Operating Systems – Concepts and Implementation, A One Semester Course

Patrick McDowell
April 9, 2020

Table of Contents

Chapter 1 – Introduction and Overview

The purpose of this book is to provide the reader with basic knowledge of computer operating systems. To do so, several topics will be touched upon, starting with an overview of operating systems, their purpose, origins, and primary structures. Following this, the discussion will head into the essentials of designing and implementing a simple OS/OS simulation. This book is not meant to be an overarching source of knowledge on Operating Systems, but more a primer that allows the reader to build a good conceptual basis from which they can create their own OS or OS simulation. The book is set up this way because in today's world of computer science curriculums there are just so many topics that must be visited, and with most programs having around 120 hours to complete a bachelor's degree, every hour is valuable.

Understanding what an operating system is, its various components, and how they work is worthwhile in its own right, but having an understanding of these topics offers other benefits. Many of the concepts that OS's are built upon are useful in the area of data collection, communications, synchronization of different programs, and the big one, understanding of the workings of Web Browsers. For now however, we will address the topic of Operating Systems by introducing definitions, concepts, and implementation strategies.

1.1 What is an Operating System?

The question of "What is an Operating System?" has many answers, and these answers usually are based on how a person is using a computer. Users will always have a different viewpoint than developers, and even among developers there will be many different perspectives. The way an applications programmer interacts with the OS is much different than that of the OS designer, or the Device Driver developer and so on. But still it is good to have a simple and generic definition. Here are a few that seem useful:

"An OS is a control program that provides an interface between the computer hardware and the user. Part of this interface includes tools and services for the user."

From Silberschatz "An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of the OS is to provide an environment in which the user can execute programs. The primary goal of an OS is thus to make the computer convenient to use. A secondary goal is to use the computer hardware in an efficient manner."

These are good definitions. One of the problems with trying to define what an operating system is and does is that for most computer users, what they see of the computer is the OS. So, for most, it is hard to separate the OS from the computer. One other problem that we have, is that of familiarity. The point here is that in today's world we take the OS for granted, sort of like walking. Think about how hard it would be to explain the

mechanics of walking. We do it all the time, so much in fact we do not think about it, much less how we do it.

A good way to get around these problems is to put oneself into the place of the first computer pioneers, those men and women who were using the very first computers to get work done. Unfortunately for them, the first computers did not have operating systems, so, in order to use the computer, they had to program it directly. If we look at how they did that, and the steps they took to make the task easier, we can learn a lot about what an OS is and how it works.

1.2 Using a Computer without the OS

The earliest computer scientists used their computers to solve many types of problems, but some of the first were designed to create range/elevation tables for artillery purposes. In essence, the Army needed to know what angle of elevation was needed in order to make an artillery shell go a certain distance. Given there were lots of kinds of shells and guns, a good set of tables would make things much easier. This was a perfect job for a computer. Suppose our scientist wanted to generate these tables using their new computers, how would they do it?

To solve the problem in modern times we would do something similar to the following:

- Develop some equations relating range and elevation, possibly including influences from wind, temperature, barrel length, etc.
- Select a language and an IDE (Integrated Development Environment).
- Develop a program that solves our equations for all the ranges from closest to furthest.
- Print out the results
- Compare the calculated results with actual field testing and adjust our equations.

This list of steps takes advantage of the computer's OS in a variety of ways. Selecting a language implies that we have more than one high level compiler and language available; using the IDE implies that we have a very useful screen editor, debugger, compiler/linker/loader combination available. Showing alpha-numeric data on the screen and even printing out the results assumes that the OS has routines available for sending individual characters to screen or printer locations, and that it was not the programmer's responsibility to do so. From the bullets above, the first and last items are the only parts of the problem that are not relying on OS facilities and services.

The first computer programmers would have tackled the problem by doing many of the same things outlined above, but they would have done it without the luxury of the OS. They would have developed their equations just as their modern counterparts did, but when it came to selecting a language, there would have been none to select. Instead the native assembly language of the available computer would have been used.

The program development process would have been somewhat like the following:

- Sketch out the equations and accompanying algorithms on paper, in some sort of high-level pseudo-code or flow chart.
- Convert the pseudo-code or flow chart to assembly code.
- Convert the pseudo-code to machine language.

So far, our intrepid programmers have been able to generate a program from the requirements, but to get that program to run they will have to do some more work. Their goal is to convert their code into a "code segment" and get that code segment into memory so that it can run. The processes of linking and loading are the modern names for these activities, but because they are automated to various degrees, modern programmers are not necessarily aware of all the work that is being done for them. Our "old school" programmers would have to do all of this work themselves, so it makes sense that they would economize their operations, i.e. reduce the amount of work that needed to be done.

The program segment consists of a symbol table, program instructions, and stack space. In the program's instructions, any function calls to library routines will have to be "linked" to the code. For the sake of argument, we will assume that our programmers are lazy and are not going to call library functions, meaning all functions called will reside in the instruction/function part of the program segment. Along with the symbol table and instructions, the program segment has a section called the stack space. This space is used to store and restore variables during function calls. It also is used as "scratch memory" for array allocation or any other memory allocation that occurs during program runtime. The stack shown in diagram 1.1 below has two pointers into it, the stack pointer (sp) and the global pointer (gp). They are set up so that they both work towards the middle of the stack. That is, as memory is used during function calls, the stack pointer is decremented, causing it to work towards the middle of the stack. As memory is used for temporary arrays and the like, the global pointer is incremented, working its way towards the middle of the stack. This system allows for maximum use of the stack's available space. The program segment structured as shown in figure 1.1 below.

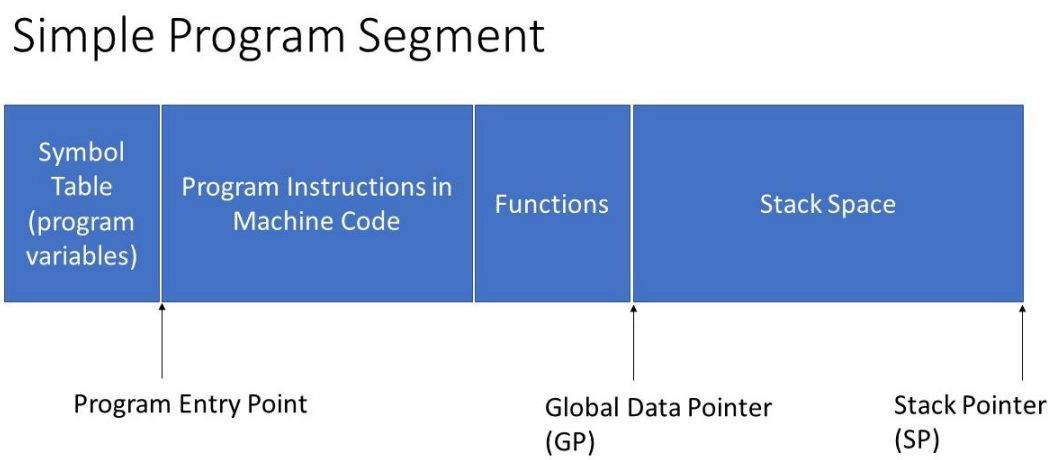


Figure 1.1 – This figure shows a diagram of a program segment. The program consists of more than the instructions. The symbol table stores the values of the variables, and the stack space provides storage for

saving values when function calls occur. It also provides a space from which to allocate memory for temporary arrays and global data.

So back to the task at hand. Our programmers would have figured out how much space the variables of the program needed, and left space for them at the beginning of the program (of course they would have allowed for any word alignment issues), then they would enter the program into the computer's memory. In modern days, this would be the job of the loader, but in a computer without an OS, there is no loader. So, the programmers would have to enter the program into the computer's memory by hand. Essentially the program would be loaded into a memory bank consisting of binary switches. Eight switches would comprise a byte, 32 a 4 byte word, and so on. The programmers would load in the machine instructions that they had translated from their assembly code (assemblers convert assembly to machine code for us in modern times). After the code was entered, a section of memory is designated for stack space, and then it is time to run the program.

To run the program the program counter register (pc) is loaded with the address of the program's entry point. The entry point is the address of the first instruction, and usually comes right after the storage for the last of the program's variables. The stack pointer register (sp) is set to the last address of the program segment (remember, as stack memory is used the sp is decremented, thus working its way towards the middle of the program stack). The global pointer register (gp) is set to the beginning of the stack. Again, all of these values are entered using binary switches. Once the pc, sp, and gp are initialized, the fetch, decode, execute process is enabled (the CPU is turned on) and the program runs, that is as long as there are no logic errors. Early programmers would have to have done several tests to make sure that they had made no mistakes in entering the program. They would run the program with input values designed to generate known results, thus testing their logic. We still do this sort of testing, it is just not as arduous of a process.

Once the program has run, we would need to get the results. That being said, where would the results be exactly? For small programs, the results could easily have been left in registers and the answers copied down on a piece of paper. Once the logic was thoroughly tested, it would be time to call a print routine, which of course would take another good bit of work.

The above narrative is provided to give a flavor of how early computers were used, and why the OS was invented. The longer programmers used this sort of a process, the more likely they would be ready to set up some tools to help them with the process. Some of the first tools that were developed were the:

- Assembler – the assembler converts assembly code to machine code. This process converts the assembly mnemonics to formatted machine instructions in binary. The table in figure 1.2 below was used in this explanation.
    - Example: the MIPS instruction "add $s0 $t0 $t1" [1,2] would add temporary registers $t0 and $t1 and place the results into program register

$s0. The MIPS assembler would convert this to the following machine code:

  - 000000 01000 01001 10000 00000 100000
  - Opcode|ornd 1|ornd 2|dest |shmnt| function
    - The first group of digits is the opcode for add.
    - Group 2 – is operand 1 as denoted by the $t0 register.
    - Group 3 – is the operand 2 as denoted by the $t1 register.
    - Group 4 – is the destination register ($s0 register).
    - Group 5 – shift amount, not used.
    - Group 6 – function code for add instruction.

- Linker – Places the address of the appropriate library functions into jump/return instructions in a program.
- Loader – Copies the output of the assembler/linker to the memory to be used by the program to run. At some point, the OS must create the program segment. This process could and most likely did occur near load time in early systems.

MIPS Register Definitions

Name	Register	Usage	Preserved on call?
$zero	$0	the constant value 0	n.a.
$v0-$v1	$2-$3	values for results and expression evaluation	no
$a0-$a3	$4-$7	arguments	yes
$t0-$t7	$8-$15	temporaries	no
$s0-$s7	$16-$23	saved	yes
$t8-$t9	$24-$25	more temporaries	no
$gp	$28	global pointer	yes
$sp	$29	stack pointer	yes
$fp	$30	frame pointer	yes
$ra	$31	return address	yes

Figure 1.2 – MIPS Register Definitions. The MIPS instruction set uses 32 registers. Some are general purpose (the $s and $t registers), while still others have a specific fixed purpose, such as the sp, and gp. [1]

- Editor – Early machines used tty (teletype) and card readers to enter programs and data. Eventually line editors and screen editors were developed.
- Monitor Program – The monitor program was the control program for the original OS's. It provided a basic command prompt-based interface for the system operator to interact with the computer.

The assembler, linker, and loader relieved the programmers of the time consuming and super tedious task of converting their code into machine code, and then having to go to memory banks and flip switches in order to enter their program. The card readers and tty machines gave the programmers a way to enter their assembly code so that the programs could be run. The monitor program gave the operator a way to keep tabs on the whole system, i.e. boot it up, load tapes with compilers from different languages, etc.

1.3 Basic Batch System

What we have just described is very similar to a basic Batch Operating System. It is named "Batch" because the system operator, would group programs that ran in the same language into batches. There would be batches of FORTRAN programs, batches of COBOL programs, batches of assembly programs, etc. Why batches? Because the system operator would have to load the appropriate compiler for each batch of programs, and then feed them into the card reader one at a time. In the first batch systems, the operator had to do this because there was not enough memory in the system to hold more than one language compiler. They ran the "batches" of programs to avoid having to continuously reload compilers.

In the previous sections, the system that was discussed was similar to that of the batch system, but loading the compilers was left out of the presentation. That discussion was probably relevant in a late 1940's or early 1950's sense; remember FORTRAN was in use by about 1954, and standardized in the 1960's. For now, we are focusing on the time right before the use of higher level languages, for simplicity and clarity in our explanations, and for being able to draw a stark contrast between what computers were like before the development of full-fledged operating systems and what we have become accustomed to in modern times.

Figure 1.3 below shows a typical memory map for a simple system that can run assembly programs. At the very lowest point in memory is the boot program and its boot strap. The "OS" in this system consists of the boot programs, and monitor. For now, we will lump the assembler and library functions in with the OS, and call the remainder the user program.

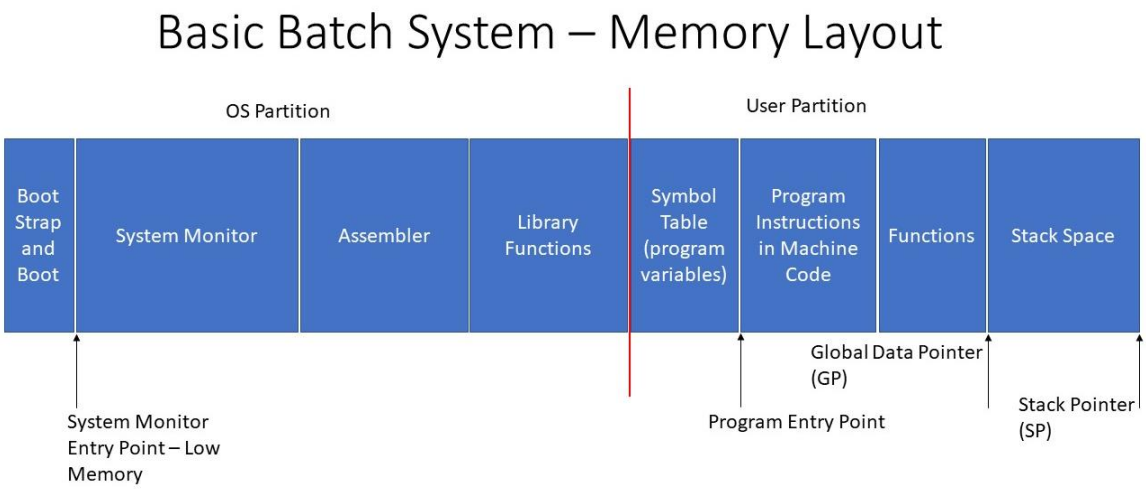


Figure 1.3 – This figure depicts a typical memory diagram of an early computer system. This system is capable of running a single assembly program at a time. The OS occupies the lower memory addresses and is represented by all the blocks to the left of the red line; the user program is to the right.

In order to use a system like this, the users would type their programs onto cards and submit them to the computer center. An operator would use the monitor program to enable the card reader to run either a single job or to process multiple jobs. Each card

stack would have a job card that identified the users account. User accounts were used for several purposes, one being the keeping track of the amount of CPU time the user used. The operator would put the cards into the reader, the cards would be read, the assembler would translate the assembly instructions into machine language and create the symbol table.

Remember that for the user's program to run, it needs to be in a program segment. In this simple OS case, the assembler can write the machine language directly to the user partition. Once the program is ready to run, the $gp and $sp registers can be set to the beginning and ending of the memory to be used for the program stack.

The user program may need to do things like read and write data to tapes, read data from cards, write to the printer, calculate square roots, sins, cosines, etc. In our system, all of these functions, the ones that many user programs may use, will reside in the library. They will be kept in memory at all times, so they are available to whoever needs them, may it be one user or another, or the OS.

For the user program to run, the OS will need to set the PC (program counter) register to the program's entry point. Essentially it will call the user program as if it were a subroutine of the OS. The MIPS function jump-return will work well (all assembly languages have a version of the jump-return function). It puts the return address in the $ra register, then jumps to the address supplied. For this example, the user program's entry point will be very near the user partition. Remember, in assembly we use the registers to supply the CPU with the values that are stored by our program's variables. The symbol table would be occupied mostly by local variables, arrays, and the like. For now, we will take this view, there will be time to get more detailed, and more complicated later.

With this design, the system operator can control the system and run single or multiple user programs. The next sections detail the pseudo code for a batch simulation. This simulation will serve as the launch point for a series of programs that will culminate in an OS simulator that will provide conceptual and hands on experience with several OS features and functions.

1.4 Batch OS Simulator

In order to make the simulator as realistic as possible we need to include as much functionality as we can. The problem is how much can we do in a semester, and how much of what needs to be done lies within the scope of an OS class. Part of the goal is to become familiar with using the various commands and tools by different OS's that we have available to us. For most of the algorithms and programs presented, we will start with a simple design sketched out in either pseudo-code or a flow chart, and then progress to bare bones versions of the code done in C/C++ in Windows and/or Linux.

The first program that will be written will be a very simple simulation of a Batch OS. Since we are simulating it on a modern computer, we will not be using card readers and boot programs. For now, we use a directory with some simple programs in it as the card

reader, create a very simple monitor program and use the compilers available to us to process the jobs. In order to make the simulation more accurate we will need a major redesign, which is a topic for another section. Our end goal is to come as close as we can to a system that has the memory layout as shown previously in figure 1.3.

For our simple simulation the basic program flow is described by the following pseudo-code:

```
Initialize Variables

While(running) {
      showMenu()

      selection = getUserSelection()

      switch(selection){
            case List Jobs:
            Use a system call to list directory contents of
            Directory where jobs are held.

            Case Run 1 job:
            jobId = getUsers job pick
            command = make a command to run that job()
            systemCall(command)

            case Run all jobs:
            command = makeJobFileCommand()
            systemCall(command)

            while(processing job file) {
                  compileCurrentJob()
                  runCurrentJob()
            }

            Case ... :
      }
}
```

A more detailed version of our pseudo-code is as follows:

```
main()
{
      /* Initialize Variables. */

      /* Main loop. */
      running = true
      while(running) {

            /* Get user input from an on-screen menu. */
```

```
valid = false;
while(!valid) {
      selection = menu();

      if ((selection >= 0) && (selection <= Last_Item)) {
            valid = true;
      }
      else {
            print("Error: Invalid Selection. ");
      }
}

/* Do user function. */
switch(selection) {
/* Case 0 turns off the program. */
case 0:
      running = false;
      break;
/* Case 1 Lists the jobs in the default job directory. */
case 1:
      /* List the c/c++ jobs in job directory. */
      systemCall("directoryList nonVerbose jobDir\*.c*");
      break;
/* Case 2 runs a job selected by the user. */
case 2:
      /* Let user input job name. */
      print("Enter job's name (no extension) \n");
      readFromScreen(myJob);


      /* Create a command to compile and link the job. */
      myCommand = createCompileAndLinkCommand(myJob);
      systemCall(myCommand);


      /* Run the job. */
      myCommand = createRunJobCommand(myJob);
      systemCall(myCommand);

      break;
/* Case 3 runs all jobs in the directory. */
case 3:
      /* Load job names in job directory to a file. */
      myCommand = createJobList();
      systemCall(myCommand);

      myFile = open(jobFile for reading);
      if (myFile has been created) {
            while(lines of data in myFile, get a line) {

                  /* Get the job from the line. */
                  readFromString(line, myJob);

                  /* Now compile and link the job. */
                  /* Create a command to compile and link
                  the job. */
                  myCommand = createCompilLinkCom(myJob);

                  systemCall(myCommand);


                  /* Run the job. */
                  myCommand = createRunCommand(myJob);
                  systemCall(myCommand);
```

```
                        }
                  }
                  else {
                        print("Error: job list could not be
                        created/opened");
                  }

                  break;
            case 4:
                  /* Set jobs directory. */
                  break;
            case 5:
            case 6:
            case n-1:
            case n:

            }

      }

}

int menuMe()
{
      int selection;

      print("0) Quit \n");
      print("1) List jobs in directory. \n");
      print("2) Run 1 job. \n");
      print("3) Run all jobs. \n");
      print("4) Set jobs directory. \n");
      print("\n");
      print("Enter selection. \n");

      readFromScreen(selection);

      return selection;
}
```

And finally here is a Windows based program written in Visual C++.

```
void main(void)
{
      bool running;
      bool valid;

      int selectMe;
      int nChars;

      char myJob[32], myCommand[132], line[80];

      FILE *in;

      printf("Hello TV Land! \n");

      /* Main loop. */
      running = true;
      while(running) {

            /* Get user input. */
            valid = false;
            while(!valid) {
                  selectMe = menuMe();
```

```
      if ((selectMe >= 0) && (selectMe <= 4)) {
            valid = true;
      }
      else {
            printf("Error: Invalid Selection. \n");
      }
}

/* Do user function. */
switch(selectMe) {
case 0:
      running = false;
      break;

case 1:
      /* List jobs in job directory. */
      system("dir /b jobs\\*.c*");
      printf("\n");
      break;
case 2:
      /* Let user input job name. */
      printf("Enter job's name (no extension) \n");
      scanf("%s", myJob);
      printf("job selected = %s \n", myJob);

      /* Run a single job. */
      /* Here is what works from the command line. */
      /* cl is the c++ compiler. */

      // cl hello.cpp /link /out:a.out
      // cl jobs\hello.cpp /link /out:jobs\a.out
      // jobs\a.out

      /* Here is what works programatically. */
      /*
      system("cl jobs\\hello.cpp /link /out:jobs\\a.out");
      system("del hello.obj");
      system("jobs\\a.out");
      */

      /* Create a command to compile and link the job. */
      sprintf(myCommand, "cl jobs\\%s.cpp /link
      /out:jobs\\a.out", myJob);
      system(myCommand);

      /* Create command to get rid of object file. */
      sprintf(myCommand, "del %s.obj", myJob);
      system(myCommand);

      /* Run the job. */
      printf("\n\n");
      system("jobs\\a.out");
      system("\n");

      break;
case 3:
      /* Run all jobs in the directory. */
      // dir /b jobs\*.cpp > jobs2Run.txt
      /* Create a file with the jobs to be run. */
      /* The /b puts in only the files in the directory, no
      extra info about the files. */
      system("dir /b jobs\\*.cpp > jobs2Run.txt");
      in = fopen("jobs2Run.txt", "r");
```

```
                  if (in != NULL) {
                        while(fgets(line, 80, in) != NULL) {
                              printf("line = %s \n", line);
                              /* Get the job from the line. */
                              sscanf(line, "%s", myJob);

                              /* Chop off the  extension. */
                              nChars = strlen(myJob);
                              myJob[nChars - 3] = (char)0;

                              /* Now compile and link the job. */
                              /* Create a command to compile and link
                              the job. */
                              sprintf(myCommand, "cl jobs\\%scpp /link
                              /out:jobs\\a.out", myJob);
                              system(myCommand);

                              /* Create command to get rid of object
                              file. */
                              sprintf(myCommand, "del %sobj", myJob);
                              system(myCommand);

                              /* Run the job. */
                              printf("\n\n");
                              system("jobs\\a.out");
                              system("\n");
                        }
                  }
                  else {
                        printf("Error: job list could not be
                        created/opened");
                  }

                  break;
            case 4:
                  /* Set jobs directory. */
                  break;

            }

      }

}

int menuMe(void)
{
      int selection;

      printf("0) Quit \n");
      printf("1) List jobs in directory. \n");
      printf("2) Run 1 job. \n");
      printf("3) Run all jobs. \n");
      printf("4) Set jobs directory. \n");
      printf("\n");
      printf("Enter selection. \n");

      scanf("%d", &selection);

      return selection;
}
```

Program Notes:

1. The c/c++ compiler used was Visual C++. It was called from the command line using the "cl" call. This call has many options, but the ones used here were for the purpose of naming the output executable "a.out". This is done because "a.out" is the default executable name in Unix and Linux, and the goal was to make this program so that it could be easily ported to the Linux environment.
2. The program makes good use of some old school methods of formatting strings. It uses the "sprintf" command extensively to write formatted strings into character arrays. This is how the various commands are built. Of special note is the double backslashes, "\\". Really there is only one backslash needed, but in order for the compiler not to interpret the character following the backslash as an escape character, double backslashes are used. (This is a standard practice.)
3. The program makes large use of the system commands. The strings are built using the "sprintf" command and submitted to the OS using the "system" function. This is a Windows only function. The Linux/Unix method is very similar; a "System" function is called.
4. The system functions used are as follows in Table 1.4:

Function	**Windows**	**Unix/Linux**
Call a system function	`system(< my command >)`	`System(< my command >)`
List files in directory, verbose	`dir`	`ls -l`
List files, not verbose	`dir /b`	`ls`
List files, send output to a file	`dir /b > <myFile.txt>`	`ls > <myFile.txt>`
Compile a c/c++ program	`cl`	`gcc`
Compile c/c++ to object file		`gcc -c`
Compile c/c++ program, specify target	`cl <myjob.cpp> /link /out: <targetFileName>`	`gcc myfun.o mymain.o –o myprog.x`

Table 1.4 – Various Windows and Unix/Linux system calls used to work with directories and compile programs.

1.5 Batch OS Simulator – A More Realistic Version

In the previous section a simple simulator that relies on system calls was developed. It can provide a glimpse of reality, but really, only scratches the surface of how a computer OS and hardware work together. In this section we will get closer; not all the way there, but close enough so that our simulator can load programs into memory and run them. We will start off very simple and slowly expand the program feature by feature. The end goal for this project is to create a system in which more than one user program can be run at once, controlled by a monitor program. The monitor program will provide the operator with the usual options, but also will protect the system from user programs that function incorrectly.

The first step is to create a simple batch system, just as we did before, but this time, we will make the system so that we can programmatically load the executable into the user section of memory and run it. In order to do that we need the following:

- Compiler – We need a compiler that we can control. After the program is compiled into assembly and the assembly is converted into machine code, the resulting relocatable executable code will need to be loaded into the memory from which it will run.
- Loader – The loader's job will be to place the executable into memory. Once the executable is in memory, its machine code instructions can be fed to the CPU one at a time.
- CPU emulator – The CPU's job is to execute the programs instructions. In order to be realistic, direct access to the CPU and its functionality is needed. This is impractical, extremely complex, and out of the scope of this course, but the problem can be greatly simplified by using an emulator tailored to the problem's needs.

1.51 Compiler for the Super Simple Language (SSL)

The compilers job is to convert high level language programs into assembly language. It follows that a high-level language is needed. The language developed for this task is very similar to C. It is like C in that it uses a semicolon to end a statement, it uses for loops, etc., etc. It is like C, except that for the purposes of this project a very simple set of keywords is enough to make a variety of programs that can be used to test the various OS functions. All that is needed is enough commands so that assignment statements, if then else statements, and looping statements can be implemented. Function calls would be excellent, but that has not been implemented as of this time.

The features that are implemented are the following:

- Variable declaration of types int, char, and float. Both scalar and arrays using open and close brackets [,] have been implemented.
- Simple assignment statements using the " + ", " – ", "*", and " / " operators of the following structure.
    - x = y;
    - x = 1;
    - x = y + z;
    - x = y + 3;
    - x[1] = 3;
- Loops using the "for" and "next" keywords.
- Branching statements using "if" and "endif" keywords
- Assignments of values to array elements such as:
    - x[3] = 3;

Table 1.5 below shows the keywords implemented for the SSL language.

Keyword	**Function**	**Example**
for/next	Create a loop	for j = 1 to 10; ….

		next

if (<condition>) then ….
endif
	Branching statement	if (j > 10) then
		….
		endif

print	Print a value of a variable or a string	print(j);
stop, end	Finish the program	stop; end;

Table 1.5 – This table shows provides a quick summary of the keywords implemented by the SSL language.

Along with the keywords, assignment statements using the "=" sign are implemented, but only in the simplest form, as described in the bullets above.

The SSL program listing below demonstrates all features of the SSL language. It includes assignment statements, for loops, if statements, and array assignments.

```
program loopy;

int sum, j, k;
int array[10];
int bigSum;

bigSum = 0;
sum = 0;

for j = 1 to 10;
      for k = 1 to 10;
            sum = sum + 1;
      next
next

if (sum >= 100) then
      bigSum = 10;
      for j = 1 to 10;
            bigSum = bigSum+1;
      next
endif

if (bigSum < 20) then
      array[0]=1;
      array[1]=2;
      array[2]=3;
      array[3]=4;
      array[4]=5;
      array[5]=6;
      array[6]=7;
      array[7]=8;
      array[8]=9;
      array[9]=10;
endif

stop;
end;
```

The SSL program below calculates the average of the numbers from 1 to 100. It includes most features of the language, except array assignments. This program contains a print

statement. Note that the print statement is not implemented for this discussion. It will be implemented by a system call and will be discussed in later sections. The reason for this is that for now, simplicity is the key. It turns out that system calls for common functions such as print, read, etc., are going to play a key part in the interaction of the OS with user programs.

```
program loop2;

int sum, j, k;
int av;

sum = 1;
for j = 1 to 10;
      for k = 1 to 10;
            sum = sum + 1;
      next
next

av = sum/j;

print(sum);
print(j);
print(k);
print(av);

stop;
end;
```

The compiler writes its output to a file called assembly.o. Below is the assembly.o file for the "loop2" program above:

```
4
0 i
1 i
2 i
3 i
***
4 setMemI 0 1
5 setMemI 1 1
6 for0: nop
7 setMemI 2 1
8 for1: nop
9 addI 0 0 1
10 addI 2 2 1
11 jleqI for1 2 10
12 addI 1 1 1
13 jleqI for0 1 10
14 div 3 0 1
15 sysCall 1 0
16 sysCall 1 1
17 sysCall 1 2
18 sysCall 1 3
```

The first line in the program is a 4. It denotes the size of the variable section (program symbol table). SSL handles variables and calculations in simplified manner in that it does not use registers. The variables are assigned an "array position" in the symbol table, references to those variables in the program are replaced by the array element number.

Of note is the implementation of the "print" statement. The print to the screen function is dependent on an OS call, thus the system call implemented by the "sysCall" instruction. In this case the system call is the print function; the print function implemented here is designed to print a memory position. Printing out a string would be implemented by a sysCall statement with a function of value 2 and an address of the string to be printed. The string would be stored at the end of the variable section of the program's symbol table.

For clarity, the assembly code is commented/explained below. Comments are indicated by the '#' sign with comment text following.

```
4     # Size of symbol table.
0 i   # Integer storage for sum
1 i   # Integer storage for j
2 i   # Integer storage for k
3 i   # Integer storage for av
***
4 setMemI 0 1     # Set sum to value of 1
5 setMemI 1 1     # Set j to value of 1
6 for0: nop       # Label for outer for loop
7 setMemI 2 1     # Set k to value of 1
8 for1: nop       # Label for inner for loop
9 addI 0 0 1      # Add immediate, add 1 to sum
10 addI 2 2 1           # Increment k
11 jleqI for1 2 10      # If k <= 10, go back to top of k loop
12 addI 1 1 1           # Increment j
13 jleqI for0 1 10      # if j <= 10 go back to top of j loop
14 div 3 0 1            # Divide sum by j and place in av
15 sysCall 1 0    # System call to print sum
16 sysCall 1 1    # System call to print j
17 sysCall 1 2    # System call to print k
18 sysCall 1 3    # System call to print av
```

Once the code is converted to assembly by the compiler, the assembler is called. The assembler does a 1 to 1 translation of the assembly file (kept in assembly.o) to machine code.

The machine code is very typical of machine code. The important thing to remember about machine code is that while there may be several types of instructions each with different formats, the opcode of all instructions is in the same place. By using this strategy, the CPU, or in this case the emulator, can decode the instruction based on the opcode, because it is always in the same place. Imagine if it were in a different place in each type of instruction; how would the CPU know where to look for the opcode so the instruction could be decoded?

The machine code for the SSL language has 4 different types of instructions:

- R Type – memory to memory operations, such as x = x + y
- I Type – immediate instructions, such as x = x+1
- J Type – jump, and jump conditional instructions.
- S Type – system calls

All instructions occupy one byte of storage. So, an executable SSL program consists of a symbol table and a stream of instructions immediately following the symbol table. Looking at the example program from above, the symbol table occupies the first 4 bytes, (bytes 0 through 3) and the instructions occupy the next 15 bytes (bytes 4 through 18). The entry point of the program is the programs first instruction, which is the setMem instruction at position 4 of the program (it immediately follows the program's symbol table). This is very similar to what we saw earlier, as illustrated in figure 1.3.

The next sections provide an overview of the rest of the process, linking, loading, and execution.

1.52 Linking, Loading, and Execution

For the scope of this project, linking is kept to a minimum. That being said, it is worth drawing a contrast to the routines that the user programs link to, vs. those that require a system call. Common routines such as math routines are kept in libraries and are linked to, but other routines that involve I/O are handled by system calls.

Library routines, such as math libraries, are used by many programs. Keeping them available in a library allows programs to link to them which helps reduce the amount of code that is in the user routine. The other thing about these routines is that they do not affect the way the OS is functioning. They could easily be replaced by a series of statements in the user program that did the same thing (increasing the amount of user code). However, functions such as printing do affect the system, so these are supported by system calls.

Referring to the "print" example, it makes no sense for users to have the need to know how to get bitmaps of pixels to the screen memory so that the hardware can display the desired text. So, part of the reason to implement these types of functions as system call is one of abstraction; the other is that of system protection. Suppose that users want to print their results to a printer; abstraction allows the system to take over the detailed work of controlling the printer's mechanical intricacies, while protection allows access to the printer one user at a time. Furthermore, since a mechanical printer is much slower than the computer system, using a system call to implement the print will allow the OS to schedule some other function while it is waiting for the printer to complete its job.

1.53 Implementation of the Process, Compiling, Linking, Loading and Execution in the OS Simulation

To make a more realistic simulation than the previous effort, as discussed in section 1.4, control over instruction processing is needed. This is the whole reason that we created

our own language, which in turn required a compiler, assembler, linker, and loader. For the SSL detailed in the previous section, linking is trivial, because we are not using any calls to library routines. It turns out that in the case of a batch simulation, loading is almost trivial also. So long as the program resides in a memory section whose starting index is 0, there is no real loading process. So why did we go through all the effort of getting control over instruction processing? The answer is: Because it will be instrumental in the development of simulating systems that run more than one process at once. But it is best to do things one step at a time. The next paragraphs detail the batch system using the added detail of control over instruction processing.

Below is the pseudo code for the system:

```
batchOS()
{

	/* Go into monitor loop. */
	running = true;
	while(running) {
		choice = showMenu();

		/* Quit. */
		if (choice == 0) {
			running = false;
		}
		/* Show current jobs. */
		else if (choice == 1) {
			jobList = listCurrentJobs();
		}
		/* Run one job. */
		else if (choice == 2) {
			/* Get a job. */
			jobList = listCurrentJobs();
			filename = get1Job(jobList);

			/* Run the job. */
			/* Compile  - put job into assembly.o  */
			SSLCompiler(fileName);

			/* Convert the assembly to byte code. It is placed
			into the b.bin file. */
			assembleMe("assembly.o", codeAreaSize, symTableSize);

			/* Read the just created assembly into temporary
			memory. */
			readRelocatable("b.bin", temp, codeAreaSize);

			/* Loader Function. */
			/* Load program segment into the user memory. */
			baseAddress = 0;
			loader(baseAddress, userMemory, temp);


			/* Run the program. */
			entryPoint = symTableSize;
			processInstructions(userMemory, entryPoint,
			codeAreaSize, baseAddress);
```

```
                    /* Show the results, by printing out the symbol
                    table. */
                    coreDump(userMemory, baseAddress, symTableSize);

                }
        } /* End while. */

        /* Shutdown. */
        shutdown();
}
```

The code above closely matches the system discussed in section 1.4, but provides much more detail. The key parts of the program are the parts that compile, assemble, link, load and run the program, as shown below:

```
/* Run the job. */
/* Compile  - put job into assembly.o  */
SSLCompiler(filename);

/* Convert the assembly to byte code. It is placed into the b.txt file.
*/
assembleMe("assembly.o", codeAreaSize, symTableSize);

/* Read the just created assembly into temporary memory. */
readRelocatable("b.bin", temp, codeAreaSize);

/* Loader Function. */
/* Load program segment into the user memory. */
baseAddress = 0;
loader(baseAddress, userMemory, temp);


/* Run the program. */
entryPoint = symTableSize;
processInstructions(userMemory, entryPoint, codeAreaSize, baseAddress);

/* Show the results, by printing out the symbol table. */
coreDump(userMemory, baseAddress, symTableSize);
```

The SSLCompiler translates the SSL program into assembly and writes it to the "assembly.o" file. That file is read by the assembler, translated to machine code and written to the "b.txt" binary file. The machine language is read into the temp array by the readRelocateable function. The loader then copies the temp array to user memory starting at the address defined by baseAddress. Currently that address is 0, but later when multiprogramming is introduced, the base address will be given different values, depending on the situation. Once the program is in memory, the CPU emulator function, processInstructions, is called. Upon completion, the coreDump function prints out the completed programs symbol table, thus providing the user with whatever calculations the program was designed to generate.

It is important to note that in a real system the process outlined above would be more efficient. For instance, it is not necessary for the compiler to write the assembly.o file. This is done as a sanity check, and also so that testing of the components downstream from the compiler can be done on an individual basis. The same is true about the b.bin file. During development, it is common to structure software in this manner; once all issues have been resolved, efficiencies can be introduced.

1.6 Exercises

1. Write an SSL program that loops from 0 to 10 and writes out the loop counter.
2. Use the SSL compiler to compile the program from exercise 1. Examine the assembly.o file and make sure that it is correct.
3. Use the assembler program to generate the b.txt file from the assembly.o file created from exercise 2.
4. Write the readRelocatable function and the loader function.
5. Call the processInstructions function to run the binary file created by the assembler program. Your program should print the numbers to the screen via the print statement implemented by the system call in processInstructions.
6. Call the coreDump program to show your SSL program's results.
7. Now integrate your code into a "batch like" system similar to the one described by the pseudo code in section 1.53. Your system should be able to run several jobs in sequence. The jobs can be loaded into a directory and then your system can process them one at a time.

True Story – "A Piece of Cake"

It turns out that it helps to have a distraction sometimes. Distractions can let a person's mind relax a bit after working on a stubborn problem. So, in this chapter and some of the others we will present a true story that may or may not be relevant to the material, but, for sure, somewhere in it there is a lesson. These stories, while related to the reader by the author, did not necessarily involve the author or anyone the author knows; they have been offered to the author by various anonymous "colleagues and computer and science enthusiasts".

This first story is offered by my twin brother; it is told in first person…

I was rewiring my house that was built in 1940, meaning that the wiring was old, and not up to code. Many older homes do not have breaker boxes, they have fuse boxes, so when a circuit draws too much current, the fuse blows. Replacement is straight forward, but compared to more modern breaker setups, it is more tedious because the fuse must be replaced, whereas a breaker can just be reset. I was having constant problems with fuses, because over the many years that the house was used, many "extra" circuits had been tied into the existing wiring, resulting in fuses that were always on the edge of being overloaded. Too many trips up to attic dinking around with fuses!

At the very least, I wanted to replace my old fuse box with a nice breaker box so that I could just flip a switch rather than replace a fuse. The first job to do was to turn off the power to the house, an easy thing to do; all that needs to be done is to go outside and flip the main breaker. It was a nice day, I walked out to flip the main breaker off knowing it would be right next to the power meter. When I got out there, I could not find it. I called my friend Herb to see if he had any ideas. Herb said, "Lots of those old houses don't have main breakers. If you can't find it, just yank out the power meter." This made sense to me, the power meter just plugs into a fixture on the side of the house, no big deal. The thing to know about power meters is that the wires from the power pole go to

the fixture, allowing all the power to flow through the meter, so it can measure it, before it goes into the house. I took Herbs advice, grabbed the meter, and gave it a tug. Nothing. I tugged again. Nothing. It must have been in there a while. I will get that sucker out. This time a big tug. Out it comes, along with yanking the pole wires out of the fixture. Oops, not good, but at least the power was now turned off.

Upstairs to the attic I went. Besides being hot, it was an easy uneventful job of putting in the new box, transferring the wires from the fuse box, and plugging in the breakers. Right about when I finished up, Herb comes walking in. "Got any more cake?" he asks. My wife now, girlfriend at the time, always made nice cakes and pies, and Herb loved them. I did too, so I would hide them in various spots, sometimes in the microwave, sometimes in the oven, because Herb would come over and eat them when I was not home. I did not mind sharing, but it was fun to make him go on an Easter egg hunt. I told him sure, got him a piece of cake, and he started gobbling it down. Herb is a human truck – he can work very hard, he is smart, and like a truck he likes lots of fuel; he can ingest nearly anything, at a rapid rate, and not blink an eye. After about half the cake, he asks me if I had run into any problems. I said "Well, just one little one. Come see."

So, there we were, staring at the pole wires hanging on the side of the house. "What do you think?" I asked. "Nothing to it" was the reply. I figured Herb wanted to call the power company, do the easy thing and let those guys put their wires back into the fixture. Right about when I was going for the phone Herb asks for a screwdriver. "Huh??" "And maybe a crate, something we can stand on." Herb adds. Uh-oh, we were going to fix this ourselves. Not good. Very scary. Bad. "What's the crate for?" I ask. "So we are not grounded." comes the reply. Makes sense to me. I did have a crate; I throw it on the ground in front of the house by the power fixture and wires. "What's the plan?" I ask. "We will just screw those lugs right back into the fixture." Herb replies. Remembering that the lugs are delivering unmetered, no breakers, no nothing, full power from the pole, my thinking is "You first?!", but before I can say anything, Herb says, "I'll give it a shot, just knock me off the crate if anything bad happens." I hear myself say "No problem." We jump on the crate, Herb takes the screwdriver, and very gingerly puts the tip of it into the slot on the screw that luckily stayed in the power lugs mounting hole. We look at each other – so far so good. Next, he very gently scooches the lug to the fixture and starts screwing the lug in. At this point I cannot believe my eyes! Its working. As he tightens it in, I can clearly see the screwdriver contacting the lug and the screw at the same time, it should be shorting out all over the place. Squeak, squeak, squeak, three turns and it is tight. We look at each other in disbelief. Herb says "That was easy." "Piece of cake." I reply.

We are somewhat relieved and totally befuddled, but happy things worked out; one more to go. Herb starts the process with the next lug. Gingerly now, the screw driver slips into the slot on the screw; he guides the lug to the mounting plate, just a few turns, and "Holy #$% !!", the whole thing blows up in a shower of sparks. I shove Herb off the crate, and we dive for the ground. We are both screaming like 7-year-old school girls, except we are saying, well mostly unintelligible curse words. "What happened?" I ask. "No clue" Herb replies. "Let's try it again.", he says. Great! #$%%# This time we somehow get

the lug back in, with just a few minor sparks. I plug the power meter in, the lights come back on, etc. I look at Herb and say “Thanks Dude!” He replies, “No worries, lets get another piece of cake”.

Chapter 2 – Multi-Programming

In the previous chapter we looked closely at various basic OS definitions and concepts. We saw that running a program without an OS was a tedious endeavor at best and could possibly become a mistake filled frustrating experience without a lot of careful planning and design to prevent it from becoming so. The first computer pioneers lived this, so they quickly came up with "tools" to help them make programming the computer easier, and to increase their productivity.

The first usable OS's were the Batch systems. They featured a monitor program from which the operator controlled the system along with a card reader so that the operator could input user programs into the computer, run them, and then a printer so that the user programs could output their results.

The general operating narrative follows: the programs were separated into "batches" based on language; the appropriate compiler was loaded into the system; then one by one each user job would be run. If a program did not compiler correctly, the compiler would print an error message to the printer, and the program with the message would be returned to the user so they could fix it. If the program compiled correctly, the system would link it, load it, and run it. If the program ran correctly, the printed output would be returned to the user, otherwise, a "core dump" would be printed and that would be returned (along with the cards).

Such was early computing. One major thing that is not ever mentioned in the discussion above is user/computer interaction. In a batch system there was none to speak of. The other thing that a batch system suffers from is excess CPU idle time. Since only one job was run at a time, whenever a program needed to output a result, or read data from a file (usually on a tape) or from standard input (usually stored on cards at the end of the users card stack), the CPU would have to wait for the electro-mechanical systems that the tape drive and card reader were made of. The CPU is orders of magnitude faster than these components, so it would sit idle for thousands if not 10's of thousands of CPU cycles, waiting for the hardware to complete it's task. The original cure for this was Multi-Programmed Batch.

2.1 Multi-Programmed Batch Systems, an Overview

Multi-Programmed Batch systems are very much like batch systems except that they load several jobs into memory at once. When the CPU has to wait for an I/O request done by the current program, it sets that operation in motion, and switches to another job. When the OS's I/O subsystem completes the I/O operation, it notifies the CPU/OS, and the CPU/OS can switch back to the original job. In order to accomplish this, the multi-programmed system has a job pool in which the jobs reside, and a scheduler that sends jobs to the CPU to be processed. The OS has to keep up with the status of each of the jobs in job pool so that it can restart them, and know when they are complete, or in an I/O phase. This has to be done in a transparent manner, meaning that the process has to take place so "smoothly" that the user's do not realize it has occurred.

The advantage of this system is that CPU idle time is drastically reduced. Downsides include increased complexity, and it still does not address the user/OS/Computer interaction issues. User/OS/Computer were first addressed by a system called "Time Sharing". As it turns out, we will see that this is a somewhat natural extension of mult-programming; it will be described it in later sections and chapters.

2.2 An Approach to Multi-Programming Implementation

In order to implement the Multi-Programmed system, an incremental approach will be taken. Starting with the simple batch system developed in the previous chapter, modifications and additions will be made and tested until a simple multi-programmed system has been developed. This system will then be modified once again in order to reduce memory waste, and then finally, it will modified once again, so that it approximates (grossly) a Time-Sharing system.

2.21 Simple Multi-Programmed System Development – 2 Fixed Partitions

The first steps that are required to create the multi-programmed system are to create the job pool, and to set up the CPU so that it can switched between 2 running processes in a transparent way. To do this, a system that loads 2 programs, and runs them concurrently until they are complete will be developed. This will be the first step in developing the complete multi-programmed system. Below is an outline of the steps to be followed in the development leading to a multi-programmed system using dynamic memory partitions.

- Develop a system with 2 partitions.
    - Alternate between jobs for a specified number of instructions. That is, run one job for "n" instructions, suspend the job, and run the other job for "n" instructions. Alternate in this fashion until both jobs are complete.
    - When an I/O statement is encountered, suspend the job, and run the job in the other partition.
- Generalize the above system to multiple fixed partitions.
    - Introduce memory management; track free memory partitions.
    - Introduce job scheduling
        - First come, first serve
        - Shortest Job First, Shortest Job First Pre-emptive
        - Round Robin
- Switch to dynamic partitions.
    - Track memory holes.
    - Introduce memory hole selection algorithms.
        - First fit
        - Best fit
        - Worst fit

In this chapter, the first two bullets will be covered extensively. The last bullet is a memory management topic and will be covered in the memory management chapter. In

that chapter, dynamic partitions will be detailed and implemented as part of the contiguous memory discussion.

For the first step in the implementation process, user memory will be split into 2 equal sized memory partitions. One program/process will reside in the first partition, and another program/process will reside in the second partition. The CPU will be controlled so that it executes a set (specified) number of instructions from a process; upon completing those instructions, it will switch to the next process, then back to the first, and so forth. It will continue like this until both processes are complete. The CPU will output a status value that indicates if the process is complete, has encountered an I/O statement, or has executed its instructions with no issues. The memory layout is illustrated in figure 2.1 below.

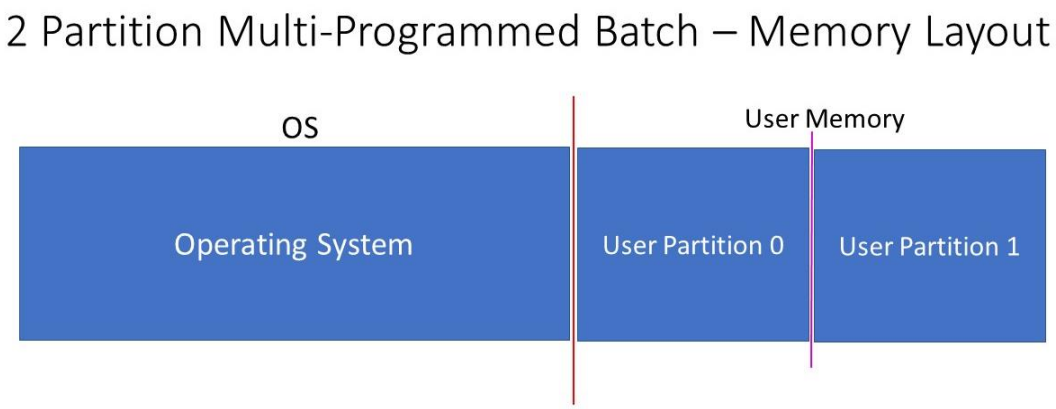


Figure 2.1 – This figure illustrates the memory layout of a 2-partition system. User memory is split into 2 equal size pieces, allowing the OS to run 2 processes at once.

For this scheme to work, a program/process must be able to run in either memory partition (It does not know where it will end up). Note, there is a difference between a program and a process. The two definitions below are not exact, but they do draw a contrast between a program and a process.

program – A set of instructions designed to accomplish a task. It is in storage somewhere waiting to be run. It is a passive entity.

process – An executable set of instructions in a program segment in memory and currently either running, waiting to be run, or waiting for I/O. It is an active entity being tracked by the OS.

From now on, entities that have not entered the OS's execution cycle/process will be referred to as programs; entities that are being executed by the OS will be called processes.

As stated earlier, processes need to be able to run in either user memory partition. To do so, the compiler/assembler builds code based on logical addressing. Logical addressing assumes that the beginning of the process is at address zero. So, the symbol table starts at

address zero, and the first instruction, the process's entry point, follows after the last symbol. For the process to run in either partition, the beginning of the process is placed at the first address in the partition, and while the process is running the address of the beginning of the partition is added to the process's logical address. The resulting address is called a physical address. The equation below illustrates this point:

physical address = base address + logical address

- physical address – This address is actual location in the computer's memory.
- base address – This the address of the beginning of the memory partition in which the process has been loaded.
- logical address – This address is generated by the CPU. It assumes that the processes symbol table starts at address zero.

