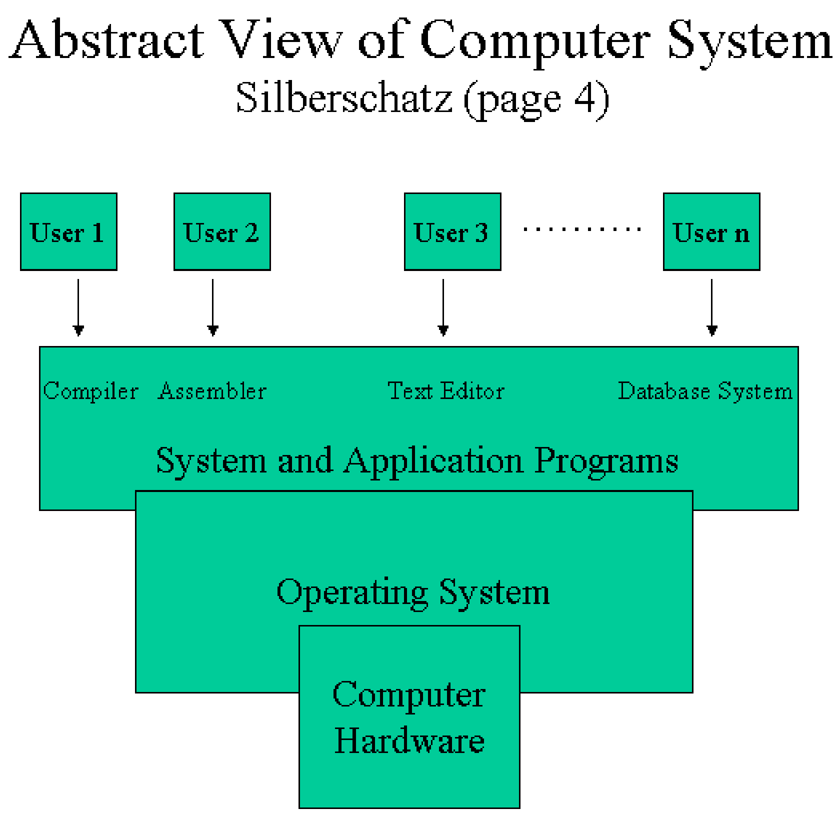
1.0 CMPS 431 (Operating Systems) Course Notes

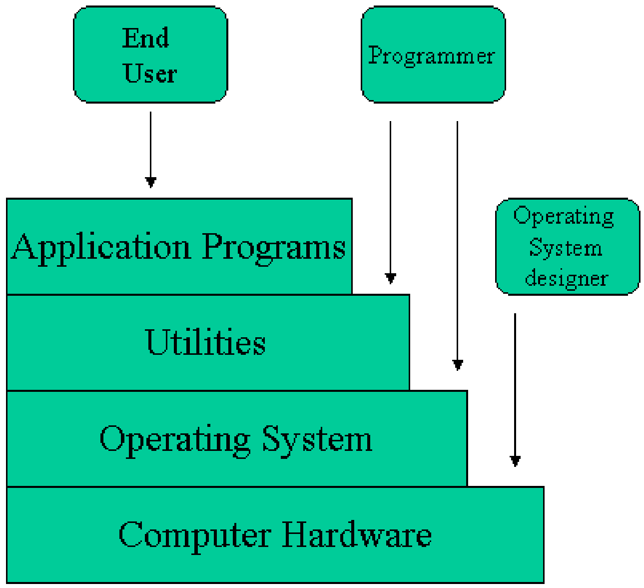
1. Overview of Operating Systems
   a. What is an OS – It is a control program that provides an interface between the computer hardware and the user. Part of this interface includes tools and services for the user.

   From Silberschatz (page 3): "An operating system is a program that acts as an intermediary between a user of computer and computer hardware. The purpose of the OS is 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."



Computer Hardware – CPU, memory, I/O devices provide basic computing resources.
System and Application Programs – Compilers, database systems, games, business programs, etc. define the ways the computing resources are used to solve the users problems.
Operating System – Controls and coordinates the computing resources among the system and application programs for the users.

Layer and Views of a Computer System
(Stallings, page 53)



End User – Views the computer system as a set of applications. The End User is generally not concerned with various details of the hardware.

Programmer – Uses languages, utilities (frequently used functions) and OS services (linkers, assemblers, etc.) to develop applications instead. This method is used to reduce complexity by abstracting the detail of machine dependant calls into APIs and various utilities and OS services.

OS – Masks the hardware details from the programmer and provides an interface to the system. Manages the computers resources. The OS designer has to be familiar with user requirements and hardware details.

b. OS attributes
    i. Convenience – make the computer easy to use
    ii. Efficiency – manage computer resources in an efficient manner
    iii. Ability to Evolve – An OS should be able to integrate new system functions and additions/modifications without interfering with service or over burdening users.

c. OS can be thought of as:
    i. Control Program and Resource Allocator – The OS is a program, like other programs in the system. Like other programs, it made of instructions, but its instructions serve the purpose to allocate resources (CPU, memory, disk storage, i/o) so that other programs can operate. To do this it must stop itself from running and let other programs run. When the other program's turn to run is over, the OS runs long enough to prepare the resources for the next process to run, and so on.

    ii. User to Computer Interface – Provides an friendly environment from which user can accomplish their goals.

d. OS from the viewpoint that it is User/Computer Interface – The OS acts as an intermediary between the Users/Programmers and the hardware, making it easier for users, programmers, and applications to access the OS's facilities, services (facilities can be thought of as the system's resources, services are the methods that the OS provides to use the facilities). Services are typically provided in the following areas:
    i. Program Development – editors, debuggers, compilers, etc.
    ii. Program Execution – loaders, linkers, system protection.
    iii. Access to I/O devices – Provides a uniform interface, usually simple reads and writes, that hides the details of i/o device operation.
    iv. File access and protection – Must be able to manage storage devices, access data in file, and provides access control to files.
    v. Error detection – Must be able to gracefully handle various errors, such as memory access violations, divide by zero, device errors, etc.
    vi. System Logging – log important system events for system tuning, error correction and/or billing information.

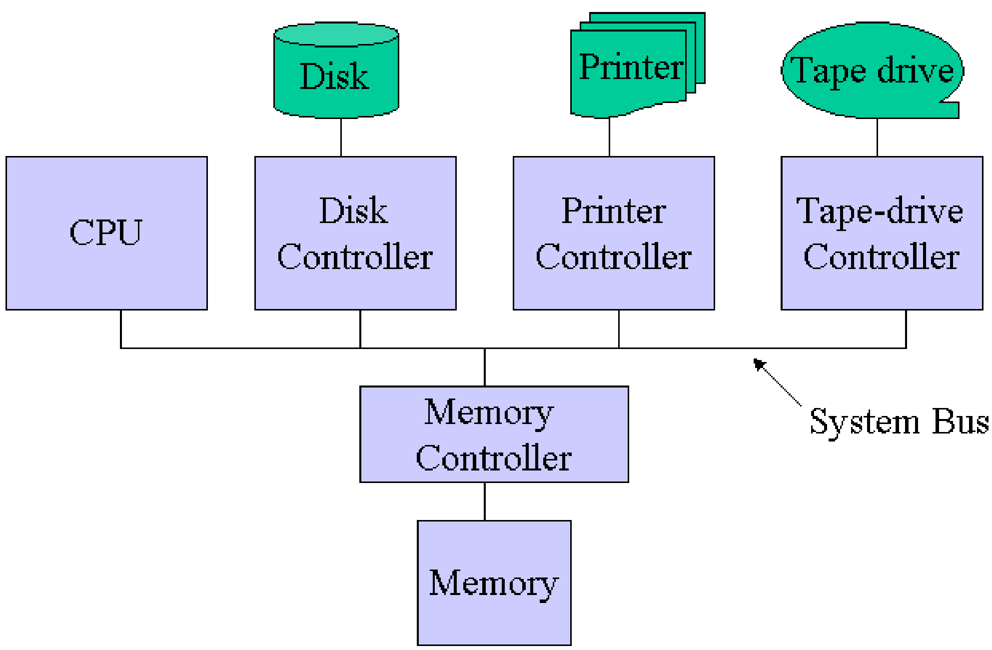
e. Brief history (Wikipedia)
    i. The first computers did not have an OS but programs for managing the system and using the hardware quickly appeared.
    ii. By the early 1960s, commercial computer vendors, such as UNIVAC and Control Data Corporation, were supplying quite extensive tools for streamlining the development, scheduling, and execution of jobs on batch processing systems.
    iii. In the 1960s IBM System/360 OS 360 was developed to run on a whole line of computers. It was a first, an OS for several different machines in IBM's product line. Features included:
        1. Hard disk storage development.
        2. Time sharing environment. Time sharing provided users the illusion of having the whole machine to themselves.
    iv. Multics was another well known OS that used the time sharing concepts. It inspired several OS's including Unix and VMS.
    v. The first microcomputers (predecessors to PCs) did not require/use most of the advanced features used for mainframes and mini computers.
    vi. CP/M (Control Program/Monitor) was created by Digital's Gary Kildall for Intel 8080/8085 and Zilog Z80 processors in about 1974. It is the predecessor for IBM's PC DOS and MS DOS. DOS's major contribution was its FAT file system.
    vii. In the 1980's DOS dominated the Intel based PC's while the Macintosh operating system, patterned after XEROX corporations early window based document editing operating systems, provided competition on the Apple platform.

f. Computer System Structures – A modern computer consists of:

i. CPU – Central Processing Unit. Is responsible for execution of arithmetic, logical, data transfer, and control operations.
ii. Device Controllers
    1. disk drive controller
    2. audio device controller
    3. video controller
    4. etc.
iii. System Bus – Serves as a communication channel between the various system components.
iv. Memory – Storage of instructions and data for system and user processes.

- The CPU and device controllers can run concurrently.
- A memory controller synchronizes the CPU's and device controller's access to memory.