2.22 Process Loading

For this step, user memory will be divided into two user partitions. Each partition will have a base address. As programs become processes, they are loaded into the partitions to be run. The loading process is essentially a simple process of copying the programs binary executable into memory. In more complicated systems (pretty much every other system in existence) the loading process also involves the creation of the program segment.

Later in this chapter memory will be divided into more fixed size partitions so that more processes can be concurrently run. And then after that the fixed size requirement of the partitions will be dropped. This will help alleviate the memory fragmentation.

2.23 CPU Control

In the simple batch system described in the previous chapter, once the process was loaded into memory, and began execution, CPU execution of the process's instructions did not stop until the natural termination of the process. The pseudo code below illustrates this point:

```
/* Run the program. */
entryPoint = symTableSize;
processInstructions(userMemory, entryPoint, codeAreaSize, baseAddress);
```

In order to run two processes, and run them concurrently, each process lives in its memory partition, and the CPU alternates between them. How is this done? In this case, the CPU is set up so the number of instructions that it runs can be specified. In the first tests, the instruction number is set to 10. So, the process in partition 0 runs for 10 instructions, then is suspended, and next, the process in partition 1 is run for 10 instructions, then is suspended. The system alternates between the two processes until they both complete.

In order to do this, the state of the process must be tracked while it is running, meaning that the CPU controller will need to be able return a status of the process. A successful

run of 10 instructions results in a status value of 1 being returned; if the program completes before the 10 instruction limit is reached, a status value of 0 is returned; and finally, an I/O statement is encountered, the status value of 2 is returned. The pseudo code for the CPU Controller is shown below.

Inputs:

- memory – a pointer to computer memory.
- entryPoint – 1st instruction of the process, it is the location immediately following the symbol table.
- progSize – the number of 4-byte words in the process.
- baseAddress – the address of the beginning of the memory partition that the process inhabits.
- N – the number of instructions that are to be executed, unless the process completes or hits an I/O statement.
- PC – the value of the program counter. This is the "array element" in which the current instruction is located.

Functions:

- processInstruction – This function decodes and executes the instruction held in the IR variable. If the instruction decodes to a system call, the return value (IO) is set to *true*. The value of PC is changed if the instruction executed was a jump or conditional jump.

```
char CPU_N_Instructions(int *memory, int entryPoint, int progSize,
                        int baseAddress, int N, int PC)
{
      int IR, limit;
      int count = 0;

      char status = 0;
      bool running = true;
      bool eyeOhh = false;

      limit = progSize;


      while(running) {

            IR = prog[PC];

            /* Process the instruction. */
            IO = processInstruction(IR, prog, baseAddress, PC);

            PC++;

            /* If program has completed, shut it down. */
            if (PC == (baseAddress + limit)) {
                  running = false;
                  status = 0;
            }
            /* If program has completed n instructions, return. */
```

```
            else if (count == (n-1)) {
                  running = false;
                  status = 1;
            }
            /* If I/O statement encountered, set flag, return. */
            else if (IO == true) {
                  running = false;
                  status = 2;
            }
            count++;


      } /* End while. */

      return status;
}
```

2.24 Twin Partition Multi-Programming OS Control

The pseudo-code for a 2-partition system in which 2 processes are loaded into the partitions is shown below. Notice, this code does not load a new job up termination of the current running jobs. It compiles, assembles, and loads 2 jobs into their partitions, and then runs them 10 instructions at a time, alternating between the jobs, until they are both complete.

```
/* Load 2 jobs into memory and run them. */
else if (choice == 4) {
      /* Load job list file. */
      buildJobFile("multiProgJobs.txt", currentJobs);

      /* Get 1st job name. */
      getNextJob("multiProgJobs.txt", fileName);

      /* Create a full path to job1. */
      sprintf(fullFilePath, "%s\\%s", currentJobs, fileName);

      /* Build the assembly for the job. */
      /* The assembly file is assembly.o */
      SSLCompiler(fullFilePath);

      /* Build job binary, place it in file "b.bin". */
      assembleMe("assembly.o", codeAreaSize0, symTableSize0);

      /* Set job 1's entry point. */
      p0Entry = symTableSize0;

      /* Load relocateable to User Partition 0. */
      loadRelocatable("b.bin", USER_0);

      /* ********************************************************** */

      /* Get 2nd job name. */
      getNextJob("multiProgJobs.txt", fileName);

      /* Create a full path to job1. */
      sprintf(fullFilePath, "%s\\%s", currentJobs, fileName);

      /* Build the assembly for the job. */
      /* The assembly file is assembly.o */
      SSLCompiler(fullFilePath);

      /* Build job binary, place it in file "b.bin". */
```

```
        assembleMe("assembly.o", codeAreaSize1, symTableSize1);

        /* Set job 1's entry point. */
        p1Entry = symTableSize1;

        /* Load relocateable to User Partition 1. */
        loadRelocatable("b.bin", USER_0 + PartitionSize);









        /* Execute the jobs, n instructions at a time. */
        p0Run = true;
        p1Run = true;
        PC_0 = USER_0 + p0Entry;
        PC_1 = USER_0 + PartitionSize + p1Entry;
        while((p0Run == true)||(p1Run == true)) {
              /* Run job in partition 0 for n instructions. */
              if (p0Run) {
                    status = processNInstructions(userMemory, p0Entry,
                    codeAreaSize0, USER_0, 10, PC_0);
                    if (status == 0) {
                          p0Run = false;
                    }
              }

              /* Run job in partition 1 for n instructions. */
              if (p1Run) {
                    status = processNInstructions(userMemory, p1Entry,
                    codeAreaSize1, USER_0 + PartitionSize, 10, PC_1);
                    if (status == 0) {
                          p1Run = false;
                    }
              }
        } /* End while. */

}
```

The next step is to generalize the code so that multiple partitions can be used. In this version, when a job completes, another job will be loaded into the now available partition. Several “book keeping” structures will be needed to accomplish this goal, the first being a simple way to track which partitions are being used and which ones are available, and the next being a way to track all of the information in each process. For the first rendition of this multi-programmed OS, the jobs will take turns running, 10 instructions at a time, just as before, except there will be many more jobs. As it turns out this is a scheduling strategy is called “Round Robin”. It along with other scheduling methods will be covered in later sections.

2.3 Multi-Programming Using N Fixed Partitions

The previous sections describe a system with 2 fixed memory partitions. It is a step up from the simple batch system, but it is limited to running only 2 processes concurrently. In this section the twin partition system is generalized into an “N Fixed Partitions

System". The advantages are that fragmentation of memory/waste of memory is reduced, and the OS can run up to N processes at once. The disadvantages are that there is still waste of memory, i.e. what happens to the memory in the partition that the process does not use? Also, what happens if a process is bigger than the partition? The short answer to that question is, the process cannot be run.

In order to create this system memory management is introduced; also the OS tracks the processes in data structure called the PCB (process control block). At this point in the development, process scheduling has not been explicitly discussed; that is a topic for discussion in later sections and chapters. So for now, process scheduling consists of checking to see if a memory partition is holding a process, if it is, the process is given some CPU time.

2.31 Required Data Structures for the Fixed Partition System

Whether a partition is occupied or not is tracked by a simple Boolean array with same number elements as memory partitions. Each time a process is loaded into a partition, the corresponding Boolean array member is set to "true". Upon completion of a process the associated Boolean array element is set to "false".

Once a process is created and loaded into a memory partition, its relevant information is kept in a data structure called a PCB (process control block). This is very important because this data allows the OS to track the processes state, allowing for the process to be suspended and resumed in a seamless manner. Below is a code snippet that illustrates the just described data structures:

```
bool partitionAvailable[nPartitions];

/* Structures/Classes used. */
struct PCB {
      int pid;
      int partitionNumber;
      int base;
      int limit;
      int entryPoint;
      int codeAreaSize;
      int pc;
      int status;
};

PCB pcbs[nPartitions];
```

As noted earlier the availability of a memory partition is tracked by the Boolean array partitionAvailable. The PCB structure holds the relevant data so that the OS can keep tabs on running processes. The various parameters in the PCB are described in the bullets below:

- pid – The pid variable is the process id. This is a unique number used to identify the process.
- partitionNumber – This variable identifies the partition number in which the process is currently loaded.

- base – the base address of the process. This is the same as the base address of the partition. For example, in a 2 partition system 1024 words for user processes, the base address of the 1st partition would be 0, and the base address of the 2nd would be 512.
- limit – Currently this variable indicates the process's exit point. It is equivalent to the size of process.
- entryPoint – This variable holds the address of the first instruction of the process. This is the point immediately following the process's symbol table.
- codeAreaSize – This is the size of the process in words.
- pc – This is the current value of the processes program counter. Initially it is set to the process's entry point. As the process runs, it gets updated so that the process can be restarted exactly where it left off.
- status – This variable records the status of the process. Possible values are 0, meaning the process has completed, 1 meaning the process completed N instructions successfully, 2 meaning an I/O statement was encountered (in the form of a system call), and 3 meaning that process is in a "running" state. The status value of 3 is the initial value.

2.4 Control Flow for the Fixed Partition System

The multi partition system works in a manner that is very similar to the twin partition system. The pseudo code below illustrates the concepts of the process.

```
Initialize variables that track memory partition usage and process
tracking (PCB array)

While(there are programs to be run) {

      for each available fixed memory partition:
            Create a process and load it into the partition.
            Record it's data in the PCB array.

      for each occupied fixed memory partition:
            Run the resident process and record its status.
}
```

A much more detailed set of code is below:

```
/* Initialize data structures. */
/* Available partitions. */
for(j=0;j<nPartitions;j++) {
      partitionAvailable[j] = true;
}

/* Load job list file. */
buildJobFile("multiProgJobs.txt", currentJobs);

/* Go into a job processing loop. */
jobBinEmpty = false;
processing  = true;
while(processing) {
      /* Load any partitions that are available. */
      for(j=0;((j<nPartitions)&&(jobBinEmpty == false));j++) {
            if (partitionAvailable[j] == true) {
                  /* Get job name. */
```

```
                gotAJob = getNextJob2("multiProgJobs.txt", fName);

                /* Create a process,
                   Compile, assemble, load, create PCB. */
                if (gotAJob == true) {

                      /* Create a full path to job1. */
                      sprintf(fullFilePath, "%s\\%s",
                              currentJobs, fName);

                      /* Build the assembly for the job. */
                      /* Compile  - high level -> assembly. The
                         assembly file is assembly.o */
                      SSLCompiler(fullFilePath);

                      /* Build job binary. By default the
                         assembler will put file into a
                         file called "b.bin". */
                      assembleMe("assembly.o",
                                  codeAreaSize, symTableSize);

                      /* Set job 1's entry point. */
                      jobEntry = symTableSize;

                      /* Load relocateable to User Partition j. */
                      base = USER_0 + j * partitionSize;
                      loadRelocatable("b.bin", base);


                      /* Create PCB for this partition. */
                      limit = base + codeAreaSize;
                      myPCB = createPCB(pid, base,
                              limit, codeAreaSize, jobEntry);
                      pcbs[j] = myPCB;
                      pid = pid + 1;


                      /* Set flags to indicate partition
                         is occupied. */
                      partitionAvailable[j] = false;
                }
                else {
                      /* Set a flag to indicate job bin empty. */
                      jobBinEmpty = true;
                }
          }

}

/* Process jobs in partitions. */
for(j=0;j<nPartitions;j++) {
      status = pcbs[j].status;

      /* Run job. */
      if (status > 0) {
            /* Get ready to run job in partition j. */
            jobEntry = pcbs[j].entryPoint;
            codeAreaSize = pcbs[j].codeAreaSize;
            baseAddress = pcbs[j].base;
            PC = pcbs[j].pc;

            /* Run job in partition j for 10 instuctions. */
            newStatus = processNInstructions(userMemory,
                                             jobEntry,
```

```
                                                codeAreaSize,
                                                baseAddress,
                                                10, PC);

                        /* Update PCB. */
                        pcbs[j].status = newStatus;
                        pcbs[j].pc = PC;
                  }
                  else {
                        partitionAvailable[j] = true;
                  }

            }

            /* Check to see if all processing is complete. */
            if (jobBinEmpty == true){

                  atLeast1JobRunning = false;
                  for(j=0;j<nPartitions;j++) {
                        if (pcbs[j].status > 0) {
                              atLeast1JobRunning = true;
                        }
                  }

                  processing = true;
                  if (atLeast1JobRunning == false) {
                        processing = false;
                  }
            }

} /* End while processing. */
```

The above detailed code follows the conceptual code very closely. It relies on most of the same functions that the Twin Partition Multi-Programmed Batch system from the earlier section used. That being said, the data structures discussed in section 2.3 are used to track the status of the system's memory partitions and the status of the processes being run.

2.5 Time Sharing Systems

The Multi-Programmed Batch system solved the problem of excessive CPU idle time, but did not make any improvements in the area of user/computer interaction. It is essentially a much more efficient batch system, controlled by a monitor program and an operator. Because the CPU can switch from job to job, thus avoiding the wait times caused by slower hardware such as printers, system throughput is vastly improved. But, users still needed to submit their cards/tapes to the computer center so that an operator could process them.

Program development suffers because of this. To put this into perspective, imagine that every syntax error incurred during program development required 8 hours to correct. That seems ridiculous, but that was nearly the case in the early days of computing, for some users. Why 8 hours? Depending on the load of the computing center, it could take that long or longer for the operator to process all the various jobs submitted. Some computing centers had card punches and card readers available so that users could submit their jobs and then correct them. This was much better than having to rely on the

operator, but still, no where near as nice as having an on screen editor, or better yet a dedicated IDE (Integrated Development Environment).

One of the first remedies for the user/computer interaction dilemma was the Time Sharing system. In this system, each user had access to a terminal on which a monitor like program ran. From this program the user could edit, compile, link, and run their programs. They also had access to a user disk and directory system. The DOS prompt on modern Windows systems, and the terminal program on Linux systems are very similar to the monitor programs used on the early time sharing systems. The bullets below briefly describe these systems.

- Mainframe computers used time sharing to give users, located at remote terminals, (sometimes called "dumb terminals") the illusion that each user had the computer to themselves.
- Introduces user/computer interaction. Interaction helps in:
    - Finding compiler errors (syntax errors).
    - File editing is much more convenient. A screen editor can be used vs cards or a teletype machine.
    - Debugging code while it is running.
    - Execution of multi-step jobs in which later jobs depend on results of earlier jobs.
- Multiple jobs are executed by the CPU switching between them. But the switches occur so frequently that the users may interact with the programs as they run. The CPU is "time multiplexed between the users".
- Requires:
    - User accessible on-line file system.
    - Directory system
    - CPU scheduling
    - Multiprogramming.

The implementation of the Time Sharing system relies on time multiplexing. Time multiplexing refers to a method of distributing CPU slices among resident jobs/processes. The designers of these systems wanted each user to be able to interact with their terminal in a pleasant, comfortable, and effective manner. This meant that each user's monitor program needs to be serviced by the OS several times per second, so that to the user, it appears that they have their own computer.

For things to appear smooth, no flickering or stop motion, the frame rate of terminal needs to be anywhere in 16 to 24 frames per second. If there are no graphics (the terminal only displays alpha-numeric data), then 16 frames per second may be adequate. People who type really fast may see some stop motion at lower frame rates. For a very basic example, imagine that a mainframe computer has 100 terminals that it is servicing. Each of the 100 terminals needs to be visited 24 times per second, meaning that each, visit by the OS to a terminal is 1/24/100 seconds long (1/2400 seconds). This seems like a really short time, but if the CPU processes 1 million instructions per second, about 417 user instructions will be executed upon each visit. In 1 second, each user will be able to

consume about 10,000 CPU slices. For I/O driven programs (programs like editors, spreadsheets, etc.) this would be entirely workable. For raw computing programs, things would slow down.

2.7 System Stability Considerations

In the previous chapter's batch implementation, once the user job was loaded into the system's memory, it was allowed to run to completion. The way it was allowed to run to completion is that it used as many CPU slices as was required until it reached the last instruction of the process. This is excellent for avoiding context switches, thus avoiding OS overhead, but in reality it is very bad. It is bad because it offers no protection against an errant process. The pseudo code shown below would cause the simple batch OS to lose control of the system:

```
x = 0;
for(j=0;j<10;j++) {
   running = true;
   while(running) {
      x = x+1;
   }
}
```

In this code snippet, the "running" variable can never be set to false, so the while loop will run indefinitely. Thus, control will never return to the OS. In the multi-programmed systems, a program like this would run indefinitely, or until the operator used the monitor program to shut it down (provided the monitor program had such provisions). In the multi-programmed OS, the user processes were given an allotment of CPU slices, then control returned to the OS. The point is, no user program should ever be given the opportunity to monopolize control of the CPU. This can be done by periodically returning to the monitor program to check the OS's status. The control loop from section 2.4 should me modified to be as show below. Notice that after the user jobs are run for a finite amount to time the OS status is checked (the status could be checked between user jobs also).

```
Initialize variables that track memory partition usage and process
tracking (PCB array)

While(there are programs to be run) {

      for each available fixed memory partition:
            Create a process and load it into the partition.
            Record it's data in the PCB array.

      for each occupied fixed memory partition:
            Run the resident process and record its status.

      Check OS status;

}
```

The status check could involve many things, but mainly it will check for flags being set. Flags are often set to indicate a process needs attention. Various flags could include a keyboard interrupt (indicating a key was hit), a divide by 0, a system timer, etc.

For example, suppose a user on a time sharing system notices that their job seems to caught in an infinite loop. In that case they may do a “control c” sequence to try to abort the job. This would set the keyboard interrupt flag for their terminal. When the OS came to check on the OS’s status, it would pick up the “control c”/abort sequence and end the program. This would end the user program, and the terminals monitor program would resume, giving the user a chance to fix their program.

2.7 Exercises

1. What is a multi-programmed batch OS and how did it improve on the regular batch OS?
2. Show some pseudo code for a batch OS, then add into the pseudo code needed to convert it to a multi-programmed batch.
3. What is a PCB and how is important to a multi-programmed batch system?
4. What are some of the typical components of a PCB?
5. Extend the simple batch system, as detailed in Chapter 1, to a two partition system as described starting in section 2.2. Make sure and to take a very incremental approach to the development.
6. Extend the Twin Partition System developed in exercise 1 to a Multiple Fixed Partition Multi-programmed system as described starting in section 2.3. Once again, be very incremental in the development cycle. It is extremely easy to make addressing mistakes during process initialization.

True Story – "Brake This!"

This story comes from the author's older brother. It is told in first person…

My first car was a 1966 Mustang. It was a durable, fun, and just dangerous enough car to instill a life-long passion for fast cars and especially fast Mustangs in me. When I got out of undergraduate school I purchased a 1987 Mustang GT convertible. This car was truly a fun car. The car would go fast, break the tires loose at will in first and second gear; in general just what everyone needs, including my wife. My '87 inspired my wife, at the time my girlfriend, to purchase a Mustang of her own. She bought a 1996 Mustang GT, standard, like mine, and she loved it.

We used to race each other out of stop lights sometimes. I could always beat her in the '87 for some reason, but she would always say "But I got a way longer and louder peel out at the light!" She was right, but that is not how you win races. To her it did not matter, what did matter was who made the most noise and tire smoke. She loved it. She was good with a stick shift and had lots of fun. After we were married, we used to pull up to our house, in a rural area thank goodness, and have peel out contests. It was her thing. She would just laugh and peel out more.

When her car finally wore out, we got her a new 2005 Mustang GT. This time it was an automatic. She loved it and had been driving it for a week or two when she pulled up to the house and said "I love my new car, but how do I make it peel out with this automatic?" My friend from undergraduate school, "Dudz", and I laughed and said "Power brake it!" "What's that?" came the reply. Dudz explained the basics; put it in gear, put your foot on the brake, bring up the rpms, let go of the brake and floor it at the same time. "Okay!" she giggled. She aimed the car up the driveway and put the car in gear. Her window was down; she said "Left foot on the brake, other foot on the gas?" Dudz and I laughed, "Yep, that's all there is to it!" The rpms came up, a lot, the car got lower on its haunches, my wife giggled with excitement, and then she hammered it! The tires spun wildly in a cloud of smoke and the car took off like a rocket, except backwards.

My sweet wife hit the brakes just soon enough to not hit the house, the truck, and everything else. She screamed with laughter "I did it! That's so fun!!" Dudz and I looked at each other in bewilderment. Not one to curb enthusiasm, I said, "Maybe try drive next time?"

Chapter 3 – Processes and Process Scheduling

In the previous chapters a progression of early OS's and the basics of their inner workings was discussed. The programming examples and details of those examples has provided the infrastructure needed to implement batch processing and multi-programmed batch processing. Those discussions have introduced several fundamental concepts, including process fundamentals, CPU control, and basic memory concepts. At this juncture in the OS discussion most texts introduce processes and process scheduling.

Looking at the state of the development of the OS simulation from the exercises of the previous chapters, development needs to progress in two directions: memory handling and process management. The current memory model is a multiple fixed partition scheme. While this memory model has its disadvantages, it is sufficient to support basic scheduling algorithms. Currently, the scheduling used is a simple round-robin system; each process is given a chance to run in a sequential fashion, repeating until all processes are complete. In this chapter we will explore the basics of processes and process scheduling; we will look at the concepts, and then implement the algorithms within the framework of the previously developed simulation.

3.1 Processes, Process Operations, and Process States

A process is an active entity that the OS is tracking during its execution lifecycle. This is different than a program. A program is a set of instructions designed to accomplish some task. It is a passive entity that is in storage somewhere. The program could be written in a high-level language, or it could be an executable binary, but it has not entered the execution lifecycle. As defined earlier:

program – A set of instructions designed to accomplish a task. It is in storage somewhere waiting to be run. It is a passive entity.

process – An executable set of instructions in a program segment in memory and currently either running, waiting to be run, or waiting for I/O. It is an active entity being tracked by the OS.

Several operations can be done to/on processes. These include the following:

- Creation – A process is created when an executable set of instructions is incorporated into a program segment and PCB (process control block) is created to hold the process's relevant operational data.
- Process Spawning – Processes can spawn (create) other processes. A parent process can create child process. The child process can execute concurrently with the parent process, or the parent process can wait until the child process completes. This is often refereed to as "blocking". The child process can be a copy of the parent, or it can be a completely different entity. In UNIX based systems the fork() function call can be used to create a child process that is

identical to the parent. Details of the fork() function will be covered in later sections.

- Deletion/Termination – When a process has completed the OS recovers the memory used by the process/program segment and releases any resources that the process may have been using during execution.
    - Normal Termination – The process terminates when its last instruction has completed. A call to exit() requests that the OS deletes the process.
    - Parent can terminate child using abort call for the following reasons:
        - Child exceeded use of its resources.
        - Child's task is no longer needed.
        - Parent is exiting, many OS's do not allow child process to continue after the parent terminates.
- Interruption – When a running process is interrupted all data needed to restart the process (context data) is stored in the process's PCB so that process can resume functioning as if it had never stopped. Context data includes register values, file handles along with read/write pointers, etc.
- Suspension – A suspended process is similar to a process that has been interrupted. Usually a process is suspended so that an I/O event or a Wait event can be serviced. Suspended processes are put in "holding pattern" until the I/O or Wait event is completed, after which they can be restarted using their context data.
- Resumption – A process is resumed after it has been interrupted or suspended. The context data allows it to restart just as if it had never been stopped.

Once a request is given to the OS to run a program, the OS creates the process and manages its execution lifecycle. The process's execution lifecycle can be modeled using the traditional Five State Process Model. It is shown in figure 3.1 below. As the name suggests, there are 5 states in the model. The states, often implemented by using queues, are described below:

- New – The process is being created. The processes program segment and PCB is being created and initialized.
- Ready – Once created, the process is placed in a job pool (the Ready Queue) where it waits for the CPU.
- Running – Instructions are being executed.
- Waiting (Blocked) – The process is waiting for some event to occur (such as I/O completion or receiving a signal() to satisfy a wait() call).
- Terminated – The process has finished execution, its memory and resources are being recovered by the OS.

The Five State Process Model is very important for process management and scheduling, but it also serves as an invaluable aid in understanding other important concepts, such as how to manage multiple CPU's. It will be referenced in later sections when and where appropriate.

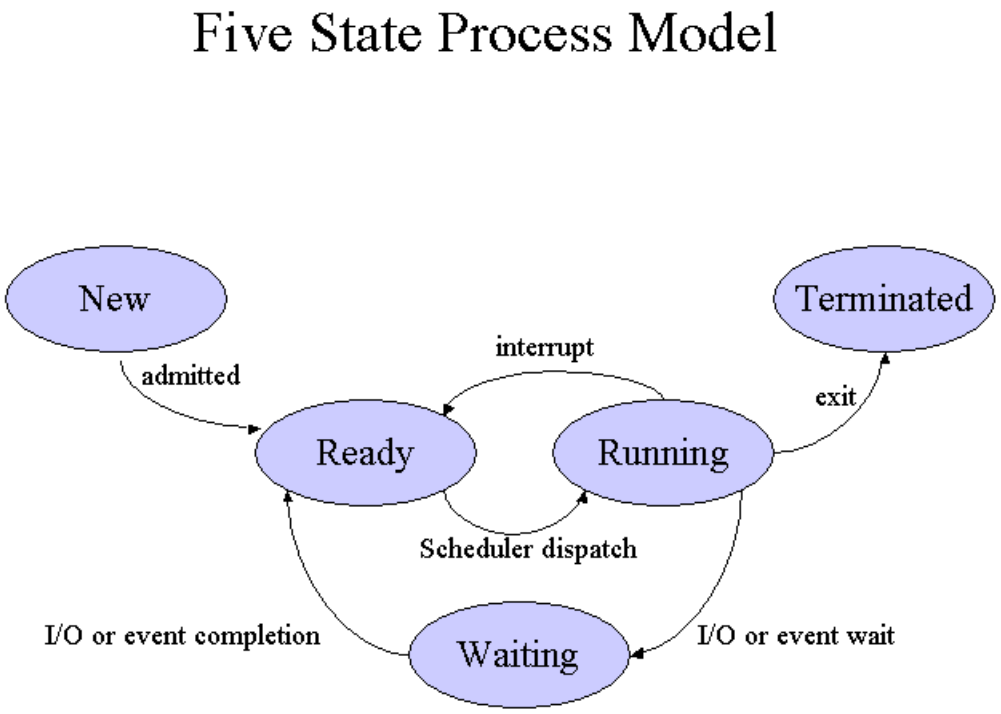


Figure 3.1 – This figure shows the 5-state process model. Processes are created in New, they wait to be scheduled in Ready, they receive CPU time in Running, they wait for I/O and wait events to complete in Waiting, and when their last instruction has been executed, their memory and resources are recovered in Terminated.

SSL programs used in the Multi-Programmed system from the previous chapter moved through several states during their execution lifecycle. These states were not called out specifically, but nevertheless, they were very similar to those of the Five State Process Model. Programs were compiled, assembled, and loaded into memory partitions, which formed a "job pool". Concurrently, their PCB's were created, recall section 2.31 for more information on the PCB data structure. These actions correspond to the work that occurs in the "New" state/queue. The job pool formed by the multiple partitions in memory, corresponds to the Ready state or Ready queue, and was tracked by the PCB array that kept tabs on each partition. In order to get to the running state, the round-robin like scheduling loop selected a process each iteration and allocated it "n" CPU slices. If the process encountered an I/O statement, its CPU time was cut short and it had to wait until it was scheduled by the round-robin scheduling system before it got another chance to run. This is akin to having to go to the Wait state, and then to the Ready state in the Five State Process Model. Finally, when a process executed its last instruction, its PCB status variable was set to complete, allowing the job pool system to create and load another process into the completed process's partition.

3.2 Process Scheduling and Scheduling Algorithms

Process scheduling is based on the CPU/IO burst cycle. In this scheme, execution alternates between periods of CPU use and waiting for I/O to complete. These phases are called CPU bursts and I/O bursts. A process starts with a CPU burst and then alternates

between I/O bursts and CPU bursts until completion. Usually a process ends with a CPU burst period, followed by a system call requesting the termination of the process. The amount of CPU bursts and I/O bursts vary greatly from process to process, depending on what the process is designed to do. Some examples are shown below.

- Database program – I/O intense (disk)
- Simulation – CPU intense (calculation of mathematical models)
- Picture Rendering – CPU intense
- Excel – I/O intense (mostly waits on the user to enter data)
- Games – both CPU intense (rendering of scenes), and I/O intense (display of scenes, and user inputs).

3.21 Schedulers, Short-Term and Long-Term

There are two schedulers; a short-term scheduler that deals with moment to moment scheduling and a long-term scheduler operates much less frequently. It is the job of the short-term scheduler to select jobs from the job pool (Ready Queue) to be placed in the Run Queue to be run by the CPU. The long-term scheduler puts regularly scheduled jobs, such as system backups, disk maintenance programs, etc., into the job pool. To think of this another way, the users, and long-term scheduler determine the contents of the job pool/Ready Queue, while the short-term scheduler selects from jobs in the job pool to put in the Rrun Queue so that the CPU can run them.

Switching a running job from one state (or queue) to another is called a context switch. Processes that are running are suspended, and processes that are in the ready queue waiting to be run are started or resumed. A context switch does involve work by the OS (storing of process states, process variable values, etc), so OS overhead goes up with more frequent context switching. Briefly, context switching involves:

- Saving the state and all relevant data of the running process in its PCB so that when the time comes for the process to restart, it can pick up where it left off.
- Resuming execution of a scheduled process. This means that register values, I/O handles, etc., are updated so that the suspended process can resume as if it had never stopped.

To recap, the function of the short and long term schedulers are:

Short-Term Scheduler – When the CPU becomes idle, a job from the Ready Queue is selected to be run. This is the job of the short-term scheduler. Characteristics include:

- Frequent operations, on the order of 10 times a second or more.
- Because it operates often, it must be fast.

Long Term Scheduler – It is the job of the long-term scheduler to place regularly scheduled jobs into the job pool so they can executed. In general, the Long-Term scheduler will attempt to balance the system between CPU and I/O bound processes to make greatest use of available resources. Some of its characteristics are:

- It operates less frequently.
    - Minutes to hours could pass between calls to the long-term scheduler.
- Because it is called infrequently, it can take time to decide which processes should be selected for execution.
- Some systems don't use Long Term schedulers, or they are very minimal. Time sharing systems are a good example. They just put every new process into the Ready Queue.

Another piece of software, the dispatcher, handles the mechanics of getting a scheduled job running. It is the dispatcher's job to give control of the CPU to the module selected by the Short-Term Scheduler. Dispatching involves:

- Switching Context.
- Switching to user mode. User programs do not have access to the full set of instructions available to the CPU. This is done to protect the system and the OS.
- Transferring control to the proper location in the user program (identified in the PCB by the value of PC (program counter)) and restarting the program.

A Pre-emptive Scheduler does not wait for an I/O request or an interrupt to move a job from the running queue to the ready queue. The round-robin scheduler used in the Multi-programmed Batch system from the previous chapter is an example of pre-emptive scheduling.

In a Time-Sharing System each process in the running queue gets an allotment of CPU cycles. If the process does not terminate or make an I/O request before its allotment completes, the scheduler moves to another running process. Time-Sharing Systems will be discussed in more depth in later sections.

3.22 Scheduling Criteria

Different scheduling algorithms try to optimize the following criteria to different degrees:

- CPU Utilization – Keep the CPU as busy as possible. In a real system a light load is about 40%, heavy 90%.
- Throughput – The number of process completed per time unit. Can range anywhere between more than 10 process per second to less than 1 per hour.
- Turnaround time – This is a measure of how long a process spends in the system. It is the time from when the process is submitted to completion.
- Waiting time – The scheduling algorithm does not effect the amount of time spent executing or doing I/O; it only affects the time spent in the ready queue. Waiting time is the sum of time periods waiting in the ready queue.
- Response time – The amount of time that from submission of the job to its first response to the user. This is often measured by recording the first time the job entered the run queue. Good response time is important in time sharing systems.

3.23 Exercises

1. How is a process different than a program?
2. Show a diagram of process, make sure and include the various parts, define what each part does.
3. Define Process Control Block (PCB).
    a. List and define all components of the PCB
    b. Which processes get PCB's
4. What type of operations can be done to a process?
5. What is the difference between threads, lightweight processes, and heavyweight processes?
6. Draw and label the 5 state process model.
7. What types of action/actions are required for a process to do each of the following:
    a. Process enters the new queue?
    b. Process enters the ready queue?
    c. Process enters the run queue?
    d. Process enters the wait/io que?
    e. Process moves from run to ready?
    f. Process moves from wait/io to ready?
    g. Process moves from run to terminate?
8. What is a long term scheduler?
9. What is a short term scheduler?
10. What are 3 algorithms are used in short term schedulers?
11. How can the OS influence the mix of jobs in the ready queue?
12. What sort of mix of jobs is most efficient from a CPU usage standpoint?

3.3 Scheduling Algorithms and Analysis Methods

The scheduling algorithms that will be discussed in detail are First Come, First serve, followed by Shortest Job First and Shortest Job First – Pre-emptive, and finally Round-Robin. Priority scheduling will be also be discussed.

When examining algorithms, a tabular approach will be used. The goal of the examination process, in the scope of this book, is to determine the average response time and the average turnaround time. The first part of the analysis is to make a timeline of the events. The timeline reflects the order that the involved processes go into the run queue, and eventually complete. Completion occurs when they have been in the run queue long enough to exhaust their allotment of CPU bursts.

The just created timeline will be used to fill out a table, and from the table the algorithms performance statistics will be calculated, i.e. average response time and average turnaround time. Remember, average response time reflects the average time a process has to wait until its initial entry into the run queue, while average turnaround time describes the average amount of time a process is in the system, defined by the time it arrived until all CPU bursts required to run it were exhausted. The paragraphs below detail the various scheduling algorithms.

3.31 First Come First Serve

Processes are served in the order they are received. This can be implemented with a FIFO queue. Characteristics of the algorithm include:

- Response time is highly dependent on order of jobs and can be long.
- Non-Preemptive
- CPU intense jobs can cause all I/O bound jobs to wait for them to complete. As I/O bound jobs wait for I/O they all move from the waiting queue to the ready queue and the CPU intense job dominates the CPU.

It is easiest to understand the analysis process by looking at an example. Suppose the table below, table 3.1, describes some processes, their arrival times, and the amount of CPU bursts they require to complete.

Process	***Arrival Time***	***CPU Bursts***
P0	0	8
P1	1	7
P2	2	4
P3	3	1

Table 3.1 shows a group of 4 processes, their arrival times, and the CPU bursts required to complete execution.

The first thing to be done is to make a timeline that reflects when the processes enter and exit the system. Using First-Come, First-Serve, the processes will execute in the order

they arrived; their execution order is P0, P1, P2, P3. This is reflected by the timeline below in figure 3.2.

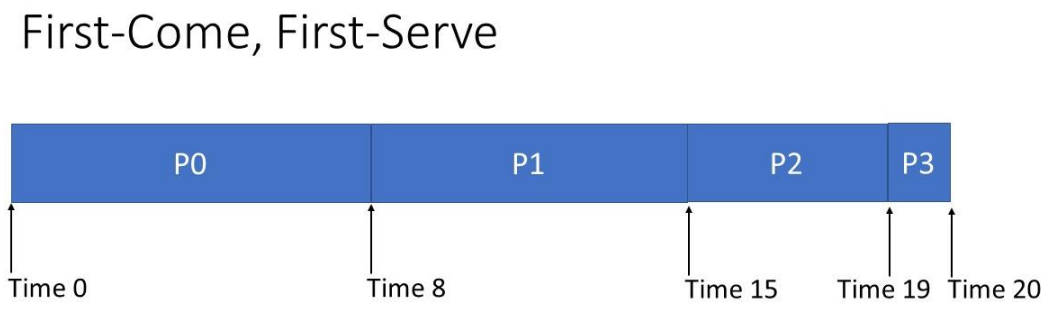


Figure 3.2 This figure shows a timeline of processes based on the first-come, first-serve scheduling algorithm. P0 arrived first, so it was run first. It completed at time 8, because it required 8 CPU burst to run. P1 arrived next, so it ran when P0 completed. It required 7 CPU bursts, so it completed at time 15. Following the same logic, P2 and P3 ran and completed at times 19 and 20.

Now based on figure 3.2 a table can be built that shows all data required to compute average response time and average turnaround time. See table 3.2 below.

Process	***Arrival Time***	***CPU Bursts***	***1st Time in Run Queue***	***Completion Time***
P0	0	8	0	8
P1	1	7	8	15
P2	2	4	15	19
P3	3	1	19	20

Table 3.2 reflects the start times and completion times as shown in figure 3.2. A table of this form provides an easy tool to calculate the average response time and average turnaround time of a scheduling algorithm for a group of processes.

Average response time is defined as the average amount of time that a group of processes needs to wait before their first entrance into the run queue. To calculate this statistic, the following formula is used:

$$\text{Average response time} = \frac{\sum(1^{st}\text{ time in run queue } p_i - \text{arrival time } p_i)}{\text{Number of processes}}$$

For the data in table 3.2 we get the following:

$$\text{Average response time} = \frac{(0-0)+(8-1)+(15-2)+(19-3)}{4}$$

$$\text{Average response time} = \frac{(0+7+13+16)}{4} = \frac{36}{4} = 9$$

Average turnaround time is calculated in a similar manner. To calculate this statistic the following formula is used:

$$\text{Average turnaround time} = \frac{\sum(\text{completion time of } p_i - \text{arrival time } p_i)}{\text{Number of processes}}$$

Using the data in table 3.2 we get the following:

$$\text{Average turnaround time} = \frac{(8 - 0) + (15 - 1) + (19 - 2) + (20 - 3)}{4}$$

$$\text{Average turnaround time} = \frac{(8 + 14 + 17 + 17)}{4} = \frac{56}{4} = 14$$

3.32 Shortest Job First and Shortest Job First Preemptive

In the shortest job first algorithm, the job in the ready queue with the shortest required CPU burst is selected. If two jobs have equal next CPU burst, FCFS breaks the tie. The algorithm is provably optimal, i.e. it gives the minimum waiting time for a set of processes. Once again, taking an example is very illustrative. Using the same data as used for first-come, first-serve, the timeline in figure 3.3 below is generated.

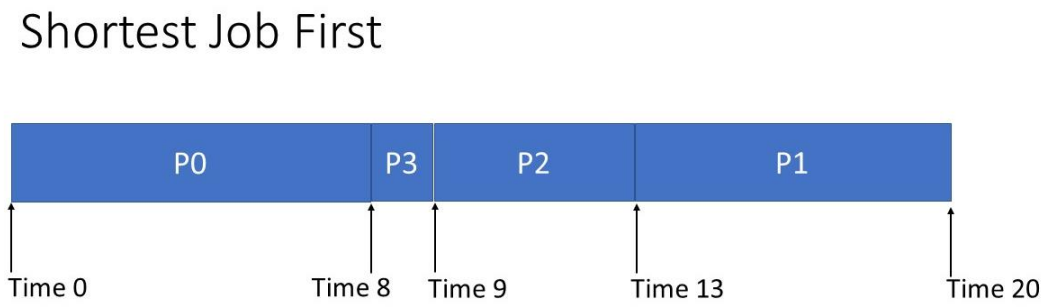


Figure 3.3 shows the timeline using the shortest job first algorithm. Note that P0 goes first because at time 0 it is the only process that has arrived. While P0 is running, all the other processes arrive.

Table 3.3 below is generated using the data from figure 3.3.

Process	*Arrival Time*	*CPU Bursts*	*1st Time in Run Queue*	*Completion Time*
P0	0	8	0	8
P1	1	7	13	20
P2	2	4	9	13
P3	3	1	8	9

Table 3.3 reflects the start times and completion times as shown in figure 3.3. Notice that the end time for all the processes is the same as that of the algorithm first-come, first serve; only the order in which they run has been changed. That being said, changing their order will change the performance statistics.

Once again, the performance statistics are calculated:

$$\text{Average response time} = \frac{(0 - 0) + (13 - 1) + (9 - 2) + (8 - 3)}{4}$$

$$\text{Average response time} = \frac{(0 + 12 + 7 + 5)}{4} = \frac{24}{4} = 6$$

And the average turnaround time:

$$\text{Average turnaround time} = \frac{(8 - 0) + (20 - 1) + (13 - 2) + (9 - 3)}{4}$$

$$\text{Average turnaround time} = \frac{(8 + 19 + 11 + 7)}{4} = \frac{45}{4} = 11\ ¼$$

Comparing the two algorithms, it can be seen that shortest job first outperformed first come first serve in both average response time and average turnaround time. From these results, it would appear that every good OS should use shortest job first or some variant of it, if the choice were between it and first-come, first-serve. However, shortest job first has one major drawback – How does the system know how long some job in job pool is going to take? How can this be known ahead of time? In practice, it is very hard to know this answer. Some possibilities may be to keep records of processes and their run times, or ask the compiler to determine a process's run time complexity. These data points could be included in a process's PCB, allowing a scheduling algorithm to make selections based on these criteria.

Shortest Job First – Preemptive is very similar to shortest job first, except that a running process can be preempted by a process in the job pool that has a lower CPU burst requirement. Looking at the data from the previous example, we see that at time 0, P0 is the only job present so it is selected to run, however during the next 3 time units, 3 processes join the job pool and each of them has a lower CPU burst requirement than P0. One thing of note, a running process should only be preempted by a process with a lower CPU burst requirement; if the burst requirements are equal, the running process takes precedent. Once again, we present an example using the previously used data. Figure 3.4 shows the timeline for the data.

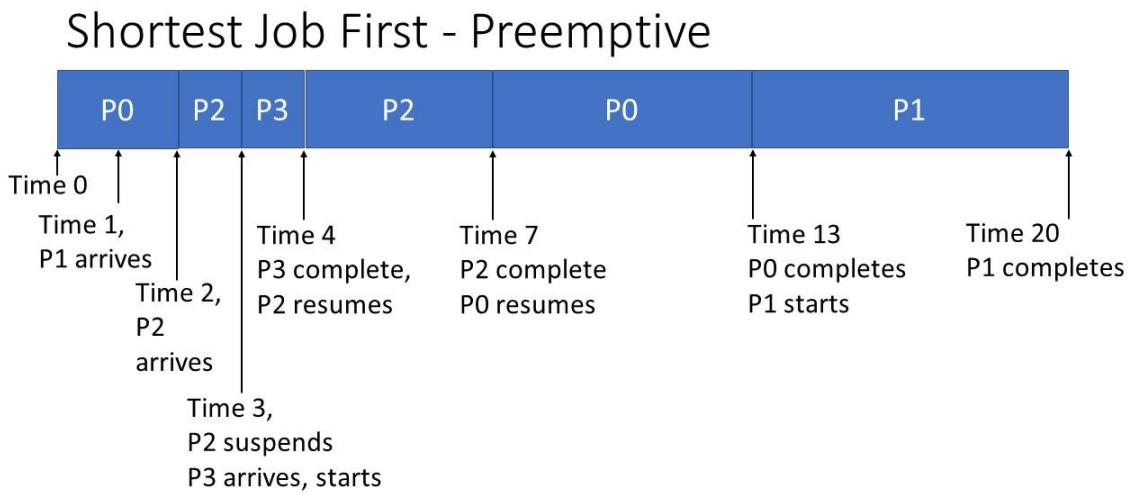


Figure 3.4 depicts a timeline for the shortest job first preemptive algorithm. P0 has a run time of 8 CPU bursts, P1 arrives a time 1 but does not preempt P0 because it requires 7 CPU bursts and at time 1 P0 is resident and also requires 7, so because it is resident, it remains. At time 2, P2 arrives and only requires 4 bursts; at that moment P0 requires 6 more, so P2 preempts P0. P2 runs for 1 burst and P3 arrives. P3 requires only 1 burst, P2 requires 3 more bursts, so P3 preempts, runs, and completes by time 4. At time 4 P0 needs 6 bursts, P1 needs 7, and P2 needs 3, so P2 is selected. When P2 completes, P0 is selected because it requires less CPU bursts than P1, and finally P1 is selected.

Table 3.4 is generated from figure 3.4

Process	***Arrival Time***	***CPU Bursts***	***1st Time in Run Queue***	***Completion Time***
P0	0	~~8, 6~~	0	13
P1	1	~~7~~	13	20
P2	2	~~4, 3~~	2	7
P3	3	~~1~~	3	4

Table 3.4 shows the relevant data for the shortest job first preemptive algorithm. Note that in the CPU Bursts column, there are more than 1 number for P0, and P2, and that there "strike throughs". These indicate that started, was suspended, and resumed. For instance, P0 started with 8 CPU bursts, was suspended after 2 CPU bursts, leaving 6 CPU bursts. When it resumed those 6 were used and the job was complete. The strike throughs are a way to keep track of the bursts when suspending and restarting the jobs.

The statistics are calculated below:

$$\text{Average response time} = \frac{(0 - 0) + (13 - 1) + (2 - 2) + (3 - 3)}{4}$$

$$\text{Average response time} = \frac{(0 + 12 + 0 + 0)}{4} = \frac{12}{4} = 3$$

And the average turnaround time:

$$\text{Average turnaround time} = \frac{(13 - 0) + (20 - 1) + (7 - 2) + (4 - 3)}{4}$$

$$\text{Average turnaround time} = \frac{(13 + 19 + 5 + 1)}{4} = \frac{38}{4} = 9\ ½$$

Comparing the numbers, it appears that the preemptive algorithm is the most efficient. But practical considerations do come into play. There is a cost for a context switch that is not reflected in these diagrams, tables, and calculations. It is true that the cost was not accounted for in the previous algorithms either, but because there was an equal number of context switches, it is a non-factor. In the preemptive algorithm there are more switches, so it must be noted. If there are excessive context switches, this adds to the OS's overhead which can mitigate any performance "gains".

Another problem with the shortest job first algorithms is that they can starve processes that have greater CPU requirements. That is, if a single high CPU burst job is in the pool and many low CPU requirement jobs are continually introduced, the high CPU burst job may never get a chance to run.

3.33 Round Robin

Round Robin is a FCFS algorithm in which each process in the ready queue is preempted after a short time. This short time is referred to as a "time quantum". The bullets below describe the relevant characteristics of the round robin algorithm.

- The ready queue is treated as a circular queue.
- The scheduler goes around the ready queue allocating the CPU each process for a maximum of 1 time quantum.
- Average waiting time can be long.
- Used in time sharing systems where response time is important (response time needs to be fast so that user/computer interaction is good). In a time sharing system each user has a terminal and if response time is slow, users can be caught wondering if the system is crashing. So, in general, the system must poll each terminal several times per second; a frame rate of about 24 times per second would be ideal.

Time quantum duration is guided by optimizing response time while minimizing the number of context switches (overhead). A general rule of thumb is that a time quantum of a length such that half the jobs finish within one time quantum is ideal. Using this rule, a time quantum of 4 would be the ideal length for the example data. Figure 3.5 below shows a timeline for the example data with a time quantum equal to 4 CPU bursts.

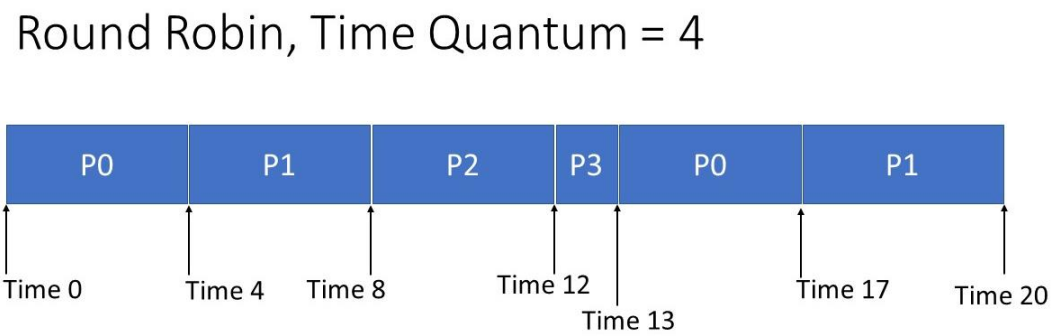


Figure 3.5 shows a timeline using the round robin algorithm. The time quantum was selected to be 4, it is 4 so that half the jobs could finish in 1 time quantum. P1 needed 4 CPU bursts, and P3 needed 1 burst, so these two jobs finished in time quantum.

Table 3.5 is generated in conjunction with figure 3.5.

Process	***Arrival Time***	***CPU Bursts***	***1st Time in Run Queue***	***Completion Time***
P0	0	~~8, 4, 0~~	0	17
P1	1	~~7, 3, 0~~	4	20
P2	2	~~4, 0~~	8	12
P3	3	~~1, 0~~	12	13

Table 3.5 shows the relevant data for the round robin algorithm whose time quantum is set to 4. Notice in the CPU Bursts column, the strike throughs record the time bursts as they subtracted 1 time quantum at a time. Also, notice that when a job needs less bursts than the time quantum, it only uses what it needs.

The statistics are calculated below:

$$\text{Average response time} = \frac{(0 - 0) + (4 - 1) + (8 - 2) + (12 - 3)}{4}$$

$$\text{Average response time} = \frac{(0 + 3 + 6 + 9)}{4} = \frac{18}{4} = 4\frac{1}{2}$$

And the average turnaround time:

$$\text{Average turnaround time} = \frac{(17 - 0) + (20 - 1) + (12 - 2) + (13 - 3)}{4}$$

$$\text{Average turnaround time} = \frac{(17 + 19 + 10 + 11)}{4} = \frac{57}{4} = 14\frac{1}{4}$$

The round robin algorithm sacrifices turnaround time performance for response time, while ensuring that no process is starved. If the time quantum is very short it increases response time performance, but adds to system overhead because of all of the context switches. If the time quantum is large, so large that it exceeds all jobs required CPU bursts, it behaves identically to first-come, first-serve.

3.34 Priority Scheduling

Instead of looking at the order in which jobs arrive and/or their estimated CPU bursts, jobs run in order of importance (priority). The bullets below provide more detail.

- Jobs of equal priority are scheduled FCFS
- SJF is a special case of priority scheduling, in which the priority is the inverse of the estimated CPU burst.
- Priority scheduling can be preemptive or non-preemptive.
- Priority Scheduling suffers from indefinite blocking or starvation. A low priority job can be caused to wait indefinitely by higher priority jobs.
    - From Silberschatz [3], "Rumor has it that , when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and not yet been run."
    - "Aging" is a process in which the priority of processes that have been waiting for a long time is increased, meaning that the longer the process is in the ready queue, the higher its run priority becomes.