Modern Computer System

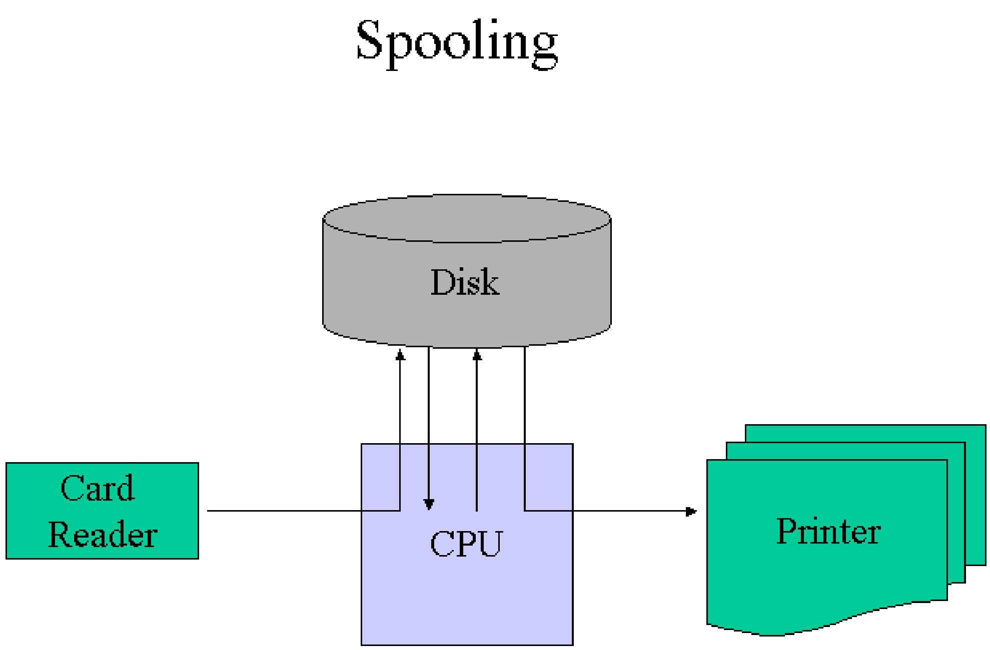


- Upon power up:
    - Bootstrap program initializes all system aspects.
        - CPU Registers
        - Device Controllers
        - Memory Contents
    - Loads OS and starts its execution.
        - Locates OS in storage (disk).

        - Loads OS kernel (basic OS functions)
        - Begins OS function (init, or monitor)
- Storage Devices:
    - Main Memory – relatively small in size (32k to 4gig), relatively fast access speed, random access, volatile. The only large storage area that the CPU can directly access. The other memory areas that the CPU can access are:
        - Registers (typically 256 bytes) really fast
        - Cache (64k – 256k) fast
    - Disk – Large in size, medium access speed, random access, non-volatile
        - Electronic disk
        - Magnetic disk
        - Optical drive
    - Magnetic Tape – Really Large, low speed, sequential access, non-volatile

g. Operating System Components (Chapter 3, Silberschatz) – Main components are process, memory, file, I/O system, and secondary storage management.
   i. Process Management responsibilities.
      1. Creation and Deletion of user and system processes.
      2. Suspension and resumption of processes.
      3. Provision of mechanisms for process synchronization.
      4. Provision of mechanisms for process communication.
      5. Provision of mechanisms for deadlock handling.
   ii. Main Memory Management responsibilities
      1. Keep track of which parts of memory are being used and by what processes.
      2. Decide which processes are to be loaded into memory when memory space becomes available.
      3. Allocate and de-allocate memory as needed.
   iii. File Management responsibilities
      1. Creation and deletion of files.
      2. Creation and deletion of directories.
      3. The support of primitives for manipulating files and directories.
      4. Mapping of files onto secondary storage.
      5. Backup of files onto stable storage media.
   iv. I/O System Management – hides the peculiarities of specific hardware devices from the user. The sub-system (in UNIX) consists of:
      1. Memory management component including buffering, caching and spooling (**spooling** refers to putting jobs in a buffer, a special area in memory, or on a disk where a device can access them when it is ready. **Spool** is an acronym for **simultaneous peripheral operations on-line**. Early mainframe computers had relatively small and expensive (by current standards) hard

disks. These costs made it necessary to reserve the disks for files that *required* random access, while writing large sequential files to reels of tape. Typical programs would run for hours and produce hundreds or thousands of pages of printed output. Periodically the program would stop printing for a while to do a lengthy search or sort. But it was desirable to keep the printer(s) running continuously. Thus when a program was running, it would write the printable file to a **spool** of tape, which would later be read back in by the program controlling the printer. Wikipedia.org)



2. A general device driver interface.
3. Drivers for specific hardware devices. Only the device driver knows the operation specifics of the device to which it is assigned.

v. Secondary Storage Management
   1. Free-space management.
   2. Storage allocation
   3. Disk scheduling

h. First Operating Systems
   i. Simple Batch (One Job at a time)
      1. Batch refers to an early type of processing in which the user would submit a job (program, data, and control information about the nature of the job (usually on cards)) to an operator.
         a. The operator would group the jobs into like "batches" (for example, all FORTRAN programs may be grouped

together, saving load time for the compiler) and put them into the card reader.

b. A program called a "monitor" signals the card reader to read a job. Once the job is loaded, the monitor hands execution over to the job. When job completes, the monitor takes over and process repeats.
    i. A monitor that always is in main memory is called a "resident monitor".

c. The output would be spooled to a printer. If the program failed, error messages from the compiler or a memory dump would become the output.

2. A definitive feature of Batch Processing is the lack of interaction between the user and program while the program is executing.
3. Jobs are usually run one at a time, first come first serve.
4. The language that instructs the monitor on the particulars (language, tape mounting, etc) is called Job Control Language (JCL).
5. The delay between job submission and completion is called *turnaround* time.
6. Because the CPU is orders of magnitude faster than the card readers and printers, there is lots of CPU idle time.
7. Two execution modes are often used to aid in system protection:
    a. User mode: The application program run in this mode. It has access to a subset of the systems commands and memory space. Certain areas of memory (the area in which the monitor resides) are off limits.
    b. Kernel mode: Sometimes called privileged mode. The monitor has access to all instructions and memory areas of the system.

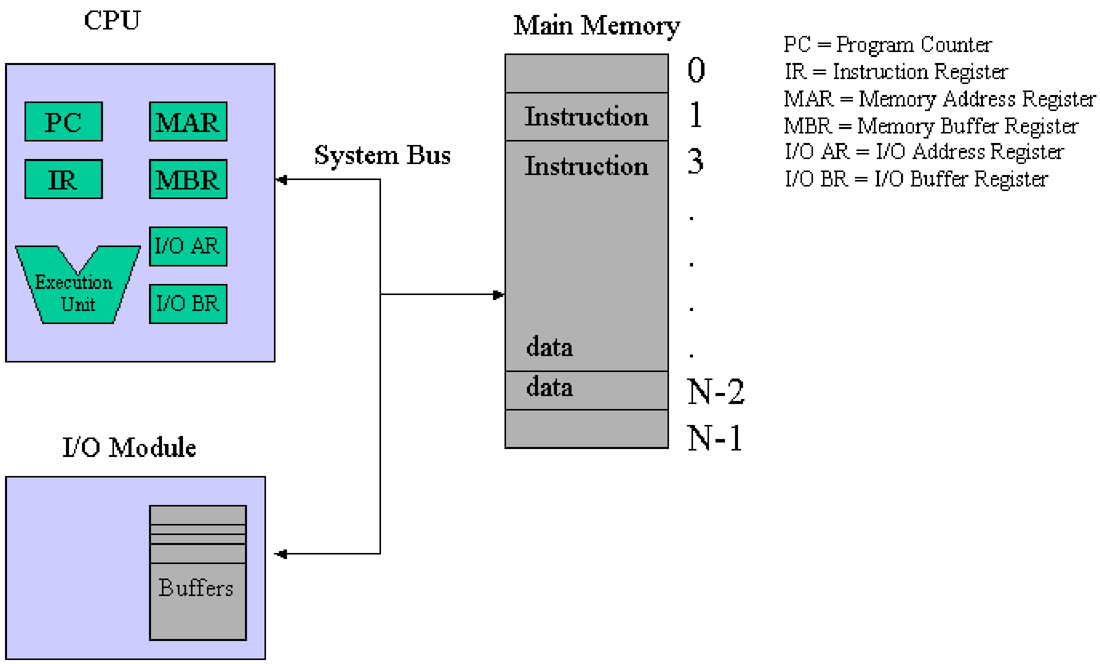
ii. Multi-programmed batch – Similar to simple batch except that the computer executes jobs from a pool of jobs that have been loaded into the memory. The advantage is that CPU idle time is reduced, and I/O devices do not sit idle.

1. Jobs are loaded continually from cards into the job pool (in memory).
2. The OS selects a job to be run.
3. When the job has to wait for some task (such as a tape to be mounted or an I/O operation (card read, or print result) the CPU switches execution to another job from the job pool.
4. When that job has to wait, the CPU is switched to another job and so on. When each job finishes its waiting time, it eventually gets the CPU back. In this way CPU idle time is reduced.
5. This scheme introduces two new concepts:

        a. Memory management
        b. CPU scheduling
    iii. Timesharing (Multi-tasking) – 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.
        1. Introduces user/computer interaction.
        2. Interaction helps in:
            a. Finding compiler time errors.
            b. File editing.
            c. Debugging code while it is running.
            d. Execution of multi-step jobs in which later jobs depend on results of earlier jobs.
        3. Multiple jobs are executed by the CPU switching between them (a switch from one job to another is referred to as "context switch"), but the switches occur so frequently that the users may interact with the programs as they run. CPU is "time multiplexed between the users".
        4. Requires:
            a. User accessible on-line file system.
            b. Directory system
            c. CPU scheduling
            d. Multiprogramming.
        5. Commonly uses:
            a. Virtual memory (a technique that allows execution of a job that may not be completely in memory).

2. Background and Basics
    a. Computer System review
        i. Architecture (Stallings, page 9)
            1. Processor – Carries out the operation of the computer. The processor consists of at least one Arithmetic Logic Unit (ALU) and several registers. The registers are used for addressing, comparisons, arithmetic operations etc.
            2. Main Memory – Stores data and programs. Main memory is usually volatile (when power is shut down, contents are lost).
            3. I/O Modules – Moves data between system and external environment (secondary memory devices - disks, terminals, etc)
            4. System bus – Provides communication among system components.

Computer Components (Top Level View)



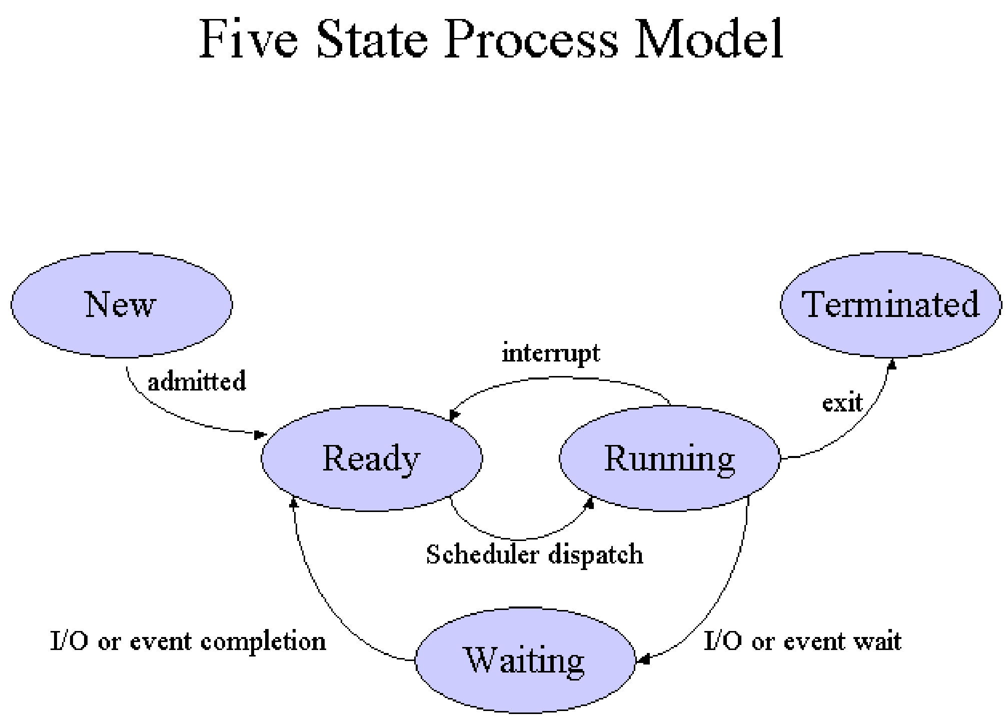
Definitions (expressed using C notation)
PC – Program Counter, contains the address of the next instruction to be fetched from memory.
IR – Instruction Register, contains the instruction most recently fetched.
MAR ≅ &memory_buffer /* Address of buffer location for next read or write. */
MBR ≅ (&memory_buffer) **/*** Contains data to be written or received from memory. ***/**
I/O AR – Denotes I/O device
I/O BR – Address in I/O buffer of data to be moved

ii. Instruction cycle
  1. A program is a list of instructions.
  2. To execute a program the instructions are executed sequentially (usually) from start to end.
  3. The instruction execution process has two steps.
     a. Fetch stage – The instruction at the location pointed to by the PC is loaded in the IR.
     b. Execute stage – The instruction in the IR is executed by the CPU. In general, there are four categories of instructions:
        i. Processor to Memory transfer
        ii. Processor to I/O transfer
        iii. Data Processing – An arithmetic or logical operation is performed on data

iv. Control – The sequence of execution is altered. For example a “jump command”.

iii. Process Control Block – Every program that is executing has a Process Control Block (PCB) associated with it. The PCB is a block of data that OS uses to keep track of the status of the running program. It contains process state information, execution scheduling data, memory management information, etc.

3. Processes
   a. Definition – A computer program is a passive entity that is stored in file. When the program is being executed, it is called a “process”. A process is more than the code being executed; it also includes information such as:
      i. The current part of the program being executed.
      ii. CPU register data, program stack data (temporary data used by the program including subroutine parameter, return addresses and temporary variables)
      iii. A global data section.
   b. Process States
      i. 5 state model



New – The process is being created. PCB is being created and initialized.
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).
Ready – The process is waiting for the CPU.
Terminated – The process has finished execution.

c. Process structure – The OS keeps track of each process by maintaining a data structure called a Process Control Block (PCB).
    i. PCB and components (typical) (Stallings, page 111)
        1. Process ID (PID) – unique identifier
        2. Process State – New, running, waiting, ready, or terminated.
        3. Priority – Priority level in relation to other processes.
        4. Program Counter (PC) – Address of the next instruction to be executed in the process.
        5. Context data – Current register values
        6. I/O status information – Outstanding I/O requests, I/O devices, open files used by the process, etc.
        7. Accounting Information – CPU time, elapsed time, run time limit, etc.
        8. Other information commonly included:
            a. Memory Information – Base and Limit registers, page table data.
            b. Scheduling information – Pointers to scheduling queues.

d. Operations on Processes
    i. Creation
        1. Assign unique PID
        2. Allocate memory for the process. Can use defaults for the type of process, or use specific numbers as requested by the process.
            a. Code space
            b. Stack space
        3. Initialize the PCB – Initialize the PCB values.
        4. Put process in appropriate queue – Most new processes are put into the new process queue (or Ready/Suspended queue).
        5. Initialize Process Accounting Parameters – Update system log files.
    ii. Process Spawning – Creation of a process by another process.
        1. Parent – Creating process.
        2. Child – Newly created process. It will need resources (CPU time, memory, files, I/O devices). They come from either the parent process or the OS.
        3. Execution of parent and child can take several forms:
            a. Concurrent execution of parent and child.
            b. Parent waits until some or all of its child processes complete.
        4. Nature of parent/child:
            a. Child process is copy of parent.
                i. UNIX fork call can be used. Child is identical except that the return code for the parent is 0 and the return code for the child is nonzero.
                ii. Example of Unix fork:

```
Pid = fork();
If (Pid < 0) {
        Printf("Child process creation failed. \n");
}
else if (Pid == 0) {
        /* Do parent operations. */
}
else {
        /* Do child operations. */
}
```

iii. Deletion (Termination)
    1. Normal Termination - Process terminates when last instruction is has completed. A call to exit() requests that the OS deletes the process.
    2. Parent can terminate child using abort call for the following reasons:
        a. Child exceeded use of its resources.
        b. Child's task is no longer needed.
        c. Parent is exiting, many OS's do not allow child process to continue after parent terminates.

e. Threads – Sometimes called a "lightweight process" or LWP. It has its own:
    i. Program Counter (PC)
    ii. Register set
    iii. Stack space
f. Lightweight processes shares with peer threads.
    i. Code section
    ii. Data section
    iii. OS resource such as open files and signals
g. Threads are used when threads work on related jobs. It can be more efficient to use multiple threads, for example providing data to multiple remote machines on a network. Having one process with multiple threads, each executing the same code, but having different data and file sections is more efficient than multiple heavy weight processes.

4. CPU Scheduling
    a. CPU - I/O burst cycle – Process execution alternates between periods of CPU use and I/O waiting. These phases are called CPU burst and I/O burst. A process starts with a CPU burst and then alternates between I/O burst and CPU burst until completion with a CPU burst, a system request for termination of the process. The amount of CPU burst and I/O burst vary greatly from process to process. Some examples are below.
        i. Database program – I/O intense (disk)
        ii. Simulation – CPU intense (calculation of mathematical models)
        iii. Picture Rendering – CPU intense
        iv. Excel – I/O intense (mostly waits on user to enter data)
        v. Games – both CPU intense (rendering of scenes), and I/O intense (display of scenes, and user inputs)

b. Context Switching - Changing of process states. The more context switching, the higher the overhead, meaning less application work is being accomplished. Context switching involves:
    i. Saving the state of the running process in its PCB.
    ii. Loading a saved process.
c. Scheduling
    i. Short Term – When the CPU becomes idle, a job from the Ready Queue is selected to run. This is the job of the short term scheduler.
        1. Operates very frequently.
        2. Could be invoked at a rate of 10 times a second.
        3. Because it operates often it must be fast.
        4. Dispatcher. It is the dispatcher's job to give control of the CPU to the module selected by the Short Term Scheduler. Dispatching involves:
            a. Switching Context.
            b. Switching to user mode.
            c. Jumping the proper location in the program (identified in the PCB) and restarting the program.
    ii. Long Term – It is the job of the long term scheduler to select jobs from the job pool and load them into memory for execution. 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.
        1. Operates less frequently.
        2. Could minutes between calls.
        3. Because it is called infrequently, it can take time to decide which processes should be selected for execution.
        4. 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.
    iii. Preemptive – Scheduler does not wait for an I/O request or an interrupt to move a job from the running queue to the ready queue.
        1. 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.
d. Scheduling Criteria – Different scheduling algorithms try to optimize the following criteria to different degrees:
    i. CPU Utilization – Keep the CPU as busy as possible. In a real system a light load is about 40%, heavy 90%.
    ii. 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.
    iii. 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.

iv. 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.

v. Response time – The amount of time that from submission of the job to its first response to the user. Good response time is important in time sharing systems.

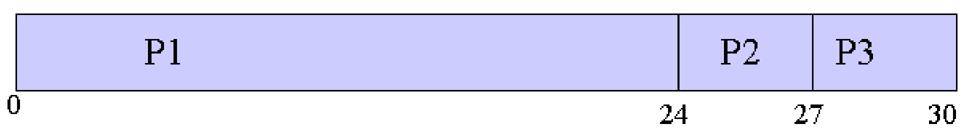
e. Algorithms

i. First Come First Serve – Processes are served in the order they are received. Can be implemented with a FIFO queue.