3.35 Exercises

1. Define the following mathematically:
    a. Response time
    b. Turnaround time
2. Describe the following scheduling algorithms:
    a. First Come First Serve
    b. Shortest Job First
    c. Round Robin
3. What scheduling algorithm/algorithms does a batch system use? Why?
4. What scheduling algorithm/algorithms does a multi-programmed batch system use? Why?
5. What scheduling algorithm/algorithms does a time sharing system use? Why?
6. Considering the shortest job first (pre-emptive) algorithm.
    a. What are its strengths, as related to response time and turnaround time?
    b. What are its weaknesses, if any?
7. What is a CPU burst?
    a. How do we know how many bursts a program will use?
    b. Devise at least 2 strategies for predicting CPU bursts for frequently used processes.
8. What is a time quantum?
    a. What happens when the time quantum is very short?
    b. What happens when the time quantum is very long?
9. Using the table below, find the average response time and the average turnaround time for each of the following algorithms.
    a. First-Come First Serve
    b. Shortest Job First
    c. Shortest Job First Preemptive
    d. Round Robin, time quantum = 2

Process	***Arrival Time***	***CPU Bursts***
P0	0	7
P1	2	3
P2	4	12
P3	6	2
P4	7	5

10. Is a time quantum of 2 ideal for the processes in exercise 9? Select a more ideal time quantum and recalculate the statistics for the round robin algorithm.
11. Context switching has a cost to the system in CPU bursts. Assume that a context switch requires 1 CPU burst each time there is a context switch. Rework problem 9, this time include the 1 CPU burst for each context switch.

12. Program a simulation of a job scheduler for an operating system. Your scheduler will read in a list of jobs with the relevant information and output the order of completion along with the time of completion for each job and other relevant statistics.

Implement 3 scheduling that were discussed in class.

a. First Come, First Served (batch, non preemptive)
b. Shortest Job First (batch, non preemptive)
c. Round Robin (preemptive)

Use C or C++ on a Linux machine.
Below is an example of an input file. This is the input that generated the example output.

ProcessID	Arrival	cpuBurst	Priority
100	0	10	1
101	6	10	1
102	8	4	1
103	12	20	1
104	19	15	1
105	30	5	1
106	35	10	1

Below is an example of the way the output should look. In this example each algorithm is run with the same data file, then the output and statistics are presented. This is a good example to use to test your algorithms and program. For the Round Robin, the time Quanta was 15.

ProcessID	Arrival	cpuBurst	Priority
100	0	10	1
101	6	10	1
102	8	4	1
103	12	20	1
104	19	15	1
105	30	5	1
106	35	10	1

number of jobs in newQ = 7
Terminated Jobs (First Come, First Served).

processId	arrival	completion
100	0	10
101	6	20
102	8	24
103	12	44
104	19	59
105	30	64
106	35	74

Run Stats
Throughput = 0.09
Average turnaround time = 26.43
Average response time = 15.86

ProcessID	Arrival	cpuBurst	Priority
100	0	10	1
101	6	10	1
102	8	4	1
103	12	20	1
104	19	15	1
105	30	5	1
106	35	10	1

number of jobs in newQ = 7
Terminated Jobs. (Shortest Job First)

processId	arrival	completion
100	0	10
102	8	14
101	6	24
104	19	39
105	30	44
106	35	54
103	12	74

Run Stats
Throughput = 0.09
Average turnaround time = 21.29
Average response time = 10.71

ProcessID	Arrival	cpuBurst	Priority
100	0	10	1
101	6	10	1
102	8	4	1
103	12	20	1

104	19	15	1
105	30	5	1
106	35	10	1

number of jobs in newQ = 7
Terminated Jobs. (Round Robin)

processId	arrival	completion
100	0	10
101	6	20
102	8	24
104	19	54
103	12	59
105	30	64
106	35	74

Run Stats
Throughput = 0.09
Average turnaround time = 27.86
Average response time = 11.1

Below is an example of a routine used to output the run statistics. For each of the states in the 5 state diagram it uses an array of type PCB. The PCB holds relevant process data like pid, cpu bursts, arrival time, completion time, etc.

```
void showRunStats(int numJobs, int time)
{
      int j;
      float tPut, turn, resp;

      /* Initialize variables. */
      turn = 0.0f;
      resp = 0.0f;
      /* Calculate turnaround time and waiting time.  In this
      calculation the waiting time is really should be called response
      time. */
      for(j=0;j<numJobs;j++) {
            turn = turn + (terminated[j].completion -
                           terminated[j].arrival);
            resp = resp + (terminated[j].start -
                           terminated[j].arrival);
      }

      /* Calculate averge values. */
      turn = turn/(float)numJobs;
      resp = resp /(float)numJobs;

      /* Show stats to user. */
      printf("\n");
      printf("Run Stats \n");

      /* Throughput. */
      tPut = (float)numJobs/(float)time;
      printf("Throughput = %.2f \n", tPut);
      /* Turnaround time. */
      printf("Average turnaround time = %.2f \n", turn);
      /* Waiting time. */
      printf("Average response time = %.2f \n", resp);

}
```

Below is an example of a routine that reads the job pool data from the file:

```
int loadJobs(char *filename)
{
      FILE *jobs;
      char string[80];
      int pId, arrival, cpuBurst, priority;
      int j;
      int nJobs;

      /* Open file of jobs to be put in the ready que. */
      jobs = fopen(filename, "r");

      /* Load the ready que from the file. */
      fgets(string, 80, jobs);
      printf("%s \n", string);
      j= 0;
      while(fscanf(jobs, "%d %d %d %d", &pId, &arrival, &cpuBurst,
            &priority) != EOF) {
            newQ[j].pId = pId;
            newQ[j].arrival = arrival;
            newQ[j].cpuBurst = cpuBurst;
            newQ[j].priority = priority;
            newQ[j].status = NEW;
            printf("%d        %d    %d          %d \n",
                  newQ[j].pId,newQ[j].arrival,  newQ[j].cpuBurst,
                  newQ[j].priority);
            j = j+1;
      }
      nJobs = j;
      printf("\n");
      printf("number of jobs in newQ = %d \n", nJobs);
      fclose(jobs);

      return nJobs;
}
```

Some helpful tips.

d. Think about the 5 state model.
e. Work through these examples by hand so that you understand why the output is in the order that it is in and the numbers are the way they are.
f. Design your program before you sit down at the computer.

13. Modify the Multiple Fixed Partition Multi-programmed from chapter 2, exercise 2.5 c, so that it can use the following scheduling algorithms:
    a. First Come, First Serve. The order of arrival can be approximated by either the time stamp on the program before it is compiled, some other metric such as alphabetical order.
    b. Shortest Job First. For this You need to develop a metric that will approximate CPU bursts for the job. You can use the following:
        i. Size of the job in bytes
        ii. Track the run time of the job and "tag" the job with that runtime.
        iii. You can analyze each program and based on the its time order complexity (n, n^2, etc.) and the size of n, a metric can be made.
    c. Round Robin. The system already uses this algorithm. Currently the "time quantum" is set to 10. Modify the program so that the time quantum is selectable.
    d. Make sure that average response time and average turn around time can be computed and displayed for each of the algorithms.

True Story – The Bucket List

This story is offered by the author's twin brother; it is told in first person. One of the participants in this saga is Herb; he was in a previous story involving electricity.

We were going on a 3 day fishing trip in the Gulf of Mexico. The guys and I were to meet at a dock off of a bayou near Lake Pontchartrain. Everyone was excited to go, my friend Dave, my old friend Herb, Herb's dad – Billy, and myself. The boat we were taking was a nice shrimp boat – a 36 foot Lafitte skiff, rigged very nicely for pulling wing nets and trawls. It was Herb's personal boat, well maintained, comfortable, and seaworthy. What could be better? A better question, looking back at it, was – "What could go wrong?".

Anyway, we were all at the dock, loading the boat. Herb, Dave, and I were doing most of the heavy work; Billy was in his mid 70's at the time so he was "supervising". We loaded the usual supplies, bait, fishing gear, food, ice, and of course cold drinks. We left the dock at 6pm on the nose with the intention of travelling all night through the intercoastal waterway and arriving at the Gulf to catch the morning bite. The boat, "Cactchalot" was efficient and pretty fast for its size; it would cruise easily at 22 mph.

Herb was/is an experienced mariner, and at the time, Dave, myself, and Billy were not, especially when compared to Herb. Herb got us to the beginning of the intercoastal, just outside of the Rigolets pass, and pointed us down the long straight and scenic waterway. Herb had been up late the night before, so he left the driving to me. "Just follow those channel markers and we will be good" he said, and he was off to sleep. Billy was resting on the back of the boat, so Dave and I talked while I drove. There was plenty of light still, so it was easy going. Dave was relaxing, drinking cold drinks, that is, ice cold beers, because the summers in Louisiana can be hot, and we were out having fun.

The more he drank the more relaxed he got, and he occasionally offered to drive. It was nearing dusk and I needed to visit the back of the boat for a quick relief, so I asked Dave to drive. He obliged, and said "Where to?". I pointed to the channel marker out front about 500 yards away and said "Go to that marker, but don't hit it!" We laughed and I went to the back of the boat. I was standing on the fantail of the boat, happy to be relieved, when a big ole reed almost hit me. Yeow! That would have hurt. I was stunned, where the heck did that come from. I zipped up and ran to the front of the boat and said "What's going on Dave?". He replied, "I think we got our channel markers confused." He was right; somehow, when I pointed out the marker to head towards, he and I were not on the same page and we were now no longer in the intercoastal; we were in some little cut or bayou off to the side. Back then, GPS and chart plotters were not in common use, so we were not quite sure where we were.

Right about then, Herb woke up. "What's up fellows?" he asked. "Nothing, just a bit off course" came the reply. By now it was dusk, but there was still a bit of visibility; we started to idle our way back to the intercoastal. Wherever we were, there were dozens, if not hundreds of crab traps, easily identifiable by their floats. I was up in the rigging spotting the traps so that Herb did not wrap one up in the propeller. Dave was down with Herb helping spot the traps. I saw a trap coming up on the starboard bow of the boat and said "Crab trap coming up, on the right!" Herb said "What?", and there was a change in the engine noise and a crushing crunch. We had wrapped the trap up into the prop. Herb reversed the big diesel and tried to unwrap it, no dice, just a horrible scraping crunching noise. So over into the water we went to get the trap out.

The water was only about 3 to 4 feet deep with the usual muddy bottom, so it was not so bad. We figured we could go under the boat with plyers and wire cutters and remove the trap. Really, that was our only choice. Herb and I were in the water discussing our options, and Dave and Billy were in the boat, when the water around our feet and legs started to feel like it was moving, not fast, but getting faster. I looked at Herb and asked – "You feel that?" "Uh-huh, I do, what is it?" came the answer. Right after that Herb yelled "Everyone in the boat! Now!" Dave, a very stout guy, grabbed us to help, really he yanked us back up into the boat, and yelled with a cross eyed wild look on his face "What the hell is going on?" The boat was rocking wildly, I looked out to the water where we were just standing and all I saw was mud. Herb pointed up into the sky right next to us – "That's what's going on!" We looked up, expecting to see some kind of alien spaceship coming to get us, but instead it was a ship, a big ship, towering about 8 stories over our boat. It was a big cargo ship going up the Mississippi Gulf Outlet, and as it moved through the shallow intercoastal its huge props sucked up the water around it.

Because of that little incident, we had to wait until the water settled down before we could go to work on getting the trap out of our propeller. That took at least an hour, and it was pitch black in the bayou when we jumped back over into the water. Dave and I went over, and started working on the trap. One of us would tie a rope to our ankle, get a big breath, and swim up to the prop, only about 6 feet or so, and start randomly cutting and yanking on the trap. It was slow going, because we could only hold our breath for so

long, and it was pitch black under the boat. There was no way of knowing what you were doing, at least not in the short time a person could hold their breath.

This went on for about an hour, and then Herb came over the side with a 5 gallon bucket. I said "What have you got there Herby?". He replied – "I got me a diving bell.". Herb took the bucket and turned it upside down and pushed it into the water; it took some effort. He slid the bucket up under the bottom of the boat and pushed it as far forward as he could get it. Next, we tied a rope around his ankle, and he said, somewhat like Arnold, "I'll be back." It seemed like he was under there forever, at least 5 minutes, maybe more, but he kicked his leg and we pulled him back up to the surface. He was red in the face, and somewhat cross eyed. He said, in a raspy tone – "Its working!".

Now it was our turn. Dave and I took turns. We would pull the bucket back to the surface, get a new batch of air, and push it back under. Diving under the boat and coming up with your head in a bucket is an eery experience, but it worked. You could concentrate on cutting the trap out of the prop. You could not see a thing, it was dark, you could hear the water rippling around the bottom of the bucket. With the bucket we could be down there long enough to get a feel for where the trap was in relation to the prop and you could use your hands to "map" out the situation. Herb would get the bucket just where he wanted it, come back up for a while, smoke a cigarette, and then go back down without replenishing the air in the bucket – a move of shear lunacy, but it worked.

In all honesty, I can say that this was an awesome thing to have done, but not something that I would want to repeat too often. Since then, we have tried going underwater with a tube in our mouths and it only works to a depth of a couple of feet, the bucket works much better. Why? It is physics, the water keeps the air in the bucket at about the same pressure as the water around it. When you use the tube, you are sucking in air that is at a lower pressure than the water, and it cannot be done once the pressure differential is too high.

It was not long after we started using the bucket that we had the trap free and were on our way to the Gulf. Did we catch a lot of fish? Not really, but after that wild experience in the marsh that night, we had a great time, whether the fish cooperated or not.

This story is dedicated to the memory of Billy Griffin.

Chapter 4 – More on Processes, Including Instruction Interleaving, Critical Sections and Synchronization

In this chapter more aspects of process will be discussed. Part of the focus will be on process interleaving, critical sections and synchronization. These topics will lead to some process management topics such as deadlocks, deadlock prevention and avoidance. However, initially, we will focus on some implementation issues that will help us down the road. These topics include process spawning, and shared memory. They are relevant because many of our examples will deal with multiple processes sharing a data buffer, and it is very useful to not only understand the concepts, but also to be able to implement them.

4.1 Child Processes Using the fork() Function

A simple way to create another process in Unix/Linux system is through the use of the fork() function. A call to the fork() function spawns a process that is identical to the calling process. The spawned process is a referred to as a heavyweight process because it while shares code with the parent process, everything else (data spaces, etc.) is its own. The fork() function returns a 0 to the chile process, and the process id of the parent process to the parent process. The C code below illustrates the use of the fork() function.

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int pid;

  pid = fork();
  if (pid == -1) {
    printf("The fork() call failed. \n");
  }
  else if (pid == 0) {
    printf("I am your child; my pid is %d \n", pid);
  }
  else {
    printf("I am your Daddy and my pid is %d \n", pid);
  }

  /* Sleep for 1 second so that the program does not end before
     the output from both threads can be put to the screen. */
  sleep(1);

  return 0;
}
```

The output from the above code (run using the online compiler repl) is shown below:

```
 clang-7 -pthread -lm -o main main.c
 ./main
I am your Daddy and my pid is 1189
I am your child; my pid is 0
```

The #include statements at the top of the code are used to tell the compiler that we are using library functions for input and output (#include <stdio.h>) and the sleep function (#include <unistd>).

A more interesting example is one that will show the two threads running concurrently. To do so the program is modified so that they both output a stream of messages to the screen. Since the programs will be running concurrently the messages will become interleaved. The code is shown below:

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
  int pid;
  int j;

  pid = fork();
  if (pid == -1) {
    printf("The fork() call failed. \n");
  }
  else if (pid == 0) {
    printf("I am your child; my pid is %d \n", pid);

    /* Send a slow stream of dots to the screen. */
    for(j=0;j<10;j++) {
      printf(".");
      fflush(stdout);
      sleep(1);
    }
  }
  else {
    printf("I am your Daddy and my pid is %d \n", pid);

    /* Send a slow stream of asterix to the screen. */
    for(j=0;j<10;j++) {
      printf("*");
      fflush(stdout);
      sleep(1);
    }
```

```
  }

  printf("\n");

  /* Sleep for 1 second so that the program does not end before
     the output from both threads can be put to the screen. */
  sleep(1);

  return 0;
}
```

The output is shown below:

```
clang-7 -pthread -lm -o main main.c
 ./main
I am your Daddy and my pid is 1670
*I am your child; my pid is 0
.*.*.*.*.*.*.*.*.*.*.
```

One thing to note is that the fflush() function is used to push output to the screen immediately. Without, on some systems, the dots and asterix may not intermingle. Later, we will use the fork() function to create two processes, one that loads a buffer and one that reads a buffer.

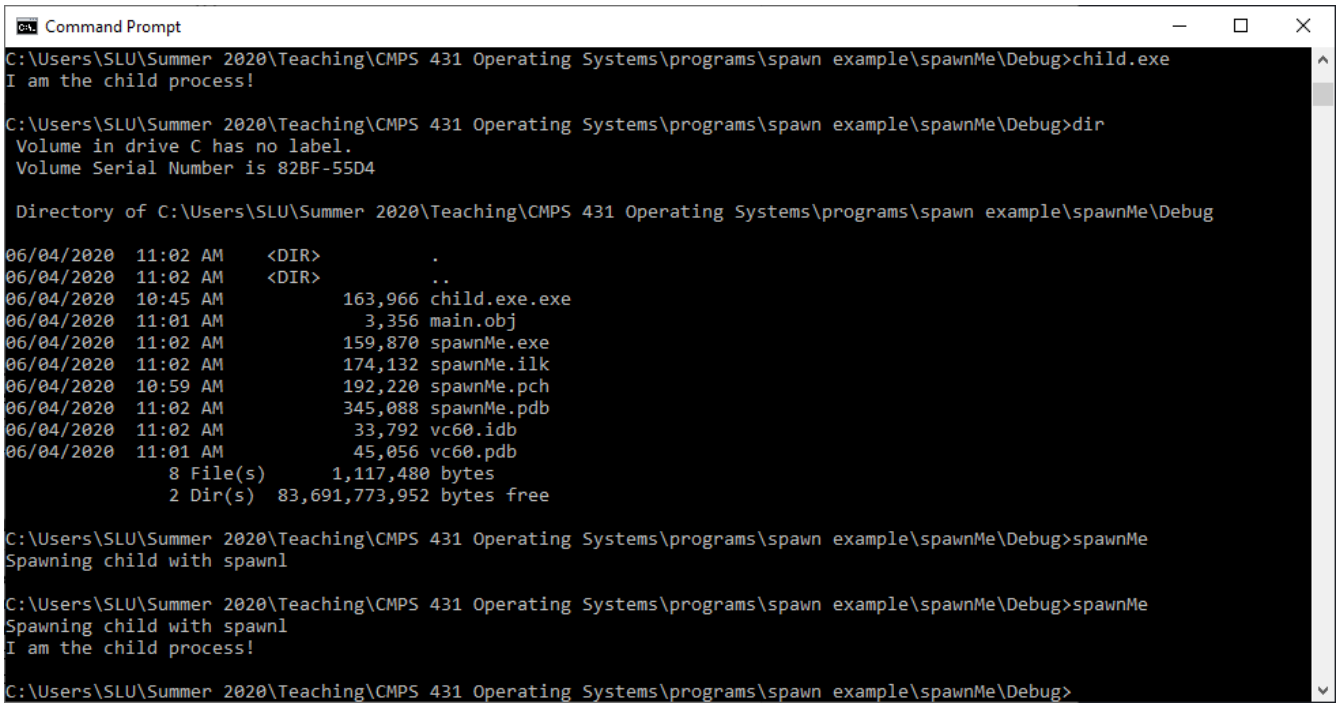
4.2 Process Spawning in Windows

The example below illustrates the use of the spawn() function in Visual C/C++.

```
#include <process.h>
#include <stdio.h>

void main(void)
{
  puts("Spawning child with spawnl");

  spawnl( P_WAIT, "child.exe",
    "child.exe", "Using spawnl", "Arg1", "Arg2", NULL );
}


#ifdef child

void main(void)
{
  printf("I am the child process! \n");
}
```

```
#endif
```

The #define can be used to create the child process during compiling. That is to create the child process, insert the line "#define child" under the last #include statement, and comment out the main program. For the program to work the child process executable must be named "child.exe". The output is shown below:

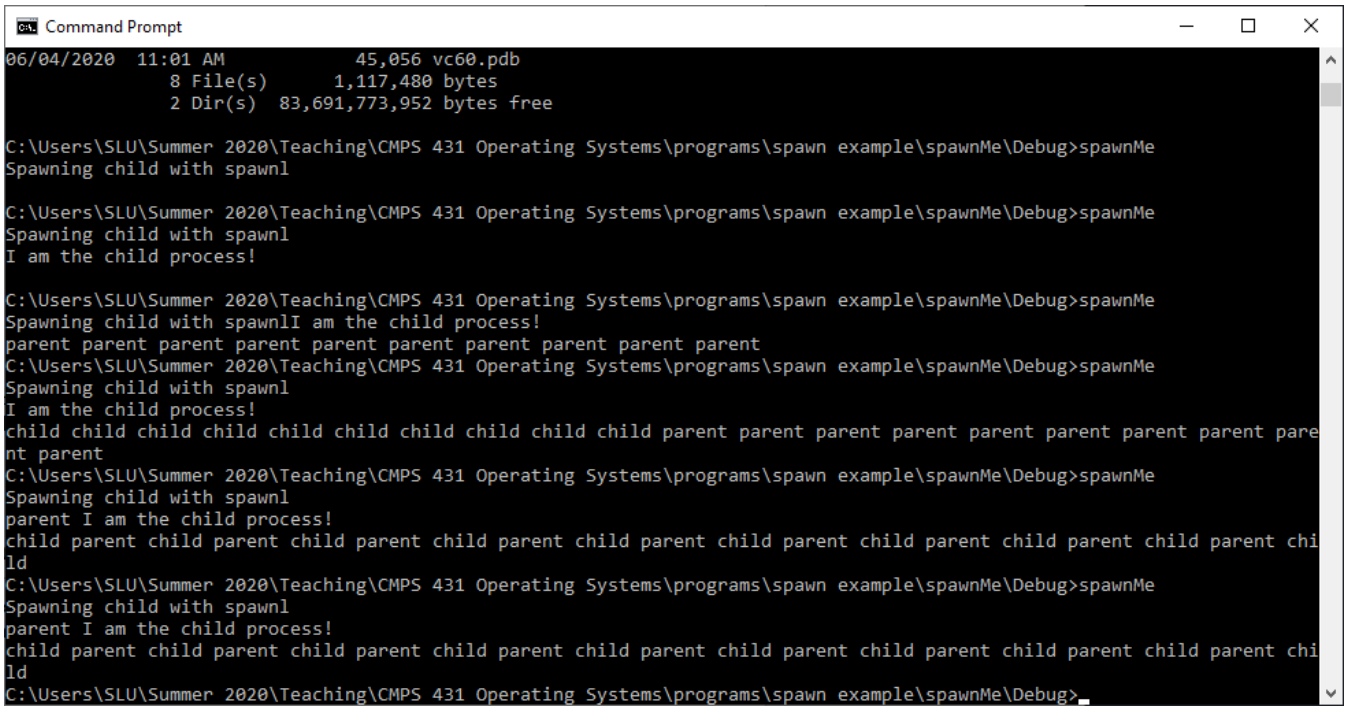


A more interesting example is shown below. In this example, the parent and child run at the same time, printing to the screen. Notice the careful use of the #define statements for building the child and parent processes.

```
#include <process.h>
#include <stdio.h>
#include <windows.h>

//#define child

//#if 0

void main(void)
{
  int j;

  printf("Spawning child with spawnl \n");
```

```
  /* P_WAIT will cause the parent to wait for the child to
     execute, P_NOWAIT will allow the parent and child to
   run simultaneously. */

  spawnl( P_NOWAIT, "child.exe",
    "child.exe", "Using spawnl", "Arg1", "Arg2", NULL );

  for(j=0;j<10;j++) {
    printf("parent ");
    /* Sleep for a half a second. */
    Sleep(500);
  }

}
//#endif


#ifdef child

void main(void)
{
  int j;

  printf("I am the child process! \n");

  for(j=0;j<10;j++) {
    printf("child ");
    /* Sleep for a half a second. */
    Sleep(500);
  }
}

#endif
```

The output is shown in the screen shot below. While it is a bit busy, looking closely reveals runs with the P_WAIT and P_NOWAIT.



4.3 Shared Memory in Unix/Linux

Shared memory is a very useful tool in the world of computing and processes that work and communicate together. The producer/consumer problem is classic example in computing literature and it relies on a buffer shared by a process that produces and item and a process that consumes that item. To implement this example we will use the shared memory functions declared in the library shm.h. The code below creates a very basic set of reader/writer threads using a fork() call and using shared memory.

The fork() function creates two processes, similar to the code in the previous section. Each of those processes, the parent and the child, attempt to create a piece of shared memory. Whichever process creates it first does not matter, if the shared memory has already been created, the process attaches to it. Once the shared memory is created, the child function assumes the role as the writer function and loads the buffer with the alphabet. It then uses the 100th element of the buffer to signal the reader function (played by the parent process). While the child was playing the writer function, the parent function sat in a loop examining the 100th element of the buffer. Once that element has a 1 in it (the signal from the writer function set it to 1 after it loaded the buffer) the reader function reads and displays the contents of the buffer. In later sections the signaling system will be improved, but for now a dedicated element of the memory will suffice.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```c
#include <sys/ipc.h>
#include <sys/shm.h>

#include <sys/sem.h>

#define SHARE_SIZE 128

int main()
{
  int j;
  int pid;
  char curr;

  /* Shared memory variables. */
  int shmid;
  key_t memKey, semKey;
  char *buffer;

    /* Send message to the user so they know
       this text program is running. */
    printf("Hello TV Land! \n");

    pid = fork();
    if (pid == -1) {
        printf("Error: fork() failed. \n");
    }
    else if (pid == 0) {
        printf("Child process! pid = %d \n", pid);
        fflush(stdout);

        /* Create a piece of shared memory. */
        if ((shmid = shmget(memKey, SHARE_SIZE, IPC_CREAT|0666)) == -1) {
            printf("Child: Error allocating shared memory \n");
            exit(shmid);
        }

        /* Attach to the memory. */
        if ((buffer = shmat(shmid, NULL, 0)) == (char *)-1) {
            printf("Child: Could not attach to shared memory.\n");
            exit(-1);
        }

        printf("Child process is writer() process \n");

        /* Initialize the buffer. */
```

```
    for(j=0;j<SHARE_SIZE;j++) {
      buffer[j] = 0;
    }

    /* Load the buffer. */
    curr = 'a';
    for(j=0;j<26;j++) {
      buffer[j] = curr;
      curr++;
    }

    /* Hang out for 5 seconds. */
    sleep(5);

    /* Send a signal to reader() to let it know buffer is ready. */
    buffer[100] = 1;

    sleep(1);
}
else {
    printf("Parent process! pid = %d \n", pid);
    fflush(stdout);

    /* Create a piece of shared memory. */
    if ((shmid = shmget(memKey, SHARE_SIZE, IPC_CREAT|0666)) == -1) {
        printf("Parent: Error allocating shared memory \n");
        exit(shmid);
    }

    /* Attach to the memory. */
    if ((buffer = shmat(shmid, NULL, 0)) == (char *)-1) {
        printf("Parent: Could not attach to shared memory.\n");
        exit(-1);
    }

    /* Wait for signal from child process. */
    while(buffer[100] == 0) {
      sleep(1);
    }

    printf("Parent reading buffer \n");
    for(j=0;j<26;j++) {
      printf("%c ", buffer[j]);
    }
    sleep(2);
```

```
    }
}
```

The output is shown below:

```
 clang-7 -pthread -lm -o main main.c
 ./main
Hello TV Land!
Parent process! pid = 346
Child process! pid = 0
Child process is writer() process
Parent reading buffer
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

4.4 Shared Memory in Windows

Here is a similar set of programs, but they are written in Visual C and run on Windows systems. The functions for creating the shared memory pieced are different; the Unix functions create and open, the Windows functions do only one thing, create or open. Also, the Windows code creates two applications, but 1 at time, depending on how the #defines are set.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define TRUE 1
#define FALSE 0

/* Module prototypes. */
void block(void);

#define app1

#ifdef app1
void main(void)
{
    HANDLE hIntArray;
    int *pInt;
    int j;

    printf("app1 says 'hello!' \n");

    /* Make the shared file. */
    hIntArray = CreateFileMapping(INVALID_HANDLE_VALUE,
                                  NULL,
```

```
                                    PAGE_READWRITE,
                                    0, sizeof(int) * 100,
                                    "sharedIntArray");

/* Check to see if it was created correctly. */
if(hIntArray == NULL || hIntArray == INVALID_HANDLE_VALUE)
{
    printf("Could not create file mapping object (%d). \n",
            GetLastError());
    hIntArray = NULL;
    exit(0);
}

/* Map an adress to the shared file. */
pInt = (int*) MapViewOfFile(hIntArray,
            FILE_MAP_ALL_ACCESS,
            0,
            0,
            sizeof(int) * 100);

/* Make sure that the mapping was successful. */
if(pInt == NULL)
{
    printf("Could not map view of file (%d). \n", GetLastError());
    exit(0);
}

/* Load data. */
for(j = 0;j<100;j++)
{
    pInt[j] = j;
}

/* Show initial contents of shared array. */
for(j=0;j<100;j++) {
    printf("%d ", pInt[j]);
}

/* Signal user to start app2. */
printf("\n");
printf("Start app2 now and follow directions there. \n");

/* Wait for other program to change contents. */
while( !_kbhit() );
```

```
    /* Get the key off the input que. */
    _getch();


    printf("\n\n");
    printf("New numbers in array. \n");

    /* Show new contents of shared array. */
    for(j=0;j<100;j++) {
        printf("%d ", pInt[j]);
    }

    printf("\n\n");

    /* Get the key off the input que. */
    printf("Hit any key to end. \n");
    _getch();
}
#endif



#ifdef app2
void main(void)
{
    HANDLE hIntArray;
    int *pInt;
    int j;

    printf("app2 says 'hello!' \n");

    /* Make the shared file. */
    hIntArray = OpenFileMapping(FILE_MAP_ALL_ACCESS,
                                FALSE,
                                "sharedIntArray");


    /* Check to see if it was created correctly. */
    if(hIntArray == NULL || hIntArray == INVALID_HANDLE_VALUE)
    {
        printf("Could not open existing file mapping object (%d). \n",
                GetLastError());
        hIntArray = NULL;
        exit(0);
    }
```

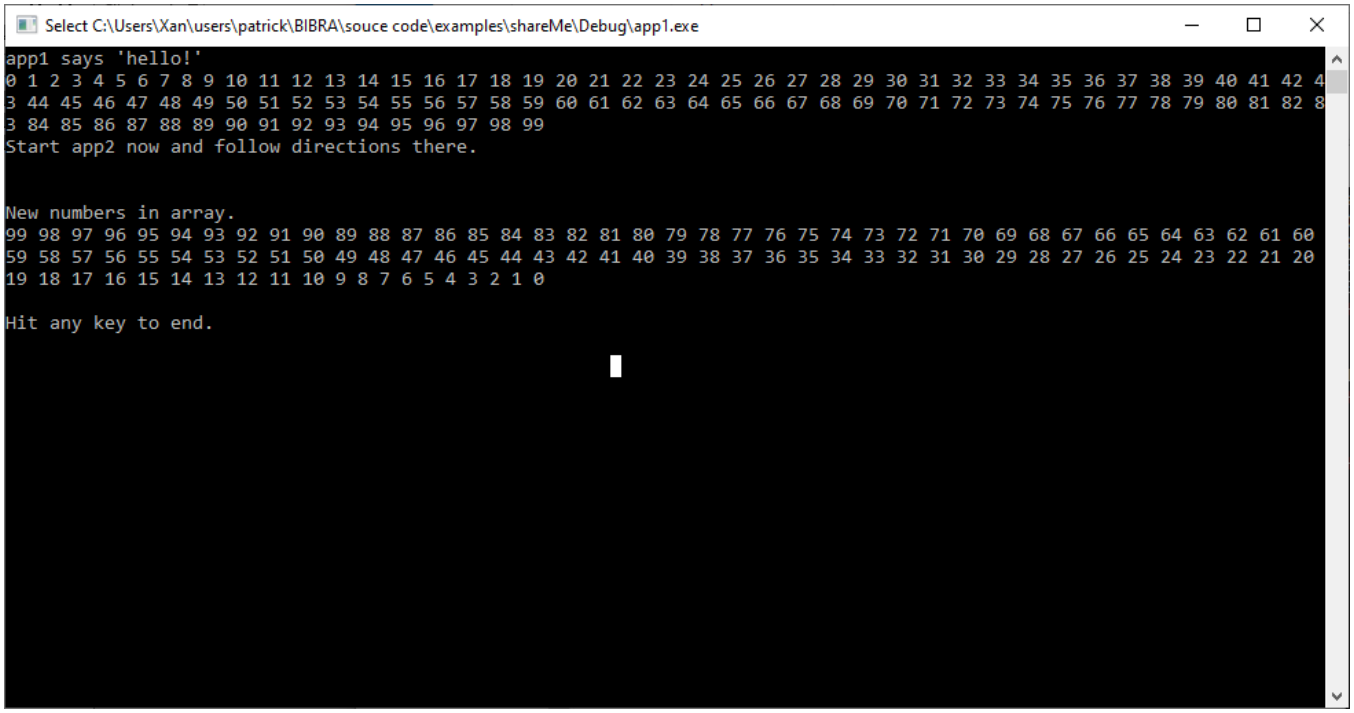
```
/* Map an adress to the shared file. */
pInt = (int*) MapViewOfFile(hIntArray,
            FILE_MAP_ALL_ACCESS,
            0,
            0,
            sizeof(int) * 100);

/* Make sure that the mapping was successful. */
if(pInt == NULL)
{
    printf("Could not map view of existing file (%d). \n",
            GetLastError());
    exit(0);
}


/* Signal user. */
printf("Contents of shared file. \n");

/* Show initial contents of shared array. */
for(j=0;j<100;j++) {
    printf("%d ", pInt[j]);
}

/* Wait to change contents of shared file. */
printf("\n");
printf("Hit key to change shared contents. \n");

/* Wait for other program to change contents. */
while( !_kbhit() );

/* Get the key off the input que. */
_getch();


printf("\n\n");
printf("Loading new numbers in array. \n");

/* Load new contents of shared array. */
for(j=0;j<100;j++) {
    pInt[j] = 99 - j;
}

printf("Hit the key in app1! \n");
```

```
    printf("\n\n");

    /* Get the key off the input que. */
    printf("Hit any key to end. \n");
    _getch();
}
#endif
```

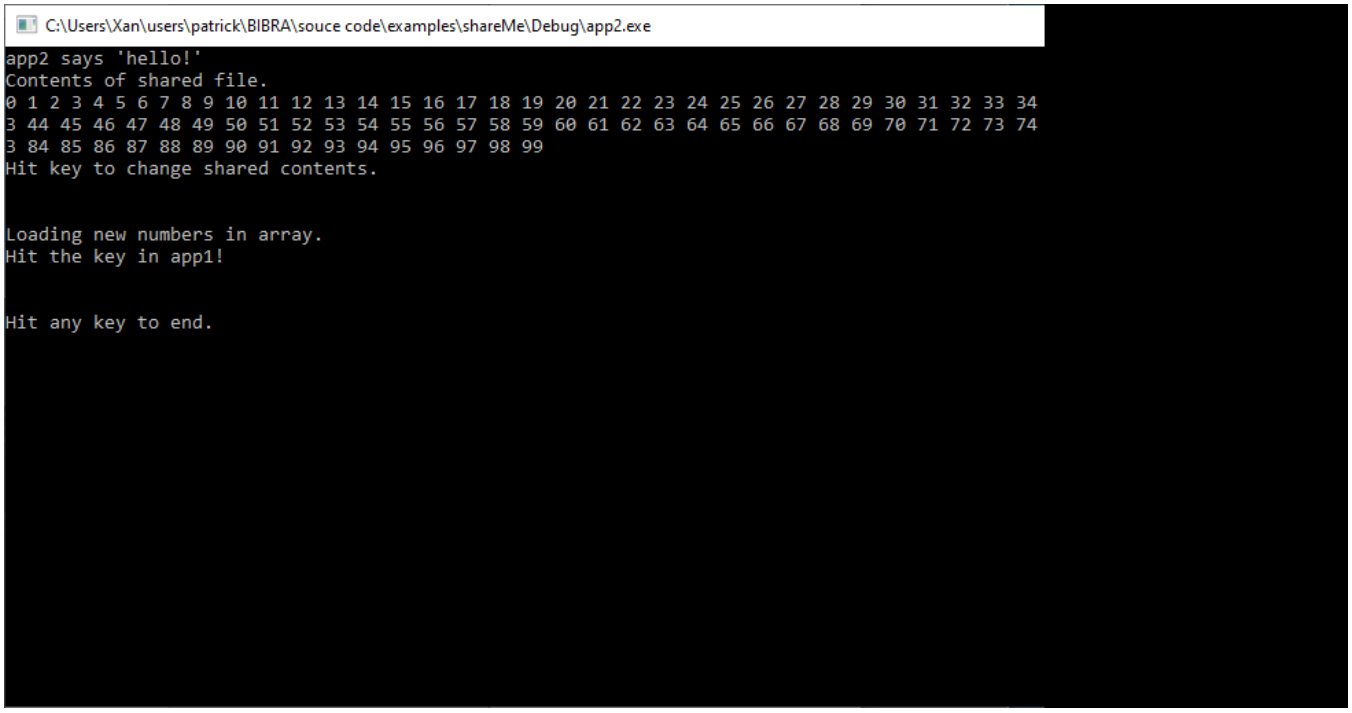
The two processes share a piece of memory that the parent process creates. The parent process creates the memory, loads it, and then asks the user to start the child process. Once the child process is started, it attaches to the shared memory and writes its contents to the screen. It prompts the user to allow it to change the contents of the shared memory. Once the user hits a key, the child process changes the contents of the memory, and asks the user to hit a key in the parent process. After the user does so, the parent process displays the contents of the memory, revealing the changes that were made by the child process.

In the above discussion the processes are referred to as parent and child, but in reality, they are only related in that they share a piece of memory. The fork() example was an example of parent/child relationship between two processes. The windows shared memory example could be made more like the fork() example by incorporating ideas in the spawn() example from section 4.2.

The output for the program app1 is shown below:



The output for the program app2 is shown below:



4.5 Interleaving of Processes and Instructions

When scientist and programmers design their various algorithms and processes, most, if not all, of the time, they do not imagine that as their creations are running, quite often the instructions are being intermixed with other concurrently running processes. This may not occur in a more primitive system, like a single partition batch system, but if the system has the ability to run more than one process at once, the phenomena is likely to occur.

Recall back to the multi-programmed batch system that ran that ran multiple processes concurrently by running them 10 instructions at a time. In that example, the instructions were very interleaved. To see this, observe the value of the PC as the SSL programs run. It will jump from partition to partition, most of time running 10 instructions from each partition. That is, unless an I/O statement is encountered, then it will switch to the next process (next partition too) and continue.

When programs are designed, the people designing them think of them as lists of instructions as illustrated by processes A and B on the left of figure 4.1. But when they run, an "instruction stream" is formed, much like what was described in the previous paragraph, and is shown on the right hand side of figure 4.1. It is akin to going to a restaurant and ordering a steak dinner. We look at our plate and see a wonderful steak, a big pile of green beans or broccoli, a pile of salad, and some potatoes. That is our perspective; to us each entrée is distinct, but our stomach has a different perspective. It is hanging our down there doing stomach things, processing the various entrées in whatever order they arrive, and how ever intermingled they may be.

Process Instruction Interleaving

Process A Instructions	Process B Instructions
A1	B1
A2	B2
A3	B3
A4	B4
A5	B5

CPU Instruction Stream
A1
A2
B1
B2
A3
A4
A5
B3
B4
B5

Figure 4.1 shows 2 processes, A and B. The instruction stream at the right of figure is there to illustrate the intermingling of instructions that could occur if A and B are running concurrently.

The CPU is at its core a machine that goes through its fetch, decode, execute cycle, continuously. Even if it has nothing to do, if there is power going to the CPU, for the most part it is in its fetch, decode, execute cycle. To put things colloquially, it just wants to process instructions, and it does not care where they come from. It is in an odd way like the stomach from our earlier description. How is that? The CPU is doing CPU things, hanging out processing instructions; from its perspective it just wants instructions to process; it does not care or make the distinction of where they come from. Strictly speaking this is not true; CPU's do not have emotions, but they do have user and supervisor modes, so the CPU does "know" when it is executing user instructions or OS process instructions.

The end result of this discussion is that instructions from different processes do get intermingled. The question to answer is "Does the fact that instructions get intermingled have any ramifications on how the processes execute?" The answer to that question is an unqualified "yes", or really a "yes, sometimes". If processes have common variables/dependencies then interleaving instructions can cause problems. If running two processes separately results in different answers than running them intermingled, then there is a problem. To see how this can happen, consider the code in figure 4.2 below.

```
Producer
count = 0
Repeat
  while(count == n)
        do nothing;
  buffer[in] = x;
  in = (in+1)%n;
  count++
Until(done);
```

```
Consumer
Repeat
  while(count == 0)
        do nothing;
  x = buffer[out];
  out = (out+1)%n
  count--;
Until(done);
```

CPU Instruction Stream
P1
C1
C2
C3
P2
P3

```
++
P1      load count r1
P2      add r1 1
P3      store r1 count
```

```
--
C1      load count r2
C2      add r2 –1
C3      store r2 count
```

Figure 4.2 Shows the classic Producer/Consumer problem. In this version from Silbershatz[3] the count variable is shared between the producer and consumer processes. If the instruction streams of the increment and decrement instructions get interleaved there is a high likelihood that the results will differ than if the instructions were run in a non-interleaved fashion.

In the code, a producer process puts items into a common buffer of size n. Each time an item, described by the variable x is put into the buffer, the count variable, which is shared, is incremented. The consumer process removes items from the shared buffer; each time it removes an item, again described by the variable x, it decrements the count variable. So the two processes share the buffer and the variable count.

If, for some reason, the increment and decrement instructions get interleaved, then the results may be different than expected. In the diagram the instruction stream {P1, C1, C2, C3, P2, P3} is shown. This means that the increment instruction was interrupted, and the decrement instruction was put right in the middle of it. How could that happen? An easy answer would be that the system used round robin scheduling and P1 was the last instruction allocated to the producer process, and C1 was the first instruction for the consumer process. Of course, a time quantum of only 3 instructions is unlikely, but even if the time quantum were 1000 instructions, and P1 was the 1000^{th}, the entire decrement instruction from the "C" process would be processed. This case could be shown as {P1, C1, C2, C3, ……. C_n, P2, P3…}; the results would be equivalent. Table 4.4 below illustrates the results of various arrangements of interleaved instructions. Assume that the initial value of count is 5. The initial instruction stream is representative of when there is no instruction interleaving, each time afterwards, there is instruction interleaving. Note that count is reset to 5 for each different group of instructions.

Instruction Number	***Instruction***	***Value of r1***	***Value of r2***	***Value of count***
P1	Load count r1	5	-	5
P2	Add r1 1	6	-	5
P3	Store r1 count	6	-	6
C1	Load count r2	-	6	6
C2	Add r2 -1	-	5	6
C3	Store r2 count	-	5	5
Count reset to 5				
P1	Load count r1	5	-	5
C1	Load count r2	5	5	5
C2	Add r2 -1	5	4	5
C3	Store r2 count	5	4	4
P2	Add r1 1	6	-	4
P3	Store r1 count	6	-	6
Count reset to 5				
C1	Load count r2	-	5	5
P1	Load count r1	5	5	5
P2	Add r1 1	6	5	5
P3	Store r1 count	6	5	6
C2	Add r2 -1	-	4	6
C3	Store r2 count	-	4	4
Count reset to 5				
C1	Load count r2	-	5	5
P1	Load count r1	5	5	5
C2	Add r2 -1	5	4	5
P2	Add r1 1	6	4	5
C3	Store r2 count	6	4	4
P3	Store r1 count	6	-	6

Table 4.4 shows various permutations of the instruction stream generated by the increment and decrement instructions from the producer/consumer processes illustrated in figure 4.2. Notice that when the instructions are intermingled the results are not the same if they were not intermingled.

Looking at the table it can be seen that when the instructions are interleaved, whichever process has the last instruction becomes the dominant process. That is, if the last instruction of decrement is executed after all of the instructions of increment, it is as if increment were never executed!

Why is this? It is because the two instructions are using different registers to do their work, in effect creating two copies of the count variable. To avoid this, the increment and decrement instructions would need to be rewritten so that they use the same register. If that were done, any combination of interleaved/intermingled instructions would work.

Getting back to the original idea of the producer/consumer processes, if a context switch from the producer to the consumer is executed while the producer is executing the count increment instruction, it will result in a miscount of the number of slots available in the

buffer, eventually causing problems. Each process keeps their own read/write variables, but they share the variable that regulates the total number of items in the buffer. So if there is a miscount, in this case count is too high, it could cause the producer to stall, and/or the consumer to unload unfilled buffer slots, or use old data (because the miscount, there are buffer slots that are assumed to be full, but there was no data transferred to them).

What is the solution to this dilemma? As a computer scientist or programmer who is doing application development, there is no way to know which registers the compiler designers decided to use, or the unpredictable ways that various different process's instructions get interleaved. So, from the applications designer's perspective the only way to guarantee correct results is make sure that these instructions do not get interleaved. This is called the "critical section problem" and is the subject of the next sections.

4.6 Critical Sections and Process Synchronization

The critical section problem can be defined as a region of code in which a process may be changing common variables, buffers, etc. While the process is in its critical section no other process can be in their corresponding critical section. Considering the example from the previous section, once the producer process entered the increment command, that code would have to complete without interruption, with the same being true for the decrement command. This is referred to as "Mutual Exclusion".

Solutions center around the following strategies:

- Increment/Decrement operations are atomic (cannot be broken down into a series of instructions)
- Protect the increment and decrement calls in critical sections using solutions devised using the following types of implementations:
    - Software solutions such as the general critical section solution some times referred to as the bakery algorithm.
    - Hardware solutions implemented by special "atomic instructions" such as TestAndSet() or Swap().
    - OS Solutions using semaphores are an OS/hardware solution that rely on special functions wait() and signal().

No matter the technique, to solve the critical section problem, the following requirements must be satisfied:

- Mutual Exclusion – If process p_i is executing its critical section, then no other processes can be executing their related critical sections.
- Progress – If the critical sections are vacant and a process p_i wishes to enter, then only processes that wish to enter a critical section can participate in the decision about which process enters next, and the decision process cannot take an indefinite amount of time.
- Bounded Waiting – There is a limit on the number of times that a process can be asked to defer entry into its critical section in favor of others.

Regardless of using a software, hardware, or OS/hardware technique, the structure of the critical section is the same. It consists of an entry point, the critical section, and an exit point, as shown in figure 4.5 below.

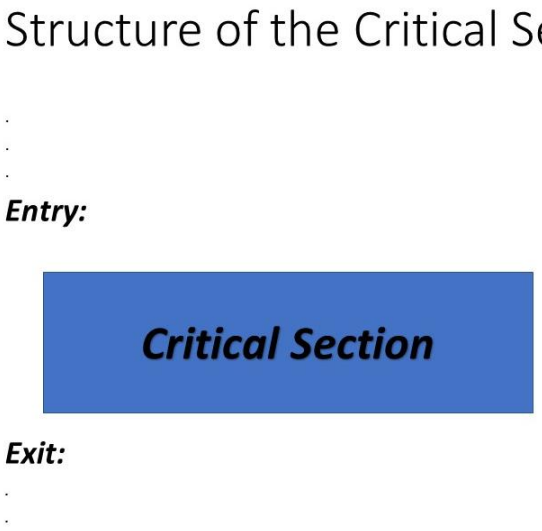


Figure 4.5 shows the structure of the critical section solution. The function of the entry section to ensure that mutual exclusion is upheld. It acts as a gate keeper. Once entry is gained, the code in the critical section is executed free of interruptions. Upon exit, variables are set so that other processes can proceed to their own critical sections.

The entry section usually consists of a loop in which a variable or variables are polled with each iteration of the loop. Depending on the variable's states, the loop at the entry section blocks entry to the critical section until the variable's states change to allow entry. Once entry is gained, the variables are set so that other processes trying to gain entry are blocked. After the code in the critical section is completed, the exit code is processed, meaning that gate keeper variables are reset again so that other processes can gain entrance to their critical sections.

As stated earlier, solutions to the critical section problem can be implemented using software, hardware, or OS/hardware techniques. In each of these types of implementations the requirements of mutual exclusion, bounded waiting, and progress must be satisfied. The actual implementations are similar in that the entry code is polling a "gatekeeper" variable, and the exit code is resetting that variable. In the next sections, the various implementation techniques will be reviewed. For this discussion the explanation will be done using just 2 processes, with the understanding that with modifications, the solutions can be extended to multiple processes.

4.61 Critical Sections – Software Solutions

The classic software-based solution is often referred to as "The Bakery Algorithm". The following paragraphs present a 3-step development of the Bakery Algorithm. The first two steps will not satisfy the requirements of mutual exclusion, bounded waiting and progress, but the third will.

4.62 First Try – Half Baked

The pseudo code below presents the first attempt at a two-process solution to the critical section problem. It uses a single variable in the entry section as a gate keeper, myTurn. The myTurn variable can be changed by either process, P_0 or P_1. Its initial value is 0.

Variables:
int myTurn;

Initialization:
myTurn = 0;

Process P_0:

.
.
.
Entry: while(myTurn != 0) {
doNothing();
}

/* Critical Section Code. */

Exit: myTurn = 1;

Process P_1:

.
.
.
Entry: while(myTurn != 1) {
doNothing();
}

/* Critical Section Code. */

Exit: myTurn = 0;

The myTurn variable will effectively enforce mutual exclusion of the critical section code. For instance, the myTurn variable is initially 0, and if the P_0 process wishes to enter its critical section it can do so. While it is in the critical section if P_1 attempts to enter, it will iterate in its entry code until the myTurn variable is set to 1 in the exit section of P_0, thus enforcing mutual exclusion. Once the P_0 exit code is run, P_1 can enter its critical section. Using this setup, the two processes will alternate, take turns, entering their critical sections.

There are problems however. Suppose P_0 is running, and P_1 never gets started. Then P_0 will enter its critical section, set the myTurn variable to 1, and never be able to go into its critical section again. It can only go into its critical section after P_1 sets myTurn back to 0 at its exit section, and if P_1 is not running this will not happen. P_0 will set in its entry section forever. This is a violation of bounded waiting.

4.63 Second Try – Half Baked Again

The second try does a better job than the first try in that the processes are not required to alternate back and forth. Instead of the myTurn variable is uses an array of flag variables as gate keepers that indicate when a process wants to enter its critical section. One thing to note about this algorithm, when a process wants to enter its critical section, it sets its flag variable indicating the desire to enter, and waits until the flags of the other processes are false. This means that processes are changing only their flags, but reading other processes flags. In the previous algorithm processes read and write the same variable.

The pseudo code below describes the algorithm for just 2 processes. To extend it to *n* processes the flag array would need n elements and for entry to be gained into a critical section all would need to be set to false.

Variables:
boolean flags[2];

Initialization:
flags[0] = flags[1] = false;

Process P_0:

```
.
.
.
Entry: flags[0] = true;
       while(flags[1] == true) {
              doNothing();
       }

/* Critical Section Code. */

Exit: flags[0] = false;
```

Process P_1:

```
.
.
.
Entry: flags[1] = true;
       while(flags[0] == true) {
              doNothing();
       }

/* Critical Section Code. */

Exit: flags[1] = false;
```

In this algorithm the flags[] array will effectively enforce mutual exclusion. If one process completes or is not started the other process will not stall, as did the first algorithm. The problem that occurs with this algorithm is when the CPU executes the 1st line of the entry code in P_0 and immediately afterwards executes the 1st of the entry code of P_1. If this occurs, both flag variables are set to true when the polling loops are entered and both processes will be locked in their entry loops forever, resulting in what is referred to as a "Deadlock". This is a violation of progress. The decision process of who gets to enter the critical section must take a finite amount of time, and since there is no provision to change the flags[] to false other than in the exit section, the finite time requirement is violated.

4.64 Last Try – The Bakery Algorithm

The first two algorithms were close to working, with the second being very, very close. What was needed was a way to break the deadlock caused by both processes waiting on the flags[] array to change state. The third algorithm accomplishes this by combining elements of the first two algorithms. The following pseudo code describes the Bakery Algorithm.

Variables:
int myTurn;
boolean flags[2];

Initialization:
myTurn = 0;
flags[0] = flags[1] = false;

Process P_0:

.
.
.
Entry: flags[0] = true;
myTurn = 1;
while((flags[1] == true) &&
(myTurn == 1)) {
doNothing();
}

/* Critical Section Code. */

Exit: flags[0] = false;

Process P_1:

.
.
.
Entry: flags[1] = true;
myTurn = 0;
while((flags[0] == true) &&
(myTurn == 0) {
doNothing();
}

/* Critical Section Code. */

Exit: flags[1] = false;

The Bakery algorithm combines elements of first two half-baked algorithms. The entry section first sets the flags elements and myTurn variable. If the algorithm were a person, and a very polite person at that, when setting the flags and myTurn variables it would be saying "I want to go in, but I let it be your turn". After saying that, it in the while loop it would be saying "I wait while you want to go in and it is your turn". After entrance to the critical section is gained, and the critical section is completed, the exit section simply states "I do not want to go in.". So, does it work? It avoids the deadlock encountered in the second algorithm by using the turn variable. If the two processes encounter their entry code at the same time, both elements of the flag array will be set to true, but whichever process hits the myTurn statement last, will allow the other process to enter the critical section, thus breaking the deadlock. If one process ends, or is not started, the other can still enter its critical section whenever needed by virtue of the other processes flags[] array element being false.

The common form of the algorithm is written so that it can be adapted to multiple processes. It is shown below.

Variables:
int myTurn;
boolean flags[2];

Initialization:
i = 0, 1;
turn = 0;
flags[0] = flags[1] = false;

Process P_i:
.
.
.
Entry: flags[i] = true;
turn = 1-i;
while((flags[1-i] == true) &&
(turn == 1-i)) {
doNothing();
}

/* Critical Section Code. */

Exit: flags[i] = false;

The Bakery algorithm for two processes is derived from the above pseudo code. Extending it to more than 2 processes is a matter of allocating more flags in the flags array and being very with the turn variable. One solution is to replace the turn variable with a set of "turn" Booleans.

In this implementation, the flags Booleans allow a process to say, just as before, "I want to go in." The turn Booleans allow a process to say "I defer to *all* of you guys!", as opposed to "I defer to you!", as was the case with the turn variable whose value was restricted to {0, 1}.

The code below supports critical sections in which up to 10 processes are allowed to enter, but only 1 at a time.