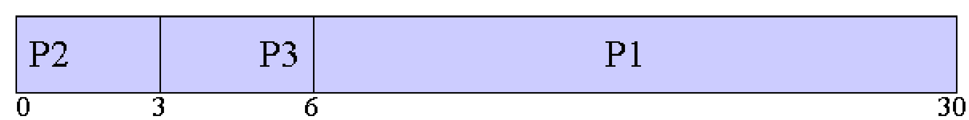
1. Waiting time is highly dependent on order of jobs and can be long.
2. Non-Preemptive
3. CPU intense jobs can cause all I/O bound jobs to wait for them to complete. As I/O bound jobs wait for I/O they all move from the waiting queue to the ready queue and the CPU intense job dominates the CPU.

First Come, First Served Scheduling

Process	Burst Time	Wait Time
P1	24	0
P2	3	24
P3	3	27



Average Wait = 17; (0+24+27)/3



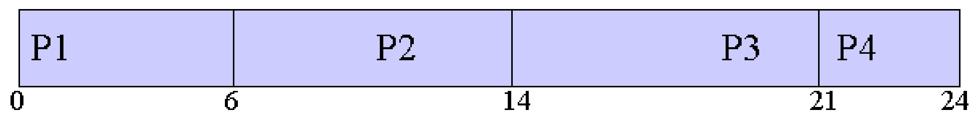
Average Wait = 3; (0+3+6)/3

ii. Shortest Job First – The job in the ready queue with the shortest next CPU burst is selected. If two jobs have equal next CPU burst, FCFS breaks the tie.

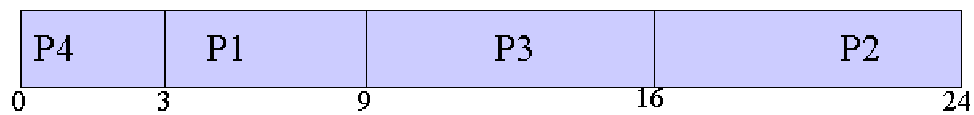
1. Algorithm is provably optimal. It gives the minimum waiting time for a set of processes.

FCFS vs. Shortest Job First

Process	Burst Time	Wait Time FCFS	Wait Time SJF
P1	6	0	3
P2	8	6	16
P3	7	14	9
P4	3	21	0



Average Wait = 10.25; (0 + 6 + 14 + 21)/4

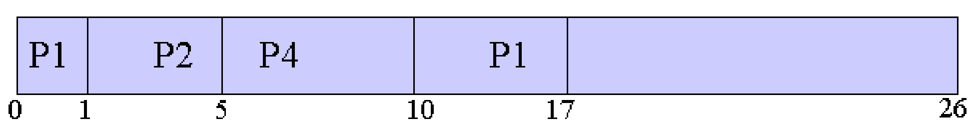


Average Wait SJF = 7; (0 + 3 + 9 + 16)/4

2. The major drawback is knowing what the length of the next CPU burst for each process will be.
    a. No practical way to know its value.
    b. Value can be predicted; could be similar in length to previous bursts.
3. SJF can be preemptive or non-preemptive.

SJF Preemptive

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



Average Wait = 6.5; (P1(0) + P2(0) + P4(2) + P1(9) +P3(15))/4

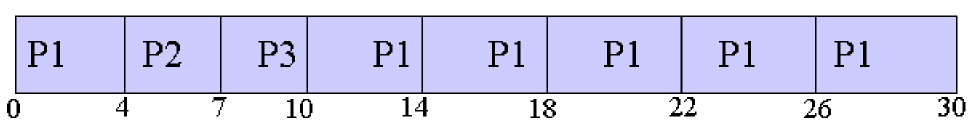
iii. Priority Scheduling – Instead of looking at the order in which jobs arrive and/or their estimated CPU bursts, jobs run in order of importance (priority).
    1. Jobs of equal priority are scheduled FCFS
    2. SJF is a special case of priority scheduling, in which the priority is the inverse of the estimated CPU burst.
    3. Priority scheduling can be preemptive or non-preemptive.
    4. Suffers from indefinite blocking or starvation. A low priority job can be caused to wait indefinitely by higher priority jobs.
        a. From Silberschatz page 134 “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.”
        b. “Aging” is a process in which the priority of processes that have been waiting for a long time is increased.

iv. Round Robin – FCFS algorithm in which each process in the ready queue is preempted after a short time (time quantum).
    1. The ready queue is treated as a circular queue.
    2. The scheduler goes around the ready queue allocating the CPU each process for a maximum of 1 time quantum.
    3. Average waiting time can be long.
    4. Used in time sharing systems where response time is important (needs to be short).

Round Robin

Process	Burst Time
P1	24
P2	3
P3	3

Time Quantum length = 4



f. Assignment 1 – Program a scheduler, implement FCFS, SJF, and Round Robin (Due in 3 weeks).

5. Quiz 1 Review – Review of sections 1 through 4 (Week 4)
6. Quiz 1
7. Process Synchronization - What methods are available for coordinating cooperating processes that share resources, such as memory, so that they execute correctly every time?

a. Instructions from the processes are interleaved in an unpredictable way.

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

b. Critical Section Problem – 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. (Mutual exclusion).

Critical Section Illustration

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

Count++                           Count—
P1     load count r1              C1     load count r1
P2     add r1  1                  C2     add r1  –1
P3     store r1  count            C3     store r1  count
```

CPU Instruction Stream (Error)
P1
C1
C2
C3
P2
P3

In the code illustration above a context switch from the producer to the consumer is executed while the producer is executing the count increment instruction. As a result an item is lost (the producer will write over it) because the count variable is decremented and incremented without any action data transfer actions occurring.

c. Solutions center around the following strategies:
   i. Increment/Decrement operations are atomic (cannot be broken down into a series of instructions)
   ii. Protect the increment and decrement calls in critical sections.
      1. Software solution (General critical section solution)
      2. Hardware solution (Test and set, semaphores, etc.)

d. To solve the critical section problem, the following requirements must be satisfied:
   i. Mutual Exclusion – If process p(i) is executing its critical section, then no other processes can be executing their critical sections.
   ii. Progress – If the critical sections are vacant and a process wishes to enter, then only processes that wish to enter a critical section can participate in the decision about which process enters next, and this process cannot take an indefinite amount of time.
   iii. 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.

e. Two Process Solutions

i. Structure of solution:

Repeat
Entry into critical section
Critical section
.
.
.
Exit from critical section
Until done

ii. Algorithm 1

Process P(i)
Entry: while(turn != i) {
Do nothing;
}
.
.
.
Exit: turn = (1-i)

turn = {0, 1}, initially 0

P0	P1	turn
Enters	Waits	0
Exit	Wait	0 to 1
wait	Enter	1
Wait	exit	1 to 0

If P0 uses its turn then (upon exit, turn is set to 1) and then P1 does not enter its critical section (it never sets turn back to 0) then if P0 wants to enter its critical section, it will wait forever (or at least until P1 resets turn back to 0). So this algorithm violates bounded waiting.

iii. Algorithm 2

Process P(i)
Entry: flag[i] = TRUE;
While(flag[1 – i]) {
Do nothing;
}
.
.
.
Exit: flag[i] = FALSE;

In this algorithm the process sets its flag indicating that it is entering its critical section and then waits for the other process to leave its critical section. If both processes hit the flag[i] = TRUE statement at the same time deadlock occurs. Both processes will wait on each other forever in the while loop.

iv. Algorithm 3

```
Process P(i)
Entry: flag[i] = TRUE;
    Turn = 1-i;
    While(flag[i-1] && (turn = 1 – i)) {
            Do nothing;
    }
    .
    .
    .
Exit: flag[i] = FALSE;
```

In this algorithm the process indicates that it desires to enter the critical region, but defers to the other process. If both process hit the entry point at the same time, the last process to defer to the other process waits. As soon that process clears its critical section it resets its flag allowing the other process to enter its critical section.

	P0		P1
1	Flag[0] = TRUE	2	Flag[1] = TRUE
3	Turn = 1	4	Turn = 0
5	While = FALSE	5	While = TRUE

v. Bakery Algorithm – The multiple process version of algorithm 3 is called the Bakery Algorithm.

f. Synchronization Hardware – Hardware provision 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.

i. Test and Set - 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) . It's function is to probe retrieve the contents of a variable, return it value, and at the same time set its value to TRUE;

```
TestAndSet(x)
{
    temp = x;
    x = TRUE;
    return temp;
}
```

Entry: while(TestAndSet(x)) {

```
        Do nothing;
    }
    .
    .
    .
```

Exit: x = FALSE;

If x is FALSE it is set to TRUE and we fall out of the loop, execute the critical section, and up exit, reset it to FALSE.

If x is TRUE, we it set it to TRUE and wait in the loop until another process sets x to FALSE;

ii. Swap.

iii. Homework problem – define the swap() function. Explain how it works. Compare and contrast it to the TestAndSet() function.

g. Semaphores

i. The previous solutions are not easy to generalize to multiple process solutions. Semaphores are a tool to help with this.

ii. A semaphore, *s*, is an integer variable. In order to use semaphores, two atomic operations are defined:

1. Wait(s) = P(s) = test
2. Signal(s) = V(s) = increment
3. They are sometimes referred to as P(s) and V(s) for the Dutch words *proberen* (wait), and *verhogen* (signal).

```
wait(S)
{
        while(S<= 0) {
                do nothing;
        }
        S = S-1;
}
```

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

4. Both operations must be atomic, that is only one process can modify the semaphore value at any given time.
5. Structure of the solution is as before:

```
/* mutex is initialized to 1 */

repeat {
        wait(mutex);

        critical section

        signal(mutex)
until false;
```

6. Consider two processes using semaphores to synchronize their operations. To do so they will share a common variable, *synch*.
    a. This time, initialize synch to 0.

```
        P1()
        {
                .
                .
                .
                S1;
                signal(synch);
                .
                .
        }

        P2()
        {
        .
                .
                .
                wait(synch);
                S2;
                .
                .
        }
```

b. Statements located after the wait() statement in P2 will only be executed after the signal() in P1, meaning all statements before the signal() statement in P1 will execute before all statements after the wait() statement in P2.
c. Implementation Details:
    i. The earlier solutions along with the semaphore solution rely on *busy waiting*. Busy waiting is when the process loops continuously in the critical section entry code.
        1. Referred to sometimes as a spin-lock.
        2. In a single CPU system that is multi-programmed, spin-locks can eat valuable CPU slices.
        3. Because they require no context switch spin-locks can be good, as long as the time in them is short.
        4. The cure:
            a. Redefine the wait statement to use blocking instead of busy waiting.
                i. Blocking means that the process puts itself into the wait queue when the wait() is encountered.
                ii. 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 it is > 0 then they pass the wait() call is exited, otherwise they return themselves to the wait queue.
                iii. So 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.

h. Deadlocks and Starvation

    i. 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:

```
/* Initialize the semaphores to 1. */
S =  1;
Q = 1;



P0            P1
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 and falls out of the wait calls. The next set puts both processes P0 and P1 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.

    ii. Note: Deadlocks can be caused by many other conditions, such as processes competing for resources, etc. More discussion on that later.

i. Classic Synchronization Problems

    i. Bounded buffer – There is a common buffer that is being used to store produced items that are to be consumed. The semaphores function to keep it from overflowing, or from being accessed when it is empty, and to make sure that only one process access the buffer at a time.

```
/* Declarations. */
buffer is a piece of memory of size n.
mutex, empty, and full are semaphores.


/* Initialize variables. */
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
		…
		produce an item;
		…
		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);
		…
		consume item from buffer;
		…
	until(false);
}
```

ii. Readers/Writers

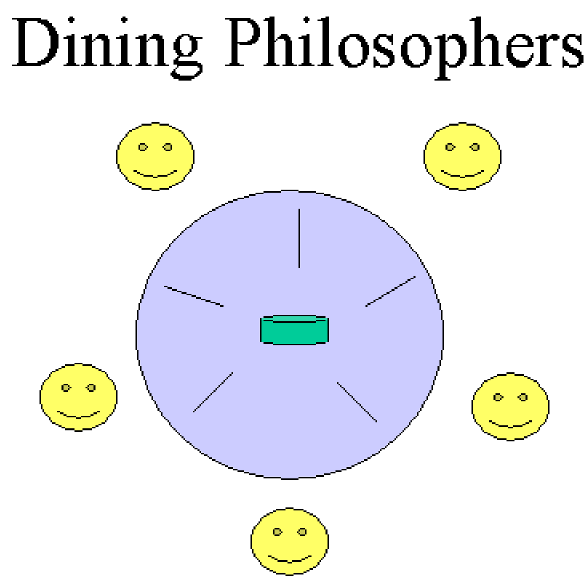
When data is to be shared by several processes, it is okay for many to read it at once, but only one process may change the object at once. When the writing process is updating the process, only it can access it, no other readers or writers are permitted access.

```
/* Declarations. */
mutex, wrt are semaphores;
readcount is integer;

/* Initializations. */
mutex = 1; /* Ensures mutual exclusion when readcount is updated. */
wrt = 1; /* Ensures mutual exclusion for the writing processes. */
readcount = 0; /* Tracks the number of processes reading the data. */
```

```
WriterProcess()
{
          wait(wrt);
                    …
                    /* Writing is performed. */
                    …
          signal(wrt);
}

ReaderProcess()
{
          wait(mutex);
                    readcount = readcount + 1;
                    if (readcount == 1) then wait(wrt); /* Protects object from being changed
                                                          while it is being read. */
          signal(mutex);

                    …
                    /* Reading is performed. */
                    …

          wait(mutex);
                    readcount = readcount –1;
                    if (readcount == 0) then signal(wrt); /* Signals writers that all reading is
                                                           completed. */
          signal(mutex);
}
```

iii. Dining Philosophers – This is a classic problem in which multiple processes must coordinate the use of multiple resources.

1. Description – 5 philosophers are seated at a round table. Their whole life revolves around thinking and eating. In the middle of the table is a big bowl of rice, the table is laid out with 5 chopsticks. When a philosopher gets hungry, they get the two chopsticks closest to them.



2. Simple solution is represent each chopstick with a semaphore.

```
/* Declarations. */
semaphore chopstick[5];

repeat
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        …
        eat
        …
        signal(chops chopstick[i]);
        signal(chopstick[(i+1)%5]);
        …
        think
        …
until (false);
```

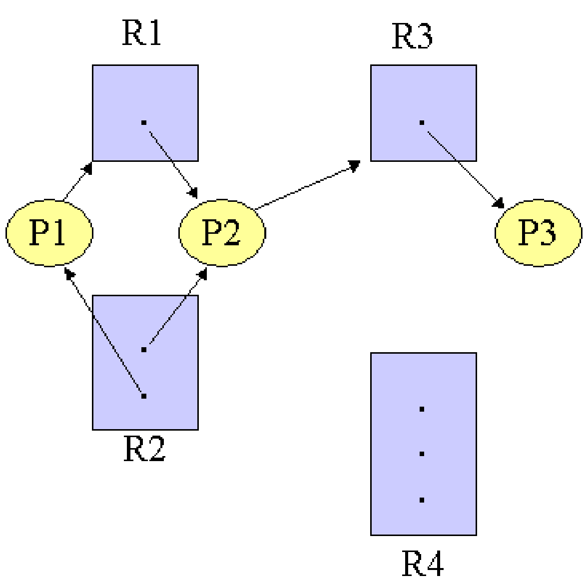
3. This solution will encounter a deadlock if all the philosophers get hungry at the same time. Some solutions are:
   a. Allow only 4 philosophers at the table at once.

        b. Allow a philosopher to pick chopsticks only if both are available.
        c. Use an asymmetric solution. Odd philosophers pick up left chopstick then right, even philosophers just the opposite.
      4. Solution must be deadlock free, but not allow a philosopher to starve. Deadlock free does not guarantee starvation free.
   j. Introduction to Knoppix
   k. Assignment 2 – Install Knoppix
   l. Assignment 3 – Run some statistics on the readers/writers programs
8. Deadlocks – In a multi-programming environment there will exist several processes and several resources. When a process request a resource that is not available the process will enter a wait state until the resource becomes available. 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.
   a. Example: We are eating steaks. I have a knife, you have a fork. I grab my knife and you grab your fork. I won't let you use my knife until you give me your fork, and you won't let me use your fork until I loan you my knife. What happens, we starve like idiots.
   b. System Model – the system is a finite number of resources distributed among a number of competing processes.
      i. Resources: several types, each having some number of instances. Examples may include memory space, CPU cycles, files, I/O devices (printers, tape drives, and whatnot).
      ii. A process requests a resource before using it, uses it, and then releases it.
         1. 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.
         2. Use: The process operates using the resource.
         3. Release: When the process does not need the resource anymore it releases it.
      iii. System calls are used to request and release resources.
         1. Examples include: open, close, malloc, free, new, delete…. etc.
   c. Necessary Conditions for a deadlock
      i. 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.
      ii. 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.
      iii. 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.

iv. Circular wait – There must be a set of processes {P0, P1….Pn} such that P0 is waiting on a resource held by P1, P1 is waiting on a process held by P2 …..Pn-1 is waiting for a resource held by Pn, and Pn is waiting on a process held by P0. Circular wait implies Hold and Wait.

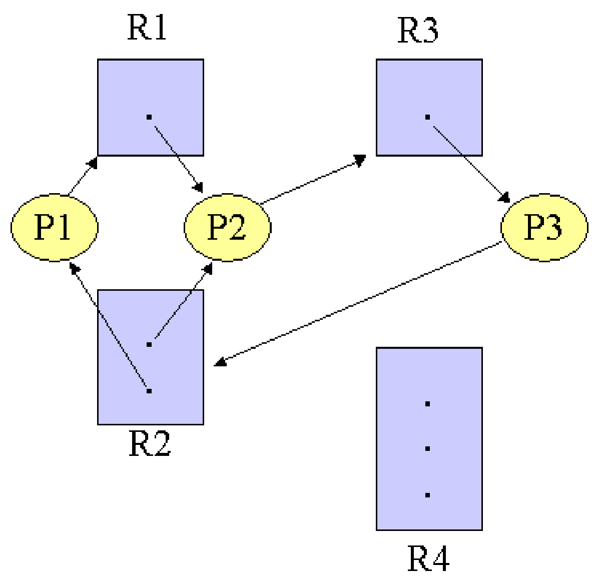
d. Resource Allocation Graphs – a set of vertices V, and edges E. The vertices represent processes and resources, the edges represent requests and assignments (allocations of resources).
   i. V consists of processes, P, and resources, R.
   ii. E consists of directed edges that go from a process to a resource, indicating a request, or from a resource to a process, indicating an assignment of the resource to the process (assignment), or think of it as the resource is allocated to the process.
   iii. Example: The graph is defined by the sets, P (processes), R (resources), and E (edges).
      1. P = {P1, P2, P3}
      2. R = {R1, R2, R3, R4}
      3. E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Resource Allocation Graph



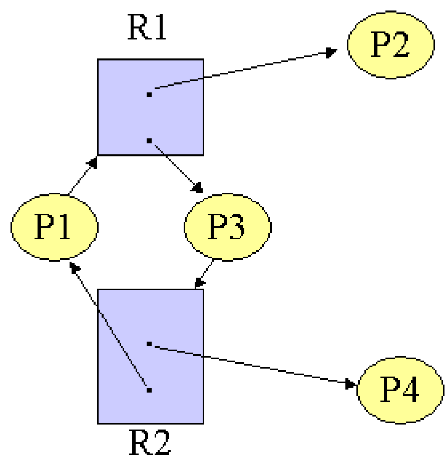
4. Notice that in the above example there are no cycles in the graph. A resource allocation graph with out cycles indicates that no deadlocks exist.