```
/* Bakery Algorithm critical section variables. */
bool flags[10];
bool turn[10];

  /* Entry Code. */
  /* Process i: I want to in. */
  flags[i] = true;

  /* Set the turn flags to defer to the other processes. */
  for(j=0;j<n;j++){
    if (j == i) {
      turn[j] = false;
    }
    else {
      turn[j] = true;
    }
  }

  /* Busy wait to get to the critical section. */
  while((checkFlags(flags, i, n) == true) &&
      (turn[i] == false)) {
    time++;
  }

  /* Critical Section. */
  value++

  /* Exit Code. */
  flags[i] = false;


bool checkFlags(bool *flags, int i, int n)
{
  int j;
  bool atLeast1isTrue = false;

  for(j=0;j<n;j++) {
    if ((j != i) && (flags[j] == true)) {
      atLeast1isTrue = true;
    }
  }

  return atLeast1isTrue;
}
```

4.65 Implementation Notes

The three algorithms previously discussed all use a polling loop as a gatekeeper to the critical section. In that polling loop there is a call to the doNothing() routine. In concept this is very nice, but as stated earlier, it is hard for the computer to not do anything. We could put in a statement that is designed just to take up a few CPU slices, like an assignment statement such as:

x = 1;

The problem with doing this type of thing is that depending on how the computers scheduling algorithms are set up (time quantum values etc.), once a process gets scheduled, and there is not I/O, because it is in the entry code doing an assignment statement, the process that it is waiting on may not get scheduled for some time. This could be a real problem with older Windows OS's such as Windows XP.

The easy remedy for this problem is use a sleep() or delay() statement instead of the assignment statement. This can be a workable solution, but it is somewhat computer/OS/OS version specific. In general the sleep() or delay() statements can put a process into the Wait Queue, which will then allow other processes to run, but can cause other problems, because once out of the Wait Queue a process goes to the Ready Queue and must be scheduled again, just to check to see if it can go into the critical section. Essentially, this is a process that must be tuned by selecting the correct delay times. It is not a good solution because once in the Wait Queue, precise control is compromised.

4.66 Critical Sections – Hardware Solutions

Hardware provisions can make critical section solutions easier. One option is to disallow interrupts while a process in a critical section. Interrupt disabling is not a good solution for multi-processor machines because it takes time to disable, and to pass the message to other processors to disable. Another disadvantage is that some system clocks are kept updated by interrupts.

Another viable option is the TestAndSet() instruction. The test and set operation is a machine instruction in the CPU's instruction set, meaning that it operates atomically (as one un-interruptible unit). Its function is to probe and retrieve the contents of a variable, return its value, and at the same time set its value to true. Pseudo code that describes the TestAndSet() instruction is below, along with some code using it in the entry and exit portions of a critical section.

```
Boolean TestAndSet(boolean x)
{
        Boolean temp;

        temp = x;
        x = true;
        return temp;
}
```

Variables:
boolean x;

Initialization:
x = false;

Process P_0:
.
.
.
Entry: while(TestAndSet(x)) {
Do nothing;
}

/* Critical Section Code. */

Exit: x = false;

If x is false it is set to true and we fall out of the loop, execute the critical section, and upon exit, reset it to FALSE. On the other hand, if x is true, we it set it to true and wait in the loop until another process sets x to false;

This is a good solution, as long as whatever system that the software is being developed upon provides user programs access to these types of instructions. Also, this is not a portable solution. Different systems have different versions of this instruction, making it somewhat system specific.

4.67 Critical Sections – Semaphores

The solutions outlined in the previous sections are good for explaining the concepts of critical sections and process synchronization, but for practical use, they are not ideal. The Bakery algorithm fulfilled the mutual exclusion, progress, and bounded waiting requirements, but because of issues associated with the entry polling loop, it requires tuning of delays in order to make it work. This causes reliability and portability problems. Likewise, using a primitive atomic instruction such as TestAndSet, presents its own problems with portability, and it also suffers from the same polling loop issues.

Semaphores are a tool to help with this. A semaphore, *S*, is a special integer variable that is associated with two OS/hardware supported functions, wait() and signal(). They are sometimes referred to as P(s) and V(s) for the Dutch words *proberen* (wait), and *verhogen* (signal).

- Wait(*s*) = P(*s*) = test
- Signal(*s*) = V(*s*) = increment

Pseudo code for wait() and signal() is given below:

```
wait(semaphore S)
{
        while(S <= 0) {
                doNothing();
        }
        S = S -1;
}


signal(semaphore S)
{
        S = S +1;
}
```

Both operations must be atomic, that is only one process can modify the semaphore value at any given time. Looking at the wait() function, it has the same polling loop as our other solutions do, complete with checking a variable and a call to doNothing(). The difference is that the wait() and signal() functions are OS supported, meaning that they are developed and tested by the OS designers. This implies that if a process is put into the wait queue because of a semaphore, it will be handled correctly as soon as its signal is sent. This is different than using delays and moving around in the wait and ready queues as would happen when tuning various versions of the Bakery Algorithm.

One thing for sure, the difference between the testing done for a commercial OS/semaphore system and the somewhat casual testing done when generating a working version of the Bakery Algorithm is substantial. The old sayings “We tested until we got it to work!” vs “We tested until we could not break it, and then tested for another 100 hours!” imply two entirely different regimens of testing.

A critical section solution using semaphores is shown in the pseudo code below. Again, the structure of the solution is as before.

Variables:
Semaphore *s*;

Initialization:
s = 1;

Process P_0:
.
.
.
Entry: wait(*s*)

/* Critical Section Code. */

Exit: signal(*s*);

Here the wait() function acts as the gate keeper to the critical section. Since the semaphore, *s*, is initially set to 1, whichever process hits the entry point first will proceed to the critical section. That process closes the door behind it because the wait() function decrements the semaphore. All other processes will go into their polling loops in their wait() functions. Once the first process hits the exit point the signal() function increments the semaphore. The first waiting process to be scheduled will then proceed into its critical section, and so on and so forth.

Semaphores have other valuable uses besides being the gate keeper to critical sections. Consider two processes using semaphores to synchronize their operations. To do so they will use a semaphore called synch.

Variables:
Semaphore *synch*;

Initialization:
synch = 0;

Process P_0:
{
.
.
.
S1;
signal(*synch*);
.
.

}

Process P_1:
{
.
.
.
wait(*synch*);
S2;

Statements located after the wait() statement in P1 will only be executed after the signal() in P0, meaning all statements before the signal() statement in P0 will execute before all statements after the wait() statement in P2.

4.68 Implementation Details

As mentioned previously, all the solutions, including the semaphore solution rely on *busy waiting*. Busy waiting is when the process loops continuously in the critical section entry code. It sometimes referred to sometimes as a spin-lock. In a single CPU system that is multi-programmed, spin-locks can eat valuable CPU slices. Because they require no context switch spin-locks can be good, as long as the time in them is short.

The cure is to redefine the wait statement to use blocking instead of busy waiting. Blocking means that the process puts itself into the wait queue when the wait() is encountered. When a signal() call is made, blocked processes are restarted by a wakeup() operation that puts them into the ready queue. Once they are in the run queue they check their semaphore. If the semaphore variable is greater than 0 then they exit the wait() call, otherwise they return themselves to the wait queue. In this manner busy waiting (the wasting of CPU cycles) is largely eliminated, but not completely. It does so at the expense of reaction time to the signal.

4.69 Examples using Semaphores

When two or more processes are waiting indefinitely for a signal that can only be sent by one of the waiting processes the processes are said to be "deadlocked". See the example below:

Example – Deadlocked processes

Variables:
Semaphore *S, Q*;

Initialization:
S = 1;
Q = 1;

Deadlock Example:

Process P_0:	***Process P_1:***
Wait(*S*);	Wait(*Q*);
Wait(*Q*);	Wait(*S*);
.	.
.	.
.	.
Signal(*S*);	Signal(*Q*);
Signal(*Q*);	Signal(*S*

The first series of waits decrements the semaphores, *S* and *Q*, to 0, resulting in an exit of the wait calls. The next set puts both processes P_0 and P_1 into the wait queue, resulting in a deadlock because the only way to get out is to increment the *S* and *Q* semaphores and the increment call is in the processes that are being blocked. Note: Deadlocks can be caused by many other conditions, such as processes competing for resources, etc. More discussion on that in the later sections of this chapter.

Classic synchronization problems such as the "Bounded Buffer Problem" present interesting material that illustrates the usefulness of semaphores, and puts a focus on how the semaphore's integer properties adds extra utility when compared to the Boolean type solutions provided by the Bakery Algorithm or TestAndSet(). In this example, there is a common buffer that is being used to store produced items that are to be consumed. The semaphores function to keep the from overflowing, or from being accessed when it is empty, and to make sure that only one process access the buffer at a time.

Example – Bounded Buffer Problem

Variables:
type item buffer[n];
Semaphore *mutex*, *empty*, *full*;

Initialization:
mutex = 1; /* Provides mutual exclusion when accessing buffer. */
empty = n; /* Counts the number of empty slots in the buffer. */
full = 0; /* Counts the number of full slots in the buffer. */

```
ProducerProcess()
{
        repeat {
                produceBufferItem();

                wait(empty);
                wait(mutex);

                add item to buffer;

                signal(mutex);
                signal(full);

        } until(false);
}
```

```
ConsumerProcess()
{
        repeat {
                wait(full);
                wait(mutex);

                remove item from buffer;

                signal(mutex);
                signal(empty);

                consumeBufferItem();

        } until(false);
}
```

4.7 Semaphore Implementation Details

The Unix/Linux operating systems provide convenient tools for creating semaphores. The code below provides an example of some code in which two processes are created using a fork() call. Both processes attach to a semaphore, and then a signal is sent from one process to the other. This code is very similar to the example provided in section 4.3. In that example the 100th element of the shared memory array was used to send a signal; this time a semaphore will be used. The semaphore is a more appropriate method; it has mutual exclusion, and will not suffer from busy waiting, or the problems associated with using sleep() or delay() style functions inside busy waiting / polling loops. Look to the comments for help on what the code is doing. This code was tested in Repl.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include <sys/sem.h>

#define SHARE_SIZE 128

int main()
{
  int j;
  char curr;

  /* Shared memory variables. */
  int shmid;
```

```
key_t memKey, semKey;
char *buffer;

/* Semaphore variables and data structures. */
int id;
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short *array;
} argument;

struct sembuf operations[1];
int retval;

/* Let user know that the program is alive. */
printf("hello tv land! \n");

/* Set up a semphore that can be used to cause the read part of the
   fork() to wait for the write portion. */
argument.val = 0;
semKey = 456;

/* Create semaphore, id is semKey, second arg is number of
   semaphores in array to create (just 1), third arg is permissions. */
id = semget(semKey, 1, 0666|IPC_CREAT);

if (id < 0) {
  printf("Error: Could not create semaphore. \n");
  exit(id);
}

/* Set the initial value of the semaphore to 0. */
if (semctl(id, 0, SETVAL, argument) < 0) {
  printf("Error: Could not set value of semaphore. \n");
}
else {
  printf("Semaphore %d initialized. \n", semKey);
}



/* Make a memory key for our shared memory buffer. */
memKey = 123;
if (fork() == 0) {
  /* Write part of fork(). */
```

```
/* We need to make sure we are attached to our semaphore. */
id = semget(semKey, 1, 0666);
if (id < 0) {
  printf("Error: Write fork() cannot access semaphore. \n");
  exit(id);
}

/* Create a piece of shared memory. */
if ((shmid = shmget(memKey, SHARE_SIZE, IPC_CREAT|0666)) == -1) {
  printf("Error allocating shared memory \n");
  exit(shmid);
}

/* Attach to the memory. */
if ((buffer = shmat(shmid, NULL, 0)) == (char *)-1) {
  printf("Could not attach to shared memory.\n");
  exit(-1);
}

curr = 'a';
buffer[110] = 0;

/* Write to buffer and also keep write pointer updated. */
for(j=0;j<26;j++){
  /* Send signal to start read process. */
  if (j == 12) {
    /* Set up a semaphore wait operation (P) */
    operations[0].sem_num = 0; /* First semaphore in the semaphore
                                  array. */
    operations[0].sem_op = 1; /* Set semaphore operation to send. */
    operations[0].sem_flg = 0; /* Wait for signal. */

    /* Do the operation. */
    retval = semop(id, operations, 1);
    if (retval == 0) {
      printf("## Write process signaled Read process. ## \n");
    }
    else {
      printf("Error: Write process semaphore operation error. \n");
      exit(-1);
    }
  } /* End if j == 12. */
```

```
      /* Set curr to the current character. */
      printf("%c ", curr);
      fflush(stdout);

      /* Load the shared buffer. */
      buffer[j] = curr;

      /* Increment current character. */
      curr = curr + 1;

      buffer[110] = (char)j;

      /* Sleep for 1 second. */
      sleep(1);
    }
    /* Unattach from shared memory. */
    shmdt(NULL);
  }
  else {
    /* Read portion of the fork(). */

    /* We need to make sure we are attached to our semaphore. */
    id = semget(semKey, 1, 0666);
    if (id < 0) {
      printf("Error: Read fork() cannot access semaphore. \n");
      exit(id);
    }

    /* Set up a semaphore wait operation (P) */
    operations[0].sem_num = 0; /* First semaphore in the semaphore
                                  array. */
    operations[0].sem_op = -1; /* Set semaphore operation to wait. */
    operations[0].sem_flg = 0; /* Wait for signal. */

    /* Do the operation. */
    retval = semop(id, operations, 1);
    if (retval == 0) {
      printf("** Read process recived signal from Write process. **\n");
    }
    else {
      printf("Error: Read process semaphore operation error. \n");
      exit(-1);
    }
```

```
    /* Create a piece of shared memory. */
    if ((shmid = shmget(memKey, SHARE_SIZE, IPC_CREAT|0666)) == -1) {
      printf("Error allocating shared memory \n");
      exit(shmid);
    }

    /* Attach to the memory. */
    if ((buffer = shmat(shmid, NULL, 0)) == (char *)-1) {
      printf("Could not attach to shared memory.\n");
      exit(-1);
    }


    /* Read the memory loaded by the parent process. */
    do {
      /* Put sleep here so that there is less chance
         of a context switch between when the buffer
         is written and when we check to see if the
         buffer filling has completed. */
      sleep(1);

      printf("\n");
      printf("shared buffer contents: ");
      for(j=0;j<=buffer[110];j++) {
        printf("%c ", buffer[j]);
      }
      printf("\n");
    } while (buffer[110] != 25);

    /* Unattach from shared memory. */
    shmdt(NULL);
  }

  return 0;
}
```

4.8 Exercises

Do the following exercises. Look to the examples in the text for hints. For programing tasks that need a Unix/Linux environment, the Repl online environment is very useful.

1. What is instruction interleaving? When is it okay? When is not okay?
2. In reference to instruction interleaving, how does this affect the CPU? How does it affect the processes involved?
3. How do we make sure that interleaved processes give consistent results?
4. What are the criteria needed to achieve consistent results by interleaved processes?
5. What is a critical section?
6. What are the criteria that we must satisfy to obtain a correct solution to the critical section problem?
7. What is the difference between progress and bounded waiting?
8. What is the structure of a critical section solution?
9. Describe the following:
   a. Bakery Algorithm
   b. Solutions using TestAndSet
   c. Solutions using Semaphores
10. What are the advantages and disadvantages of the solutions listed above?
11. When are the solutions efficient, when are they inefficient? What can be done lower wasted CPU cycles?
12. Describe how semaphores are more versatile than the other solutions we discurssed?
13. How are semaphores more reliable than the other solutions we discussed?
14. Can a deadlock occur using semaphores? If so, provide an example.
15. Show some pseudo code that implements the semaphores operations.
16. Practice yodeling the words verhogen and proberen. Make sure you enunciate and are easily understood by your fellow students. This will be part of the quiz for sure.
17. Use the fork() function to create a parent and child process. Program the parent process to write out a stream of ‘a’s and the child process to write out a stream of ‘b’s.
18. Use the Linux/Unix shared memory functions to create 2 programs. Set the first program up to read some input from the user. Once the input is read, place it into a shared memory buffer. When the first program sets the 100^{th} element of the buffer to a value of 99, let that be the trigger for the other program to output the contents of the buffer. To do this, both programs will have to share the buffer.
19. Implement the 1^{st} Half Baked algorithm, the 2^{nd} Half Baked algorithm and the Bakery algorithm in Linux/Unix using 2 processes. Use a piece of shared memory to hold the myTurn and flags[] variables. For the critical sections, just print out a message. To sleep for small amount look to the example below for guidance [4].

```
#include <time.h>
#include <errno.h>
#include <stdio.h>

int msleep(long msec);
```

```c
int main(void)
{
  int j;

  for(j=0;j<10;j++) {
    printf("j = %d \n", j);
    msleep(100);
  }

}

int msleep(long msec)
{
    struct timespec ts;
    int res;

    if (msec < 0)
    {
        errno = EINVAL;
        return -1;
    }

    ts.tv_sec = msec / 1000;
    ts.tv_nsec = (msec % 1000) * 1000000;

    do {
        res = nanosleep(&ts, &ts);
    } while (res && errno == EINTR);

    return res;
}
```

20. Program the bounded buffer example from section 4.69. Use the fork() function to create the producer() and consumer() processes. Use a piece of shared memory for the buffer array. Use semaphores as detailed in section 4.7 for the mutex, empty, and full semaphores.
    a. Let the producer() function load the buffer with the alphabet, starting at ‘a’ and ending at ‘z’, and then repeating. Let it operate on a frequency of one letter per second.
    b. Let the consumer() function read and display the buffer, one item at a time, at frequency of 1 item every 3 seconds.

True Story – The Revenge of Mrs. Lincoln

This story comes to the author from his older brother's wife. It is told from her point of view.

My husband and I were taking a summer vacation in the Northwest. We liked to take driving vacations, originally it was because of my husband's nerves, but then it evolved into something with a bit of a life of its own. You know, the open road, listening to music, and to audio books, and eating fun gas station food. I like to call it our "Griswold Vacation".

We had just left Park City and were headed to Walla Walla to check out the town where my husband's family lived when he was an undergraduate at Idaho. He had said that it was beautiful up in the Northwest during the summers and that we should visit Walla Walla and Moscow, not in Russia but in Idaho, so we were on our way. I used to plan these vacations pretty thoroughly, printing out maps, and researching the various places to stay. I still look for the places to stay, but with google maps on my phone and the navigation system in the car, there really is no need for the paper maps anymore.

Anyway, for some reason, my husband wanted to take a detour, to some little lake in Oregon, called Jubilee Lake. He said he had gone up there in the summers in the early 80's and it was really pretty. It seemed off the beaten track, but no worries, we had GPS, google, and the car's navigation system. My husband was a bit worried about it though because he had always gone to the lake from the Northern Oregon side (almost 30 years ago), and we were approaching it from the south side.

As we zipped along, we had the google app on the telephone talking to us, and the Lincoln navigation talking to us also. Mrs. Lincoln, as my husband calls her, was not quite as up to date as google (google girl), so in more rural areas we would ignore Mrs. Lincoln. As we motored along google girl and Mrs. Lincoln would direct us, and when they disagreed, I would settle the differences between the gals. The terrain was becoming more and more remote and neither Mrs. Lincoln nor Google Girl were giving solid advice. Often times, one or the other would tell us to make a turn and the road would look like a logging road that had not been used in 20 years. The thing is, it seemed like when we did not take Google Girls advice, she was okay with it, she would just find another route. Mrs. Lincoln, however, was not happy. The more we ignored her, the more insistent she seemed to get. She would talk louder, make bing, bing, bing, noises, an insist we turn around and take her route. Somehow, we arrived at Jubilee Lake, it must have been out of shear luck, but we where there. It was beautiful, but it was getting late so we did not hang around long and we jumped back into the car and headed off towards Walla Walla, by way of Milton Freewater, a small town in northeast Oregon.

Mrs. Lincoln advised us to take a right just after getting on a nice paved road soon after leaving Jubilee. The road looked nice, and my husband said "We may as well take it, it looks like a good road, and it will make her feel better." We zipped up the pretty road for a ways, and it led us up a steep climb. The road then narrowed and before we knew it, the road was unpaved solid rock, narrow, too narrow to turn around and there were shear

drop offs to either side. That being said, the scenery was beautiful, but one false move on my husband's part and we would have rolled down into a deep gorge on either side of the road. This went on for almost two hours, until the road turned from rock to being completely covered with gravel and choking dust. Finally we arrived in Milton Freewater, our car covered a quarter of inch thick from front to back with fine dust, except on the windshield that was gummy gross brown blue mud from my husband running the windshield wipers in an effort to see through the caked up dust.

I was flabbergasted, but my husband seemed happy. "Mrs. Lincoln seemed so happy while we were up on that mountain ridge, she did not complain at all like she did earlier. I think she feels much better!" he said. From that day on, we have been very careful not to cross Mrs. Lincoln.

Chapter 5 – Deadlocks

Suppose we are eating steaks, but we do not have enough silverware. On this particular day, I have the only knife, and you have the only fork. We sit down to eat, and I grab my knife and you grab your fork. We are both very hungry, but because we are stubborn, selfish louts we will not share our silverware. I will not put down my knife, and you won't put down your fork. If you did, I would grab it, but you won't, and vice versa. I want to poke you in the eye to make you drop your fork, but my mamma told me she would not let me eat any cake if I did something like that, and she told you the same thing. So, we just sit there. This is a deadlock. If we do not share our silverware, we will go hungry.

In a multi-programmed environment, deadlocks can occur between processes. Similar to the steak eating analogy, there will exist several processes and resources. When a process requests a resource that is not available the process will enter a wait state until the resource becomes available. For example, if a process "A" needs to write some of its results to the printer, but the printer is already in use, process "A" will need to wait until the printer becomes available, i.e. once the resource is acquired, the process can continue it execution thread. It can happen that a process will never leave its wait state because the resources requested are held and not released by other waiting processes. This situation is called a deadlock.

In order to study/analyze deadlocks, we use a tool called the "Resource Allocation Graph". This is a graphical way of viewing the system. Later we will use an algorithmic method that is equivalent. For this discussion we have a system model. It is defined as:

System Model – The system is a finite number of resources distributed among a number of competing processes.

Resources – There are several types of resources, each having some number of instances. Examples may include:

- Memory
- CPU cycles
- Files
- I/O devices
  - Card readers
  - Printers
  - Mass Storage Devices (Tapes)

Processes: These are the various user and system programs that are currently active in the system. A process requests a resource before using it, uses it, and then releases it.

- Request: If the request cannot be granted immediately (some other process is using it) the requesting process must wait until it can acquire the resource.

- Use: The process operates using the resource.
- Release: When the process does not need the resource anymore it releases it.

System calls are used to request and release resources. Examples of these calls are: open, close, malloc, free, new, delete, etc.

There are 4 necessary conditions for a deadlock. They are:

- Mutual Exclusion – At least one resource is held in a non-sharable mode, that is only one process at a time can use the resource – example, a critical section. If another process requests that resource while its being used, that process will have to wait.

- Hold and Wait – There must exit a process that is holding at least one resource and waiting to acquire other resources that are held by other processes.

- No Preemption – Resources cannot be preempted (like the CPU is preempted in a Time-Sharing system), they are only released voluntarily by the process using them.

- Circular wait – There must be a set of processes $\{P_0, P_1 \ldots P_n\}$ such that P_0 is waiting on a resource held by P_1, P_1 is waiting on a resource held by P_2 ….. P_{n-1} is waiting for a resource held by P_n, and P_n is waiting on a process held by P_0. Circular wait implies Hold and Wait.

5.1 Resource Allocation Graphs

A Resource Allocation Graph is a graphical method of showing the interactions between a group of processes and resources. It consists of a set of vertices V, and edges E. The vertices represent the system's processes and resources, the edges represent requests and assignments of the resources.

The set of vertices, V, consists of:

- Processes, P
- Resources, R.

The edges, E, consists of directed edges:

- Edges that go from a process to a resource indicate a request. If an arrow is going from process P_1 to resource R_1 that means that process P_1 needs an instance of the resource R_1 in order for it to run.
- Edges that go from a resource to a process indicate an assignment of the resource to the process. In this case we sometimes say process P_i "holds" resource R_j. We say "holds" because that the process will retain the resource until it has collected all needed resources. Once it has all the required resources it can run. Once the

process has run, it releases all of its resources, making them available for other processes.

Figure 5.1 below shows a resource allocation graph for the steak night example from the beginning of the chapter.

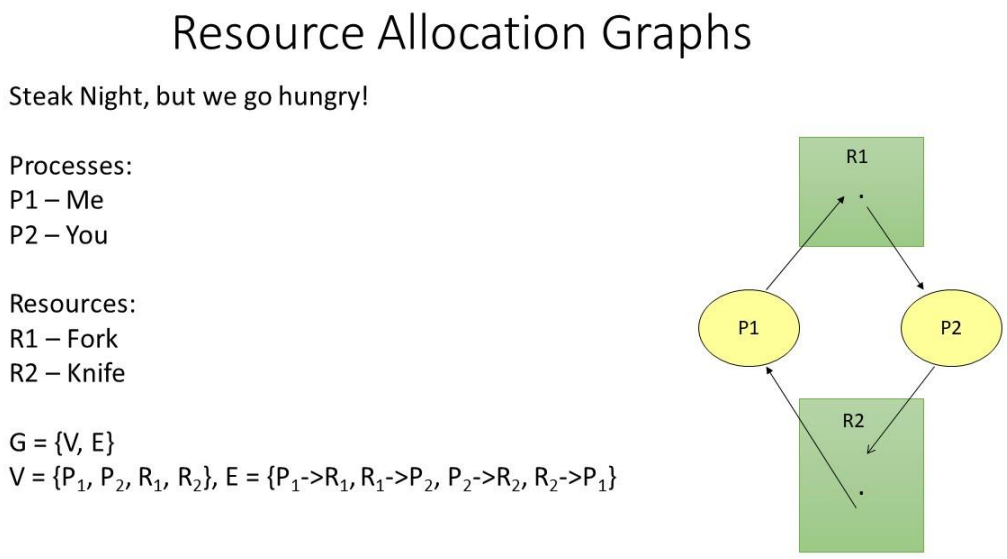
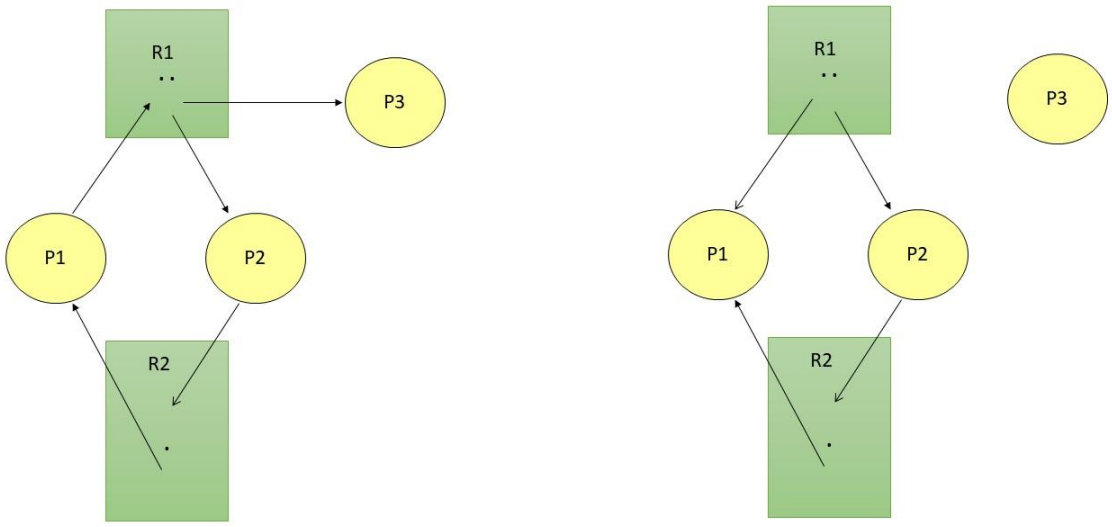


Figure 5.1 shows a resource allocation graph for the steak night example. As in the example from the text, there is a deadlock here. Looking at the graph on the right side of the figure, the cycle in the graph indicates that there is a possibility for a deadlock.

When looking at the graph in figure 5.1 it can be seen that a cycle exists; {P_1, R_1, P_2, R_2, P_1}. The cycle exists because once a process obtains a resource, it will not release the resource until gets all the required resources to run (Hold and Wait). Since processes are not allowed to interrupt other processes in order to recover resources, the cycle cannot be broken (in the case of the steak night example, no poking people in the eye). In this case there is a deadlock, but we say that cycle in the graph indicates the possibility of a deadlock because if there were a spare resource, say a spare instance of R_1, then the arrow from P_1 to R_1 would switch directions (go from R_1 to P_1) because the request would be fulfilled, and the cycle would be broken.

Looking at figure 5.2 below we see a similar situation. Here there is a cycle, but no deadlock. Process P_3 holds an instance of resource R_1. When P_3 completes it will release its instance of R_1 which will then be available to P_1. This event will cause the arrow from R_1 to be turned the other direction, back to P_1 breaking the cycle. Said another way, when P_3 completes it will release its resources allowing the request by P_1 to be satisfied. P_1 will have all the resources it needs, so it will run. After it runs, it will release its resources allowing P_2 to run.



Initial state, cycle exists, P3 running

Here, P3 has completed, released its instance of R1, allowing P1 to satisfy its request for an Instance of R1. P1 can now run.

Figure 5.2 shows two resource allocation graphs. The graph on the left is the initial state. Here there is a cycle, but no deadlock. There is no deadlock because P_3 will eventually finish running, releasing an instance of R_1. The diagram on the right side of the figure shows what happens when P_3 completes, allowing P_1 to satisfy its request for R_1.

In a nutshell, the way to determine if there is a deadlock is as follows:

- Draw the resource allocation graph.
- Look for any cycles.
- If a cycle exists, this indicates the possibility of a deadlock. However, if the cycle can be "unraveled", no deadlock exists. Unravelling a cycle can only occur if there is spare resource, or a process that can complete can release a resource to a process in the cycle.

Example 5.3 : The graph, shown in figure 5.3 is defined by the sets, P (processes), R (resources), and E (edges). There is no deadlock here; it is very similar to the example depicted in figure 5.2.

P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = {P1 $\rightarrow$ R1, P2 $\rightarrow$ R3, R1 $\rightarrow$ P2, R2 $\rightarrow$ P2, R2 $\rightarrow$ P1, R3 $\rightarrow$ P3}

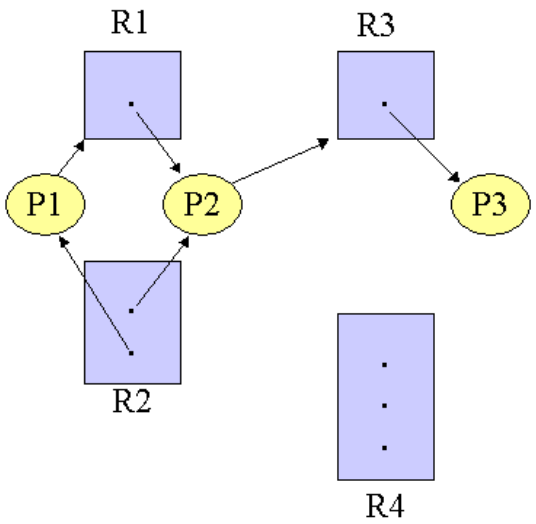


Figure 5.3 Notice that in the above example there are no cycles in the graph. A resource allocation graph without cycles indicates that no deadlocks exist.

Figure 5.4 below shows resource diagram in which there is deadlock. There are two cycles:

{P1→ R1→ P2→R3→ P3→ R2→ P1}
{P2→ R3→ P3→ R2→ P2}

The deadlock exists because no process can release a resource because they cannot run because they are waiting for resources held by other processes.

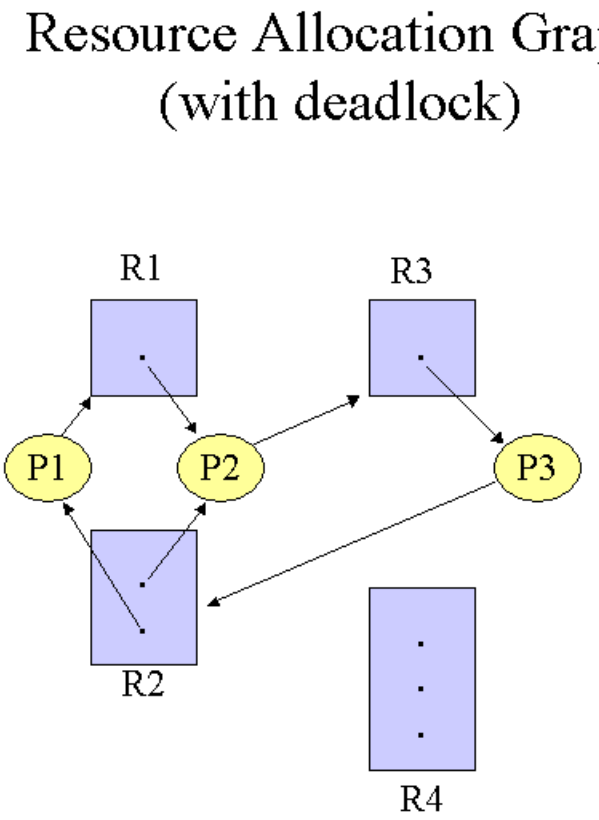


Figure 5.4 shows resource allocation graph with a deadlock.

Figure 5.5 shows a resource allocation graph in which there is a cycle, but no deadlocks, this similar to the diagram in figures 5.2 and 5.3.

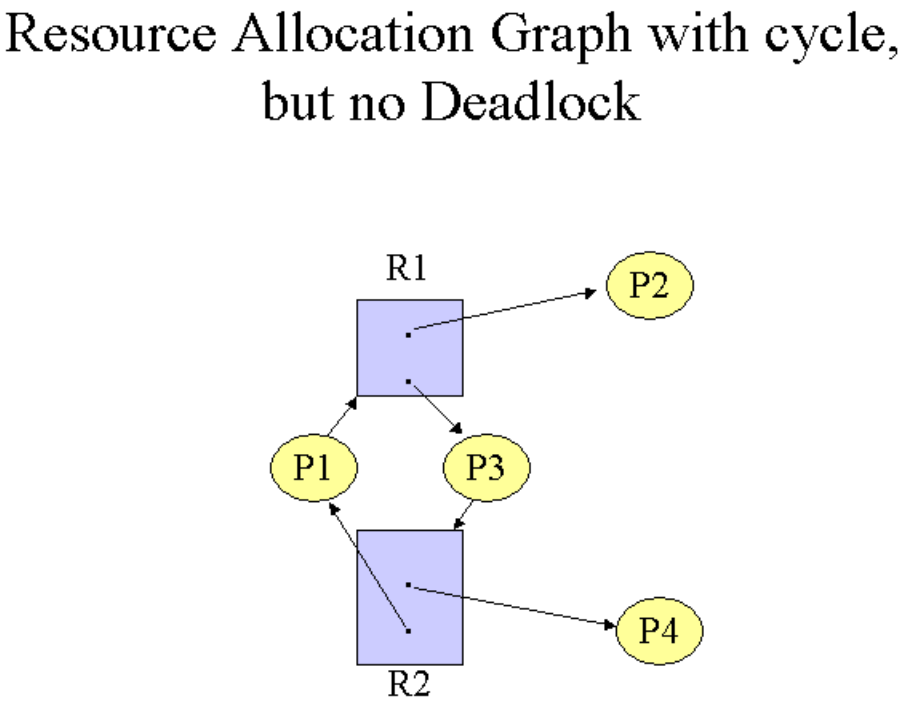


Figure 5.5 shows a resource allocation graph with a cycle but no deadlock. Here processes P_2 and P_4 hold resources that the other processes are requesting. Once they finish and release their resources, the cycle will be broken.

Cycle is:

{P1→ R1 → P3→ R2→ P1}

There is no dead lock because P_2 or P_4 can complete processing and release instances of R_1 and R_2, allowing P_1 and P_3 to acquire the resources needed to complete.
Methods for handling deadlocks and the Bankers algorithm.

5.2 Handling Deadlocks – Prevention and Avoidance

There are two methods of dealing with deadlocks, three if you count doing nothing at all. Surprisingly enough, this is common. On smaller systems, it is often up to the user to manage deadlocks. It used to be very common for a system to grind to a halt right before the user's eyes. This happened when there were many applications open, meaning system memory was largely taken up by all the applications. If an application needed more memory as it was running, the system would commonly slow down and sometimes stop. Mouse reaction times would slow, opening a window would result in long wait times, and empty window frames on the screen. The system had no means of dealing with these problems, i.e. the do nothing approach, so the user would shut down applications until the system recovered.

The do nothing approach is okay for PC's because only one person is using the CPU and its resources, but for a multi-user system, this approach is less than ideal. Two possible ways to solve these issues are deadlock prevention and deadlock avoidance. They are discussed in the following sections.

5.21 Deadlock Prevention

Deadlock prevention is a method of addressing the deadlock issue that attempts to make sure that it is not possible for a deadlock to occur. In order to do this, the conditions necessary for a deadlock to exist must be challenged. Said another way, deadlock prevention is the use of a protocol to ensure that system never enters a deadlock state. This method tries to ensure that at least one of the conditions for deadlock cannot occur by constraining how resources requests are made. Some techniques are outlined in the bullets below:

- Mutual Exclusion: This must hold for non-sharable resources, such as printers, but resources such as read-only files can be shared, thus breaking mutual exclusion. In general deadlock prevention cannot be accomplished by denying mutual exclusion; some resources are not sharable.

- Hold and Wait: Processes can be asked to request and acquire all resources before they begin; all resources are obtained by using system calls, which are moved to the beginning of the process. Alternatively, they can be forced to request resources only when they have none.

Problems with these methods:

- Starvation: A process may have to wait indefinitely for resources before it can start.
- No Preemption: To ensure that this condition does not hold, the following logic can be used. If a process is holding resources and has outstanding requests for more resources that cannot be satisfied, then it can be forced to release the resources it is holding. The process will be restarted only when it can regain its old resources and the other ones that it needed. Or when a process is starting, the OS checks to see if the resources the process needs are available. If so, they are allocated, if not, the OS checks to see if the needed resources are being held by other waiting processes. If so, these resources are preempted and the process runs, otherwise, the process with its partial list of resources is put into the wait queue, where its resources can be preempted by other processes.

- Circular Wait: In order to break the circular wait condition a number system can be assigned to the resources. If a process is not allowed to request any resources that are assigned numbers greater than the resources it holds, then the circular wait conditions can be minimized if not eradicated. As an example, Suppose two processes, P_0 and P_1 both need resources R_9 and R_{10}. If the processes make their requests at the same time, according to the rule they will request R_{10} first.

Whichever process gets R_{10} will then be able to request R_9 and then get R_9, finish, release its resources, thus allowing the other process to complete. See figure 5.6 below.

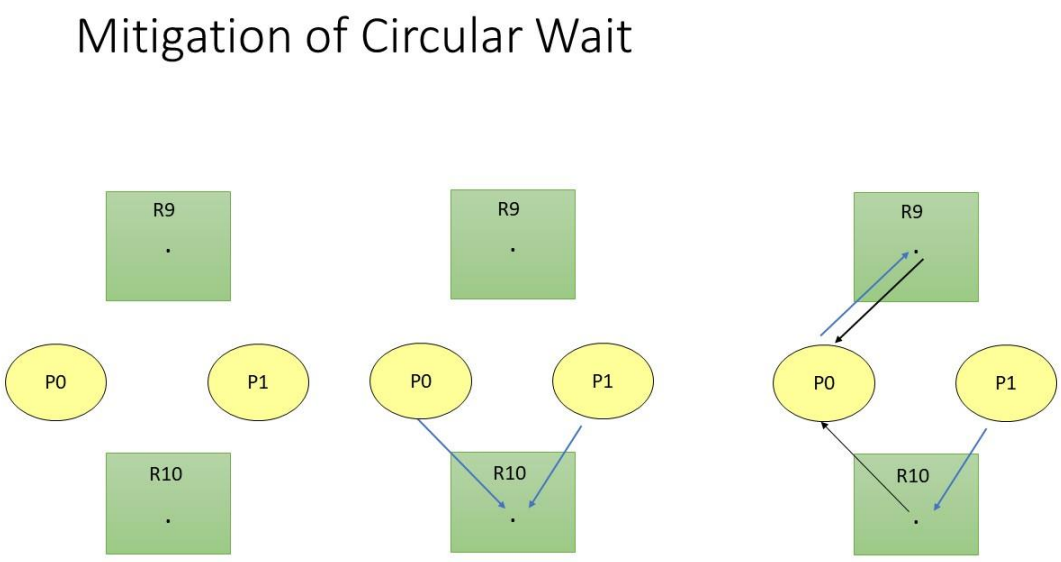


Figure 5.6 shows a sequence of 3 diagrams intended to illustrate a numbering technique designed to stop cycles from occurring, thus stopping deadlocks. The figure to the left shows the processes and resources before any requests are made. The middle diagram shows the two processes requesting R_{10}. If R_{10} is awarded to P_0 first, then P_0 can request and then hold R_9 and complete its processing.

These rules would work, as all long processes request their highest numbered resources first.

5.22 Deadlock Avoidance

If the OS is given information about the required resources of the processes during their lifetimes, it is possible to come up with an order in which the processes can run and not enter a deadlock condition. For each resource request, the system must consider which resources are currently available, the resources currently allocated to each process, and the future requests and releases of each process. Using this information the OS can then decide if the resource request should be satisfied or delayed. Algorithms that avoid deadlocks by using the resource state information work by avoiding circular wait conditions. In order to not have a deadlock, the OS has to be in a Safe State. A safe state exists if there exists a safe sequence.

Safe State: A state is safe if the system can allocate resources up to its maximum in some order and still avoid a deadlock. The OS is in a safe state if there exists a safe sequence.

Safe Sequence: For each process, P_j, the resources needed can be satisfied by currently available resources and the resources held by the processes $P_{j, j-1, j-2\ldots}$ that came before it.

Given a sequence of processes: {P_1, P_2, P_3…..Pj} All the resources Pj will need are available (not allocated to any process) or are/were held by P_1 to P_{j-1}.

There is a table-based algorithm that can be used to find a safe sequence if it exists. It is called the Bankers Algorithm. It is analogous to the resource allocation graphs in that it when we analyze a cycle in the resource allocation graph, we are basically looking for a spare resource so that we can break the observed cycle. Once we find that resource we identify a progression of processes that can run, then release their resources so that other processes can run. This works well, until the graph becomes two complicated for use to analyze visually. The Bankers Algorithm provides an automated alternative to this process. The pseudo code for the Bankers Algorithm is presented below.

Bankers Algorithm

Variables:
n – number of Processes
m – number of resource types
$Avail_{1\ to\ m}$ – number of resources of each resource type currently available
Max[i][j] – max number of resources of type j that process i will need
Alloc[i][j] – process i has this many resources of type j
Need[i][j] = in order to run process i needs this many resources of each type to run. Need[i][j] = Max[i][j] – Alloc[i][j];
Finish[] – an array of Booleans indicating if a process has completed or not.

Algorithm:
```
/* Set the working array to whatever resources of each type are currently available. */
Work = Avail;

/* Set all finish array elements to false. */
Finish[0 to (n-1)] = false;

While (there exist a i such that:
        Finish[i] = FALSE
        Need[i] <= Work) {


                Work = Work + Alloc[i];
                Finish[i] = TRUE;
}
```

If all of Finish = TRUE, then system is in a safe state.

Below we present an example. In this example, we first will show all the resources that are allocated, and the resources needed for each of the processes to complete. See Table 5.7 below.

i	Process	Allocated			Max			Avail			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3							False
1	P_1	2	0	0	3	2	2							False
2	P_2	3	0	2	9	0	0							False
3	P_3	2	1	1	2	2	2							False
4	P_4	0	0	2	4	3	3							False

Table 5.7 shows the initial state of the problem. We have 3 resource types, A, B, and C. We have 5 processes, P_0 through P_4. Each process has been allocated some of each type of resource, shown in the Allocated columns. Also, each process has a maximum number of resources it needs so it can start running, the Max columns. No process has completed, the Finish column.

The next step is to calculate what each process needs so that it can run. See table 5.8 below. In this table we have calculated the Need column by subtracting what has already been allocated by the system to each process from what is needed by each process.

i	Process	Allocated			Max			Avail			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3				7	4	3	False
1	P_1	2	0	0	3	2	2				1	2	2	False
2	P_2	3	0	2	9	0	0				6	0	0	False
3	P_3	2	1	1	2	2	2				0	1	1	False
4	P_4	0	0	2	4	3	3				4	3	1	False

Table 5.8 shows the calculation of the Need column. Looking at process 0, Max = (7, 5, 3) and Allocated = (0, 1, 0). We get the values for Need by subtracting Allocated from Max. (7, 4, 3) = (7, 5, 3) – (0, 1, 0).