Resource Allocation Graph
(with deadlock)



5. P1, P2,and P3 are deadlocked. There are two cycles:
   a. {P1, R1, P2, R3, P3, R2, P1}
   b. {P2, R3, P3, R2, P2}
   c. The deadlock exists because no process can release a resource because they cannot run because they are waiting for resources held by other processes.
6. Here is a graph with a cycle that is not deadlocked.

Resource Allocation Graph with cycle, but no Deadlock



7. Cycle is:
    a. {P1, R1, P3, R2, P1}
8. There is no dead lock because P2 or P4 can complete processing and release instances of R1 and R2, allowing P1 and P3 to acquire the resources needed to complete.
9. Methods for handling deadlocks and the Bankers algorithm.

e. Assignment 5 – Solve the Resource Allocation Graphs

f. Handling Deadlocks – Methods of for dealing with the deadlock problem:
    i. Deadlock Prevention: 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.
        1. 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 though deadlock prevention cannot be accomplished by denying mutual exclusion; some resources are not sharable.
        2. 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:

        a. Starvation: A process may have to wait indefinitely for resources before it can start.
    3. 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.
    4. Circular Wait: An of resource types is imposed and each process requests resources in an increasing order of enumeration.

ii. Deadlock Avoidance: OS is given information about a processes required resources of a process during its lifetime. 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, to decide if the resource request should be satisfied or delayed. These algorithms avoid deadlocks by using the resource state information to avoid circular wait conditions.
    1. Safe State: A state is safe if the system can allocate resources up to its maximum in some order and still avoid a deadlock.
    2. The system is in a safe state if there exists a safe sequence.
    3. Safe Sequence: For each P(i), the resources needed can be satisfied by currently available resources and the resources held by the processes P(j) that came before it. P(j): j < I

       Sequence of Processes: {P1, P2, P3…..Pn} All the resources Pi will need are available (not allocated to any process) or are held by P1 to P(i-1).

Bankers Algorithm –

n – number of Processes
m – number of resource types
Avail[i] – number of resources of type i available
Max[i, j] – max # of resources of type j that process i will need
Alloc[i,j] – process i has this man resources of type j
Need[i, j] = max[I, j] – alloc[i, j];

Work = Avail
Finish[i] = FALSE;

While there exist i such that:
    Finish[i] = FALSE
    Need[i] <= Work

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

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

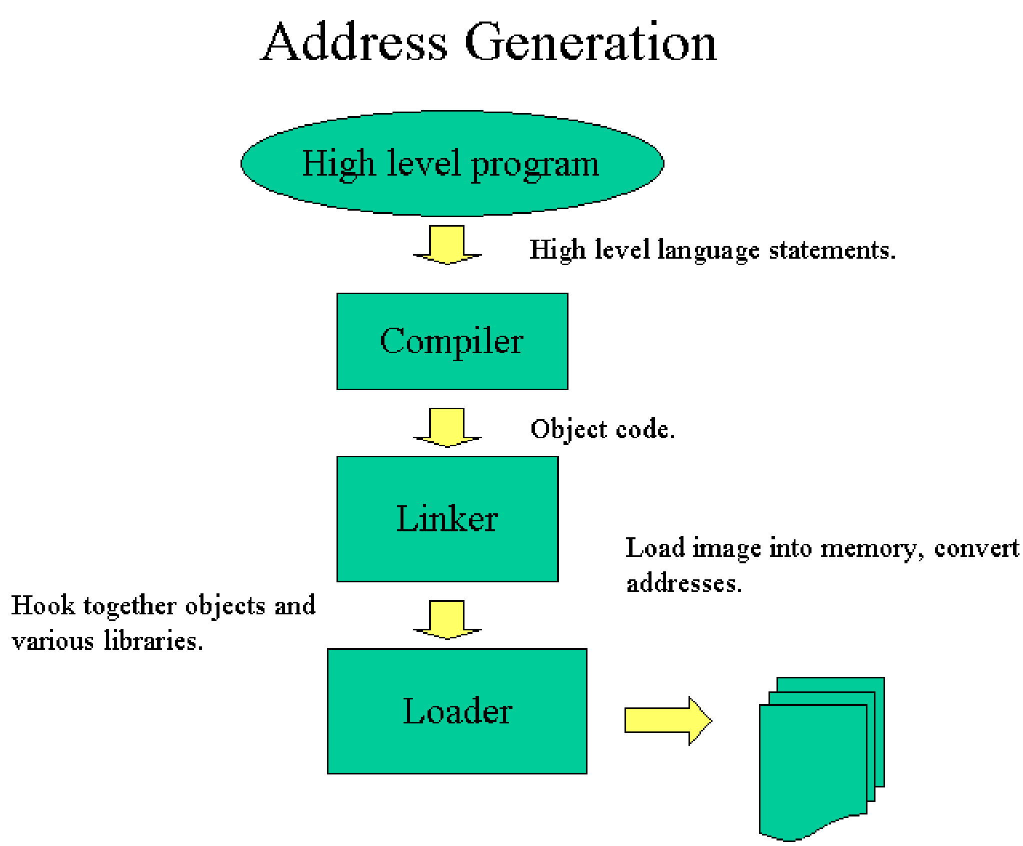
(Enter my example here)

iii. System is allowed to enter a deadlock state and recover. An algorithm monitors system state to determine if a deadlock has occurred, and uses a deadlock recovery algorithm when one is detected.
iv. Ignore the problem all together. Give the responsibility for handling deadlocks to the users. Many systems, including UNIX have used this strategy. In some systems deadlocks occur infrequently (maybe once a year). When this is the case, it may make more economic sense to ignore the problem.

9. Quiz 2 Review (Week 8)
10. Quiz 2
11. Memory Management – Main memory is a large array of bytes, each with its own address.
    a. Memory basics
        i. One unit (like a switch) Bit (0/1)
        ii. Nibble – 4 bits
        iii. Byte – 8 bits

iv. Word – 8, 16, 32, 64 bits
v. Memory addressing - The smallest piece of space that a CPU can usually address is a byte. A word is 4 bytes, some CPUs address on word or half word boundaries. Access to and from memory is handled by the memory unit (memory controller).
    1. byte addressable – all bytes are addressable
    2. word addressable – can only address on word boundaries
    3. bus error – attempt to access memory on incorrect boundary.
        a. How do we get around this? We use masking and shifting.

b. Binding – a mapping from one address space to another.
c. Process of binding:



d. Linking types:
    i. Static – library routines included in code.
    ii. Dynamic – libraries linked in at runtime. The DLL is located by looking at a stub inserted at compile time. Saves memory, but more complex, takes longer to come up initially, and is not as portable.
e. Address types:
    i. Symbolic – address in a program, like a variable name (COUNT)

   ii. Relocatable – These address are calculated from a reference point, like the beginning of the program. The compiler will generate these at compile time.
      1. offset – is position of address calculated from program start.
      2. Physical Address = Program Start address + offset.
   iii. Absolute (unrelocatable) – These are physical addresses of the machines main memory.
   iv. Position independent – runs anywhere. Variables use relative locations, relative to PC. Base and Limit method.

f. Address Binding
   i. Compile time – Usually the compiler turns symbolic addresses into relocatable addresses, but if it is known ahead of time where a process has to reside in memory, physical address can be generated. Absolute code is generated.
   ii. 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.
   iii. Execution time – When processes are moved around during their execution, final binding is saved for when they run. This binding is done by the memory management unit (MMU)

g. Dynamic Loading – In some cases (many in fact) 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.
   i. In this technique, routines are kept on disk in relocatable format.
   ii. Program starts, main routine is in memory. When a sub routine is called, the calling routine checks for its presence in memory, if there it control transfers to it, otherwise, the relocatable linking loader is called and the routine is loaded from disk.
   iii. Good when program contains large amounts of code that are rarely called.
   iv. Does not require support from OS. Programmers responsibility to design program to use scheme. OS provides library routines in support of dynamic loading sometimes.

h. Dynamic Linked Libraries – Widely supported. Linking of system libraries is postponed until run time. A stub is included in the binary image that indicates how to locate or load the appropriate routine if the routines is not already present.
   i. Under this scheme all processes that use the DLLs execute only one copy of the code.
   ii. Advantages include:
      1. Code savings.
      2. Easier to implement updates and bug fixes throughout the system. (saves re-linking of all programs using new libraries.
   iii. Usually requires OS support.

i. Overlays – 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.

j. Example: 2 pass assembler that requires 200k 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 – 130k

   Pass 2 with needed routines = 80k + 20k + 30k = 130k with overlay driver – 140k

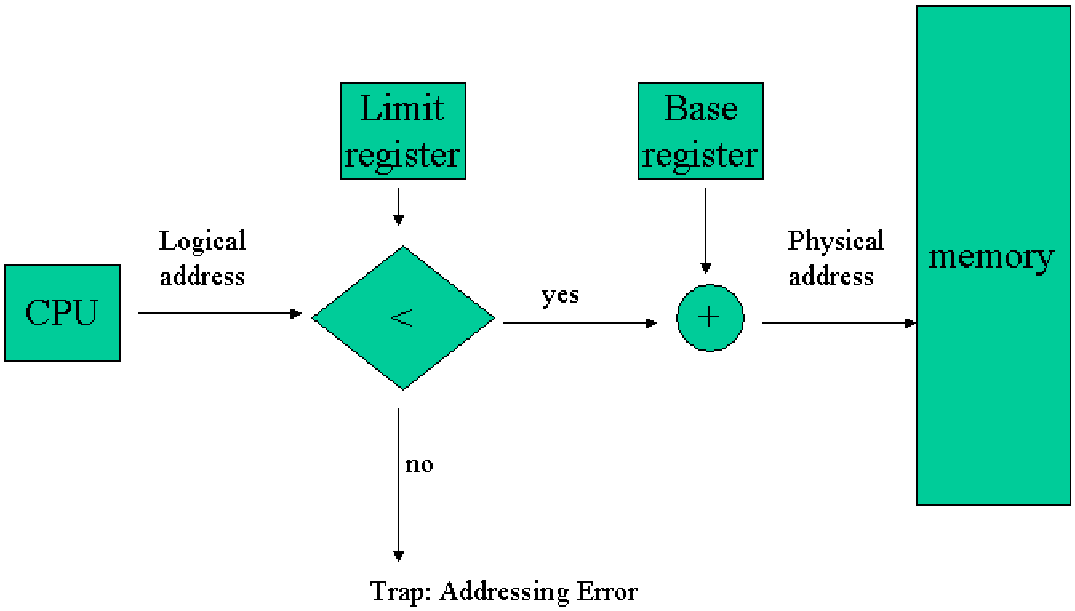
   So the assembler can be run with 150k of memory. This scheme does not require special OS support but does require the programmer to have a good knowledge of the program structure.