Now we are ready to start. Given that the system has some resources still available, 3 of resource type A, 3 of resource type B, and 2 of resource type C, we start by looking at the need column and finding which process can run if we allocate what we have available to that process. So we need to find a Need "vector" that is less than our Avail "vector". Starting from process 0 and working down, we see that our Avail vector (3, 3, 2) is less than the first Need vector (7, 4, 3), so we cannot run process 0. Process 1 has a Need vector that is (1, 2, 2) which is less than or equal to our avail vector so we can run that process. Once the process is run, it will complete and we will collect its released resources and add them to our available resources resulting in an Avail vector of (5, 3, 2). Table 5.9 below shows this step.

i	Process	Allocated			Max			Avail = (3, 3, 2)			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3				7	4	3	False
1	P_1	2	0	0	3	2	2	5	3	2	1	2	2	True
2	P_2	3	0	2	9	0	0				6	0	0	False
3	P_3	2	1	1	2	2	2				0	1	1	False
4	P_4	0	0	2	4	3	3				4	3	1	False

Table 5.9 shows the first iteration in the Bankers Algorithm. Process 1 was able to be run. Once it was run its resources went back to the system resulting in a change in the available resources. (3, 3, 2) + (2, 0, 0) = (5, 3, 2).

The first process in the safe sequence that we are searching for is process 1, and we mark it as complete as shown in the Finish column. The next step is to search for the next process that can be run with our available resources (5, 3, 2). Starting at the top, P_0, we can see that it requires more resources than what we have. We move sequentially down the chart, and it is not until we reach process 3 that we see that we have enough resources to run a process. We use our resources, run the process, and then collect the released resources, which is now, as before reflected in the Avail column. Table 5.10 below illustrates this step.

i	Process	Allocated			Max			Avail = (3, 3, 2)			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3				7	4	3	False
1	P_1	2	0	0	3	2	2	5	3	2	1	2	2	True
2	P_2	3	0	2	9	0	0				6	0	0	False
3	P_3	2	1	1	2	2	2	7	4	3	0	1	1	True
4	P_4	0	0	2	4	3	3				4	3	1	False

Table 5.10 shows the next iteration of the Banker Algorithm. Here we see that P_3 was run and once its resources were released, we added them to the those available. It is reflected in the avail column for process P_3.

The process of finding processes which need less or equal resources to what is available continues. If we find that we can run all processes, we have found a safe sequence. If at some iteration, no processes Need column is less than or equal to what is in the latest Avail column, there is no safe sequence and a deadlock exists. The next 3 tables below complete the process for this example.

i	Process	Allocated			Max			Avail = (3, 3, 2)			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3	7	5	3	7	4	3	True
1	P_1	2	0	0	3	2	2	5	3	2	1	2	2	True
2	P_2	3	0	2	9	0	0				6	0	0	False
3	P_3	2	1	1	2	2	2	7	4	3	0	1	1	True
4	P_4	0	0	2	4	3	3				4	3	1	False

i	Process	Allocated			Max			Avail = (3, 3, 2)			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3	7	5	3	7	4	3	True
1	P_1	2	0	0	3	2	2	5	3	2	1	2	2	True
2	P_2	3	0	2	9	0	0	10	5	5	6	0	0	True
3	P_3	2	1	1	2	2	2	7	4	3	0	1	1	True
4	P_4	0	0	2	4	3	3				4	3	1	False

i	Process	Allocated			Max			Avail = (3, 3, 2)			Need			Finish
		A	B	C	A	B	C	A	B	C	A	B	C	
0	P_0	0	1	0	7	5	3	7	5	3	7	4	3	True
1	P_1	2	0	0	3	2	2	5	3	2	1	2	2	True
2	P_2	3	0	2	9	0	0	10	5	5	6	0	0	True
3	P_3	2	1	1	2	2	2	7	4	3	0	1	1	True
4	P_4	0	0	2	4	3	3	10	5	7	4	3	1	True

The safe sequence found by the algorithm is: 1, 3, 0, 2, 4. Looking at the Avail column for P_4 it can be seen that after all the processes have been run there are 10 resources of type A, 5 of type B, and 7 of type C. An easy way to make sure that there no mistakes made is check the total system resources before the process was started; it should be the same as when the process completed. We do this by adding what was initially available to everything that was allocated. To do this we sum up each of the allocated columns, resulting in (7 A, 2 B, 5C). We now add that to what was initially available (3A, 3B, 2C)

(7, 2, 5) + (3, 3, 2) = (10, 5, 7)

This result, the total resources before we went through the process is equal to what we found once all processes had been run and released their resources, indicating that either we did everything correct, or we got super lucky and made offsetting mistakes.

5.3 Exercises

1. What is a deadlock?
2. What are the conditions required for deadlock to occur?
3. Show a deadlock diagram for the restaurant steak challenge in which there are three people, each with 1 knife. There are only 2 forks and person 1 holds a fork, and the other 2 people are requesting forks.
4. How do we graphically represent a system of processes and resources when illustrating deadlock?
5. What is a safe state? How does it related to a safe sequence?
6. Describe some methods of deadlock prevention.
7. How is deadlock prevention different from deadlock avoidance.
8. Draw the allocated portion of the resource allocation graph for the Bankers Algorithm from table 5.7. The Allocated column reflect the "hold" arrows.
9. Now add in the Need columns as calculated in table 5.8. The Need columns will reflect the "request" arrows. How do you like your graph?
10. Make sure you can draw a deadlock/resource diagram and convert it to a Bankers Algorithm table and vice versa.
11. Show some pseudo code for the Bankers Algorithm.
12. Now convert your pseudo code to a program that can solve the example given starting in Table 5.7.

True Story – Pounds vs Kilograms

This story comes to the author from his twin brother. It is told in first person from his point of view.

Uncle Johnny had a nice sailboat. It was a 36 foot twin mast boat. Along with the sails, it had an auxiliary motor with about 30 horsepower. Most sail boats of any size have these motors for docking, or maneuvering in a harbor. Uncle Johnny had just bought the boat and was going through the systems to make sure it was ship shape for the summer/fall sailing season, and his nephew, Herb, and I were helping him with the various tasks. The thing is, Uncle Johnny really could not see very well. That was understandable, seeing as how he was about 92 at the time. We knew that he could not see well, so we wanted to make sure that nothing bad happened because of some little oversight, or something he missed, or did not notice.

The point was really brought home when I walked down to the dock where the sailboat was tied up. For some reason I was looking for Uncle Johnny, maybe to bring him his lunch; he had told me he would be down by the dock, but I did not see him. It was kind of nice being around Uncle Johnny, he reminded me of a combination of my long passed grandpa, and a super muscled up Mr. Magoo. Anyway, I hollered for him, "Hey Uncle Johnny, I got your lunch!" I heard him call back, "I'm down here!" I went down into the sailboats cabin and he was sitting at the table in the galley. He asked me to look at his back; he said he had scratched it. Sure enough, there was a decent abrasion on his back, but it was not bleeding. "How did you do that?" He replied, "Well, I was up on the deck looking for my paint brush and I the next thing I knew I was lying on the galley table. You sure my back is not bleeding?" I looked over the table, and yep, there was open

hatch about 6 feet over it. Holy moly, this guy just fell through that, and that all that happened is what I am thinking. I resolved to make sure that nothing else like that happened.

Our task for the day was pull the auxiliary motor out of the boat and bring it to a marina to have it checked over. According to the manual it only weighed about 435 pounds. That sounds heavy, but for a diesel motor that is not much, plus Herb and Dave were there, so there was plenty of stout help. Herb proposed that we use the boom on rear mast and hoist the motor out of the access hatch and then swing it over to the dock. The boom was made of wood, about 5 inches around, so it seemed like it would be okay, and after all the motor was not that big or heavy. After the motor was loosened up, we used a come along attached to the boom and lifted the motor out. There was a lot of creaking and the come along seemed overstressed, but it was rated for way more than 430 pounds. It took everything Dave, Herb, and I had to get the motor onto the dock, and then using a dolly, roll it up to the truck. In fact, when we set it down on the dock, I swore that it seemed like it was going to break the boards and fall though to the water below. Dave quipped "Maybe its that it just small in size, all 400 something pounds in a small package." Whatever, that thing was dense or what. Anyway, another a few moments of the three of us turning blue, red and purple, we got the motor into the truck.

We were sitting on the back of the truck happy to be finished when Uncle Johnny walked up. "You all got it with no problems, good for you guys! Thanks!" he said. "No problem-o I replied, but we must be getting weak, that is the heaviest 400 pound thing I have ever seen." I replied. Uncle Johnny laughed hard and said, "400 pounds my butt, that thing weighs 435 kilos!". We all looked at each other, 435 kilos comes out to about 960 pounds. Whoops, so much for us saving Uncle Johnny from his bad eyesight. Sometimes it is great to be lucky; we sure did not notice the pounds vs kilograms thing, and could have hurt someone, sunk a boat, or whatever. Thank god for luck.

Chapter 6 – Memory Management

In this chapter we will look at memory management. We will approach it by first looking at the fundamentals, the basics of addressing, and simple memory systems. From these we will gain perspective on why modern computers use more complex systems, i.e. paging. From an implementation standpoint, it is worth recalling that some of the OS simulation exercises from the first two chapters touched on the basic concepts; once we have expanded our conceptual knowledge, we will go back to these simulations and integrate some of these newfound ideas into our simulations from the earlier chapters.

6.1 Memory Basics

Memory comes in a variety of sizes. The most fundamental memory unit is the bit. It is a single 1 or 0 in the computer, and in hardware is stored in a single "flipflop". Most of the time the bits themselves are not directly addressable. In order to change single bits "masking" strategies are commonly employed.

The next size up from the bit is the byte. A byte consists of 8 bits grouped together. There is a size that is between the bit and the byte; it is called a nibble, and it is 4 bits. Cool name, but not really used in modern times. It is often said that memory is consists of "words". A word is not necessarily some specific unit of memory, like a byte, but more precisely it is the size of memory that gets transferred to and from the CPU to main memory. The bullets below summarize the various memory units.

- Bit – A 0 or 1, represented by a flipflop, or a switch
- Nibble – 4 bits, not commonly used in modern times
- Byte – 8 bits
- Word – 8, 16, 32, 64 bits

Main memory can be viewed as a large single dimensional array of memory words. Each word of the memory can be accessed by the CPU's low-level memory transfer instructions, i.e. load and store. Typically, when variables are accessed in a program, what is occurring is that load and store instructions are transferring data to and from the programs symbol table (variable storage in the program's segment) in and out of the CPU's registers. This transfer between the CPU and the memory occurs on the computer system's data bus. The data bus consists of a group of data transfer lines, along with some control lines. It is the configuration of this data bus that determines the word size of a system. Older systems that had 8 data lines were 1 byte word systems, but as systems became progressively more powerful, more and more data lines were added.

As alluded to earlier, a memory word can be 1 byte, 2 bytes, 4 bytes, 8 bytes, or even 16 bytes. The size of the memory word is dependent on the OS/hardware configuration of the computer, and is defined as the amount of memory that is transferred on the system bus from the CPU to the computers main memory. Older PC's such as the x86 based systems often had 2 byte or byte word sizes. Most modern computers usually use 4 or 8 byte word sizes, while the newest systems use 8 and 16 byte word sizes. The MIPS

instruction set that was mentioned in chapter 1 uses a 4 byte word, as does the SSL language and simulator and development tools used in chapters 1 and 2.

When working with memory at a low level, word alignment issues can come into play. For example if a 32 bit system (4 byte word size) is moving data that is 10 bytes long, the end of the data does not fall on an even word boundary. That is, since the end of the data is at the 10 byte mark, the nearest word boundaries are at the 8 byte and 12 byte marks. When the computer reads the data it must retrieve 12 bytes to get the 10 bytes it needs, and likewise, when it writes the data, it will write 12 bytes out. Those last two bytes on the end are not used, but they are along for the ride. This is illustrated in figure 6.1 below.

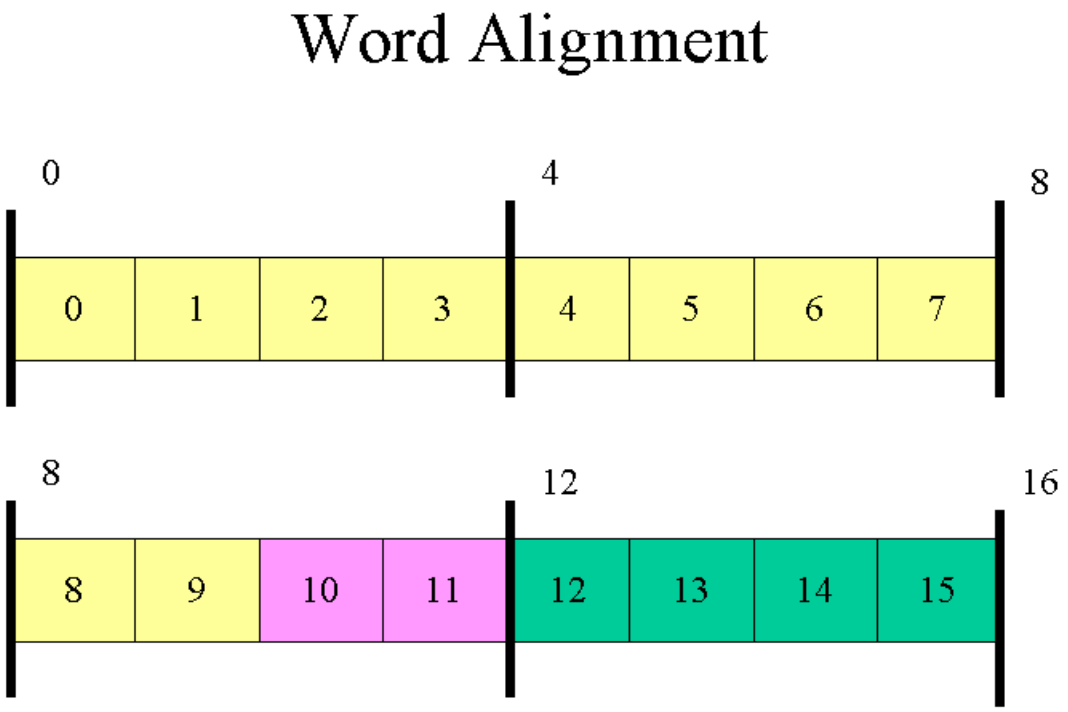


Figure 6.1 This figure shows a memory map of bytes 0 through 15 in computer with a 32 bit word (4 bytes). Each rectangle represents a single 8 bit byte; each of the heavy black vertical lines shows a word boundary. As in the example described in the text, bytes 0 through 9 (the first 10 bytes) represent our data. When the computer reads or writes the data, it moves bytes 0 through 11 in and out of the computer, because it will can only transfer memory at the word level.

Another consequence of this system is the way data is put into classes and structures. Most computers will not put a data item that requires 4 bytes of memory across a word boundary. The effects of this can be seen by checking the size of the data in a class or structure. Looking at the two structures in figures 6.3 and 6.4 below we can see how the arrangement of the data (in the structures in figure 6.2) can effect the size of each class.

Two classes, same data, different memory requirements

```
class myData0 {
    short integer height;
    long integer hairCount;
    long integer numFreckles;
    short integer weight;
}
```

```
class myData1 {
    short integer height;
    short integer weight;
    long integer hairCount;
    long integer numFreckles;
}
```

Figure 6.2. Two classes with same data content, but because the data is arranged differently, the class sizes will be different. This effect is caused by word alignment.

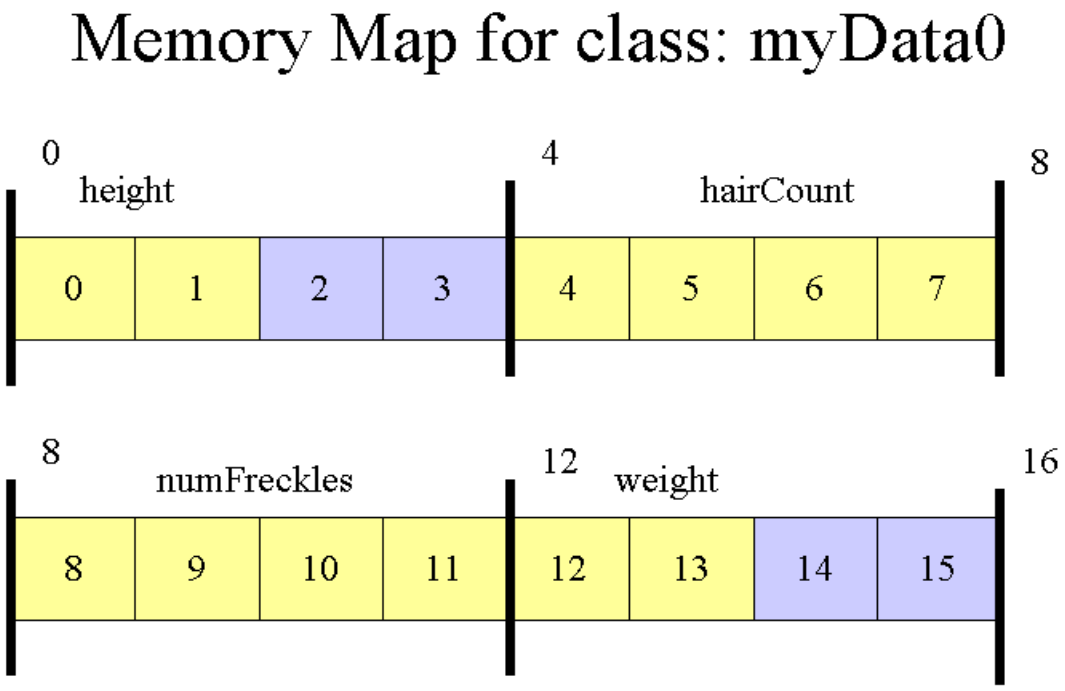


Figure 6.3. Data map for the myData0 class. The yellow block represents bytes that hold data. The light blue bytes represent bytes that are not used because the computer wants to start word length data items on a word boundary. Class size is 16 bytes.

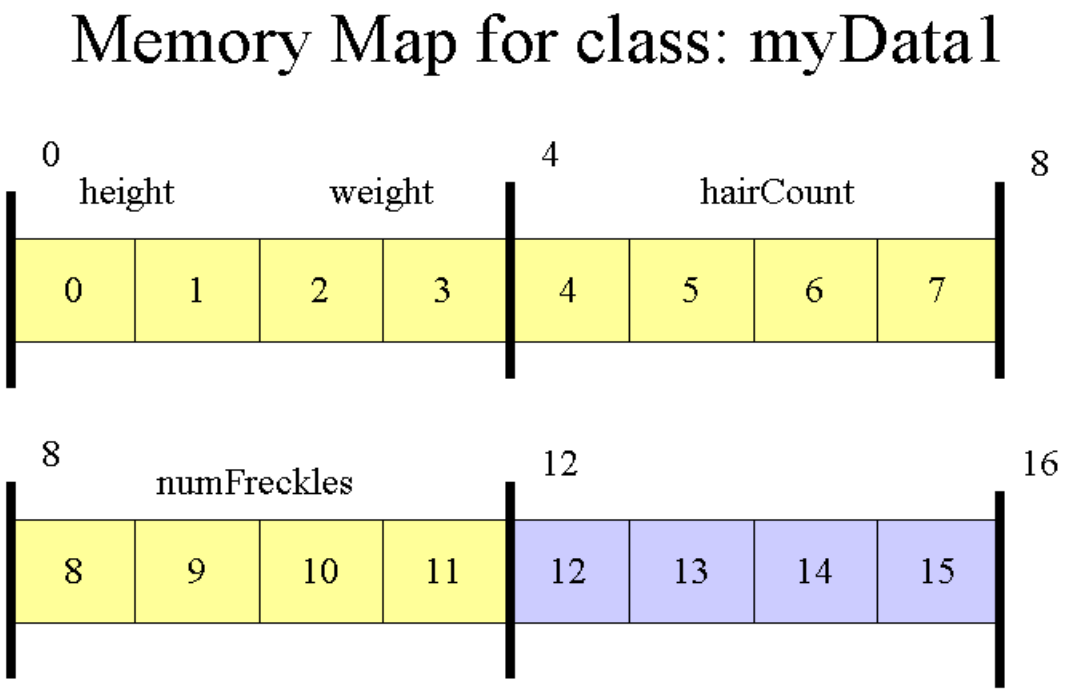


Figure 6.4. This figure illustrates how rearranging of the data in the class can save 4 bytes. Here the class size is 12 bytes versus 16 for the myData0 class as shown in figure 6.3. Once again yellow blocks represent bytes that are used by the class; blue blocks represent blocks not used by the structure; the heavy vertical lines represent word boundaries.

In summary, the computer system reads and writes data to memory in words or multiples of words. Each system has a word size; most often a modern PC will have a word size of 32 or 64 bits (4 or 8 bytes). The system does not like split a word across word boundaries, so if needed, it will move down to the next word boundary and start there.

For the most part this process is transparent. One of the times when it is not is when reading and writing formatted binary data. This type of data is often composed of data packets created by the data portion of a class or structure. In this case the structure can usually be read and written in binary with no consideration of the word boundaries, unless the reading and writing machines have different word sizes. Another time when it is not transparent is when reading and writing an indexed binary file such as that of a compressed image with a table of contents portion. In these cases, sometimes it is sometimes necessary for the program to "hop" from location to location in the file in order to retrieve the data of interest. In these cases, and others like it, being aware of word boundaries can be helpful.

As for common ways that memory is accessed; some systems are:

- byte addressable – all bytes are addressable
- word addressable – can only address on word boundaries
    - bus error – attempt to access memory on incorrect boundary.
    - How do we get around this? We use masking and shifting

6.2 Address Generation, Binding, and Binding Times

The process of address generation starts when the programmers are deciding which variables are needed to accomplish the task at hand. Generally, programmers use "symbolic addresses", they refer to the data that they working with by the names of the variables that they are using. The compiler uses the variable names to creates a symbol table which eventually becomes the variable section (data section) of the executable program. At this point in the process the variable names are no longer being used; what is being used is the numerical position of the space that represents the variable in the programs data section. Figure 6.5 below illustrates the progression from symbolic address to an address in memory of some variables in a program.

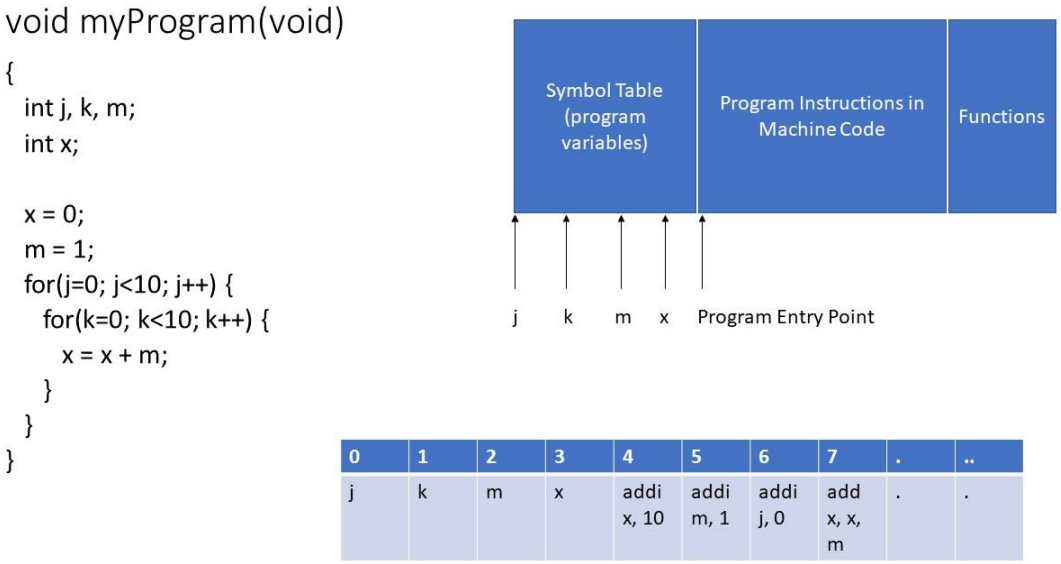


Figure 6.5 shows a simple program on the left side of the figure. The compiler/assembly process converts the high-level code to object that resembles the diagram in the upper right portion of the figure. Note the distinct regions of the object code. The diagram in lower right of the figure shows the same code, this time it appears as an array of data and instructions. The numbered elements from 0 to 3 house the programs variables, while the numbers after that are the instructions.

The diagram in the lower right of figure 6.5 above illustrates the compiled program in a fairly realistic manner (the assembly is MIPS – ish), as an array of data and instructions. The real difference is that at locations 0 to 3, the value of the variables j, k, m, and x are held. Also, in the instructions, instead of referring to the variables by their names, they would be referred to by their address. Table 6.6 below illustrates this point.

Address	***Assembly using Symbolic addresses***	***Assembly using logical addresses***
0	j	Value of j
1	k	Value of k
2	m	Value of m
3	x	Value of x
4	addi x, 10	addi 3, 10
5	addi m, 1	addi 2, 1

6	addi j, 0	addi 0, 0
7	add x, x, m	add 3, 3, 2
8	…..	…..

Table 6.6 shows assembly from figure 6.5 using symbolic address (the middle column) and using the logical addresses from the program's symbol table.

Looking at the table at address 4, the symbolic instruction is:

addi x, 10

This corresponds to:

x = 10;

The "addi" mnemonic stands for add immediate, and it is used to place a 10 in the location corresponding to x.

Looking at the column using "logical addresses" the instruction is:

addi 3, 10

Here the x is replaced by the 3. The 3 is the address or of the variable x. In this case if we were to say the program was in an array program[3] would hold the value of the variable x.

So, in summary, the compiler converts the high-level code to assembly, the linker switches the code to machine language. Along the way, the symbolic addresses get converted to their logical locations in executable program. We use the term "logical" because the programs symbol table starts from address 0 and increments from there. Eventually we will convert these logical addresses to physical addresses, meaning that instead of starting at 0, the addresses will start from wherever the program is loaded in memory. It will certainly not be address 0, because that is the beginning of the OS.

Several processes help to convert the high-level program to something that will run in the computer and along the way, addresses are translated from symbolic to various other forms such as relocatable and physical. The process of address generation is described in figure 6.7 below.

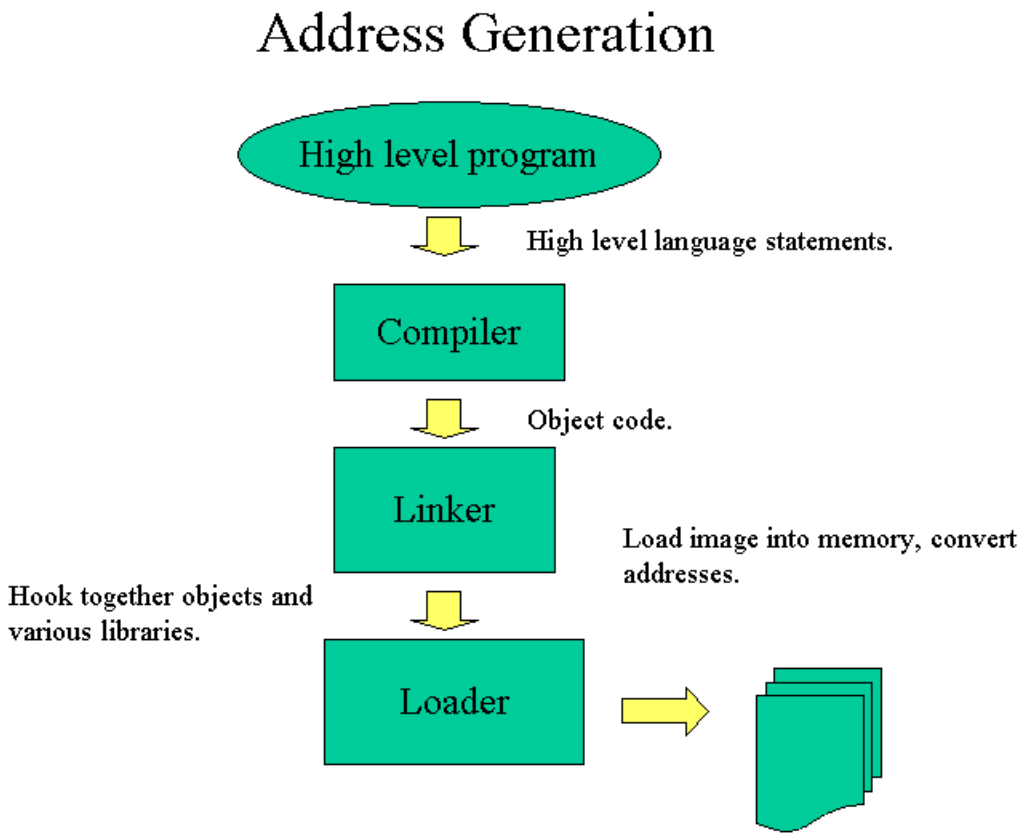


Figure 6.7 shows the process of converting a high-level program to form that can run in the computer. Along the way addresses are converted from symbolic to relocatable to physical. The functions of the compiler, linker, and loader are discussed in chapters 1 and 2.

6.21 Address Types

There are several types of addresses, some of which have already been mentioned. Logical Addresses are used by the CPU and start at address 0. These are generated by the compiler and are converted to other types of addresses during the process of creating a running process and loading that process into memory. Logical addresses are commonly generated by the compiler, and for the most part, while a process is running, it runs using logical addresses. These addresses are converted to physical addresses by the system as the process runs.

If the location that the process will reside at while it is running is known, then the compiler/linker/loader combination can generate physical addresses, negating the need for logical to physical address conversion. This is not so common in modern times, but in the early days of computing it was common. Batch systems used a single user partition with a known, fixed, starting point. This would allow the addresses of all user programs to be generated and saved so that the program would work in this partition. When using an IDE to develop code for a microprocessor, such as an Arduino, physical addresses can be used, because for many of these small processors, the programs they run are located in a user code area whose starting address is always the same; this is in essence very similar to the batch system.

Most of the time the location in which a program is going to reside while it is running is not known. In fact, sometimes, the process may start running at one location, get pre-empted by other processes, then resume running at different location. This mode of operation requires that addresses be relocatable. Relocatable addresses are formed by using logical addresses and a base address. The logical addresses are as discussed earlier in that the addresses are calculated assuming the entry point of the program is at location 0. Once the base address is known, it is added to the logical address in order to form the physical address. We saw this earlier when working with the batch system, and the multi-programmed batch system. Physical addresses are formed from relocatable addresses as follows:

Physical address = base + logical;

Or

Physical address = base + offset;

Here the base is the beginning of the program/processes partition. This partition can be either a fixed location partition as was introduced in chapter 2, or a dynamic partition, which will be covered in the upcoming sections. Many times, the logical address is referred to as the offset; it called that because that address is thought of as an offset from the beginning of the program.

Another address variable that goes along with base and offset is the limit variable. The limit variable reflects the maximum address value that is within the running process's program segment. Programs are not/should not be allowed to access memory that is outside of their program segment. Doing so could endanger the stability of the system. Imagine if programs could freely write data into any place in memory. That would mean that they could write over other processes, or even the OS. Actually, this kind of activity was very common when using older MS-DOS operating systems on the PC's of the 1980's and early 1990's. It was very useful to be able to access anything on the system, but it also causes many problems when it was done by accident. It was not uncommon for a user program to crash the entire system.

Because logical addresses need to be converted to physical addresses for nearly every instruction executed, there needs to be a way to do this quickly. Most computer/OS system have hardware support for this activity. Figure 6.8 below illustrates the process.

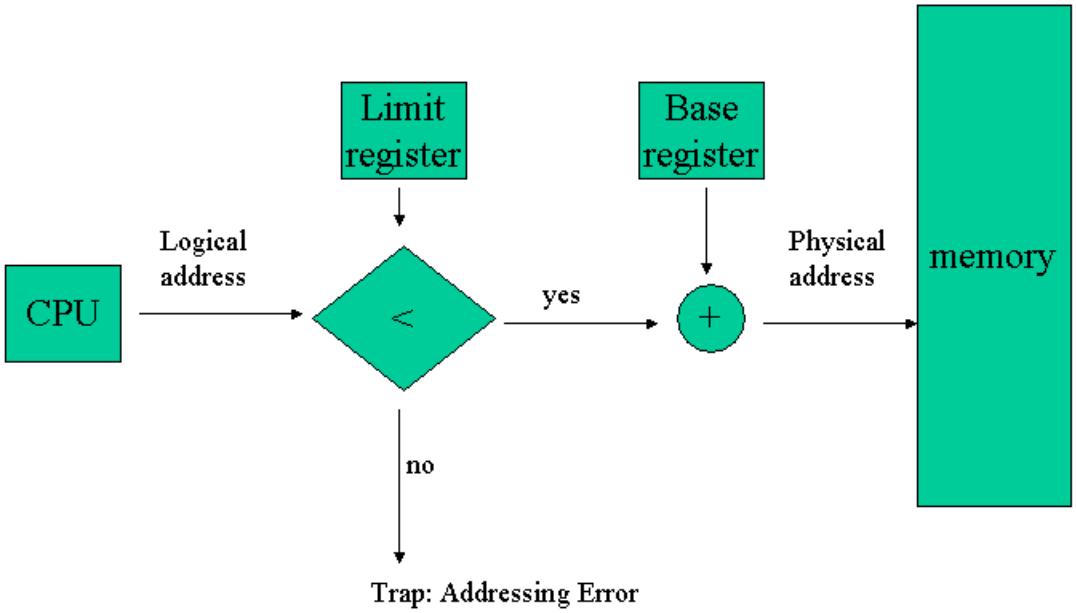


Figure 6.8 shows the process of converting logical addresses to physical addresses using the base/limit system. If the logical address is not inside the allowed address space, as regulated by the limit registers value, an error generated and the program is terminated. If the logical address is in bounds, the value of the base register is added to it forming a physical address.

Think of this example. Suppose are using multi-programmed batch system with 8 partitions, and we have compiled our program and it is ready to run. If partition 5 is available, we set the base register to the address of partition 5, and the limit register to value of the last address in the program (we could set it to the last address in the partition) and we let it run. Later on our program can run again in some other partition, all that needs to be done is load it in the partition, and set the base and limit to the appropriate values. Using this system, processes can run in any memory area without any programmatic changes.

Looking back at the example illustrated in table 6.6, we need to add the base address to addresses that reference variables. For example the instruction:

addi 3, 10

Would functionally become:

addi base + 3, 10

6.22 More on Addressing and Related Topics

In this section we try to tie together and summarize the information from the previous sections, and introduce a few more terms and concepts.

Main memory can be thought of as a large array of words, each with its own address.

Memory size basics

- Bit – one unit (like a switch).
- Nibble – 4 bits
- Byte – 8 bits
- Word – 8, 16, 32, 64 bits

Memory addressing

The smallest piece of space that a CPU can usually address is a byte. A word, is a machine dependent number of bytes. A very common older word size is 4 bytes, but modern computers have 8 or even 16 bit words. Some CPUs address on word or half word boundaries. Access to and from memory is handled by the memory unit, sometimes referred to as the MMU (Memory Management Unit).

Types of addressability include:

- Byte addressable – all bytes are addressable.
- Word addressable – can only address on word boundaries.

A bus error is an attempt to access memory on incorrect boundary. How do we get around this? We use masking and shifting.

Address types

- Symbolic – address in a program, like a variable name (count, numDats, etc.)
- Relocatable – These addresses are calculated from a reference point, like the beginning of the program. The compiler will generate these at compile time.
    - offset – is position of address calculated from program start.
    - Physical Address = Program Start address + offset;
- Absolute/physical/unrelocatable – These are physical addresses in the machines main memory.

Address Binding and Binding Time

Address binding is a mapping from one address space to another, commonly logical to physical. It can occur at different times. They can be:

- Compile time – Usually the compiler turns symbolic addresses into relocatable addresses, but if it is known ahead of time where a process is to reside in memory, physical address can be generated.

- Load time – If at compile time it is not known where the process will physically reside in memory the compiler generates relocatable code. The loader converts this to absolute addresses when the program is loaded into physical memory.

- Execution time – When processes are moved around during their execution, final binding is saved for when they run. This binding is done by a memory management unit (MMU).

Addressing During Execution

In modern systems, while a process is running, from the CPU's and process's perspective, all addresses that are generated (incrementation of the PC, and from various jump calls) are relative to the beginning of the program, which is a logical address of 0. All of these addresses, called Logical Addresses are translated to the computers physical address space. Said another way, the CPU generates logical addresses (sometimes called a virtual addresses) and memory management unit, MMU, turns the logical addresses into physical addresses. We use an MMU because this translation happens very often; so in order to speed up the process (and not burden the CPU) we have a dedicated piece of hardware for the purpose. In summary:

- The set of all addresses generated by a program/process is called the logical address space. The hardware memory addresses corresponding to these addresses are referred to as physical address space.
- Compile time binding and load time binding result in logical addresses that are the same as physical addresses.
- Execution time binding create logical and physical addresses are different from one another.
- Address translation handled by the MMU.
    - Physical Address = Logical + base register (sometimes called relocation register)
    - The program only works with logical addresses (0 to program max address). They are mapped to R + max program address in physical memory, where R is the value for the base register.
    - Logical to Physical address mapping is the central concept in memory management.

Linking types

There are two commonly used ways to link in commonly used functions. They are called Static and Dynamic. The two methods are detailed below:

- Static – The library routines are included in code. This means that routines such as square roots, window display routines, all commonly used routines are linked into the code; i.e. they are present in the program segment. This makes for code that can easily be transferred from like machine to like OS machine. Host machines do not have to rely on libraries or environments being loaded because the executable takes all of those routines and functions with it. The downside is

that the code can take up a lot of space and similar programs on the same machine will have duplicate code.

- Dynamic – The Libraries are linked in at runtime. A Dynamic Linked Library (DLL) is located by looking at a "stub" inserted into the executable at compile time. The stub tells the OS where to find the needed DLL. One thing to note is that the OS will almost always look in the current directory for the DLL; if not found there it looks at various other repositories for DLL's; many times these repositories are pointed to by environment variables. Using DLLs saves memory, but is more complex, takes longer to come up initially, and is not as portable. However, Dynamic Linked Libraries are widely supported. Linking of system libraries is postponed until run time; if the needed DLL is not present on the host system, an error is reported. As stated earlier, a stub is included in the binary image that indicates how to locate or load the appropriate routine if the routines are not already present.
    - Under this scheme all processes that use the DLLs execute only one copy of the code.
    - Advantages include:
        - Code savings. For example, all programs that display a window frame can share the same code. The data size and location data sent to the code is most all of the time different, but the code that creates the window is the same, creating a huge saving in code size when viewed from a system wide perspective.
        - Along the same lines as the above bullet, changes in system properties can be easily executed, just by selecting a different DLL. Various themes, etc. can be executed system wide by using different DLL's. Note, this is one way that these changes can be made, it does not have to be done this way.
        - Easier to implement updates and bug fixes throughout the system. (saves re-linking of all programs using new libraries).
    - Usually requires OS support.

Dynamic Loading and Overlays

In some cases (many in fact, especially in the days when memory prices were at a premium) the entire program may not fit in physical memory. If nothing is done about this, process sizes are limited to the size of memory. A work around this is Dynamic Loading. In this technique, routines are kept on disk in relocatable format. When a program starts, its main routine is loaded to memory. When a function is called, the calling routine checks for its presence in memory; if there, control is transfered to it, otherwise, a relocatable linking loader is called and the routine is loaded from disk. This is good when a program contains large amounts of code that are rarely called. It does not require support from OS, but puts the responsibility on programmers to design and use this scheme. The OS provides library routines in support of dynamic loading sometimes.

A similar method to Dynamic Loading is called Overlays. Overlays are a programmer implemented technique of executing code that is bigger than the physical memory. In

this method the programmer divides the code into sections that will fit into memory. When the program has to switch code sections an Overlay driver swaps in the needed part of the code and restarts the program.

Example: A 2 pass assembler that requires 200k memory, may not fit into memory. If only 150 k is available, the overlays can be designed with the following memory requirements.

- Pass 1 – 70k
- Pass 2 – 80k
- Symbol table – 20k
- Common routines – 30k
- Overlay driver – 10k

Pass 1 with needed routines: 70k + 20k + 30k = 120k

With overlay driver: 120k + 10k = 130k

Pass 2 with needed routines: 80k + 20k + 30k = 130k
With overlay driver: 130k + 10k = 140k

So, the assembler can be run with 150k of memory. This scheme does not require special OS support but does require the programmer to have a good knowledge of the program's structure.

Processes are memory resident while they are running. If there are more processes than the memory can hold, then the OS swaps them in and out of memory from disk so they can run. When address binding is done at compile or load time, the process must reside at a specific place in physical memory to run, and must be swapped back and forth from disk to that exact place or any programmatically generated addresses (such as those generated by jump calls) will be inaccurate. This requirement is relaxed with execution time binding.

As an extra note, processes that are swapped from memory to disk and vice versa are tracked by the OS, using the PCB. Swapping takes time, so the less times a process is swapped the better. The time is proportional to the amount of data that must be moved from disk to memory.

Example: A 100k process can be moved at a transfer rate of 1000k per second with an 8-msec. disk latency time would take:

100k/1000k + 8 ms = 108ms.

To swap whatever was in memory out and this process back in would take:

2 * 108ms = 216ms.

For efficiency reasons, memory resident running time should be longer than swapping time. Remembering back to the process scheduling algorithms, if processes are constantly undergoing context switches, and having to be moved to and from the disk, this swapping time must be added into the job's overall turnaround time.

6.3 Contiguous Memory

There are two main memory management strategies, contiguous and paged. Contiguous memory schemes have their advantages and disadvantages. The main advantage is that they are simple to understand and relatively easy to implement. Also, because of their simplicity, the software/hardware requirements needed to implement them run efficiently. The downside is that they waste memory in the form of fragmentation. Because of this, modern systems use paged memory. Paged memory is more complicated and requires more calculations to resolve addresses, but because of its structure, it eliminates almost all fragmentation. In this section we will review the basic concepts of contiguous memory systems; we will relate these concepts to the work done in the simulator in chapter 2, and will extend the memory systems of the simulator in order to reduce fragmentation.

Contiguous, in reference to memory, means that the memory is sequential and uninterrupted. In a computer with contiguous memory system, often times the OS would occupy the lower part of the memory, and user programs the upper portion. For example if a system had 1 megabyte of memory and the OS needed 256k, the address space for the OS would be 0 through 256k – 1, and the space for the user processes would be from the 256k boundary up to 1 megabyte – 1.

The types of contiguous memory systems that will addressed are:

- Single Partition
- Multiple Fixed Partitions
- Dynamic Partitions

Single Partition Batch

In a Single Partition the OS resides in low memory. User processes reside in high memory. Figure 6.9 below shows how memory is laid out in a single partition system.

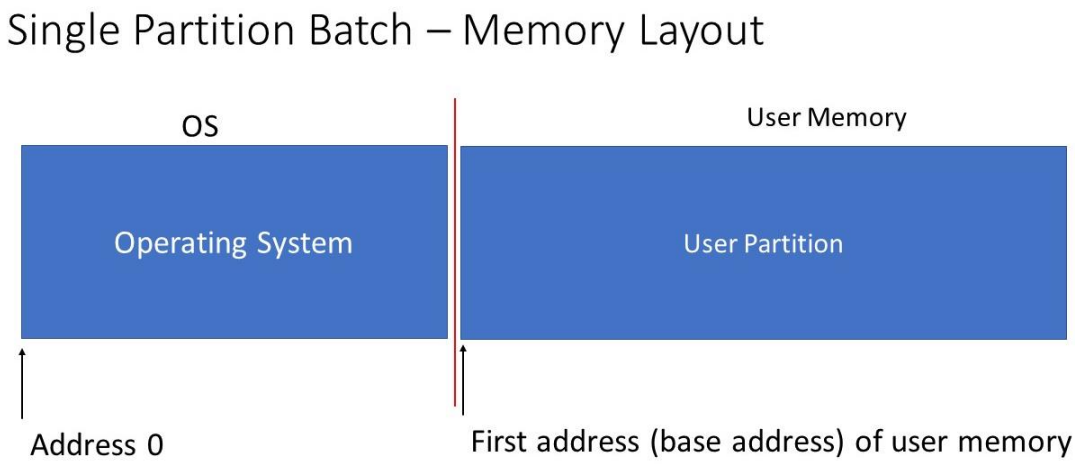


Figure 6.9 – This figure shows the memory layout of a single partition batch system. The OS occupies the low memory addresses, and the user programs reside in the user partition, starting at its base address.

Some features of the single partition system are:

- Processes and OS are protected from each other by using the base and limit register scheme, as detailed in figure 6.8.
- Every address the CPU generates is checked against these registers. Hardware is used so it can be done quickly.
- It provides OS and user programs protection from running processes.

Multiple Fixed Partitions

In the most basic case of multiple partition memory management memory is divided into "n" equal size partitions (IBM 360 used this scheme). The number of partitions in the memory determines the degree of multiprogramming. When a partition is free, the OS selects a process from the input queue and loads it into that partition. Figure 6.10 below illustrates a system with 8 fixed partitions.

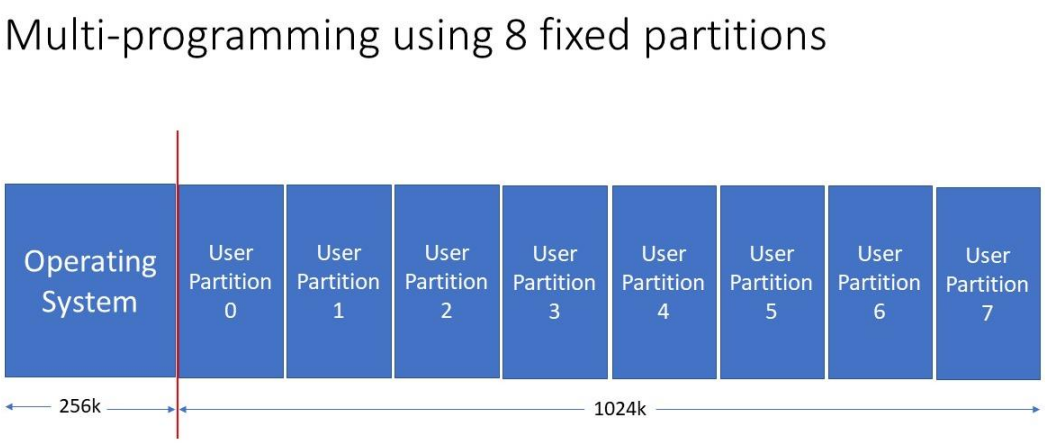


Figure 6.10 illustrates a system in which user memory is divided into 8 equally sized fixed partitions. The OS occupies "low memory", 0 to 256k, while user memory is divided into 8 128k partitions.

Looking back to chapter 2, section 4, the simulator that was developed supported multiple fixed partitions. It used an array of Booleans to track whether a partition was occupied or not. It also used first come first served job scheduling, and by default each job was put into the first available fixed partition.

The multiple fixed partition system is not ideal in terms of memory usage because:

- The maximum number of processes that can concurrently run is equal to the number fixed partitions.
- If a process is much smaller than the partition it occupies, a large chunk of memory is wasted.
- If a process is larger than a partition, it cannot be run. Even if other partitions are available, and even if the available partitions are adjacent, the process cannot be run.

With these limitations, it would seem that fixed partitions are useless. However, there are several good features of this system. Some are listed below:

- Supports multi-programming, thus reducing CPU idle time.
- Supports time-sharing, thus providing an avenue to user/computer interaction resulting in increased user productivity.
- It is a logical extension of the simple batch system, meaning that it is simple to implement. It could be ideal for a small microprocessor based system that does not have lots of spare memory for the OS to support the more complex memory management systems.

Dynamic Partitions

Later, the equal number of partitions was dropped for a dynamic system in which memory is kept track of by using a table that tells which part of memory is empty and which is occupied.

Initially all user process memory is empty. As processes are scheduled, they are moved into "holes" (areas of unoccupied memory). The OS finds a hole large enough to hold the process. Memory is allocated until there is no hole large enough to hold the next process. The OS then must wait until a process completes and frees its memory or skip down the input queue until it finds a process small enough to fit in one of the holes. Adjacent holes can be merged to form larger holes. Figure 6.11 below shows a memory map for a small system with 32k for the OS and 64k for user processes.

Processes:

Process	Size in Kbytes
OS	32
A	25
B	45
C	10
D	30
E	15
F	5

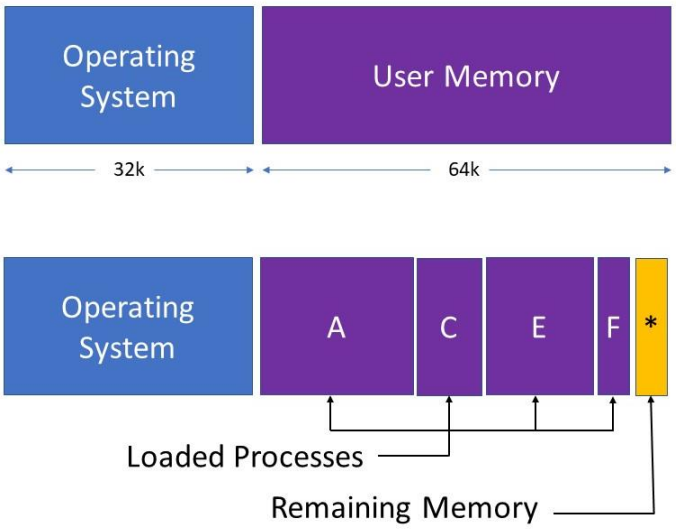


Figure 6.11 shows the memory map for a small system that has 32k for the OS and 64k for the user processes. The table on the left shows the OS and process sizes. The diagram in the upper shows memory before processes are loaded. The diagram on the lower right shows memory after the first group of processes is loaded. Note process B cannot be loaded while process A is in memory because it would not fit; 25 + 45 > 64. So process B will have to wait until there is room. The loaded process in the diagram in the lower right occupy 55k of memory, leaving 9k free. No job in the que can fit in that 9k so the OS must wait for one of the resident jobs to complete and release its memory.

Looking at the figure in 6.11, it can be seen that the OS loaded processes A, C, E, and F. It could not load B or D because there was not enough memory. Once some of the resident process complete and release their memory, other process can be loaded into memory. Which process gets selected to be loaded depends on the allocation scheme that the OS is using.

There are several allocation schemes:

- First-Fit – Put the process in the first hole found that will accommodate its memory requirements. Searching stops as soon as a hole is found that works.
- Best-Fit – Put the process in to the smallest hole that will accommodate its memory requirements. This strategy produces the smallest remaining hole/holes.
- Worst-Fit – Put the process in to the largest hole that will accommodate its memory requirements. This strategy produces the largest remaining hole/holes.

Simulations show that first-fit and best-fit work best in terms of decreasing time and storage utilization. Neither is clearly better, but first-fit is generally faster.

One thing that must be considered, there is a potential conflict that exists between the job scheduler and the memory scheduler. If the job scheduler and the memory scheduler are both set up to look at jobs as they are submitted, then conflicts can arise. For example, if a job is short, but requires more memory than is available, and a shortest job first algorithm is guiding the short-term scheduler, it could that that particular job is not in

memory. One way to resolve the conflict is to require that the job scheduler look at the memory resident jobs only when selecting a job to put into the run queue.

Internal and External Fragmentation

External Fragmentation is when the sum of the memory of the holes is enough to satisfy a memory request but it not contiguous, it is fragmented into a large number of small holes. Figure 6.12 illustrates the situation.

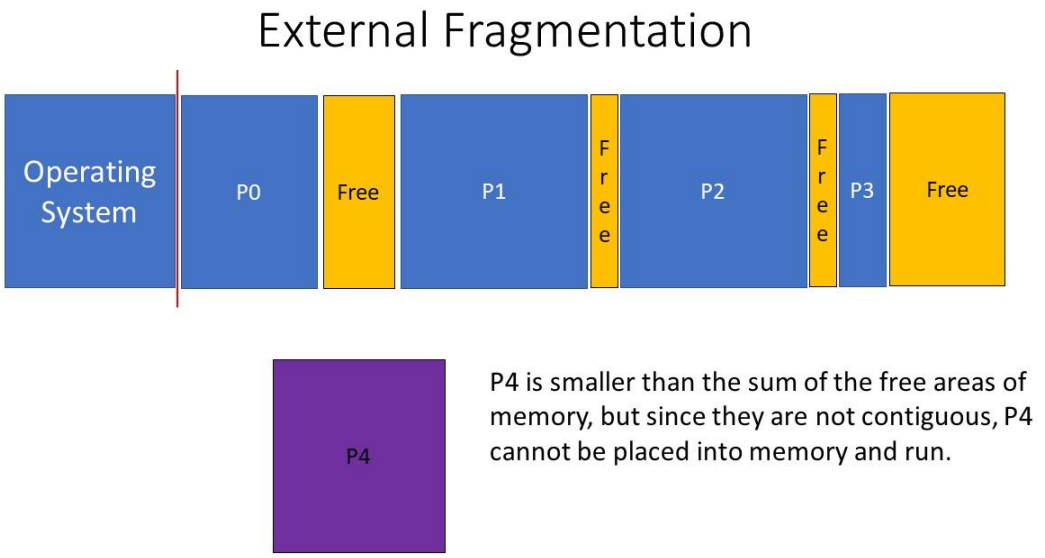


Figure 6.12 illustrates a situation of external fragmentation. The yellow free areas hold more than enough memory to accommodate process P4, but they are not contiguous.

To solve this problem (can waste up to a third of the memory) compaction can be used. Compaction algorithms periodically shuffle the memory and create a big block of free memory. Figure 6.13 illustrates the concept.

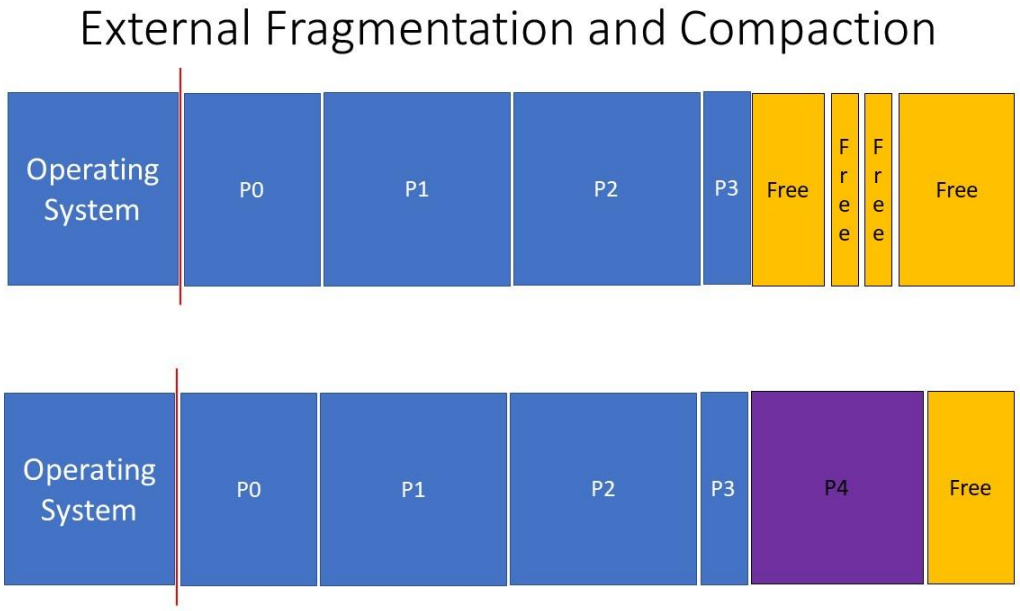


Figure 6.13 continues the ideas from figure 6.12. In the upper diagram, the free memory blocks have been aggregated into one large block (compaction). The lower diagram shows that process P4 can now be loaded into memory.

Sometimes a hole may be barely larger than the process it is allocated to. In order to avoid the overhead of tracking very small holes (the memory required to track the tiny hole could be more than the hole itself) more memory than is required would be allocated so that the tracking problem can be lessened (allocate the tiny hole as part of the larger request). The memory that is the difference between the hole size and process size is referred to as internal fragmentation. See figure 6.14 below.

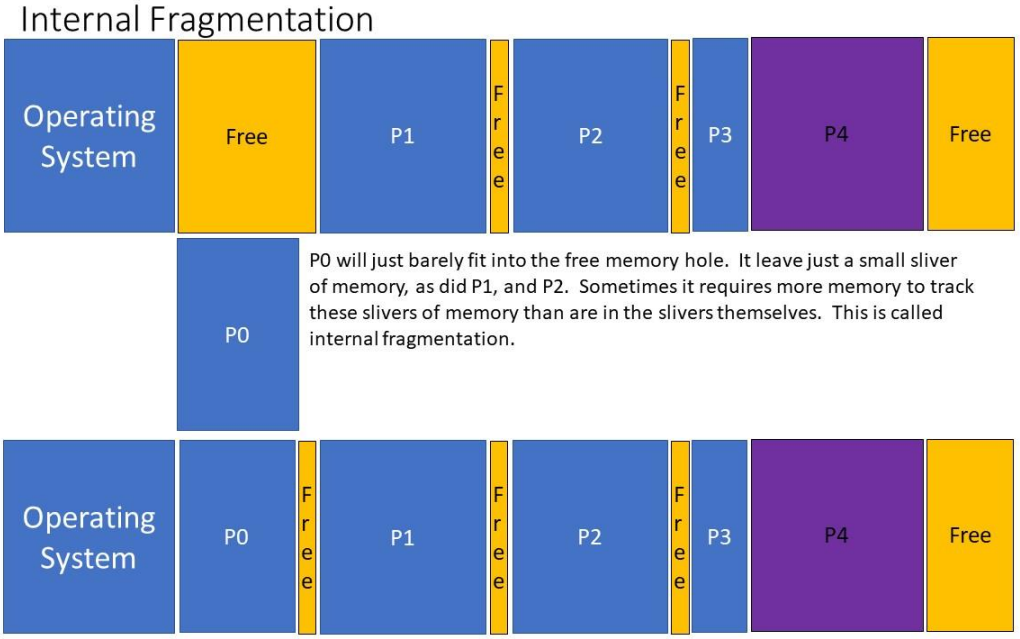


Figure 6.14 shows an example of internal fragmentation. P0 is to be put into the free memory block, leaving just a small slice of leftover memory.

In figure 6.14 above, the upper diagram shows a large piece of memory that is available immediately before process P_1. P_0 will just fit into that memory slot. Once it is loaded, see the lower diagram in the figure, a small piece of memory remains. If the piece of memory is 32 bytes, but in order to track its location and size takes 64 bytes, there is not really any point in tracking it. Main memory can become riddled with these untracked pieces of memory. This called internal fragmentation.

6.4 Paging and Virtual Memory

To alleviate the fragmentation problem, paged memory systems can be used. Paging provides a method of linking together small pieces of memory, so that the requirement of memory being contiguous is no longer needed. In doing so, external fragmentation is eliminated, and internal fragmentation is reduced to occurring in only the last page of a process.

In paged memory systems, memory is divided into "n" equal sized memory pages. A process can occupy pages that are spread throughout the system by using a page table. From the process's viewpoint, it lives in a piece of contiguous memory. A data structure called a page table is used to map the contiguous "logical" memory of the process to pages in the page table, and ultimately to physical memory frames. Figure 6.15 below shows a conceptual model of paging.

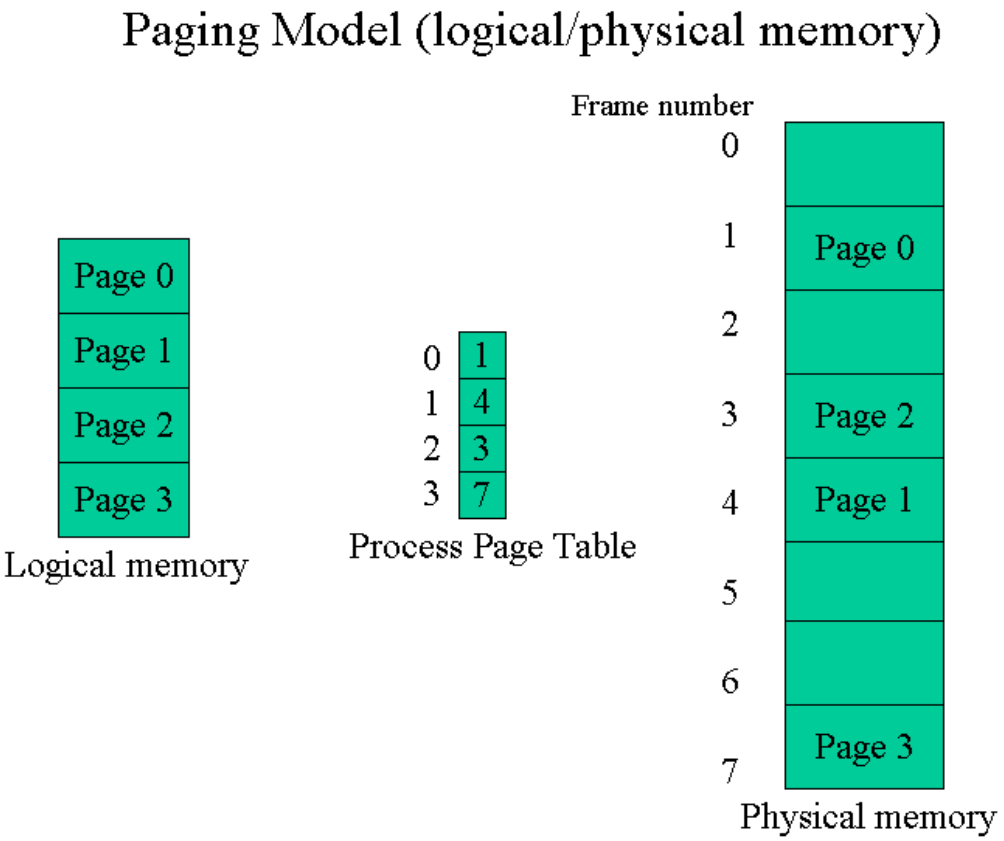


Figure 6.15 illustrates a conceptual model of paging. The process is in logical memory, as shown on the left of the diagram. Its page table, shown in the center, maps the logical pages to physical frames in physical memory, as shown on the right side of the diagram.

Looking at the diagram in figure 6.15, notice that the process is actually spread throughout physical memory. The CPU and process are not aware of it, because they are operating with logical addresses. The process's page table, along with the systems other hardware make the translation from logical addresses to physical, in a manner transparent to the CPU and the process.

Figure 6.16 provides another example of how paging works. The example is purely for illustration as the page sizes are very small at just 4 bytes. In the diagram there is a 16 byte process in logical memory. The numbering in the process and the page table starts at 1, this is for clarity, normally it would start at 0. In fact, as will be seen from the paging equations in later paragraphs, it must start at 0.

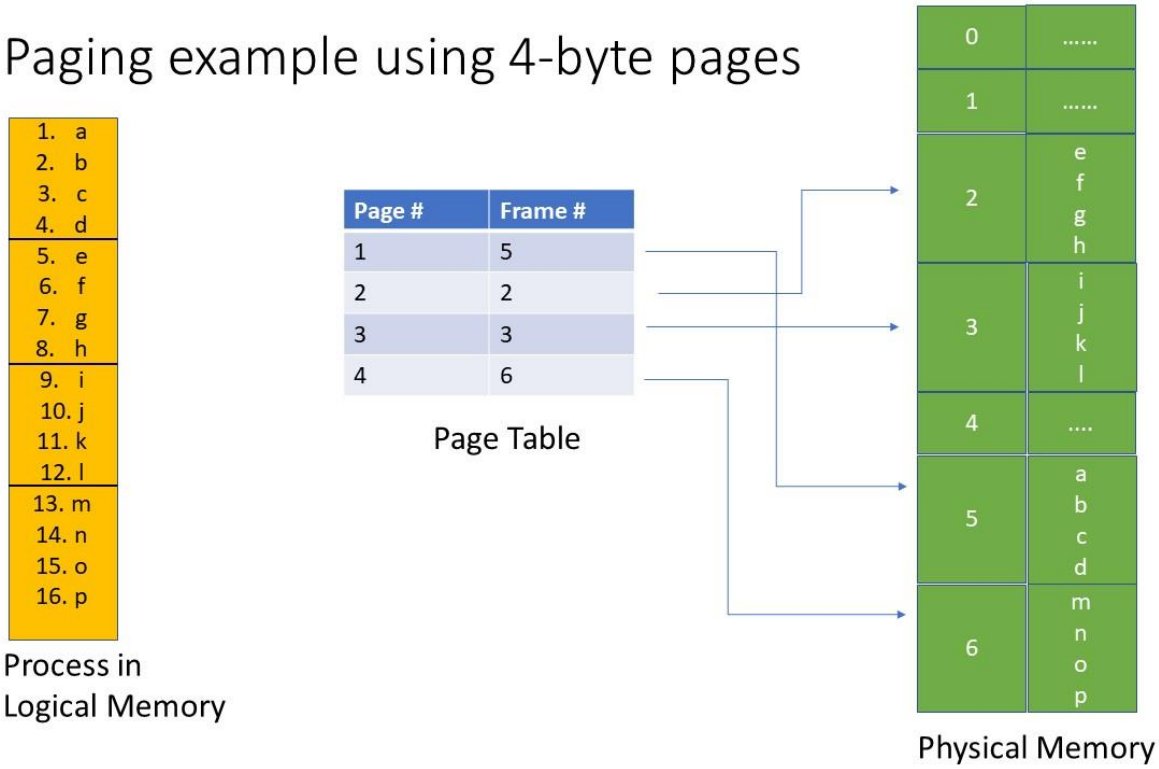


Figure 6.16 shows a process in logical memory that is mapped to physical memory using its paging table. The logical memory, on the left, is divided into sequential pages. The page table in the middle matches memory frames to the pages. The physical memory on the right, is divided into frames, which are the same size as the pages.

6.41 Paging Basics

To see how this works, consider the process and the page table that maps it to physical memory.

- A process lives in logical memory which is contiguous. The first logical address of every process is 0.
- Logical memory is divided into equally sized pages. Page size is a compromise; a very small page size will reduce internal fragmentation in the last page of the

process, but will increase the page table size. A large page size does the opposite; it increases internal fragmentation, but it reduces page table size.

- Physical memory is divided into "frames". Frames and pages are the same size. The page table maps logical memory in the form of pages to physical memory in the form of frames.

Logical addresses from the process are mapped to physical addresses in physical memory using the following equations:

page size = frame size

page number = integer(logical address / pageSize)

frame number = pageTable[page number]

offset = logical address – (page number * page size)

physical address = (frame number * frame size) + offset

Page sizes are usually sized to be a power of 2, typically between 512 bytes and 16 megs. This makes page and offset calculation a matter of shifting bits. Figure 6.17 below shows the composition of a logical address. If there are "m" total address bits, shifting n bits to the right (and dragging in 0's) results in the page number. The offset can be found by using the equation above or by a combination of (m -n) left shifts and then (m – n) right shifts.

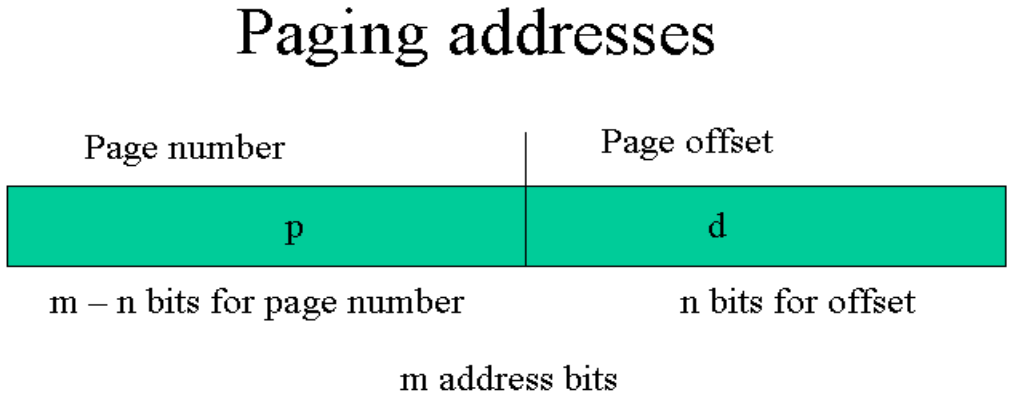


Figure 6.17 shows a logical address. The bits to the left of the divide are the page number; to the right is the offset.

Suppose the PC (program counter) is pointing at a logical instruction in binary at:

PC = 0101 00010001

If our memory is dived into 16 pages, each 256 bits in size, what is the page and offset?

PC = 0101 00010001

Here, the least significant 16 bits is the offset, as shown by the magenta colored text, while the page number is most significant 4 bits, as shown by the yellow bits.

This means that the current instruction resides in the 5th memory page at location 33. The details of the page number are below:

0101 = 0 + 4 + 0 + 1 = 5

and the offset follows:

00010001 = 0 + 0 + 0 + 32 + 0 + 0 + 0 + 1 = 33

One thing to remember is that each and every time an address is generated, it must be converted from a logical address to a physical address. This takes time, so modern systems have paging hardware to speed up the process, and to take the burden off the CPU. The hardware may consist of special bit shifting registers, and special memory to house the paging tables. Since the paging tables are consulted for every address used, instead of just one memory access per instruction executed, it now takes 2; one for the page table, and the second to retrieve the instruction. Figure 6.18 below illustrates the interaction of the CPU, paging hardware, and memory.

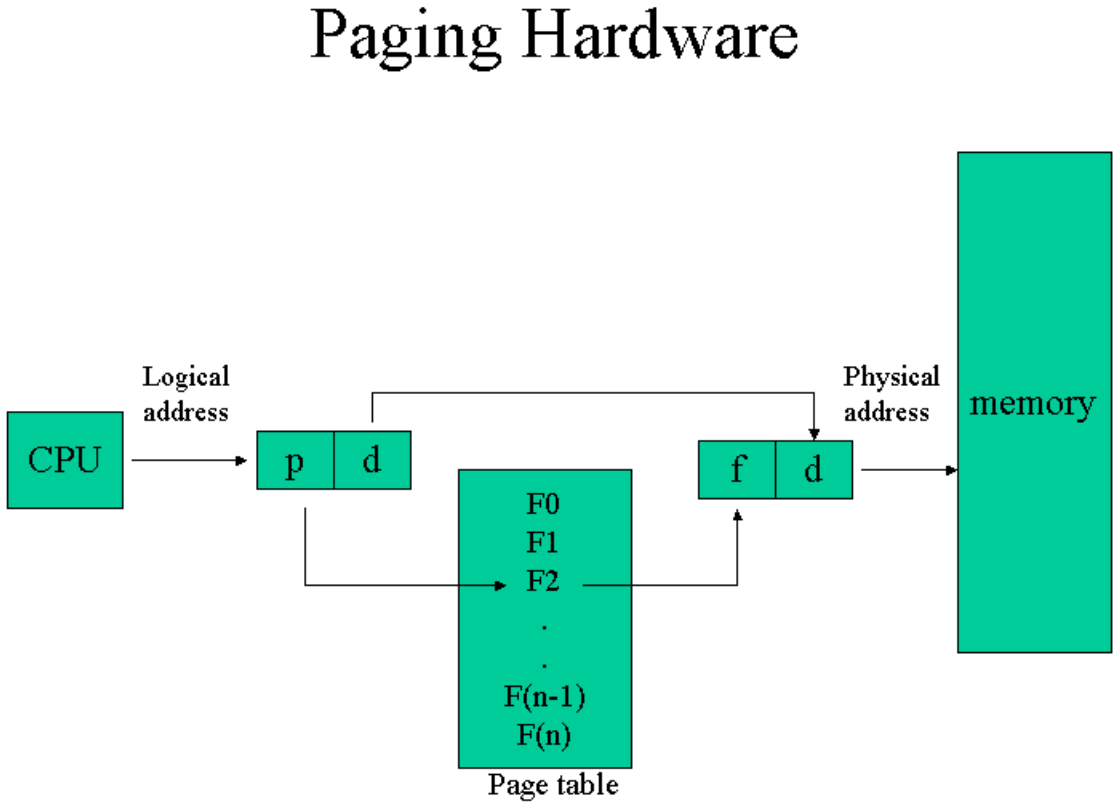


Figure 6.18 shows a logical address generating by the CPU being converted to a physical address. The address is separated into page and offset (sometime called displacement), the page number is used to retrieve the frame number from the process's page table. Then the frame number is combined with the offset (displacement) and sent down the main memory's address lines so that the memory at that location can be read (load),or written (store).

6.5 Summary of Paging Basics

Paging attributes:

- Paging is a form of dynamic relocation.
- Logical addresses are bound by the paging hardware and page table to physical addresses.
- No external fragmentation, internal fragmentation in last page of process.
- Any free frame can be allocated to any process that needs it.
- Page size is a compromise between wanting to minimize internal page fragmentation and minimizing the number of disk accesses, and the overhead required to store the paging tables.

6.6 Narrative of Paging Process

Narrative of Paging Process:

- Process arrives in system
    - Its size is expressed in pages.
    - If it requires "n" pages, there must be "n" frames available in memory.
- If the frames are available, they are allocated to the process, each frame location being recorded in the process's page table.
- User program views memory as one contiguous piece even though it is physically scattered throughout memory.
- As the program runs, logical addresses are converted to physical addresses.
- OS keeps track of which frames are used and which are available.
- When memory is full (page table full) pages must be swapped in and out from disk.

6.7 Issues with Paging

To access a byte of memory it takes 2 memory accesses, one for the page table, one for the access to memory. This takes time; twice as much time if nothing is done to remedy the situation. Translation look-aside buffers (TLB) are used to alleviate this problem. A TLB is a small fast piece of associative memory that holds part of the page table in it.

The key to this technique is the use of associative memory. Normal memory must be searched one element at time, comparing the contents of the memory position to what we are looking for. A match is found when they are the same. Associative memory is different in that it is "content addressable", meaning that the it is not referenced by location, but instead by what it holds. It works by comparing an input search tag (in this case the we are looking for a process's page table and an entry associated with the page that our instruction is on, so we the search tag presented to the associative memory would be the processes PID/page number. The associative memory would compare that PID to its tables of data and return the address and the data from appropriate page table. This type of memory is very fast compared to normal memory and is commonly used for high speed searching applications.

In the bullets below, the use of TLB are outlined:

- A Logical address generated by CPU.
- The page number is calculated and presented to the TLB. The process's page table is not guaranteed to be in the TLB.
- If the page table is present and the page number is found, the TLB outputs the matching frame number. The whole Process takes 10% longer than a direct memory reference.
- If page table and page number are not in the TLB, the page table is referenced in normal memory and the physical address is formed. The TLB can now be updated. If it is full, one of the less used entries is replaced.
- The percent of times that a page number is found in the TLB is called the hit ratio.

Below is an example that shows how the use of the TLB can increase OS efficiency.

Using the TLB hit ratio we can calculate the effective memory access time.

Assume:

80 percent hit ratio

20 nanosecond TLB search time

100 nanosecond memory access time.

Effective Access Time = page table access time + memory access time

Effective Access Time = .80 **(**20 + 100) + .20(20 + 100 + 100) = 140 nanoseconds

Think of the calculation like this, 80 percent of the time we get the paging data from the TLB and then the data (usually an instruction point to by the PC) from the memory, but 20 percent of the time it is not in the TLB, so we have to reference normal memory for the page table, and then get the our data from memory.

If we were to do this without the TLB, the page table data would have to always be accessed from normal memory, so the effective access time would be:

Effective Access Time = 100 + 100 = 200 nanoseconds

When using TLB's we can increase efficiency by improving the hit ratio, or by getting a faster TLB. If you cannot get a faster TLB, increasing its size will allow for holding more process's page tables, which would improve the hit ratio.

6.8 Locality of Reference and Paging Implementation Details

There are many important aspects regarding paging. One of the most important is the phenomena of "Locality of Reference". In a way, it is what makes paging work. To get a feel what locality of reference is, consider how a process accesses memory when it runs.

When a process runs, the program counter moves through the program in a sequential manner, unless a branching instruction is encountered. If the conditions are met from the branching instruction a jump is executed, meaning that the PC's value is changed to the entry point of a new function, is set the entry point of a loop, or is set to the entry point of a new block of code.

The above statements are actually only mostly true. Older languages and styles of programming implemented statements such as "goto xxx", where xxx was some address label in the program. Some of these older pieces of unstructured software where riddled with such statements, earning them the moniker "Spaghetti Code". Good software practices, structured programming, object oriented coding, etc., have mostly eliminated the unconditional branch statements (no more goto statements in modern code).

So, when considering how memory is accessed by a well written process, remember that the PC moves through the process's instructions sequentially, unless it hits a branching statement. Keeping this in mind, it can be said the PC either moves slowly through a region of memory and repeats itself (as in the case of a loop), or moves to a nearby region and moves through it slowly (as in an if – then – else statement, or a function call). We could say that the PC tends to mill around a certain local area of memory, before moving on. The end result is that because it is remaining in local regions of memory for extended periods of time, it tends to remain in the same memory page for and extended periods of time. This is a good thing because it lowers the number of times the OS has to go out to the disk to get a page that is not resident in memory. This is the subject of some of the latter sections of this chapter.

Another key concept that has been alluded to that of the Page Fault. A page fault is an event that occurs when a needed page is not in memory (RAM).

When a process is being loaded into memory, there are a few strategies that the OS can use. They are:

- The entire process can be loaded into memory.
- The main body of the process can be loaded to memory, and the rest can be left on disk. This is a handy way of doing things, especially when the program is so big it will not fit in memory. When a reference is made to a page that is not in memory, that is called a page fault. A page fault causes the OS to retrieve the needed page and place it into memory.
- Just the page of the code that has the process's entry point is loaded. Pages are then loaded when page faults occur; i.e. pages are only brought in when they are needed. This called "Demand Paging".

- If a page has been changed while in memory, it will need to be copied back to disk at some point. All pages can be written back to disk to make sure that the data on disk is current, but that is a waste of time if a page has not been changed. In order to be more efficient a "Dirty Bit" is used. If a change is made to a memory resident page, the dirty bit is set, indicating that the page will need to be written back to disk, otherwise, it is not copied back to disk when being replaced.

In summary, we put forward the following low-resolution definitions for locality of reference and the other afore mentioned terms:

Locality of reference – Programs tend to work in a particular memory area and move slowly through them.

Page Fault – Event that occurs when a page is not in RAM.

Demand Paging – Pages are only brought in when they are needed.

Pure Demand Paging – Initially all pages on backing store (fast disk or fast part of disk). On initial startup, or restarting after a context switch, many page faults occur.

Demand Paging with pre-paging – The compiler tells system which pages are needed for startup. After startup pages used can be swapped. This method helps to reduce the amount of page faults on startup.

Thrashing – When a process cannot be allocated the minimum number of pages it needs to run it can generate many page faults. At some point the OS is spending most of the CPU cycles servicing page faults, instead of executing user processes.

Dirty Bit – Indicates if the page has been changed since it came into memory. If it has, when it is ejected, it needs to be written back to disk before it is kicked out.

6.9 Page Replacement Algorithms

The previous section introduced the concept of page faults, an event that occurs when the OS's paging system determines that the address it is trying to access is in a page that is not currently in memory. If there is a free frame in physical memory the required page can be fetched from disk and loaded straight away. However, if all memory frames are occupied, then a page that is in a memory frame must be kicked out so that the new page with the currently needed data can be loaded into it. The relevant question to be asked is – "Which page should be replaced?". To answer this question, several page replacement algorithms will be reviewed, including the following:

- First in first out /FIFO
- The Optimal algorithm, sometimes referred to as Belady's algorithm.
- Various counting algorithms such as:
    - Least Recently Used / LRU
    - Least Frequently Used / LFU

6.91 Overview of the Algorithms – FIFO

The simplest strategy is to use is the first in first out strategy. The advantage of this strategy is that it is simple to understand and easy to implement; the disadvantage is that there is no consideration of which page is being used most or least by the system, or any other types of operational issues. Because of this, sometimes a page can be kicked out, only to be retrieved just a short time later.

For example, given the following page reference string:

{1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}

And a memory with 3 frames, determine how many page faults and page faults with replacement would occur, if the memory is initially empty. To solve this we use a tabular approach. The top row of the table will hold the page reference stream, and each column represents the which page numbers are currently in the memory frames. Table 6.19 below shows an example table using the FIFO algorithm. At the moment, the first 3 pages, {1, 2, and 3} have been placed in their memory frames, each time throwing a page fault, as seen in the "Fault" row. The blank column at the beginning of the page stream is there to indicate that the memory started off empty.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1									
Frame 1			2	2									
Frame 2				3									
Fault		f	f	f									

Table 6.19 shows the first 3 pages of the of the page stream using the FIFO algorithm. They are marked with the 'f' in the Fault row to indicate a page fault.

Table 6.20 shows the results of the FIFO algorithm when page 4 is referenced. Since page 4 is not currently resident (only pages 1, 2, and 3 are), a page fault is thrown – this time it is page fault with replacement. Since page 1 was the first to be placed in memory, it will be the first to be ejected.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4								
Frame 1			2	2	2								
Frame 2				3	3								
Fault		f	f	f	fr								

Table 6.20 shows the results of the FIFO algorithm when page 4 is referenced. Page 1 is ejected because it was the first one placed into memory. Also, notice the fault indicator is set to "fr", meaning "fault with replacement".

Table 6.21 shows the results of the FIFO algorithm for the next few page references. Notice that every time a page is kicked out, it is needed by the next page reference; a definite shortcoming of the FIFO algorithm.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5					
Frame 1			2	2	2	1	1	1					
Frame 2				3	3	3	2	2					
Fault		f	f	f	fr	fr	fr	fr					

Table 6.21 shows the results of the FIFO algorithm for the next few page references. For each refererence a page fault with replacement is generated.

Table 6.22 shows the completed solution. Notice that when the pages 1 and 2 were introduced, no page faults were thrown; the pages were already in memory frames, 1 and 2. Also, there was no page fault on the last page reference; page 5 was resident in frame 0. There was a total of 9 page faults, 6 with replacement.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	5	5	5
Frame 1			2	2	2	1	1	1	1	1	3	3	3
Frame 2				3	3	3	2	2	2	2	2	4	4
Fault		f	f	f	fr	fr	fr	fr			fr	fr	

Table 6.22 shows the final results of the FIFO algorithm for page reference stream {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5}.

Table 6.23 below shows the results of the FIFO algorithm, this time with an extra memory frame. Extra memory usually helps efficiency, meaning less page faults, but sometimes, as in the case, it does not. This is an example of "Belady's Anomaly", meaning more resources does not always improve results. Here there were 10 page faults, 6 with replacement.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1			5	5	5	5	4	4
Frame 1			2	2	2			2	1	1	1	1	5
Frame 2				3	3			3	3	2	2	2	2
Frame 3					4			4	4	4	3	3	3
Fault		f	f	f	f			fr	fr	fr	fr	fr	fr

Table 6.23 is an illustration of Belady's Anomaly, meaning that more resources (an extra memory frame) does not always guarantee better results.

6.92 The Optimal Algorithm / Belady's Algorithm

The Optimal algorithm looks at future page requests and in the page stream and selects the page used furthest in the future as the page that will be ejected. This algorithm is provably optimal. The problem with algorithm is that it is hard to predict the future, so it is used more as a baseline for other algorithms. Table 6.24 below shows the results of the algorithm using 3 frames and same page stream as previously used. Here the first 4 page references are shown. The first 3 generate faults without replacement, just as FIFO did, while the 4th page reference will generate a fault with replacement. Looking at the pages

that are resident, pages 1, 2, and 3, and looking at the incoming page stream after the 4, we see that page 3 is referenced much further in the future than pages 1 or 2, so page 3 is selected to be ejected (Haku in computer science!).

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1								
Frame 1			2	2	2								
Frame 2				3	4								
Fault		f	f	f	fr								

Table 6.24 shows the first 4 page references using the Optimal Algorithm. Notice page 3 was ejected because it is referenced in the reference stream farther in the future than pages 1 and 2.

Table 6.25 shows the completed solution, up the point in which it is still possible to look to future for references that in the frames. At this point it is appropriate to revert to FIFO, but for clarity we will omit this part of the solution. Because there is only a reference to page 5 in the remain stream, it is not possible to use the Optimal Algorithm to decide which of the 3 resident pages is used furthest in the future. Notice the improvement in the numbers of page faults to this point in the progression through the page reference stream.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1	1	1	1	1	1			
Frame 1			2	2	2	2	2	2	2	2			
Frame 2				3	4	4	4	5	5	5			
Fault		f	f	f	fr			fr					

Table 6.25 shows the solution using the Optimal Algorithm. The pink blocks indicate the point at which it is appropriate to revert to FIFO.

Table 6.26 shows the completed solution including the part in which FIFO is employed. The final page fault count is 7 with 4 needing replacement, an improvement over FIFO.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	1	1	1	1	1	1	3	3	3
Frame 1			2	2	2	2	2	2	2	2	2	4	4
Frame 2				3	4	4	4	5	5	5	5	5	5
Fault		f	f	f	fr			fr			fr	fr	

Table 6.26 shows the solution using the Optimal Algorithm.

6.93 Counting Algorithms, LRU and LFU

Since the future cannot be predicted, LRU and LFU try to use metrics that will help keep pages that are more likely to be reused resident in memory. The LRU algorithm looks backwards in the page stream to see which page was used the farthest in the past. Note that often times it returns the same results as FIFO. Table 6.27 shows the results of the

LRU algorithm using 3 memory frames. The algorithm generates 10 page faults, 7 with replacement.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	3	3	3
Frame 1			2	2	2	1	1	1	1	1	1	4	4
Frame 2				3	3	3	2	2	2	2	2	2	5
Fault		f	f	f	fr	fr	fr	fr			fr	fr	fr

Table 6.27 shows the results of the LRU algorithm and 3 frames. The overall page fault count is the same as the results generated by FIFO but, the page order is different.

The LFU algorithms examines the past page stream and ejects the page that has been used the least number of times. Table 6.28 shows the results of the LFU algorithm up to the reference to page 5, using 3 memory frames. Like LRU, its results can be similar to FIFO. One thing to note is that when there is not enough data to make a decision, in this case, FIFO is used as a default. For example, when page 4 is referenced, pages 1, 2, and 3 have all been referenced just once – a tie, so 1 is ejected by defaulting to FIFO.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4						
Frame 1			2	2	2	1	1						
Frame 2				3	3	3	2						
Fault		f	f	f	fr	fr	fr						

Table 6.28 shows the LFU algorithm up the reference to page 5.

The reference to page 5 brings up an interesting decision for the algorithm. Pages 1 and 2 are resident and have both been referenced twice, making page 4 the one to be ejected. Under FIFO page 4 would have been ejected also, but for a reason that did not have any statistical reasoning behind it. Because they are based on statistical reasoning, in the long run LFU and LRU should do better than FIFO. In these small examples, there is not enough data to make any determinations that hold meaning. Table 6.29 completes the LFU algorithm for the data. The overall results are 10 page faults, 7 with replacement, the same as FIFO and LRU.

Page stream		1	2	3	4	1	2	5	1	2	3	4	5
Frame 0		1	1	1	4	4	4	5	5	5	3	4	5
Frame 1			2	2	2	1	1	1	1	1	1	1	1
Frame 2				3	3	3	2	2	2	2	2	2	2
Fault		f	f	f	fr	fr	fr	fr			fr	fr	fr

Table 6.28 shows the results of the LFU algorithm.

In summary, the page replacement algorithms discussed all have their place. FIFO is a good basic algorithm; it has the added benefit that it can be used as a fall back algorithm by the other algorithms when there is not enough data to make a decision. The optimal algorithm is the best performer, but is not realistic because it relies on knowing the future references in the page stream. That being said, it provides a good basis of comparison for the other algorithms. LRU and LFU try to incorporate decision metrics into the process, making them good alternatives to FIFO.

6.94 Exercises

1. What is address binding?
2. What is the difference between the following address types?
    a. Symbolic
    b. Non relocatable
    c. Relocatable
3. What is the difference between static and dynamic linking?
4. What are the pro's and con's of using DLLs?
5. If you are going to deliver software to a customer and you want to give them just an executable, what type of link do you want to use? Justify your answer.
6. What is the difference between logical memory and physical memory?
7. Explain how an overlay driver can be used to run a program that is larger than physical memory.
    a. What would happen if there was no overlay system available?
8. What is swapping?
9. What does contiguous allocation mean?
10. Differentiate between the following:
    a. Single fixed partition
    b. Multiple fixed partition
    c. Dynamic partitions
11. What is the difference between internal and external fragmentation?
12. Describe how a base/limit system works.
13. Define:
    a. Base
    b. Limit
    c. Offset
14. In reference to memory allocation system that uses pages, define/provide equations for the following:
    a. Frame
    b. Page
    c. Page number
    d. Frame number
    e. Offset
    f. Logical address
    g. Physical address
15. Given the following address stream below, and a page/frame size of 100, generate the page reference stream.

    a. Address stream = {721, 43, 121, 222, 44, 327, 45, 428, 223, 328, 45, 329, 224, 122, 225, 46, 123, 722, 47, 124}. Calculate the page reference stream using a page/frame size of 100.
    b. How many processes are running, based on the numbers in the address stream.
    c. Does locality of reference help with the above question? How?

16. Paging algorithms:
    a. Given a page size of 128 bytes, what is the page reference stream for the following addresses.
        i. 128, 159, 513, 1096, 2064, 789
    b. Process the following page reference stream (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) using a frame table size of 3 the following algorithms. Indicate page faults and faults with page replacement.
        i. FIFO
        ii. Optimal
        iii. LRU
        iv. LFU
17. Using the code provided (mainStubs.cpp) as a guide make a page replacement program that does LRU, FIFO and Belady replacement algorithms. You can use this code or write your own.
    a. You can use the input deck in the "address.txt" file.
    b. Do it in LINUX!

Contents of address.txt

```
721
43
121
222
44
327
45
428
223
328
45
329
224
122
225
46
123
722
47
124
```

```c
/* mainStubs.cpp - a primer for a simple page replacement program. */

#include <stdio.h>

/* Logical defines. */
#define TRUE 1
#define FALSE 0

/* Page defines. */
#define PAGE_SIZE 100
#define MAX_FRAMES 5

/* Paging algorithm modes. */
#define FIFO 1
#define LRU 2
#define OPT 3

typedef struct {
  int pageNum;
  int usage;
  int lastUsed;
  int timeStamp;
} pageData;

int adds[100];
pageData frames[MAX_FRAMES];

int readAddressStream(char *filename);
void showAdds(int numAdds);
int pageReplace(int numAdds, char mode);
int searchFrameTable(int pageNum, int nFrames);
void showFrameTable(int nFrames);
int getIndexOfOldestPage(int nFrames);
int getIndexOfLRUPage(int nFrames);
int getIndexOfBeladyPage(int nFrames);

void main(void)
{
  int numAdds;
  int pageFaults;

  /* Send message to user. */
  printf("Hello TV Land! \n");
```

```
  /* Read the incoming address stream from the input file. */
  numAdds = readAddressStream("address.txt");
  printf("numAdds = %d \n", numAdds);
  /* Show the addresses to the user. */
  showAdds(numAdds);

  /* Implement the FIFO page replacement algorithm. */
  printf("Page replacement (FIFO) \n");
  pageFaults = pageReplace(numAdds, FIFO);
  printf("pageFaults = %d \n", pageFaults);

  /* Implement the LRU page replacement algorithm. */
  printf("\n");
  printf("Page replacement (LRU) \n");
  pageFaults = pageReplace(numAdds, LRU);
  printf("pageFaults = %d \n", pageFaults);

  /* Implement Belady's page replacement algorithm. */
  printf("\n");
  printf("Page replacement (LRU) \n");
  pageFaults = pageReplace(numAdds, OPT);
  printf("pageFaults = %d \n", pageFaults);
}


int readAddressStream(char *filename)
{
  FILE *in;
  int address;
  int j;

  in = fopen(filename, "r");

  j=0;
  while(fscanf(in, "%d", &address) != EOF) {
    adds[j] = address;
    j=j+1;
  }
  fclose(in);

  return j;
}


void showAdds(int numAdds)
```

```
{
  int j;

  printf("Address Stream. \n");
  for(j=0;j<numAdds;j++) {
    printf("%d \n", adds[j]);
  }
}


int pageReplace(int numAdds, char mode)
{
  int j;
  int pageNum, offset;
  int frameNum, nFrames;
  int repFrame;
  int pageFaults;

  /* Initialize variables. */
  nFrames = 0;
  pageFaults = 0;

  /* For each address in the incoming address stream, manage the the page
table. */
  for(j=0;j<numAdds;j++) {
    /* Calculate page and offset. */
    pageNum = ;
    offset = ;

    /* Search frame table to see if page is present in memory. */
    frameNum = searchFrameTable(pageNum, nFrames);

    /* If the page is not found in the page table, add it. */
    if (frameNum == -1) {
      /* If there is room in table add the frame. */
      if (nFrames < MAX_FRAMES) {

      }
      else {
        /* Pagefault. */
        switch (mode) {
        case FIFO:
          /* Find oldest frame. */
          repFrame = getIndexOfOldestPage(MAX_FRAMES);
          break;
```

```
        case LRU:
          /* Find the least recently used frame. */
          repFrame = getIndexOfLRUPage(MAX_FRAMES);
          break;
        case OPT:
                    /* Find the frame used furthest in the future. */
          repFrame = getIndexOfBeladyPage(MAX_FRAMES);
          break;
        }
        /* Replace the frame. */


        /* Update the pagefault count. */

      }
    } /* End if page was not in frame table. */
    else {
      /* Frame was found in the table. */
        /* Update the usage count and last time used. */

    } /* End else. */

    /* Show the frame table to user. */
    showFrameTable(nFrames);
  } /* End for. */

  return pageFaults;
}



int searchFrameTable(int pageNum, int nFrames)
{
  int j;
  int frameIndex;
  char searching;

  return frameIndex;
}

int getIndexOfOldestPage(int nFrames)
{
  int j;
  int old, oldIndex;
```

```
  return oldIndex;
}


int getIndexOfLRUPage(int nFrames)
{
  int j;
  int leastRU, lIndex;


  return lIndex;
}


int getIndexOfBeladyPage(int nFrames)
{
  int j;
  int opt, oIndex;


  return oIndex;
}

void showFrameTable(int nFrames)
{
  int j;

  printf("Frame Table - ");
  for (j=0;j<MAX_FRAMES;j++) {
    if (j<nFrames) {
      printf("%d  ", frames[j].pageNum);
    }
    else {
      printf("#  ");
    }
  }
  printf("\n");
}
```

True Story – Hammered, But Not in a Good Way

This story comes to the author from his twin brother; it is told in first person.

After hurricane Katrina most of the people along the Gulf Coast were having to deal with the aftereffects of the storm. Some people's homes and property were completely destroyed, while others had minor damage. We were lucky because we had built our home on pilings; it had made sense because in 1995 there had been two large rainstorms that flooded all of our town, and our property. The house on the property at the time had been built in 1940; the water nearly came into the house, so in 1997, when it was time to build a home for our family, we decided to raise the house so that if we ever had a water problem again we would not as vulnerable. The hurricane had put 8 feet of water on our property, but none in the house. Luck is a good thing.

However, my elderly neighbor was not as lucky. When I got back to town, I checked on him and he was okay, but his truck was toast. I had the equipment and wherewithal to repair the truck, so that's what I set out to do. The truck had sat underwater for a few days, and the motor, the transmission and all related mechanical gear were exposed, actually filled with nasty Katrina water.

I thought the motor was repairable, if it could be gotten to in time. I removed the motor from the truck, drained everything out of it, mostly gallons of salt water. The motor could not be turned over, so I set out to take it apart. I had done this several time before, so I sort of knew what I was getting into, or so I thought. Once I got down to the pistons and crankshaft, it started to get looser. I could turn it a bit, maybe 20 degrees or so, and then it would hang up. I began to remove the pistons one at a time, each time I would gain a bit more rotation out of the crankshaft. The cylinder walls did not look horrible so I still thought this thing was repairable. The motor still would not rotate freely, even though there was only one piston out of 8 left in it. By this point, it was me or the motor. I was determined to get this thing apart and repaired.

The way that pistons are removed from motors is pretty straight forward. You remove the connecting rod bolts, pop off the bearing cap, and then slide the piston out. If it is stubborn, you use the handle of a hammer and tap it out. If you put the crankshaft in right place, you can push a piston out through the bottom of the motor, and since this motor was so seized up, this how I was doing it.

One piston to go. The crankshaft and everything was out of the way, the piston would move down about a third of the way into the cylinder and that was it. No matter how much WD40, WD41 or whatever was put in there to ease the process, nothing worked. So I did what any regular guy would do; I got a bigger hammer. It worked, the piston moved a bit, but then locked up. No worries, more lube, and a bigger hammer. Each time I did this, it would move down and stop, so I worked my way through all of the hammers I had, and finally I was using my biggest hammer, a big happy sledge hammer.

Several judicious, careful, thumps with the base of the sledge hammer handle had moved the piston down to about the halfway point of the bore. But now it was not moving at all.

So close too. Maybe a bit less judicious is needed. Whack! It moved. Another one, nothing. WHACK! A better whack it moved a bit more. All it needed was a really good whack, and I was just the guy for it. WHACK!!! Nothing. One more time, this time I gave it everything I had. WHA-WHACK!!!! It is really interesting how time can slow down during troubling events. I clearly saw the base of the sledge hammer hit the piston squarely in the middle; I also clearly saw, and base of the sledge hammer bounce off the piston; I clearly saw the pretty round face of the business end of the sledge hammer coming right at my head. And then, clearly, I saw nothing.

However, I heard something. It was me, laughing. I was laying on the floor of the garage laughing, after knocking myself out with a sledgehammer. Way to go! Aside from a mild scar that makes me even more ruggedly handsome, everything was fine. I junked that motor and build a good one, and after a bit more work, the truck ran like a top.

Chapter 7 – Secondary Storage

The previous chapter talked extensively about memory systems, from contiguous to paged systems. One of the important aspects of the paged systems was the fact that pages were moved in and out of memory as required by processes. When a page fault occurred, and algorithm such as FIFO, Optimal, or LRU, selected a page to be ejected so that the new page could be read from disk and loaded into memory. In this chapter the basics of secondary storage will be introduced. We will see how the user views the secondary storage, and how the system manages the disk and secondary storage in order to support the file system and other aspects of OS's operations such as paging.

Remembering the relative speed of the various components that a computer is comprised of, the CPU, its caches, and TLB's operate at the highest speeds. In second place is the computer's main memory, but as far as being second place, it is still much slower than the CPU and its closely linked components. Secondary storage is a distant third from the computer's main memory when it comes to operational speed. At least it was until the advent of modern solid-state drives (SSD). Modern SSDs have closed the gap somewhat but are still not as fast. However, when compared to disk drives and tape drives SSDs are quite speedy. Speaking of tape drives, some people would ask "Do we still even use those things?". The answer is an unqualified no, or yes, well maybe, meaning that in common practice, most users do not ever use them, but there is still lots of data that is archived on tapes, and that depending on where a person is working and what they are working on, they may in fact need data from these tapes.

When we think of secondary storage, we usually are thinking of the systems file system and the components that it is comprised of. In order to adequately discuss secondary storage a few definitions are needed. They will be covered in the next section.

7.1 Preliminary Definitions and Discussions

In order to create a basis from which to develop an understanding of topic at hand, secondary storage, some definitions are needed.

File – A file is collection of related data items. It could be a program, a document, an image or any of a myriad of other types of data.

Files can have any of a number of formats/structures. It used to be that files fell into two primary categories, alpha-numeric or binary, and while this characterization still has applicability, it is more common now to use a few more categories when characterizing the type of a file. For example, we may say that file is an image file. Technically, an image file (for the most part) is a binary file, but it holds an image, as opposed to a binary file that holds music.

Files can have complicated structures embedded within them. Many image files have several sections; they may have a section that stores compression information, another

section that holds a table of contents that points to different tiles of the image, and so on and so forth.

Alpha-numeric files typically hold ASCI characters. These files are easy to look at with an editor, but are not the same as files that word processors create. The files that word processors create have an abundance of alpha-numeric data, but they also hold vendor specific formatting data.

So, what does it all mean? It means that the first definition, as vague and non-specific as it is, is a good definition, when considering all the various types of files and file formats that exist currently, and that will be created in the future.