k. Logical versus Physical Address Space – CPU generates a logical address (sometimes called a virtual address), memory management unit MMU turns the logical address into a physical address and loads it into the MAR (memory address register).
   i. The set of all addresses generated by a program is called the logical address space. The hardware memory addresses corresponding to these addresses are referred to as physical address space.
   ii. Compile time binding and load time binding result in logical addresses that are the same as physical addresses.
   iii. Execution time binding – logical and physical addresses are different from one another.
   iv. Address translation handled by the MMU.
      1. Physical Address = Logical + base register (sometimes called relocation register)
      2. 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.
      3. Logical to Physical address mapping is the central concept in memory management.
   v. User Mode – accesses virtual address space
   vi. Supervisor Mode – accesses physical address space

l. Swapping – Process are memory resident when 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.
    i. With 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 place.
    ii. This requirement is relaxed with execution time binding.
    iii. Processes that are swapped out are tracked by the OS.
    iv. Swapping takes time. The time is proportional to the amount of data that must be moved from disk to memory. Example:
        1. a 100k process at 1000k per second with a 8 ms disk latency would take:
            a. 100k/1000k + 8 ms = 108ms.
            b. To swap whatever was in memory out and this process back in would take: 2 * 108ms = 216ms.
    v. For efficiency reasons time in memory and running should be long compared to the time to swap the process from disk to memory.

m. Contiguous Allocation – Main memory accommodates the OS and the user processes. OS resides in one partition, user process in the other. OS is commonly placed in low memory (address numbers starting at 0) because interrupt vectors is often in low memory. Advantages: Simple, Fast. Disadvantages: Fragments memory, there may be programs too big for memory.
    i. Single Partition – OS resides in low memory. User processes reside in high memory.
        1. Processes and OS are protected from each other by using the base and limit register scheme.
        2. Every address the CPU generates is checked against these registers. Hardware is used so it can be done quickly.
        3. It provides OS and user programs protection from running processes.

Base and Limit Hardware Support



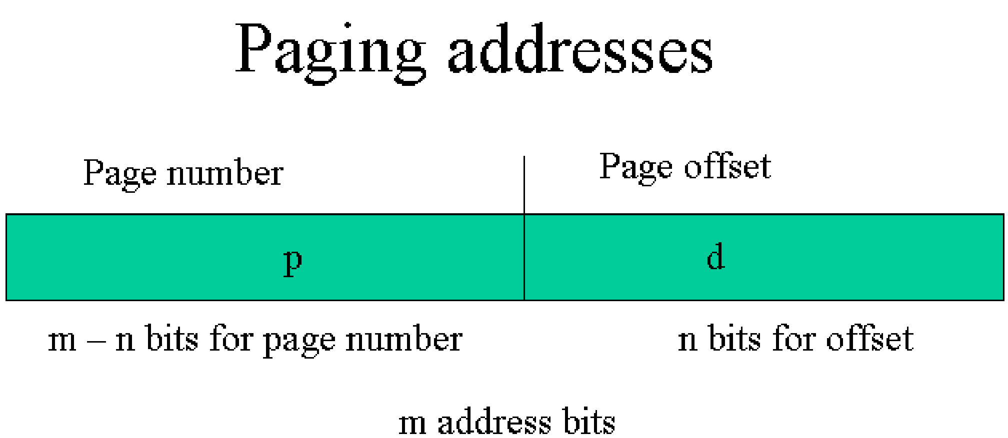
ii. Multiple Partition – In the most basic case of multiple partition memory management memory is divided into n equal size partitions (IBM 360 used this scheme). The number of partitions in the memory determines the degree of multiprogramming. When a partition is free, the OS selects a process from the input queue and loads into that partition.

iii. 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.

1. Initially all process memory is empty.
2. 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. There are several allocation schemes:
    a. First-Fit – Put the process in the first hole found that will accommodate its memory requirements. Searching stops as soon as a whole is found that works.

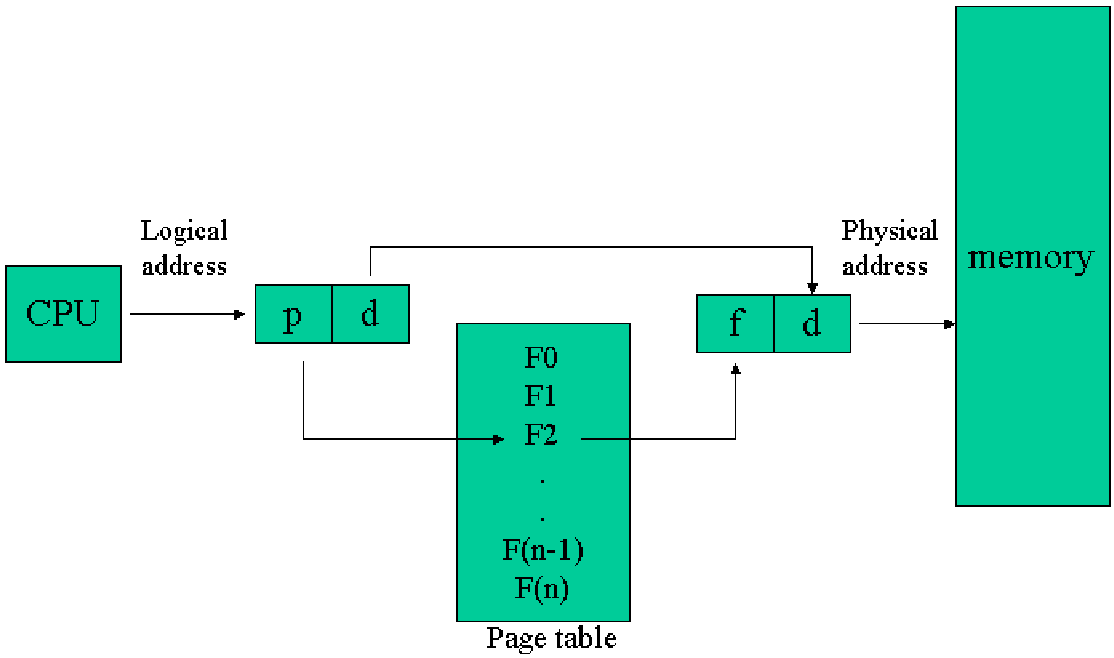
        b. Best-Fit – Put the process in to the smallest hole that will accommodate its memory requirements. This strategy produces the smallest remaining hole/holes.
        c. Worst-Fit – Put the process in to the largest hole that will accommodate its memory requirements. This strategy produces the largest remaining hole/holes.
      3. 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.
    iv. Internal and External Fragmentation
      1. 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. To solve this problem (can waste up to a third of the memory) compaction can be used. Periodically shuffle the memory and create a big block of free memory.
      2. Internal Fragmentation – 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 would 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 internal fragmentation.

12. Paging and Virtual Memory – Used to avoid fragmentation.
  a. Basics – Each process has a page table.
    - PageNum = logical address/pageSize;
    - FrameNum = pageTable[pageNum].page
    - Offset = logical address – pageNum*pageSize
    - Physical Address = FrameNum * FrameSize + Offset

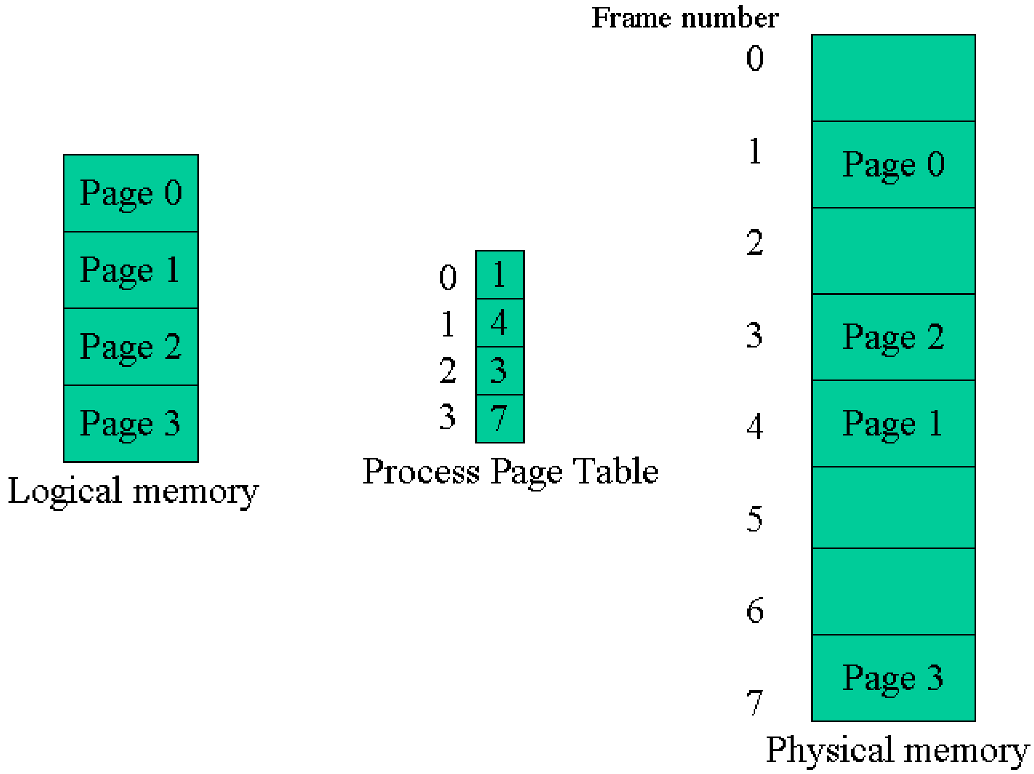


Page sizes are usually kept to be a power of 2, typically between 512 bytes and 16 megs. This makes page and offset calculation a matter of shifting bits.