Determining what is in a file can be done a few different ways:

- User Convention – Files names follow a set of conventions. This is the way things were commonly done in the past. For instance, a file may have a ".c" suffix meaning it was a "C" programing language file. If the programmer wanted to indicate that they had moved on from that version, they may have saved a copy of it with the ".cold" suffix, indicating that it was an "old" "C" file. Another example: the executable file generated by Linux "C" compilers is commonly called "a.out". It is common to rename that file to a more descriptive name such as "averageFileNumbers.exe", with the ".exe" extension (suffix) meaning it is an executable file.
- OS Defined – In modern times, this is a common method. The OS and applications vendors work together to associate a files extension, such as ".doc" with a specific program that is designed to open/work with files of that type. That being said, users still have the ability to rename files at their convenience, which sometimes results in conflicts.

File access methods, or more descriptively access to the data in a file, follow a few norms:

- Sequential Access – Most commonly used files are read and written sequentially, that is from the start of the file to the end of the file. That being said, it is possible to use special file positioning commands to place the read pointer at a desired location and then read. The default action of the read pointer, however, is sequential, meaning that after it has been repositioned, reading will be sequential. Writing to the middle of a file usually causes all data that was after the write pointer to be lost, so writing in a non-sequential manner is not viable.
- Random Access – Random access files allow reading and writing to and from any position of the file. In some cases, the file can be viewed in a similar manner as an array would be, that is, having elements. Some languages, Java for one, support random file types.
- Indexed Access – An indexed file consists of an index file and one or more associated data files. Some of those data file, as we shall see later, can actually be

index files also. Typically, a program designed to read an indexed file, knows the structure of the file, and takes the appropriate actions to retrieve the desired data.

File Paths are either relative meaning that the path is relative to the current working directory, many times that is the directory that the file is in, or absolute, meaning the path is based on the root directory of the file system.

- Relative – The path is relative to the current working directory.
    - Example: When in the C:\users\bob directory, a user may want to change directories to another user's, nancy's, directory.
        - cd ..\nancy would go up one directory from bob, and then down to nancy. This is a relative path from the bob directory.
- Absolute – The path is based on the root directory.
    - Example: When in the C:\users\bob directory, a user may want to change directories to another user's, nancy's, directory.
        - cd C:\users\nancy would go start from the root directory (C:\) and go to the nancy directory.

File protections are commonly based on two things, who is allowed to interact with the file and what are they allowed to do.

The who part is:

- Owner – whoever owns the file.
- Owner's group – whoever is in the owners group.
- Everyone else – the rest of the users.

The what part is:

- Read
- Write
- Execute

For example, if an executable file, "a.out", is copied to a more user friendly name, "myAverage.exe" the mode of file may need to be changed so that everyone can use it. Below is a Linux example.

```
$cp a.out myAverage.exe
$chmod 777 myAverage.exe
```

The "cp" command does the file copying; the chmod command changes the protections of the "myAverage.exe" file so that owner/owner's group/everyone else has read/write/execute permission.

File attributes normally include:

- Name – All files have names. The name must be unique to that file.

- Type – Files usually have types associated with them. It can be by convention and or by OS/vendor determination as discussed earlier.
- Location – Files are stored on secondary storage media, whether on disk, tape, or some other type of storage. Each file has a location. Generally, the file system has a logical method of locating the files in the system that abstracts the details of the particular hardware that used to store the actual file.
    - Locations of the file from the user standpoint are implement using the file path as earlier described.
- File size – Each file has size in bytes associated with it.
- Protection information – Files have protection information that tells the system what kind of actions the file can be used to do, and who can do them.
- Owner – Files, especially on multi-user systems, have owners. This information works with the protection information to determine who has access to the file and what can be done with it.
- Last modified – Each time a file is changed, the data and time of the change is recorded. This allows users to revert to older versions of the file if need be.

Operations that can be done on files include the following:

- Creation – A file can be created. This is usually done by a user when they are using an IDE, a word processor, or some other program that works with files. Programmatically, files are created using one of the many variants of the "open" keyword.
- Truncate – Files can be truncated by specifying a maximum file size. The truncate operation will release all data that is in the file that follows the maximum byte size.
- Delete – Files are deleted by users for many reasons; some being the file is not needed anymore, it has been replaced by new versions of the file, etc.
- Open – The open keyword is a part of many languages. It allows programmatic reading and writing of files.
- Close – The close keyword works with "open" to regulate access of a program to file. Close signals the OS that the file that the program is linked to is no longer needed by the program.
- Append – The append command/keyword allows data to be added to the back of an existing file.
- Read – A file can be read by a program by opening the file in read mode. When opened in read mode, a read pointer is placed at the beginning of the file and each read statement retrieves data from the file and moves the readpointer.
- Write – A file can be written by a program by opening the file in write mode.
- Seek – The seek command moves the read/write pointer to a byte position in the file. In this way "sequential" files can be read as if they are random access. However, writing to a place other than the end of the file (in a normal file) will usually truncate the file at the end of the data just written.
- Tell – The tell command returns the byte position of the read pointer in a file.

***Directory* –** A directory is a collection of files in a file system. The file system and its directories are usually organized like a tree, where the root of the tree sits at the top of the system, whether it is a disk, tape, or a compressed archive. Usually directories and directory systems do not allow cycles (just as trees do not allow links from children to their ancestors). It can be said that:

- Directories are tree structured acyclic graphs.
- A file may be referenced from more than one directory.
- A file is deleted only when there are no more links to it.

7.2 File System Implementation

File systems are implemented in many different ways. This section provides an overview of the common methods used. It begins with a few definitions, and then by identifying and describing the major structures of the file system, and moves on to methods of allocation.

7.21 Definitions

Each file in the system has a File Control Block (FCB) associated with it. The file control block is defined as:

File Control Block (FCB) – The File Control Block holds file metadata, that is data about file itself. Implementations vary, but typical FCB data includes the following:

- Name
- Owner
- Date created/modified
- Path
- Type
- Location
- Size
- Protection information

The secondary storage system is organized into an array of logical file blocks. File blocks are mapped to physical locations on the systems hard drive, or whatever type of mass storage the system is using. Using the logical block system allows the upper layers of the OS to abstract away the details of the hardware. Each file block has physical address on in hardware.

The File System consists of all files, FCBs and data, required hardware, directories and format information (block size, free blocks, etc.) needed to implement the secondary storage system.

7.22 Hard Drives

As stated previously, there are many types of secondary storage, but the currently the most common is the hard disk drive. The disk drive consists of a spinning disk platter on which data is written and read by a disk head; some drive systems have multiple platters and read/write heads. Figure 7.1 below shows a diagram of a disk drive. The drive is organized into tracks and sectors. It is these tracks and sectors that allow us determine where a file block is; each file block has a unique address based on a track/sector pair.

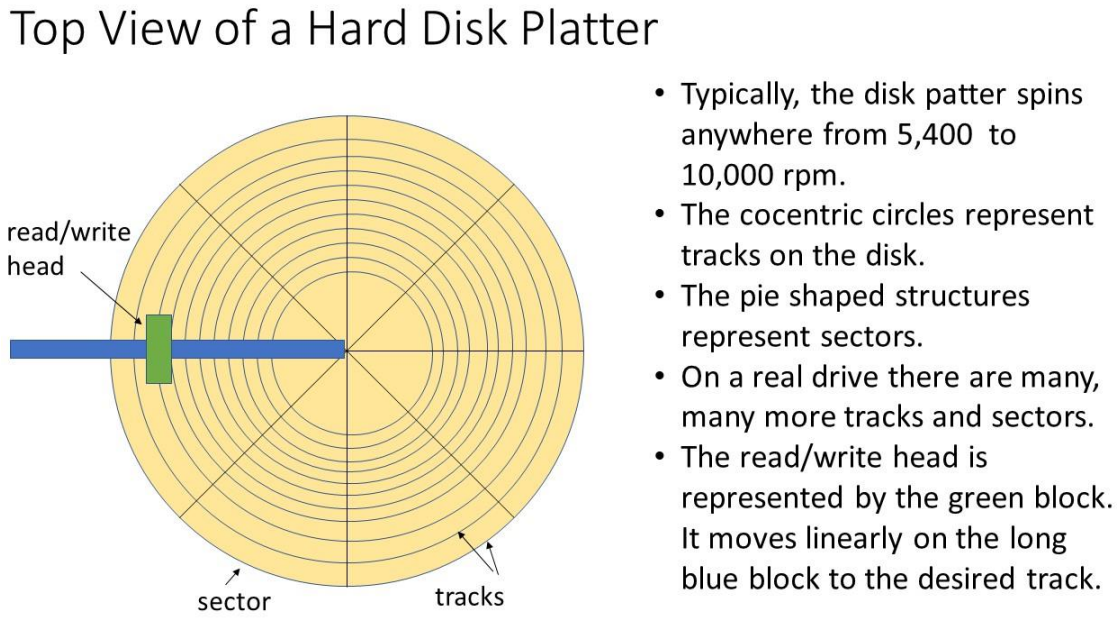


Figure 7.1 shows a diagram of a hard disk platter. The platter spins rapidly just under the read/write head. When a block, defined by a track/sector pair rotates under the disk head, data can be read or written. The linear speed of the platter rotating under the head is much higher than the lateral speed of the head itself.

One thing to remember about hard drives, as compared to memory, or the other electronic components of the computer, they are mechanical devices, meaning that their components are subject physical forces such as momentum and inertia. These factors influence the way that blocks are laid out on the drive. Figure 7.2 below illustrates this point.

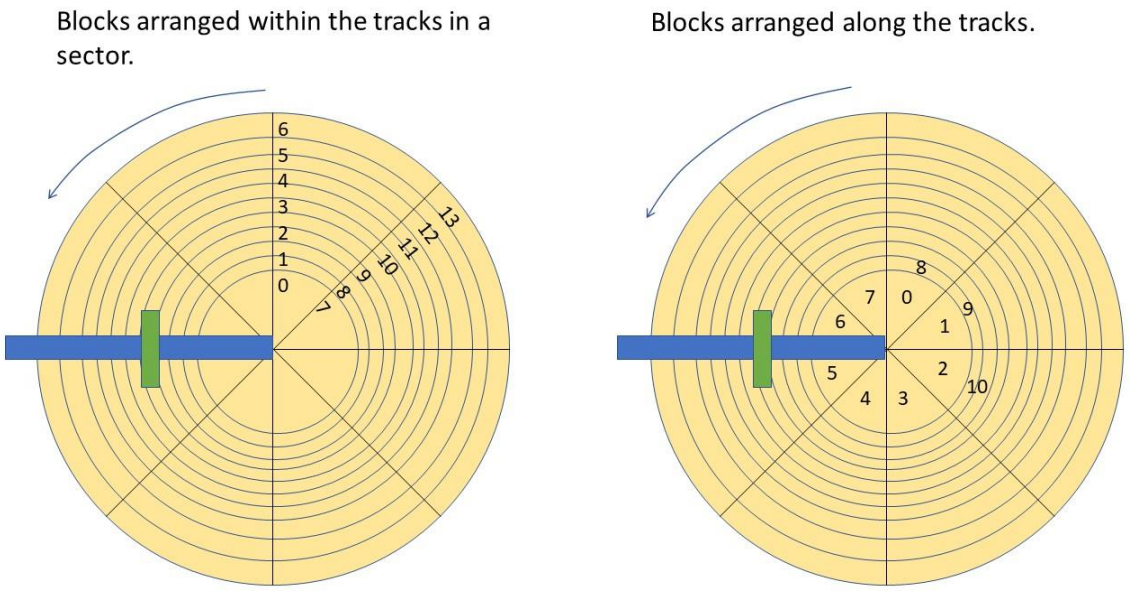


Figure 7.2 shows 2 possible arrangements of the blocks on a hard disk. The one on the right allows all of the blocks on a track to pass under the read/write head without moving it.

Since the spinning platter conserves momentum, and is spinning much faster than the disk head can move, arranging the blocks along the tracks as seen in the diagram on the right hand side of figure 7.2 is more efficient. It minimizes read/write head movement for reading sequences of blocks. For example, to read blocks 2 through 5, using the arrangement on the left, the read/write head would have to move to the correct track, wait for the block 2 to move under it and read it, then move 1 track, wait for block 3 to move under it, and so on, and so forth. Using the scheme illustrated by the diagram on the right, once the read/write head is over the correct track, all that must happen is that the blocks pass under the head.

One last point, no matter how fast the read/write head can be moved, it is still inefficient and likely much slower than the speed that a block moves on the rotating disk. Why? Because the head must move and stop at precise location. In more physical terms, the mass of the head must be accelerated and decelerated to very precise locations. Contrast this with the spinning disk; once it is up to speed, it does not stop, reverse directions, etc; no big momentum shifts. As long as its controller can keep its rotational speed within specs, everything is good. So it makes sense to arrange the blocks along the tracks, from physical world perspective.

In later sections these ideas will be used as a basis for optimization of read/write head movement.

7.3 File Storage Allocation Methods

This section will provide an overview of the basic methods of allocation. The most basic is contiguous, and as expected there are some similarities between contiguous secondary storage allocation and contiguous memory allocation; i.e. mainly both suffer from external fragmentation. The next method that will be reviewed is a simple method of

linked allocation. This method helps alleviate the fragmentation caused by the contiguous methods, but suffers from its own set of problems. Following that is the File Attribute Table (FAT); it is a step up from the first two methods, and finally there will be a discussion of multi-level indexed systems.

7.31 Contiguous Allocation

In contiguous allocation schemes all blocks that comprise a file must be contiguous, meaning that they are sequential and uninterrupted from the first block of the file to the n^{th} block of the file. The contents of the directory file hold, among other file metadata, the file name, its starting block and the number of blocks in the file. Table 7.3 below provides an example of directory data for a contiguous system.

Directory – Contiguous		
Filename	***Start Block***	***Number of Blocks***
biff	2	7
baff	10	4
eek	0	1
beek	17	4

Table 7.3 depicts a directory file for a directory that 4 files in a contiguous allocation system.

Figure 7.4 below shows how the files are laid out in secondary storage. Remember, this is in logical blocks, meaning that the logical block numbers are translated to track/sector addresses in order to be accessed.

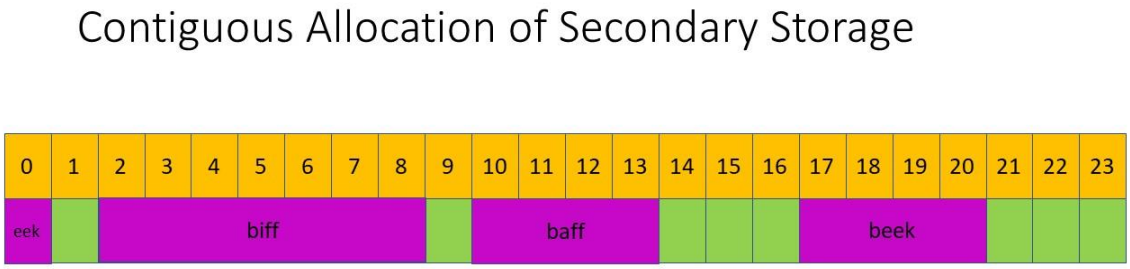


Figure 7.4 shows the files from table 7.3 in their logical blocks. Notice that there are 8 free blocks (the green blocks), but no file of size larger than 3 blocks can be accommodated because of fragmentation.

While the contiguous allocation scheme suffers from fragmentation it has the strengths of simplicity, and speed. For example, if a programmer wants to access some data item in the middle of the biff file, they can use the seek() function to move the read pointer directly to that byte position. As we shall see, with some other systems, it takes more work to do such an operation.

7.32 Linked Allocation

In order to work around the fragmentation problems that arise from the contiguous allocation scheme, a linked method can be used. In this method, much like a linked list in data structures, the last few bytes of each of a file's blocks, points to the next block. The last block in the file will have an End of File (EOF) indicator where the next block link

would normally reside. The directory data is similar to that of the contiguous scheme, except that instead of storing the number of blocks in the file, we store the last block in the file. (Actually, it makes sense to store both of these data items in the directory). Table 7.5 below shows the directory structure for a simple linked allocation. Notice that the file "squeak" has been added, and because of the links, it can be larger than the size of 3 that the earlier contiguous system was limited to.

Directory – Linked			
Filename	***Start Block***	***Number of Blocks***	***Last Block***
biff	2	7	8
baff	10	4	13
eek	0	1	0
beek	17	4	20
squeak	21	5	9

Table 7.5 shows the directory information for a simple linked allocation system. The file squeak has been added; its blocks are spread into 3 separate groups as can be seen in Figure 7.6.

Figure 7.6 below illustrates the block arrangements for the linked system described by table 7.5. The squeak file is distributed into 3 groups of linked, non-contiguous blocks. In the figure, each block holds the address of the next block, the last block in the file has the EOF marker.

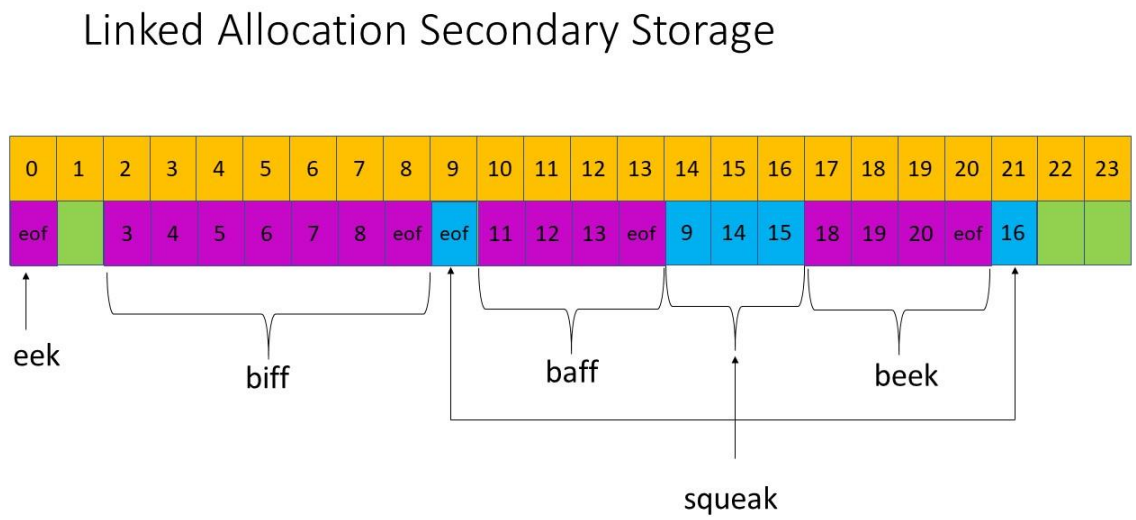


Figure 7.6 shows the arrangements of the blocks for the linked allocation system described by Table 7.5. The squeak file has 5 blocks in 3 groups. This file could not be accommodated by the contiguous scheme discussed/described earlier because of fragmentation.

The linked scheme solves the fragmentation problems that plagued the contiguous scheme, but it introduces the following issues:

- If the end of a block is somehow damaged and the link to the next block is lost, all data in the file after that block is lost.

- In order to skip ahead in a file (perhaps by using the seek() command), the entire file must be examined in some way shape or form so that the links can be followed. Because of this, access can be slow.

Systems using a File Attribute Table can avoid the issues associated with the linked scheme and the fragmentation problems prevalent in contiguous systems.

7.32 The File Attribute Table (FAT)

The File Attribute Table (FAT) system is a linked allocation whose file information is held in a table at the beginning of the disk partition. It is similar to the linked system as described in the previous section, except that all of the link data is concentrated in the FAT making it possible to remove the links from the end of each block. Often times the FAT is kept in cache memory so access to the information is quick. The bullets below describe the particulars of the system:

- FAT is a form of linked allocation.
- The FAT inhabits the beginning of each disk partition.
- There is one entry in the table per disk block.
- Each entry holds the block number of the next block in the file, unless it is the last block. In that case it holds and EOF marker.
- Empty blocks are demarked by zero entries.

The FAT is held in cache memory (fast memory, usually on board the CPU) so that it can be accessed quickly. If it were on disk, every disk access would require 2 access, 1 for the FAT and 1 to get the desired data from the disk. Table 7.7 below shows the directory for a FAT system. Notice that the "eek" file inhabits block 1 instead of 0. Block 0 now houses the FAT. In this table we no longer track the last block, that information is held in the FAT. The number blocks is retained so the file size (in blocks) can easily displayed.

Directory – FAT		
Filename	***Start Block***	***Number of Blocks***
biff	2	7
baff	10	4
eek	1	1
beek	17	4
squeak	21	5

Table 7.7 shows the directory entries for a FAT system. The "eek" file has been moved to block 1 so that the FAT table can occupy the block 0, the start of the partition.

Figure 7.8 shows the data of the FAT table. Notice that this figure represents binary data that would be stored in disk block 0 of a disk partition and would also be held in cache memory. Also notice that blocks 22 and 23 have 0's in them, indicating that they are free blocks.

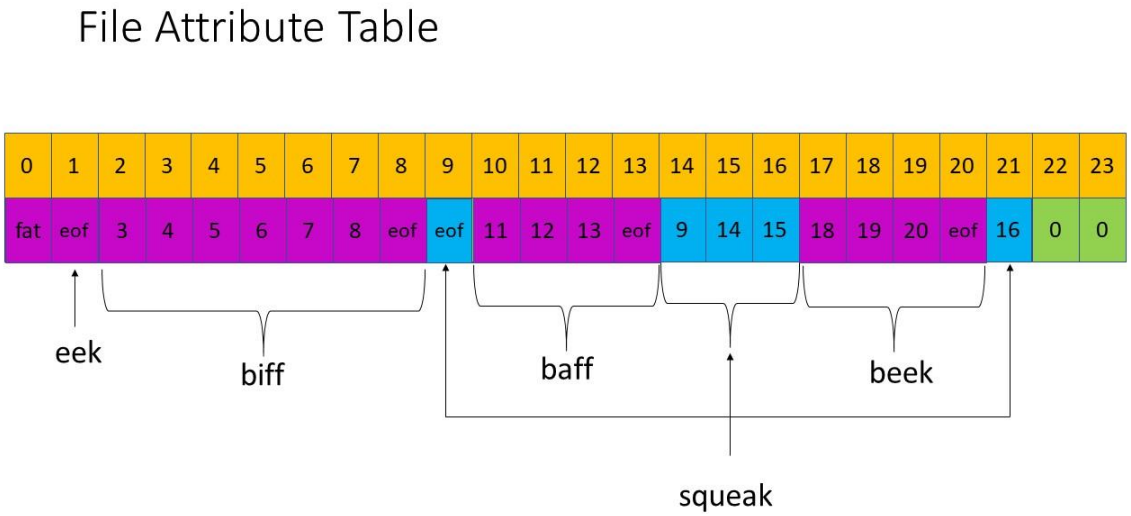


Figure 7.8 shows a diagram of the FAT for the data in Table 7.7. The 0th entry is set to indicate that block 0 holds the FAT table, the "eek" file has been moved from block 0 to block 1 (the FAT resides in block 0), and blocks 22 and 23 have 0 entries to indicate that they are empty blocks.

The FAT system is excellent and was used by many computers for years. The biggest disadvantage to it is when a disk crash occurs and wipes out the FAT. This would happen if the disk head damages the data in block 0. In this case, if the FAT data has not been put into backup storage, the entire file system could be ruined. Since the file system data is held in a central location, the FAT, the consequences of such a failure are much more so than if the data is distributed throughout the system.

7.33 Multi-Level Indexed Systems

In contrast to the centrally located data of the FAT based systems, multi-level indexed systems store the files block data in the FCB. Because of this, it is far easier to avoid catastrophic file system failures such as those that happen with a system like FAT. The FCB of a multi-level indexed system stores all of the usual FCB data, and it stores file data block location data.

The FCB of a multi-indexed file has a file data section and a block map section. Table 7.9 below provides an example.

Multi-Indexed FCB	
File name	biff.txt
Owner	Jethro
Created	7/6/2020, 14:19
Path	C:\users\jethro
Type	txt
Size	14kb
Block Data	
B0	2
B1	3

B2	4
B3	8
B4	9

Table 7.9 shows an example of an FCB for a multi-indexed file. The first section holds the file attributes that a normal FCB would hold; the Block Data section holds the logical block numbers for file blocks B0 though B4.

The FCB illustrated in table 7.9 uses only direct links to the file's data blocks. The advantage of this technique is that it is quick; the direct links point directly to data blocks on the disk. Also, the blocks do not have to be contiguous, so the fragmentation issues are avoided. Since each file has its own FCB, if something happens that corrupts a portion of the disk, only the FCB's of the files local to the corrupted area will be damaged, not the entire file system.

Usually the structure of the FCB is a balance between limiting the size of the FCB and providing enough room in the FCB so that the maximum size of a file is not unduly limited. For example, if the FCB is only big enough to hold 16 file block addresses, and each file block is 4k, then the maximum file size is:

Max file size = 4k × 16 direct links = 64k

A 64k file size limit is pretty low; it is really not viable in todays world. The solution is to use indirect blocks. Suppose that we keep the 16 direct links and add 4 indirect links. In this case, an indirect link is a link that points to a block that is full of links to data blocks. Figure 7.10 below illustrates the point.

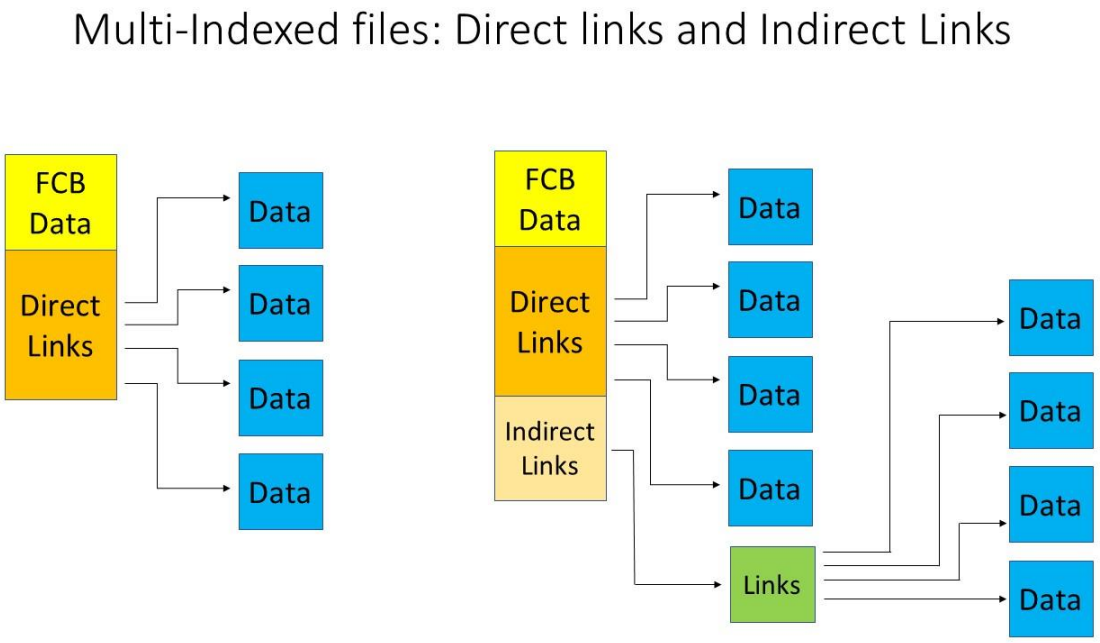


Figure 7.10 the diagram on the left shows an FCB with 4 direct links. The diagram on the right shows an FCB with 4 direct links, and a single indirect link. The indirect link has the disk block addresses of another 4 data blocks.

By doing this, the FCBs can remain a manageable size, and are adaptable for large file sizes. For instance, small files can use just the direct links in the FCB, while larger files can use the indirect links.

Consider the following: Suppose that the block size for the system is 8k, and each disk block address (logical) has 2 bytes. Question 1, what is the size of disk?

To answer that question, we need to know the maximum integer supported by 2 bytes.

$2^{16} = 65, 356$

This is how many disk blocks our address system can support. So, if the disk will hold all these, then we have a disk that is

Disk size $= 2^{16} \times 8k$

$= 2^{16} \times 2^3 \times 2^{10}$

$= 2^{29}$

Our disk holds 2^{29} bytes. In real numbers that is pretty decent sized, but not nearly as big as it looks. Consider that a kilobyte is 2^{10} bytes, a megabyte is 2^{20} bytes, and a gigabyte is 2^{30} bytes. Our disk drive is not quite a gigabyte big, so it is big by 1995 standards. By the way it is:

Disk size = $2^9 \times 2^{20}$ bytes

= 512 * 1 megabyte

= 512 megabytes.

To figure out a maximum file size we need to know the structure of the FCB. Suppose our FCB supports 16 direct links, and 2 indirect links. The block size is 8k, the addresses size in bytes is 2.

That means that the direct links support:

$16 \times 8k = 128k$

The indirect links support

$2 \times ((8k \div 2) \times 8k) = 64$ megabytes

The $(8k \div 2)$ is how many block addresses are in 1 disk block. 8k block size divided by 2 bytes per address results in 4k block addresses held in the indirect link.

Total file size is 64 megs + 128k.

This is a decent file size, but still not enough to support some images, music data, or film clips. To increase the max file size, more indirect links can be added. If 2 more indirect links are added, the equation becomes:

$4 \times ((8k \div 2) \times 8k) = 128$ megabytes

Here the total file size is:

128 megs + 128k

This is bigger, but if that is not big enough, a set 2nd level indirect links can be added. These are links that point to links that then point to data. Figure 7.11 below illustrates a system with 2nd level links.

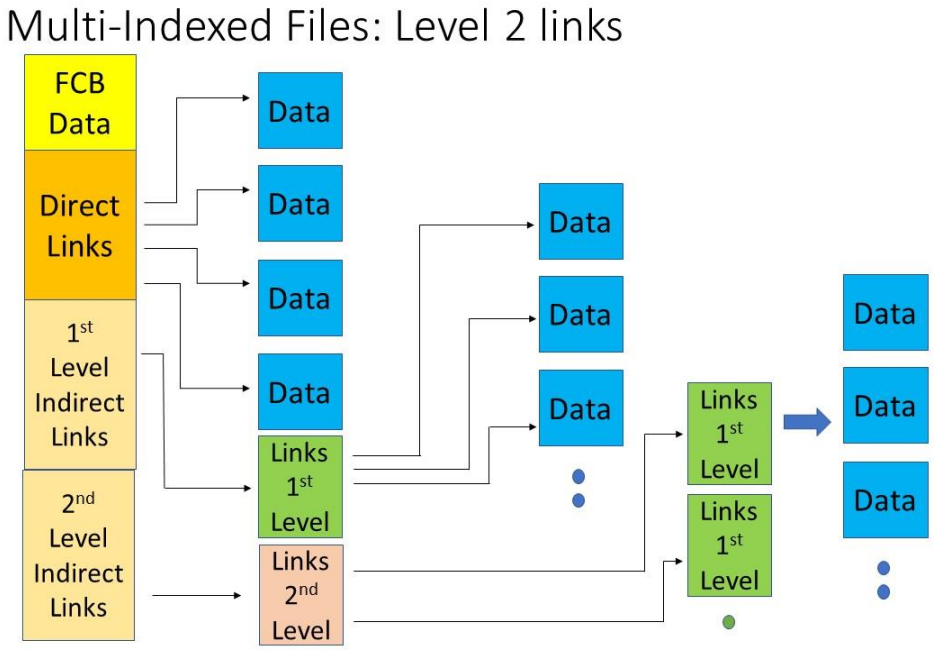


Figure 7.11 shows a diagram of multi-level indexed system that supports direct, 1st level, and 2nd level links. The first level links in the FCB point to index files that have addresses that point to data. The second level links in the FCB point to index files that point to index files that then point to data.

Suppose we modify our FCB from above and add a single 2nd level link to it. That means that our FCB will support 16 direct links, 2 1st level indirect links, and 1 2nd level indirect link. The math for the 1 all but the second level links remains as before:

The direct links support:

$16 \times 8k = 128k$

The 1st level indirect links support:

$2 \times ((8k \div 2) \times 8k) = 64$ megabytes
And the 2nd level indirect link supports:

$1 \times ((8k \div 2) \times (8k \div 2)) \times 8k$

Which works out to:

$4k^2 \times 8k = 32$ gigabytes

Total file size is 32 gigs + 64 megs + 128k. That is a large file. If it is not big enough then these ideas can be extended. Note, that the disk drive in the earlier example was only 512 megs, so for this exercise, we would need a much larger drive.

7.4 Disk Platter Latency

The hard drive's platter spins at a high speed; 7200 rpm is a common speed for modern disks. When a logical block request is received by the disk controller software it is translated to a track and sector pair. In order to fulfill the request the disk read/write head is moved to the track and when the sector rotates under the read/write head the requested action (read or write) occurs. Approximate values for the time that it takes to fulfill the request can be arrived at by making some reasonable assumptions.

Assume that our computer is using a common drive that has the following specs:

- Rotational speed: 7200 rpm
- Platter diameter: 3.5 inches

If the requested track is equally likely to be any of the tracks on the disk, then on average it is the middle track. This assumption allows us to use the middle of the disk as the diameter, so then our average diameter is 1.75 inches.

If it is equally likely for any sector to be requested then, we can assume that on average, the desired is the average of the closest sector to the read/write head and the furthest sector from the read/write head. This line of reasoning brings us to the conclusion that we will have to wait ½ of a disk rotation for the desired sector to pass under the read/write head.

The following equations describe the problem:

- Working Diameter = Platter Diameter ÷ 2 = 3.5/2 = 1.75 inches
- Circumference = $\pi \times D = \pi \times 1.75 = 5.4979$ inches
- Speed of a Block on the Platter = 5.4979 * 7200 rpm = 39,584.067 inches per minute

From these calculations it can be seen that the sectors are moving pretty fast. We are interested in their speed in inches per second, so we divide the results by 60:

- 39,584.067 inches per minute/60 seconds per minute = 659.734 inches per second

That is still really fast; it is almost 55 feet per second. Considering that 60 miles per hour is 88 feet per second, the sectors on the disk are travelling at about 37 ½ miles per hour. To solve for the latency time we use the following equations:

- ½ rotation = circumference ÷ 2 = 5.4979/2 ≈ 2.75 inches.

To solve for the latency time we divide the distance that the sector needs to move by the speed that it is moving:

- Average Latency Time = ½ rotation ÷ rotational speed = 2.75/659.734 ≈.0041 seconds, or about 4 milliseconds (ms).

A second and much easier, but less illustrative way to come to almost the same answer is to say that the disk needs to rotate about ½ of a revolution for the desired sector to arrive under the read/write head.

- Time for 1 revolution = 60 seconds ÷ 7200 rpm = .00833 seconds
- Time for ½ revolution = .00833 ÷ 2 = .00416 seconds, or about 4 ms.

One last thing to note about disk drives. We saw that the speed that the sectors were moving was about 660 inches per second. This speed is very fast. Imagine if the disk head had to move from track to track at this speed, meaning that in the space of, for examples sake, it had to move an inch. Is it within physical reason to expect that something would accelerate to just over 37 mph and stop in the space of an inch? The answer is emphatically "no", not with commonly available, mass produced technology. So, this is the reason why we arrange the logical blocks along the tracks as shown in figure 7.2 on the right, versus the figure on the left.

7.5 Disk Head Scheduling

Earlier, in section 7.22, and the preceding section, the basics of hard drives were discussed. Here we learned that because the hard drive is a mechanical system, physical factors such as momentum and inertia play a role in how the blocks will be laid out on the disk drive. In essence, it is best to lay the blocks out sequentially along the disk tracks so that the blocks can pass under the read/write head as the disk rapidly spins. We also learned that as the disk drive spins, it conserves its momentum, which is different than the linear movements of the read/write head. Since the read/write head is required to start and stop in very precise positions above the tracks, and it cannot conserve momentum. All of these factors result in the desire to limit disk head movement. This section will discuss several algorithms for optimizing disk head movement.

7.51 Scheduling Algorithms

As logical disk block requests flow into the disk controller software, the requests are translated to track, sector pairs. The disk head controller moves the head to the desired

track based on the incoming stream of the track requests. For the reasons discussed earlier, it is efficient to limit the disk head's trackwise movement. It can be beneficial to limit/economize its total movement and also limit the number of times it changes directions. As a baseline, the first algorithm that will be examined will be first come first serve.

Frist Come First Serve

This algorithm is similar operates similarly to the first come first serve algorithms studied in past sections, notably in the process scheduling section and in the contiguous memory section. Given a stream of track requests, the first come first serve algorithm services the tracks in the order they arrive.

Example: Track requests = {77, 42, 107, 3, 145, 166, 13}

If the read/write head is currently above track 60 then it moves to track 77, then 42, and so on. To measure the algorithm's effectiveness, the total number of tracks (track distance) is calculated. Figure 7.12 below shows the path of the head given the set of track requests.

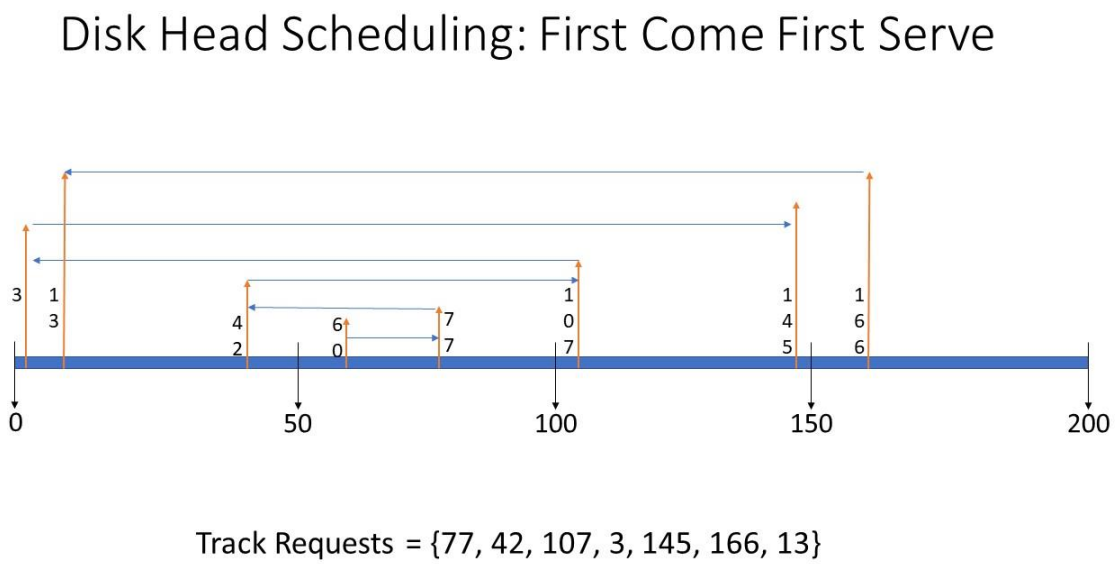


Figure 7.12 shows the path of the disk read/write head above the tracks of the disk platter. The diagram shows a side view of a disk with 201 tracks, 0 – 200. The head is initially located above track 60. It then precedes to visit the tracks in the order that they arrive, thus the name of the algorithm, First Come, First Serve.

The disk head covers 537 tracks while servicing the incoming requests. Table 7.13 below details the process.

Current Track	***Next Track***	***Tracks Travelled***
60	77	17
77	42	35
42	107	65
107	3	104
3	145	142
145	166	21
166	13	153
Total Tracks Travelled		537

Table 7.13 shows the distance travelled by the disk read/write head as it services the track requests using the First Come First Serve disk head scheduling algorithm. Notice that the total tracks travelled is 537. This will be used as a baseline of comparison for the other algorithms.

Looking at the table, the "Tracks Travelled" column is found by using the following equation:

Tracks Travelled = | next track – current track|

The absolute value is used because all distances travelled are positive.

Shortest Seek Time First

The shortest seek time first algorithm attempts to economize the number of tracks covered by move the disk head to the nearest track from its current position. It, like all the algorithms that will be discussed, will visit all of the tracks, however, the order will be different. Using the same set as used earlier, with the same starting position, 60, it can be seen that the closest track is 77, the same as before, but after that instead of moving to track 42 the head will move to track 107 because the is closer (distance = 30) to 77 than track 42 (distance = 35). Figure 7.14 below depicts the path of the read/write head.

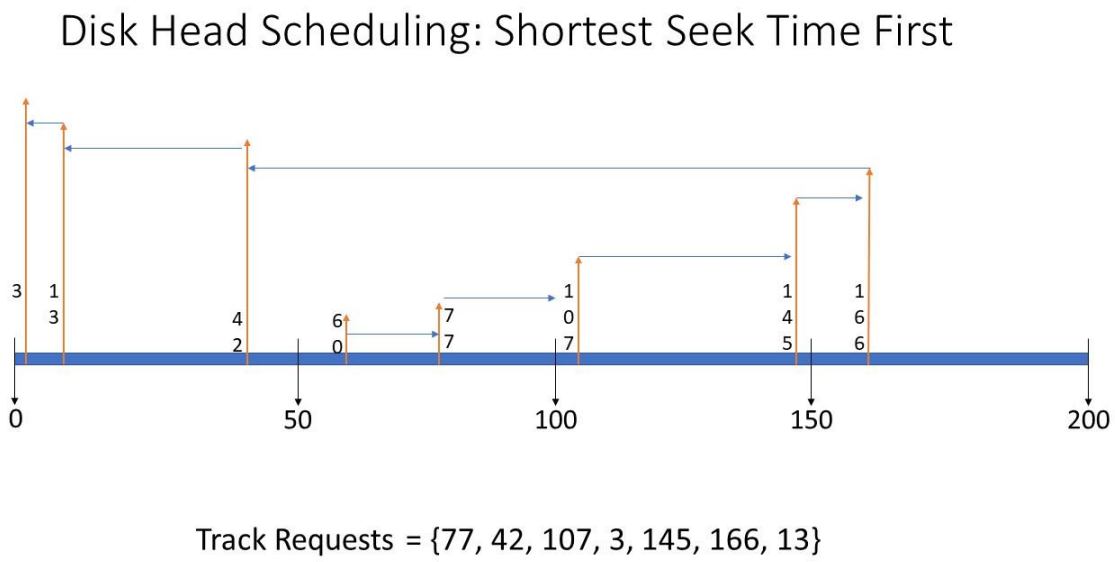


Figure 7.14 shows the path of the disk head using the Shortest Seek Time First scheduling algorithm. Notice how there is only one change of direction in the heads movement as compared to the multiple changes produced by the First Come First Serve algorithm (see figure 7.12).

Table 7.14 details the order that the requests are serviced, and the tracks travelled per head movement. The total tracks travelled is substantially lower; 259 vs. 537.

Current Track	***Next Track***	***Tracks Travelled***
60	77	17
77	107	30
107	145	38
145	166	11
166	42	124
42	13	29
13	3	10
Total Tracks Travelled		259

Table 7.14 shows the order that the tracks requests are serviced using the Shortest Seek Time First disk head scheduling algorithm.

One of the benefits, other than the lower number of tracks traversed, that this algorithm seems to have shown is that there are less changes in direction of the disk head. This could be a good thing, reversing directions is often associated with big changes in momentum, which is not always a good thing for mechanical systems. That was not be design however, and is really just an artifact of the chosen data set. The next algorithm attempts to resolve this issue.

SCAN

The SCAN algorithm is sometimes called the elevator algorithm because it moves the disk head in one direction, picking up all of the requests until it hits the last track on the disk and reverses direction. Often times elevators in tall buildings do this; it keeps the

elevators from repeatedly reversing directions and it minimizes their movements. Using the same data set:

Track requests = {77, 42, 107, 3, 145, 166, 13}

And head starting position:

Current track = 60

Head Direction: Head moving towards track 200

We will need one more piece of information, the direction of the head's movement. For this example, we will say that the head is located above track 60 and moving towards track 200. Given these initial conditions, the head would move all the way to track 200, visiting tracks, 77, 107, 145, and 166, along the way. Then it would reverse direction and move to track 0, visiting track 42, 13 and 3 along the way. The total movement would be:

Tracks Traveled = |200 – 60| + |0 – 200| = 340

Is this worse than Shortest Seek Time First? It is more tracks, but it is guaranteed to have a low number of head direction reversals. The other factor to consider is that after track 166 was visited, why did we go all the way to 200 when there were no tracks to be visited after the 166. The literature describes this both ways. One way to think of it is, suppose that while servicing the current set of tracks, more requests arrive. This is plausible, since the disk is a mechanical system, and is much slower than the computer's memory and CPU, which are generating the requests. If this is plausible, it is quite possible that a track request between 166 and 200 could be generated while servicing the current set of tracks. This same argument applies for the trip back to track 0. Even though all the tracks between track 60 and 200 have been serviced, requests for tracks in this range could arrive while in the servicing process.

Cyclic SCAN

The Cyclic SCAN algorithm is very similar to the SCAN algorithm. The difference is that Cyclic SCAN only services disk track requests while moving in one direction. For our example, we will say that the read/write head services requests while moving from the track 0 towards the track 200 direction. If the head is located above track 60, it will service all requests on the way to track 200, and then come zipping back to track 0. It will then start servicing the remaining requests; the ones between track 0 and 60. The total movement would be:

Tracks Traveled = |200 – 60| + |0 – 200| + |42 – 0| = 382

What is the reasoning behind this algorithm? Two things can be said:

- The disk read/write head only has to stop at precise locations going in one direction.
- Since it does not have to stop on the return trip, it is possible for the disk hardware to move the head at a higher velocity (no stops ever, so go for it) for the trip back to 0 from 200.

If the head can be moved twice as fast for the return trip, our equation can be modified to reflect this reduction in time:

Tracks Traveled = $|200 – 60| + (½ \times |0 – 200|) + |42 – 0| = 282$

This makes the Cyclic SCAN algorithm competitive with the Shortest Seek Time First Algorithm, and it provides a potential reduction in the number of head reversals.

7.6 Exercises

1. Regarding File Systems:
    a. What is the definition of a file system?
    b. What are its components?
    c. In relative terms, where does the speed of access of the secondary storage fit with main memory and CPU speeds? How does it compare to tapes storage and printer speed?
2. Secondary storage can be sequential or random access.
    a. Give some examples of each.
    b. Which is better?
    c. Which is cheaper?
    d. If one is better, is it always better? Justify your answers.
3. Some files are indexed. What does that mean? Draw a conceptual diagram of an indexed file.
4. What is a File Control Block (FCB)? What information does it hold?
5. What do the seek() and tell() file operations do? When would they be used? Would indexed files use them? If so, why?
6. If a.out is an executable file and I type in the command "chmod 666 a.out" what will happen when I run a.out? How do I fix it?
7. Define the following:
    a. Disk
    b. Track
    c. Sector
    d. Block
8. What is the difference between contiguous allocation and linked allocation?
9. What are the benefits of linked allocation as compared to contiguous allocation?
10. How did the file attribute system improve on linked allocation?
11. Regarding disk drives:
    a. If the drive is not a solid state drive, does it have moving parts?
    b. If it has a disk platter, what is a typical speed in rpms that it spins?
    c. If is has a platter and read/write heads that move to center themselves over a specified track, which moves faster, a point on the disk platter or the read/write head? Why?
    d. Does the disk platter have momentum? If so, is it conserved? If so, is because it is spinning?
    e. Does the disk head have momentum? If so, is it conserved? If not, is it because it moves linearly and must reverse directions depending on the track request?
12. How do we select a block size for our disk drive?
13. How do we arrange the logical to physical mapping of the blocks? Why?
14. If a multi-level indexed file has a block size of 4k and a block address size of 4 bytes, and has the following properties, what is the size of the file:
    a. 4 direct links
    b. 2 1st level indirect links.
15. Suppose that a multi-level indexed file has a block size of 4k and a block address size of 4 bytes, and has the following properties, what is the size of the file:

a. 8 direct links
b. 4 1st level indirect links
c. 2 2nd level indirect links

16. Differentiate between the following disk head control algorithms:
    a. First Come, First Serve
    b. Shortest Seek Time First
    c. Scan
    d. Cyclic Scan
17. Using the following track requests: Track requests = {71, 142, 7, 39, 15, 106, 130, 44, 87} with the read/write head located above track 110 and heading towards the last track, 200, find the number of tracks for the following algorithms:
    a. First Come, First Serve
    b. Shortest Seek Time First
    c. Scan
    d. Cyclic Scan
    e. Cyclic Scan, with return trip discounted by a factor of 4 (it takes ¼ the time to get back).

True Story – 40, Maybe. The 3rd Time is NOT the Charm

This story comes to the author from his older brother, thank goodness. It is told from his point of view.

I had built a new home on our property just a couple of years before, mostly because I realized that when I got married, my new wife was probably going to want to live with me, although it was not a given. The only reason that this is mentioned is that I had lived in the old house on the property for several years before I built our new home. Getting rid of the old house was now something that had to be done. It was a sad thing because it had served me well; except for some minor electrical issues (fuses and whatnot), and the fact that it was riddled with termites, the house was in pretty good shape. But there was no real way to update it (it was built in 1940) without spending more money than I had, and there was no way to integrate it into the new house, so the bottom line is that it had to go.

Being a somewhat handy guy with a good truck, some chains, and some power tools, I figured that the demolition job would be fun, and hey, good exercise. Once I got started, I re-learned what I had figured out in college; after a few good crunches and whacks, demolition done by hand is really just a lot of hard work. There had been several serious mishaps during the job, but luckily no one had gone to the hospital. There was the first day when we (the usual collection of friends who want to break stuff) were taking down the breezeway that connected the house to the car shop. We had cut the columns that held up the breezeway and were bringing down the roof. For some reason some of us needed to be near the whole thing while one of us pulled with the truck. The roof ended up falling on a couple of my buds, but thank goodness we had put on motorcycle helmets, just in case. It turns out that a motorcycle helmet will protect someone's head from a falling roof; who knew?

There was the other time when I was taking down the roof and almost fell through. My dear friend Marv was helping that day. He knew that I did not like heights and was throwing boards down on the rafters so that I could walk on them as I carried a bucket of power tools in one hand and extension cords in the other. Just as I took a step onto one of the boards, I noticed it had a knot on it – a weak spot. As soon as I stepped on it, it broke, but I guess because I had seen the knot my body went into automatic mode. My right arm dropped the cords and caught a rafter above me allowing me to break my fall. Marv rushed over and grabbed my legs and stuck them on some "good boards" and we finished up.

So, with the house about halfway taken down I decided to go work on it. Knowing that my 40th birthday was the next day, I decided that I was not going to do anything dangerous, because I wanted to make it to 40. No roofs, no pulling columns, none of that, just good ole sledgehammer work. It was all working out pretty well, I had knocked down a few kitchen walls, and busted down about half the ceiling of the kitchen. The only problem was that the wires from the house's various lighting and outlet circuits were starting to get in the way (the walls that they used to be in were mostly gone). I had turned off all the power, for safeties sake, and would only turn on power when I needed to use a tool, like a circular saw, or a sawszall. That being said, old houses sometimes have weird added in circuits so you still have to be careful. And I was, I checked everything that morning and it was all off.

I busted up several 2 by 4s holding up the kitchen wall by the sink. I was in a rhythm with the big sledgehammer and it felt good. It is awesome getting in a rhythm, each swing smacks something and busts it up in a bunch of pieces; I loved it. I went to take a big swing at some cabinets, I wanted to see those things bust off the wall and go crunch, but what happened instead is my sledgehammer hooked a wire behind me and nearly flipped me. Didn't see that coming. I got up, blurting out a long string of good-natured melodious curse words and appraised the situation. Those wires needed to be moved. It was really just the wires that used to be in the ceiling that were the problem. I got the 8 foot step ladder, promised my self that I would not go high; high is bad, we already established that, climbed two thirds of the way up and went to cut the wires; they were all off; we already established that too. I grabbed the first wire, cut it with the dykes (dykes are a good wire cutting tool), grabbed the second wire cut it too, grabbed the third wire and squeezed the dykes hard. BUZZZZAPPPP! A huge shower of sparks blew up in my face and I saw my life flash before me. I heard myself hollering a barely unintelligible, but well put together stream of curse words. The bad thing about this is, I was on a ladder. Luckily, I dropped the dykes and the wire went with it. I climbed down the ladder, looked at the charred wire and looked for my dykes.

I could not believe it, I had checked the power, but there must have been a circuit that was added somewhere after the fact. I found my dykes and looked at them. The nice blue handles were still blue, but a bit dingy; I think they kept most of the electricity from getting me. However, the jaws now had a nice big round hole melted right through them. With that, I decided that I had had enough for the day. I was going to make it to 40, if it killed me or not.

Chapter 8 – Overview of I/O Systems

In this chapter we will take an extremely generic view of I/O and the various parts of the system that implement I/O events. The first few paragraphs offer a conceptual discussion from the perspective of how I/O would work if there was no I/O subsystem and how the I/O subsystem helps economize OS operation.

Suppose the CPU is executing an output operation for some process that needs to write data to a printer. In general, the process will carry out its output using OS system calls that support the I/O process. These calls will take the data from the process and put it into a buffer that will be then transferred to the requested I/O device. The usual way to accomplish this is by using a "device driver". The OS and CPU are not required to know all of the operational details of the I/O device, that is the job of the device driver.

A device driver is a piece of software that communicates with the operating system and an I/O device in order to carry out an I/O event. In the case above, a process is in the midst of writing some output to a printer. The process does not actually write directly to the printer, instead, it writes its data to a buffer and signals a device driver to handle the details of delivering the data to the printer. So, the process basically says, "Hey, I just put a bunch of data into the I/O buffer at this location, Mr. Device Driver, can you get it printed for me?" The device driver says something to the effect "Hey Mr. Process, no biggy, I got this. I know all the details of the printer. I will do it, and let you know when I am finished." At that moment, the process can then keep doing what it does, and the device driver starts the interactions with the printer.

Interactions with the printer can be pretty involved (at low level), depending on the printer. That is why it is usual practice for the printer vendor to provide a device driver for their product.

Continuing our example, suppose the printer is a simple teletype (tty) unit with little to no memory, and minimal onboard control systems; meaning that the unit will need to be controlled by the OS. If the OS did not use the device driver model, it would need to feed the data to the tty machine one character at a time. The process is outlined by the bullets below:

- Some process that the OS is running, writes data to the printer.
- The print statement used by the process is implemented by a system call. It used to be that a write to the screen used a different logical unit number than a write to the printer. Logical unit numbers are/were used to describe where the output is going to go; possibilities include writes to the screen, the printer, or a file.
- The data to be written goes to an I/O buffer. If there is no device driver, then the CPU must take charge of the control lines from the CPU to the printer, meaning that data transfer control signals, data transfer, and acknowledgements, must be all be handled at the CPU level. The CPU must:
    - Let the printer know that it wants to send data.
    - Wait for the printer to tell the CPU that it is ready for new data.

    - Send the data to the printer.
    - Wait for the printer to acknowledge that it received the data.
- The above process would need to be done for each piece of data sent to the printer.
- Once the data from the print/write statement has been sent to the printer, the process from which the print statement originated can continue.
- Remember, the printer is a mechanical device, so it is operating on a way slower timeframe that the CPU, so if the CPU is burdened with these responsibilities, the OS will slow down and many CPU slices will be wasted while waiting on the I/O devices.

A better way to handle this problem is to use the device driver. The device driver is a process that the CPU can kick off and let run concurrently with other processes. Its job to transfer data from the I/O buffer to the designated I/O device. Using the device driver, the process that is using print statements to output data writes the data to a buffer and continues on with its processing. The OS uses the device driver to handle the details of the interactions with the printer.

In the example from above, the printer was a very simple tty machine, so the device driver would monitor the control lines of the tty and feed it a character at time from the I/O buffer. This system works because the device driver is an OS process that is running way faster than the printer, and it one in several OS processes that are running concurrently. Remember, concurrently running processes are processes that the OS is running by allocating them some number of CPU slices at some specified frequency. If the device driver gets "n" CPU slices 20 times per second, and the tty printer can print 20 characters per second, then as long the number of CPU slices in "n" is enough to transfer 1 character per CPU session, the printer will keep operating at full capacity with no real slowdown in OS operations.

The key to making all of this work is communication and data transfer between the primary OS processes, the device drivers, and the devices. The communications and data transfers can be implemented by using a combination of specific hardware signals (interrupts), shared memory, and software signals, often implemented using semaphores.

Another way to help the situation is to employ the use of I/O controllers. An I/O controller can take the burden of monitoring control lines from the CPU. Many if not most modern computers and I/O devices rely on these to accomplish data transfers. It is not really much different than what was initially discussed, it just that now, the OS transfers data to a buffer, and asks the device driver to handle the I/O event. The device driver works with the controller (the controller usually has its own small microprocessor) to transfer the data, usually to another controller in the device (modern printers are practically computers in their own right). The device driver can now run and take up less CPU slices because its job is to kick off the controller, feed it data, and find out when the data transfer is complete. It no longer has to monitor individual control lines and the like; that is the controller's job, and controller has its own microprocessor, so it does not have to burden the CPU for slices.

The next sections provide some detail on hardware devices and device types.

8.1 Hardware Device Operation – Polling, Interrupts, DMA

Hardware devices, such as keyboards, mouses, etc. work one two ways; either they are polled or they work using interrupts. At first glance, polling seems much different than interrupt. In polling, a device is "asked" if it has some data to be transferred; in an interrupt driven system, a device that needs attention sets an interrupt line. It is the difference between a school teacher asking each student if they understand the topic of discussion, versus the teacher looking for a hand raised, indicating that the student has a question. At first blush polling seems far different, but in the above example, just as in the computer, the teacher still must occasionally look to see if someone has raised their hand.

Keyboard Example – Polling vs Interrupt

A keyboard makes a great example. If a keyboard were a polled device (it is not), the process managing the keyboard would ask each key if it were pressed. There are lots of keys, so that is a lot of polling. Since the keyboard must be serviced on a regular basis, so that keyboard response is good, the number of polling requests really adds up quickly. To try to get a little closer to reality, imagine that the system worked this way:

- When a key on the keyboard is pressed, a Boolean flag is set that indicates the keyboard needs servicing.
- At the same time, the ASCII value of the pressed key is written to a pre-specified location.
- The keyboard managing process checks the Boolean flag at a regular interval. If it is set, it retrieves the value of the pressed key from the specified location and places into an I/O buffer.

While neither one of the above examples is exactly how systems work, they offer insight into how interrupt-based systems economize operations. Notice that there is still "polling" involved here, in the form of checking the Boolean flag at regular intervals.

Interrupt with DMA

The discussions centered around transfer of a single piece of data from the keyboard. We saw that by using an interrupt, far less status checks need to be made, thus economizing operations. Another common data operation is to transfer a buffer of data from the computer/CPU to a peripheral device. This can be accomplished using interrupts and a system not too unlike the producer/consumer problem. That is, there is a buffer in computer that needs to be transferred to the peripheral. In this scenario, the two buffers are separate. The data is transferred from one to the other using two concurrently running processes. The thing to note here is that the CPU is involved in this transfer.

An alternative to this is to use a direct memory access (DMA) approach in which the DMA controller manages the memory transfer, using interrupts, and handles all of the data bus/control signals, thus relieving the CPU of the memory transfer burden.

8.2 Device Types

In this discussion we will differentiate the types of the devices by the amount of data that they send in one transfer. That is do they send a single character, like a keyboard or a communications port, or do they send a chunk of data, like a disk or a tape drive.

Character Devices

Keyboards and Comm ports send one character at a time in a serial fashion. That is, they sent a character of data down the communication lines, 1 bit at a time. These devices followed a serial data transmission protocol, such as RS232. Actually, RS232 was succeeded by PS/2 and then later by USB.

To transmit a byte, a typical example may be that there was a start bit, 8 data bits, a parity bit, and a stop bit. Also, sometimes, an acknowledge bit would be sent back from the host. Below we briefly describe the functions of the various bits/signals involved:

- Start bit – The start bit lets the host computer know that the peripheral is getting ready to transmit a data byte.
- Data bits – The peripheral will transmit 8 data bits, in the case of a keyboard, it would be an ASCII character, but other times, it could just be 1 byte of data among many, such as in the case of transferring data file from one computer to another using an RS232 cable and interface. The data bits themselves are usually transmitted from least significant bit first to most significant bit.
- Parity bit – The parity bit acts as a simple error checking protocol. If the parity is set to even parity, if there are an even number of 1's in the data bit, the parity bit is set to 0, otherwise it is set to 1. Odd parity is just the opposite. One thing to note, both the sending and receiving device's data transmission protocols must be in agreement.
- Stop bit – The stop bit indicates that the data packet – the byte of data along with any control bits (stop, parity, etc.) has completed transmission.
- Acknowledge bit – This bit is sent by the receiver to the transmitter to indicate that it received the data. Often times people refer to these types of signals as "handshaking" signals.

Data rates for keyboards are not extremely high; even the fastest typing done by someone who just drank 7 cups of coffee is not that fast when compared to the speeds that electronic signal can be sent, so anywhere from 7 to 12kbits per second would suffice.

Other serial devices, such as dumb terminals, would operate at higher data rates. That being said, older terminals such as a VT 100 made by Digital only supported about 24 lines and 80 characters per line, so the need for stratospheric data transfer rates was not that high. A typical baud rate for one of these was 9600 baud or even 19200 baud. A VT 100 is shown in figure 8.1 below.

Figure 8.1 – This is an image of a very common "dumb" terminal, a Digital VT100. Photograph courtesy of Wikipedia. The screen supported alpha-numeric data. It held 24 lines of data, 80 characters each.

One thing to note about the data transmission rates, they did not have to be high, there was not that much data being transferred. To put it in perspective, the 24 by 80 screen of the VT100 holds 1,920 bytes (if being used monochrome mode – 1 color). A modern icon on the screen of any version of Windows is typically between 16 by 16 up to 256 by 256. These are 32 bit icons, so for a typical 32 by 32 icon, it would take $32 \times 32 \times 4$ bytes to represent the icon. That is 4,096 bytes, or a little over double the amount of data in the entire VT100 screen for just one icon.

As the amount of data being transferred has increased, the speed of the serial interface as also moved up. Modern USB connections sport data rates of up to 5 gigabits per second for USB 3.0. The older versions such as USB 1.0 clocked in at 1.5 megabits per second, USB 1.1, 12 megabits per second, USB 2.0 480 megabits per second. Even the slowest USB data rates are blazingly fast compared to RS232; this is what allows for USB devices such as disk drives, and monitors.

Block Devices

Disk drives and tape drives are block devices. Block devices are different from serial devices in that data is transferred to and from the devices in chunks called blocks. Blocks vary widely in size; some are 1kb for an old school table drive to 64kb and upwards for their more modern counterparts.

Tape drives and disk drives differ from each other in many ways, but the one of the most important is that the tape drive is a sequential access device, while the disk is a random-access device.

8.3 Exercises

1. What is difference between a system that uses polling vs a system that uses interrupts?
    a. Provide an example of a system that uses polling.
    b. Does the entry into a critical section use polling or an interrupt?
    c. When is polling better than interrupts?
    d. When are interrupts better than polling?
2. What differentiates a character device from a block device?
3. What is the CPU's role in a DMA data transfer?
4. What is the role of the parity bits in a serial data transfer?
5. Why was it okay for "dumb" terminals use low speed data transfer rates?
6. What do you think the lowest data rate for a Windows remote desktop application would be? Remember, for graphics to work, the screen would need to refresh at about 16 to 24 frames per second. Assume that the screen is VGA resolution, 640 by 480.
7. Why are tapes sequential devices?

True Story – Academic Arrogance

This story came to the author from his older brother. This story is in many ways very relevant to life, but it is not a funny story in my view; it is something to take note of, and to avoid if possible. To give a little background, my brother had been working for a defense contractor for some years at the time the events in this story occurred. He is a non-confrontational person and did not seek out or instigate the events described here. Also, he said that this was a very, very rare event. In all of his years working as a professional computer scientist, nearly 33 at the time he discussed this with me, no one he knew had experienced something quite like this. It is told in first person, from my older brother's perspective.

I had been given the chance to work at our engineering facility a while back. I really enjoyed working with the electrical and mechanical engineers, interfacing our hardware to computers, and collecting data from the various sensors. This type of work was so different from the work done at the facility in town or out on the base; it was more raw and gritty – it was fun.

I was on maybe the second or third project out here. We had built some equipment for data collection in deep water and had sent it to Oak Ridge. It was working well up there for a few months and then it just sort of quit. I had not programmed this part of the project; I was brought in to see why it would occasionally glitch.

The first day that I got to the engineering center, one of the guys, a very talented mid-level electrical engineer, asked me to come into his office. He gave me a little background on the project, and then proceeded to explain something else to me. At first I had no clue as to what he was getting at, but after a moment or two it started to sink in. He drew a picture of a triangle, and then drew some lines through it horizontally; his lines sectioned the original triangle into a triangle sitting on top of several trapezoids (the 3 sides equal variety). He then explained – "At the very top here, this little triangle, these

are the theoretical physicist, because they are the smartest of the smart. Now right under them, right here.." He pointed to the first trapezoid under the triangle – "These are the applied physics guys, and some electrical engineers. Not the circuit layout guys, these are the signal processors and controls guys, and the guys that use wave mechanics. Now under here.." He pointed to the next layer and sectioned it up a bit – "are the double E circuit guys, power guys and the like. This next layer is the mechanical engineers, chemical engineers, and some of the smart civil engineers." The explanation continued on through the layers of the triangle. He put the computer science folks in either the lowest layer or the one above it, lumped in with artist, musicians, and perhaps below the accountants and business guys. In the lowest layer he put labor-based professions, like machinist and the like. I was hoping he was done with this, but he wasn't.

He explained that he kind of straddled the top triangle and the layer under it, after all, he was a talented double E. He then said that if you are on the top layer you can do anything that any of the people in the other layers can do, and do it better and faster. "I can think of and do anything better than any of these guys." – he said as he pointed to all the layers below. He added – "For example, these mechanical engineers here can do any thing better than all the people under them, but they are just not smart enough to solve problems that the people above them can. Its just the way it is." What do you say to that? All I said was "Okay, where's my computer and the equipment?".

It took a bit of time, but I eventually discovered that the engineers had bought and installed a nice dual port ram board into the system. It had come with some library routines, which they did not look at, load or use. "We already know how to program, what do we need that BS for?" was their explanation. After a bit of work I learned that they had used the wrong type of pointers in their code. Back then, a standard pointer declared in C/C++ was a 16 bit pointer, meaning that it could address 2^{16} bytes, and then, if incremented it would go back to 0 (binary math works that way when you are bit limited). The result was that as they wrote data into their arrays, once they exceeded the pointer size limit, the writing address went back to 0. Location 0, as has been noted in this book, is the location of the OS. The more they wrote over the OS, the more unstable the system became, and eventually it would crash.

They also were not acquiring the memory correctly from the dual port ram card; they were not using the allocation functions or the read/write functions from the provided library, so the likelihood of a crash increased. These problems were fixed by writing a memory manager that used the library routines and some others. The repaired code was installed in Oak Ridge and ran successfully.

When the project was over, I went to talk to my double E engineer friend, who by the way, I am still close with. I cordially drew a picture of the triangle and we had a "candid" discussion. I took some joy in being able to fix this project, especially after the little talk on the first day. The problem is that this sort of thing happens, maybe not as confrontational and stark in nature, but academic arrogance does exist and it is not uncommon. It really is not good for anyone, and the biggest problem is that it is an easy habit to fall into. "Oh, those guys, they don't have to take any math, they are morons!"

or “Those math guys, they can’t program their way out of a wet paper bag!” It does not matter who is doing it, it is not good.

The triangle explanation leaves out a lot. It leaves out respect for other people’s craft; it completely ignores creativity, teamwork, and many other qualities that help people solve tough problems. It is definitely something to guard against. You do not want to be the guy giving the triangle explanation, and if you are the person listening to it, do not let it ruin your day. What defines a person is not some level on a little diagram, it is among other things, hard work, talent, and kindness.

Chapter 9 – Overview of Miscellaneous Topics

This chapter provides a quick overview of a handful of relevant OS topics. The purpose of the chapter is to make the reader aware of the topics, more so than providing an abundance of information about them. Each of these topics are important, so much so that each has several, if not hundreds of books written about them. Here, the topics are introduced; the reader is encouraged to explore these topics as guided by their interests and necessity.

9.1 OS Security

The development philosophy of the OS is important. The developers can do their best to create a reliable and useable system, but problems in these complex systems are inevitable. How these problems are handled is up to the developers. Below are two diametrically opposed methods of dealing with the problems when they arise.

- Full Transparency – OS designers publish know problems with OS and work with User community, security vendors, etc. to develop patches and solutions. These can be integrated into next versions of OS.
- Head in the Sand – OS designers do not acknowledge problems and hope that people will not find them or exploit them.

In the long run Full Transparency is better. It will inspire trust among customers and companies. That being said, for this to actually work, the OS manufacturer must uphold high standards so that the occurrence of problems is kept to a minimum.

9.11 Types of Security

The sections below briefly outline common security threats.

Physical

It is best to put valuable systems and data in well protected places. Floods and fires, as well as electrical storms, heat, and moisture can cause massive damage to critical infrastructure and data.

Human

Who gets access to important systems and data is a big part of security. People must be well screened before given access to systems, especially system that house sensitive data and programs. Also, it may be prudent have different levels of access; this way screening can be based on a record of trust.

Password Security

Passwords can be made so that they are difficult to guess. But the biggest problem with passwords may not be that they are strong or not. It could be that they can obtained by illicit means. The following are some examples:

- Password Guessing – Some people have a knack for this. Usually the person guessing the password knows something about the person or knows some of the person's past passwords. Most of the time this happens when people forget their own passwords.
- Accidental Exposure – This occurs when someone purposely looks over the shoulder of someone while they are entering their password.
- Illegal Transfer – Various methods of obtaining information including sniffers and cameras can be used to collect passwords.
- Legal Transfer – This is probably the most common method of losing passwords. A user can give their password to someone they trust. Then this person transfers the password to someone they trust, and so on. Sooner or later the password can get into the hands of someone that is unknown to the original person.

9.12 Password Encryption

The password file is located on the computer somewhere. A common way to store passwords is to use a two stage hashing algorithm.

```
passwordIndex = hash(username);

if (hash(entered(password) != table[passwordIndex]) {
      /* Bar entry to system. */
else {
      /* Allow entry. */
}
```

The above piece of code uses a hash function to generate an index into a password file. If the value of the hash of user's password is located at the location indexed by passwordIndex then there is a match.

9.13 Other Security Problems

- Trojan Horses – These are programs that masquerade as useful programs in order to collect data about users, or to install malicious software.
- Trap Door – A back way into a system secretly left by the designers.
- Worm – A program that replicates itself and deteriorates system performance.
- Virus – A program that replicates itself, and attacks the system, or the system's resources or user's data.

9.2 Error Detection During Data Transmission

The most common methods of error detection are the use of parity bits and checksums.

Parity Bits

Parity bits are based on adding a bit to a group of bits in order to make an even count of 1's (even parity) or an odd count of 1's (odd parity) It is useful in detecting errors but cannot determine where the error is. The transmitting computer adds the parity bits, the receiving computer checks them. If a transmission error has occurred (a 1 flipped to a 0 or vice versa), a parity mismatch will occur. Remember, transmission error can occur in the parity bits, just as they can in the data bits. Take a look at the examples below.

Byte = 1001 | 0001

For even parity, the parity bit for the Byte above would be 1:

1 | 1001|0001

For odd parity, the parity bit for the Byte above would be 0:

0 | 1001|0001

In order to locate an error, a group of horizontal and vertical parity bits can be used. Look at the table below. The vertical parity bits are shaded blue; the horizonal are green. The yellow shaded bit is incorrect. The parity bits in the vertical and horizontal directions do not agree with the data, thus locating the error.

parity	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	1
1	0	0	1	1	1	0	0	0
0	0	1	0	1	0	1	1	1
0	0	0	1	1	1	1	0	0

Checksums

A check sum is used when transmitting packetized data. The checksum is a number that is the integer sum of all the data in a data packet. The sum is calculated by the transmitting computer and checked by the receiving computer. If the sum that the receiving computer does not equal what was sent by the transmitting computer, and error has occurred; the receiving computer can request that the transmitting computer resend that packet. Take a look at the table below:

checksum	Data								
21	1	2	3	0	0	0	4	5	6

The checksum is 21, and it is equal to the sum of the data. If it were not, then an error would have occurred.

Data Packet Numbers

Data Packets almost always have a packet number associated with them. It helps with retransmission requests, as alluded to earlier, but it also helps identify when data has been missed. If the receiving computer detects a that a packet number has been missed, it can flag the error and request a retransmission of the missing packet.

9.3 Exercises

1. How is a Trojan Horse different than a virus?
2. What is full transparency referring to when talking about OS security?
3. What are the most common ways that password-based systems are breached?

4. Given that we are using **EVEN** parity with 4 data bits, find and correct the error.

Horizontal Parity Bits	**Vertical Parity bits** **0**	**0**	**1**	**1**
0	0	0	0	0
0	1	1	0	0
1	1	0	0	0
1	1	0	0	1
1	1	1	1	1

5. Complete the following packets using checksum error checking.

Checksum	**Data bytes**							
	1	3	2	5	4	7	6	0
	8	6	7	0	5	3	0	9
	19	11	8	2	27	8	12	7

True Story – An Awesome Trojan Horse

This story comes to the author from his older brother. It is told by his brother in first person.

When I was in undergraduate college at the University of Idaho, computational and programming work was done using the university's main frames, a pair of IBM 4341 processors running IBM's 370 OS. For that time, it was an impressive system. The details of the processors are very weak, even compared to today's PCs. For example, each of the mighty processors had about 2 megabytes of memory – but to put it in perspective, this was a commercial computer that went on sale in around 1979.

Details of the hardware aside, from an empirical standpoint the system was impressive. It could service 100's of terminals at once without the users noticing too much of a slowdown. When writing a program, it could compile a program of around 1000 lines in a literal blink of the eye – meaning if you blinked or looked away from your screen after you hit the return button to compile, when you looked back, the program would be ready to go. Pretty decent, and magnitudes better than using the punch card systems in the basement of the admin building.

The only problem with this system was that it was expensive, and the students felt it. At the beginning of the semester each student received an allotted amount of CPU funds for each of their computer science courses. For a class like Data Structures, a student would

get about $50 of CPU money. If you used it wisely, no playing, careful program design and debugging, and used the computers in non-peak hours, the $50 would carry you through the whole semester. On the other hand, if you were a real enthusiast and really liked to explore computer science by writing your own programs for fun, the funds would run out quickly. So that is what brings us to this story.

My dear friend Martin was a very bright and talented student. He and his friend Kurt were big into anything computing. While other guys were just getting their homework done and heading to the bars or the gym, Martin and Kurt were burning the midnight oil writing programs and exploring the world of computing, so much so that they continually exhausted their CPU funds for all of their classes. And then, suddenly, they didn't, at least for most of a semester.

We were in class, OS class I think, when the campus police showed up and arrested Martin and Kurt. No one had any clues why. Martin told me what happened later, and it was brilliant and awesome, in an illegal way. He and Kurt had written a Trojan Horse (this is back in about 1982) that emulated the university's official login that ran on the terminals. It would allow a student to "login", that is provide their login name and password, then it sat there for a few moments and said that the login failed, and then the school's login screen would come up again. Martin's program would have collected a login name and password from some unsuspecting student, and stored it in a file for later use. His program would send an error message to the student and end. When his program ended, it logged out Martin and the school's official login would be started. No one knew, that is, until they did.

This is how Martin and Kurt could play on the computers all the time. They would just go to their file and get a username and password and use it to login. That way they could have all their fun, writing games, and playing games, and burn up everyone else's CPU time. It all worked out really well until the admin folks were installing upgrades to the OS and they isolated a file that was hanging up the install. Unfortunately for Martin, it was his stash of stolen usernames and passwords. Martin had taught himself how to do this by ordering the IBM manuals from the university bookstore. He was brilliant. I do think that the university and the CS department were secretly proud of him, but the U of I police were not. He received 600 hours of community service for his escapade. He did have good company in serving it though, his friend Kurt also collected up his share of hours, and for unrelated transgressions our mutual friend Dudz and myself were already hanging fire extinguishers in university housing, so everyone was "happy".

Martin graduated on time with very good grades. He was hired directly out of school by, you guessed it, IBM.

Chapter 10 – Designing a Basic OS

In this chapter we design a basic OS using the knowledge from the previous chapters. The product of the implementation of this OS can be an OS simulator, or these ideas can be used to create an OS for some one or more of the commonly available microchips, such as the Arduino. For either route, a kernel will be needed. For the curious and energetic student that is focused on building an OS for a microchip, the route that makes the most sense is to do both. That is, create the simulation; then once a microchip is selected, rewrite the kernel code to handle all OS services and hardware interactions. The rest of the OS, from the simulation, can then be ported to the run on the microchip and its supporting hardware.

At this point, the question of "Why do this?" should be asked. The answer is straight forward in nature. As stated earlier, by doing this, the material from the previous chapters can be integrated into a functioning system. The other part of the answer is that a greater understanding of the OS components and how they interact can be gained by actually building an OS, versus just studying the concepts. That being said, this paragraphs and sections ahead describe a substantial amount of work; it may be more suitable as a team project. The transition of the system to hardware would be suitable for a 1 or 2 semester capstone project, depending on the amount of complexity that is included.

10.1 System Overview

The goal is to create a simple OS from which multiple users can accomplish computing work. The users should be able to do the usual functions such as:

- Write and edit programs.
- Compile and run programs.
- Save programs to disk.
- Manage secondary storage.
- Send output to printers and peripherals.

In order to do this several application programs that support these activities will need to be available, along with basic OS services. All of the applications, and many of the OS services should be available from the command prompt of the monitor program.

The most basic version of this system, the OS simulation, will be somewhat pared down from what is described above in that there will be a single prompt program that runs the OS services and will serve as the monitor program. In order to make this a multiple user system, a job pool will be created, that all user programs are entered into.

In order to make the OS simulator as realistic as possible, and have multiple users, initially the users will be simulated, meaning that simulated users will be used during the development and testing of the system. The simulated user is a process that runs concurrently with the OS and its processes. As it runs, it submits jobs to the job pool at

random times. It is the OS's responsibility to run the processes and handle any output and secondary storage activity generated by the simulated users. Anytime that a user submits a command to the system it is OS's responsibility to carry it out. For example, if a user wants to edit a program that is on secondary storage, that program will need to be retrieved from the disk, causing track and sector requests to be generated and so on and son forth. The part of the OS that manages secondary storage will manage these requests and carry them out.

In order create an OS simulation, the initial versions will employ the simplest structures and algorithms for each of the many components of the OS. For example, the memory management model will be a contiguous system initially; a paged system can be implemented and installed after the basic version of the system is up and running. The same goes for the secondary storage system. Since the disk will need to be simulated as a group of available blocks on simulated tracks, the initial controller will use a first com first serve control algorithm, etc. etc.

10.2 Users – Modeling and Interaction with the OS

The process used to simulate the users is essentially a loop that moves SSL programs from a jobs directory into the system job pool. They wait there for processing. The details of the logic for processing were reviewed in chapter 2. What is new here is that user modules need to be in communication with the OS. The easy way to view the interaction of users, whether simulated or not, with the OS is to cast it as a producer/consumer problem. In this case the users are the producers, and the OS is the consumer. An outline of the modules and structures relevant to user modelling is provided in the bullets below:

- User generator – This module will create an instance of the simulated user process. It will also make sure that there are never any more than some specified number of user modules running at once.
- User module – This module simulates a user. It is created by the user generator and runs for some random amount of time. During this time, it picks jobs from the universal job directory and submits them to the OS to be run.
- User Jobs Directory – This is a directory of SSL programs that the user modules draw programs from. The SSL programs can be the same ones created for the Batch OS simulator in section 1.5 or they can other SSL programs. Initially this will be a directory on whatever system the OS simulation is being developed upon. Later, however, this will be a directory that is managed by the simulated OS on the OS's simulated disk drive.
- OS Job Pool – This is the user partition for jobs that have been submitted by the users. After a job has been submitted, the OS "turn it into a process", as detailed in chapter 2, and then the process must be loaded into memory. The memory area, or user partition, forms the job pool. From this job pool, the OS's scheduler selects jobs and submits them to the CPU in order to run.

Secondary storage also plays a role regarding the structures outlined by the bullets above. It acts as a repository for temporary files and structures required to run the various jobs

submitted by the users. Specifically, any intermediate files generated by the compilation process will be stored in secondary storage; once the process is built, these files can be deleted. Figure 10.1 below illustrates the relationships between the various entities involved.

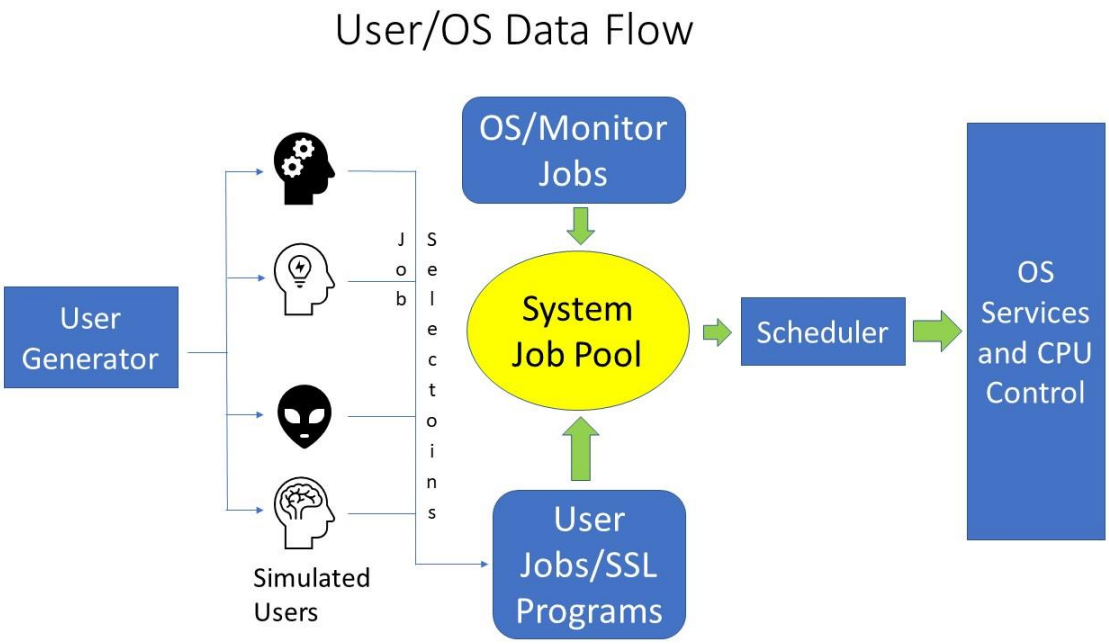


Figure 10.1 illustrates the relationships between the various modules and structures required to simulate multiple users in a simple OS simulation. In this diagram the users are represented by processes that select jobs from a job directory and submit them to the OS. The jobs enter the job pool after the OS has converted them to processes. They reside in the job pool, along with other OS and monitor submitted jobs until the Scheduler selects them to be processed. Once selected, they receive OS Services and CPU time slices until they complete.

Simulated User Pseudo Code

The pseudo code below implements a basic simulated user. This process is intended to run concurrently with other simulated users; its parent process is the User Generator. In order to bring "variety" to the users, each user simulator has an assigned number of jobs to run, with an average delay between them. The code can be written with other operational details; some users may select jobs that are more I/O oriented versus CPU intense.

The code below shuts itself down when it has submitted its prescribed number of jobs, or the system issues a shutdown order via a flag in shared memory. The "checkShutDownFlag()" function handles the details of this operation.

```
void simulatedUser(int nJobs, int averageDelay)
{
  bool running;
  int j;

  /* Initialize */
  jobList = getJobNames(jobDirectory);
  jobCount = length(jobList);

  j = 0;
  running = true;
  while(running) {

    myJob = random(0, jobCount-1);

    subMit(jobList[myJob]);

    delay(averageDelay);

    j++;

    /* Shutdown conditions. */
    if (j == nJobs) {
      running = false;
    }
    if (checkShutDownFlag() == true) {
      running = false;
    }
  }
}
```

User Generator

The User Generator is a piece of code/process that spawns simulated user processes, as described in the previous section. This process is part of the simulation and should be kicked off by the monitor program. Its responsibilities are:

- Spawn simulated users occasionally.
- Keep the number simulated users below a given threshold.
- Shutdown itself and any running simulated users when signaled.

Its logical structure is very similar to that of the simulated user – it is while loop that does its work in a controlled random manner and shuts down when signaled to do so. A sample of pseudo code for the User Generator process is shown below.

```
void userGenerator(int maxUsers)
{
  bool running;
  int nUsers;

  /* Initialize */
  nUsers = 0;
  running = true;
  while(running) {

    /* Keep nUsers relatively close to maxUsers. */
    if (nUsers < maxUsers) {
      r = getRandom(1, 100);
      if (r < 50) {
        nJobs = random(1, 20);
        spawn(simulatedUser(nJobs));
      }
    }

    /* Check to see how many users are still running. */
    nUsers = checkUserStatus();

    /* Shutdown conditions. */
    if (monitorShutdown() == true) {
      setFlags(userShutDown);
      running = false;
    }
  }
}
```

Appendix A – SSL Programs

The Super Simple Language (SSL) was developed in support of the OS simulator design/development project. The simulator and its development plays a huge part in the explanations provided in this book of OS's and OS concepts, and OS design. In order to test many of an OS's features (and the develop the simulator), a CPU emulator is needed. The CPU emulator needs to be able to process instructions; the source of those instructions are programs written in SSL.

This appendix holds 6 example programs that have been used to test the basic batch system, the twin fixed partition batch system, the Fixed Partition Multi-Programmed Batch System. These examples have no function calls, thus no recursion.

As a note for future reference, basic function calling in SSL has not been implemented, is not overly difficult to do. It can be done without adding a program stack, as long recursion is prohibited.

Below are listings of the 6 programs. The adopted file convention is as follows – "mySSLprogram.ssl".

Program 1

```
program average1to20Sum;

int sum, j;

int av;


sum = 0;

for j = 1 to 20;
      sum = sum + j;
next

av = sum/j;
print(av);

stop;
end;
```

Program 2

```
program average2;

int sum, j;
int sum2;
int av;
int bigAv

sum = 0;
bigAv = sum;

for j = 1 to 10;
      sum = sum + j;
      sum2 = j + j;
      bigAv = bigAv + sum2;
next

av = sum/j;
print(av);

stop;
end;
```

Program 3

```
program average2Print;

int sum, j;
int sum2;
int av;
int bigAv

sum = 0;
bigAv = sum;

for j = 1 to 10;
      sum = sum + j;
      sum2 = j + j;
      bigAv = bigAv + sum2;
      print(sum);
      print(sum2);
      print(bigAv);
next

av = sum/j;
print(av);

stop;
end;
```

Program 4

```
program average1000;

int sum, j, k;

int av;


sum = 1;

for j = 1 to 100;
      for k = 1 to 10;
            sum = sum + 1;
      next
next

av = sum/j;
print(sum);
print(j);
print(k);
print(av);

stop;
end;
```

Program 5

```
program loop2;

int sum, j, k;
int array[10];
int bigSum;

bigSum = 0;
sum = 0;

for j = 1 to 10;
      for k = 1 to 10;
            sum = sum + 1;
      next
next

if (sum >= 100) then
      bigSum = 10;
      for j = 1 to 10;
            bigSum = bigSum+1;
      next
endif

if (bigSum < 20) then
      array[0]=1;
      array[1]=2;
      array[2]=3;
      array[3]=4;
      array[4]=5;
```

```
      array[5]=6;
      array[6]=7;
      array[7]=8;
      array[8]=9;
      array[9]=10;
endif

print(bigSum);
```

Program 6

```
program loop2Print;

int sum, j, k;
int array[10];
int bigSum;

bigSum = 0;
sum = 0;

for j = 1 to 10;
      for k = 1 to 10;
            sum = sum + 1;
            print(sum);
      next
next

if (sum >= 100) then
      bigSum = 10;
      for j = 1 to 10;
            bigSum = bigSum+1;
            print(bigSum);
      next
endif

if (bigSum < 20) then
      array[0]=1;
      array[1]=2;
      array[2]=3;
      array[3]=4;
      array[4]=5;
      array[5]=6;
      array[6]=7;
      array[7]=8;
      array[8]=9;
      array[9]=10;
endif
```

Appendix B – Semaphores Example Using the Bakery Algorithm

The code below was developed in Microsoft Visual C++. It is a version of the producer/consumer problem, except this time there is an extra consumer, and a controller program. The 2nd consumer is there to test the semaphores in a system in which there are more than 2 processes needing access to the semaphores. That being said, the indexing of the items removed by the consumers is not ideal. Fixing this is left as an exercise to the student or curious computer scientist.

The controller program uses a semaphore to turn off the other 3 programs. The semaphore is only changed by the controller, and read by the other programs, so it is not a true 4 process test.

The semaphores are implemented using the Bakery Algorithm as discussed in 4.61. This version of the Bakery Algorithm is designed to use up to 10 processes, and does not use any delays in the entry code. It has been tested on Windows 10 and works well. However, it may not work as well on systems that do not distribute CPU slices evenly throughout running processes. Some older OS's such as Windows XP may encounter problems with this code – in the form of processes freezing.

A few programming notes:

- The code was developed in VC++, version 6.
- This example is housed in 4 files.
    - producerMain.cpp – contains the main programs for the producer process, the two consumer processes, and the controller process.
    - semaphoreSubs.cpp – contains the code for creating shared memory based semaphores. This code uses a modified version of the bakery algorithm, as discussed in chapter 4.64.
    - shareSubs.cpp – contains shared memory utility programs that help in creating and connecting to pieces of shared memory.
    - structsAndClasses.h – contains the structure template for the homegrown multi-process semaphores.
- This code used several #define statements along with #if compiler directives. The code for the 4 processes is all in the producerMain.cpp file. The user can build any of the 4 processes by setting the appropriate #define.
    - Once a process is built, the programmer/user should go to the /debug directory and rename the executable appropriately. For this project the following generic names were used:
        - producerSemaphoreP0
        - consumerSemaphoreP1
        - consumerSemaphoreP2
        - controllerSemaphore

- Once the 4 processes are built, start them, in this order listed above.
- Once the processes are running, hit the enter key on the producer process. In order to turn off the processes, hit any key in the controller process.
- There is a deadlock in this code. It can be found by removing code lines in the producer process that allow it to produce 5 more items after the shutdown signal has been sent.

File: stuctsAndClasses.h

```
struct semaphore {
  int id;
  int value;

  /* Bakery critical section variables. */
  bool flags[10];
  bool turn[10];
};
```

File: producerMain.cpp

```
#include <stdio.h>
#include <windows.h>
#include "structsAndClasses.h"

#define buffSize 128

char* createShare(int nElements, int elementSize, char *shareName);
char* connect2Share(int nElements, int elementSize, char *shareName);

void waitBake(semaphore *S, int i, int n);
void signalBake(semaphore *S, int i, int n);

void setFlags(bool *flags, bool value, int n);

#define _Controller 0
#define _Producer 1
#define _Consumer1 0
#define _Consumer2 0

#if _Controller
void main(void)
{
  semaphore *running;
  char myKey;
```

```
  /* Message user that program is coming online. */
  printf("Hello TV Land! I am the Controller! \n");

  /* Create a shared piece of memory for semaphore to live. */
  running = (struct semaphore *)createShare(1, sizeof(semaphore), "myShare");

  running->id = 99;
  running->value = 1;
  setFlags(running->flags, false, 4);
  setFlags(running->turn, false, 4);

  printf("Start the producer, and the two consumers. \n\n");
  printf("Hit enter key to turn off the system. \n");

  scanf("%c", &myKey);

  /* Set semphore value to 0, with call to wait(). */
  waitBake(running, 0, 4);

  Sleep(2000);

  printf("Goodbye \n");
}
#endif


#if _Producer

void main(void)
{
  semaphore *running;
  semaphore *mutex, *empty, *full;
  char *buffer;
  char myKey;

  /* Message user that program is coming online. */
  printf("Hello TV Land! I am the Producer! \n");

  /* Create a shared piece of memory for semaphore to live. */
  running = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare");

  /* Create a shared piece of memory for semaphore to live. */
  mutex = (struct semaphore *)createShare(1, sizeof(semaphore), "myShare0");
  empty = (struct semaphore *)createShare(1, sizeof(semaphore), "myShare1");
```

```
full  = (struct semaphore *)createShare(1, sizeof(semaphore), "myShare2");

buffer = createShare(buffSize, sizeof(char), "myShare3");

printf("semaphore size = %d \n", sizeof(semaphore));
printf("mutex = %d, emtpty = %d, full = %d buffer = %d \n",
      mutex, empty, full, buffer);

mutex->id = 100;
mutex->value = 1;
setFlags(mutex->flags, false, 3);
setFlags(mutex->turn, false, 3);


empty->id = 101;
empty->value = buffSize;
setFlags(empty->flags, false, 3);
setFlags(empty->turn, false, 3);


full->id = 102;
full->value = 0;
setFlags(full->flags, false, 3);
setFlags(full->turn, false, 3);


printf("Please Start Consumer Process.  When started hit enter key. \n");
scanf("%c", &myKey);

/* Producer Process. */
{
  int j;
  int count = 5;
  char c;
  bool stillProducing = true;

  c = (char)33;

  j = 0;
  while(stillProducing) {
    if (running->value == 0) {
      count--;
    }
    if (count == 0) {
      stillProducing = false;
```

```
      }

      waitBake(empty, 0, 3);
      waitBake(mutex, 0, 3);

      buffer[j] = c;
      j = (j+1)%buffSize;

      c = c + 1;

      if (c == 127) {
        c = 33;
      }

      signalBake(mutex, 0, 3);
      signalBake(full, 0, 3);

      Sleep(500);
    }
  }

  Sleep(2000);
  printf("Producer says goodbye!. \n");

}

#endif


#if _Consumer1

void main(void)
{
  semaphore *running;
  semaphore *mutex, *empty, *full;
  char *buffer;

  /* Message user that program is coming online. */
  printf("Hello TV Land! I am the Consumer \n");

  /* Connect to a shared piece of memory for semaphore to live. */
  running = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare");

  /* Connect to a shared piece of memory for semaphores. */
  mutex = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare0");
```

```
  empty = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare1");
  full  = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare2");

  /* Semaphores initialized by producer. */

  buffer = connect2Share(buffSize, sizeof(char), "myShare3");

  printf("semaphore size = %d \n", sizeof(semaphore));
  printf("mutex = %d, emtpty = %d, full = %d buffer = %d \n",
        mutex, empty, full, buffer);


  /* Consumer Process. */
  {
    int j;
    char c;

    c = (char)33;

    j = 0;
    while(running->value == 1) {
      waitBake(full, 1, 3);
      waitBake(mutex, 1, 3);

      c = buffer[j];
      j = (j+1)%buffSize;


      printf("%c ", c);

      signalBake(mutex, 1, 3);
      signalBake(empty, 1, 3);

    }
  }

  printf("Consumer 1 says goodbye! \n");

}

#endif


#if _Consumer2
```

```
void main(void)
{
  semaphore *running;
  semaphore *mutex, *empty, *full;
  char *buffer;

  /* Message user that program is coming online. */
  printf("Hello TV Land! I am the Consumer \n");

  /* Connect to a shared piece of memory for semaphore to live. */
  running = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare");

  /* Connect to a shared piece of memory for semaphores. */
  mutex = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare0");
  empty = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare1");
  full  = (struct semaphore *)connect2Share(1, sizeof(semaphore), "myShare2");

  buffer = connect2Share(buffSize, sizeof(char), "myShare3");

  printf("semaphore size = %d \n", sizeof(semaphore));
  printf("mutex = %d, emtpty = %d, full = %d buffer = %d \n",
        mutex, empty, full, buffer);

  /* Semaphores initialized by producer. */


  /* Consumer Process. */
  {
    int j;
    char c;

    c = (char)33;

    j = 0;
    while(running->value == 1) {
      waitBake(full, 2, 3);
      waitBake(mutex, 2, 3);

      c = buffer[j];
      j = (j+1)%buffSize;

      printf("%c%c", c, c);

      signalBake(mutex, 2, 3);
      signalBake(empty, 2, 3);
```

```
    }
  }

  printf("Consumer 2 says goodbye! \n");

}

#endif

void setFlags(bool *flags, bool value, int n)
{
  int j;

  for(j=0;j<n;j++) {
    flags[j] = value;
  }
}
```

File: semaphoreSubs.cpp

```
#include <stdio.h>
#include "structsAndClasses.h"

/* Utility Routines. */
bool checkFlags(bool *flags, int i, int n);

void waitBake(semaphore *S, int i, int n)
{
  int time = 0;
  int j;

  /* Wait on value.*/
  while(S->value <= 0) {
    time++;
  }

  /* Change Value. */

  /* Entry Code. */
  /* Process 0: I want to in. */
  S->flags[i] = true;

  /* I defer to you guys. */
  for(j=0;j<n;j++){
```

```
    if (j == i) {
      S->turn[j] = false;
    }
    else {
      S->turn[j] = true;
    }
  }


  /* Busy wait to get access to value of S. */
  while((checkFlags(S->flags, i, n) == true) &&
      (S->turn[i] == false)) {
    time++;
  }

  /* Critical Section. */
  S->value = S->value - 1;

  /* Exit Code. */
  S->flags[i] = false;
}


void signalBake(semaphore *S, int i, int n)
{
  int time = 0;
  int j;

  /* Change Value. */

  /* Entry Code. */
  /* Process 0: I want to in. */
  S->flags[i] = true;

  /* I defer to you guys. */
  for(j=0;j<n;j++){
    if (j == i) {
      S->turn[j] = false;
    }
    else {
      S->turn[j] = true;
    }
  }
```

```
  /* Busy wait to get access to value of S. */
  while((checkFlags(S->flags, i, n) == true) &&
      (S->turn[i] == false)) {
    time++;
  }

  /* Critical Section. */
  S->value = S->value + 1;

  /* Exit Code. */
  S->flags[i] = false;
}


bool checkFlags(bool *flags, int i, int n)
{
  int j;
  bool atLeast1isTrue = false;

  for(j=0;j<n;j++) {
    if ((j != i) && (flags[j] == true)) {
      atLeast1isTrue = true;
    }
  }

  return atLeast1isTrue;
}
```

File: shareSubs.cpp

```
#include <stdio.h>
#include <windows.h>

#define _Null 0


char* createShare(int nElements, int elementSize, char *shareName)
{
  HANDLE hIntArray;
  char *p;


  /* Make the shared file. */
    hIntArray = CreateFileMapping(INVALID_HANDLE_VALUE,
```

```
                                    _Null,
                                    PAGE_READWRITE,
                                    0, elementSize * nElements,
                                    shareName);

    /* Check to see if it was created correctly. */
    if(hIntArray == _Null || hIntArray == INVALID_HANDLE_VALUE)
    {
        printf("Could not create file mapping object (%d). \n", GetLastError());
        hIntArray = _Null;
        exit(0);
    }

    /* Map an adress to the shared file. */
    p = (char *) MapViewOfFile(hIntArray,
                FILE_MAP_ALL_ACCESS,
                0,
                0,
                elementSize * nElements);

  printf("p = %p \n", p);

    /* Make sure that the mapping was successful. */
    if(p == _Null)
    {
        printf("Could not map view of file (%d). \n", GetLastError());
        exit(0);
    }

  return p;
}


char* connect2Share(int nElements, int elementSize, char *shareName)
{
  HANDLE hIntArray;
  char *p;


  /* Make the shared file. */
    hIntArray = hIntArray = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, shareName);

    /* Check to see if it was created correctly. */
    if(hIntArray == _Null || hIntArray == INVALID_HANDLE_VALUE)
    {
```

```
        printf("Could not create file mapping object (%d). \n", GetLastError());
        hIntArray = _Null;
        exit(0);
    }

    /* Map an adress to the shared file. */
    p = (char *) MapViewOfFile(hIntArray,
                FILE_MAP_ALL_ACCESS,
                0,
                0,
                elementSize * nElements);

  printf("p = %p \n", p);

    /* Make sure that the mapping was successful. */
    if(p == _Null)
    {
        printf("Could not map view of file (%d). \n", GetLastError());
        exit(0);
    }

  return p;
}
```

References

1. MIPS Register Definitions: https://www.cs.fsu.edu/~hawkes/cda3101lects/chap3/index.html?$$$F3.13.html$$$
2. MIPS Instruction Reference: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
3. Silberschatz
4. Stack Overflow - https://stackoverflow.com/questions/1157209/is-there-an-alternative-sleep-function-in-c-to-milliseconds