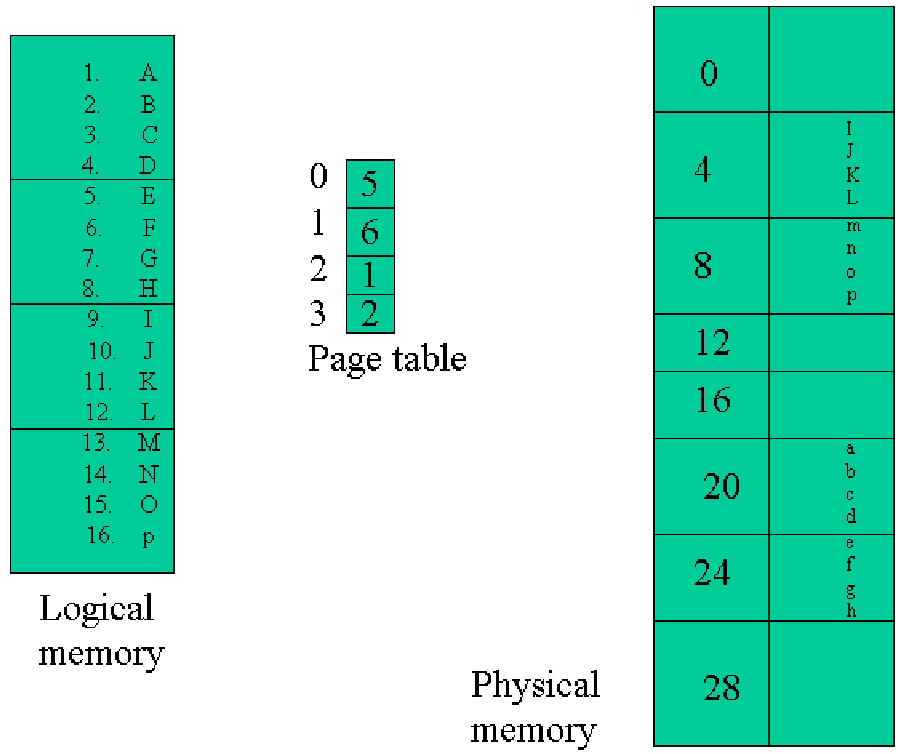
Paging Hardware



Paging Model (logical/physical memory)



Paging example using 32 byte memory with 4 byte pages



Paging attributes:

- Form of dynamic relocation.
- Logical addresses bound by paging hardware to physical addresses.
- No external fragmentation.
- Any free frame can be allocated to any process that needs it.
- Will have internal fragmentation on last frame of process.
- 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.

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 processes page table.
- User program views memory as one contiguous piece even though it is physically scattered throughout memory.
- As program runs logical addresses are converted to physical addresses.
- OS keeps track of which frames are used and which are available in the Frame table.

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.
    - Translation look-aside buffers (TLB) are used to alleviate this problem. TLB is a small fast piece of associative memory that holds part of the page table in it.
        - Logical address generated by CPU, page number presented to TLB. If the page number is found, TLB outputs the matching frame number. Process takes 10% longer than a direct memory reference.
        - If page number not in TLB, page table is referenced and address is formed. TLB updated, if full one of the entries is replaced.
        - The percent of times that a page number is found in table is the hit ratio. Using the hit ratio we can calculate the effective memory access time.
            - 80 percent hit ratio
            - 20 nanosecond TLB search time
            - 100 nanosecond memory access time.
            - Effective access time = .80 **(20 + 100) + .20**(20 + 100 + 100) = 140 nanoseconds
    - When memory is full (page table full) pages must be swapped in and out from disk.

13. Paging implementation and algorithms
    a. Locality of reference – programs tend to work in a particular memory area and move slowly through them.
    b. Demand Paging – pages are only brought in when they are needed.
    c. Page Fault – event that occurs when a page is not in RAM
    d. 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.
    e. Page Replacement
    f. Page Replacement Algorithms
        i. FIFO – First In First Out
            1. Belady's anomaly – give a system more resources (memory frames) and it makes more faults. FIFO is susceptible, LRU is not.
        ii. Optimal (Belady) – farthest in the future. This is good, but we have a hard time knowing what the future will bring.
        iii. LRU – Least Recently Used. Good in theory and in practice. (Dirty bit set when page changed, each time page accessed, time stamp updated or reference bit set. If using reference bit, each time quanta shifts reference word to the right.
    g. Example of FIFO, LRU, and optimal; page/frame size = 100; maxframes = 5

```
hello tv land!
numAdds = 20
Address Stream.
721
43
121
222
44
327
45
428
223
328
45
329
224
122
225
46
123
722
47
124
Page replacement (FIFO)
Frame Table - 7  #  #  #  #
Frame Table - 7  0  #  #  #
Frame Table - 7  0  1  #  #
Frame Table - 7  0  1  2  #
Frame Table - 7  0  1  2  #
Frame Table - 7  0  1  2  3
Frame Table - 7  0  1  2  3
Frame Table - 4  0  1  2  3  >>> page fault
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  7  1  2  3  >>> page fault
Frame Table - 4  7  0  2  3  >>> page fault
Frame Table - 4  7  0  1  3  >>> page fault
pageFaults = 4
```

Page replacement (LRU)

```
Frame Table - 7  #  #  #  #
Frame Table - 7  0  #  #  #
Frame Table - 7  0  1  #  #
Frame Table - 7  0  1  2  #
Frame Table - 7  0  1  2  #
Frame Table - 7  0  1  2  3
Frame Table - 7  0  1  2  3
Frame Table - 4  0  1  2  3  >>> page fault
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 4  0  1  2  3
Frame Table - 7  0  1  2  3  >>> page fault
Frame Table - 7  0  1  2  3
Frame Table - 7  0  1  2  3
pageFaults = 2
```

Here is an optimal (Belady) example.

```
7  #  #  #  #
7  0  #  #  #
7  0  1  #  #
7  0  1  2  #
7  0  1  2  #
7  0  1  2  3
7  0  1  2  3
4  0  1  2  3 >>> page fault
.
.
.
.
.
.
.
.
.
7  0  1  2  3 >>> page fault
7  0  1  2  3
7  0  1  2  3
```

h. Belady’s anomaly – sometimes with FIFO, having more resources results in more page faults.
    i. Page reference string – 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
        1. FIFO – with 4 frames results in 10 page faults

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
			4			4	4	4	3	3	3

        2. FIFO – with 3 frames results in 9 page faults.

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5			5	5	
	2	2	2	1	1	1			3	3	
		3	3	3	2	2			2	4	

i. Counting algorithms
    i. LFU – Least frequently used.
    ii. MFU – Most frequenty used.

14. Assignment 5 – Program a page swapper, implement FIFO and LRU, show some run statistics for various provided data sets. (Due in 2 weeks)
15. 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.
16. Demand Paging with pre-paging – 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.
17. 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.
18. Storage
    a. Files Systems – A file is a collection of related data items (program, text, image….)
    b. Two ways of deciding what is in a file:
        i. User convention (*.old in Unix is usually an old version of a file, while *.cold would usually be an old C source code file.)
        ii. OS defined (*.doc in windows is a word file)
    c. Access Methods:
        i. Sequential
        ii. Indexed – Consists of an index file and a data file
    d. Attributes
        i. name

        ii. type
        iii. location (on disk)
        iv. size (number of records, or bytes)
        v. protection info
        vi. owner
        vii. last modified
    e. Operations
        i. creation
        ii. truncate
        iii. delete
        iv. open
        v. close
        vi. append
        vii. read
        viii. write
        ix. seek
        x. tell
    f. Directory – collection of files. In DOS links can only go to leaves. This keeps cycles from occurring.
        i. Tree Structured – acyclic graphs
        ii. A file is deleted only when there are no more links to it. A file may be referenced from more than one directory.
    g. Pathnames
        i. Absolute – back to root
        ii. Relative – relative to current working directory
    h. Protection
        i. Against other users
        ii. Against self
        iii. Control
            1. what – read/write/execute, append/delete/list
            2. who – you, your group, everyone

19. File System Implementation
    a. File Control Block – FCB: Directory Block, data
    b. File System – Consists of all files (FCB, data), hardware, directories, format information (block size, free blocks)
        i. Blocks – each block has physical address (track, sector)
        ii. Disk is abstracted to an array of blocks (8k – 16k)
            1. To get data from disk – move head (seek time), wait for platter to spin around to right place (rotational latency).
            2. To read/write quickly – group blocks of file close to each other. We would like to allocate contiguously in hardware and software.
    c. Methods of Allocation
        i. Contiguous – suffers from external fragmentation
        ii. Linked allocation
            1. Eliminates external fragmentation

2. Leads to non-contiguous file data distribution
3. Only good for sequential accessed files.
4. Lost file pointers can cause a file to be damaged.
5. Directory will store:
    a. Name
    b. first block
    c. last block
    d. Each block has a pointer to the next block.

iii. FAT (File Attribute Table)
1. Simple and efficient
2. Linked allocation method.
3. If FAT is not cached, it can lead to lots of disk seeks.
4. FAT –
    a. FAT at beginning of each disk partition.
    b. It has one entry for each disk block and is indexed by number.
    c. The directory entry has entry for first file block, FAT has entry for the next block.
    d. The last block in file is signaled by EOF value.
    e. Empty blocks marked by 0's.
    f. If FAT table gets wiped out, look out.

iv. Indexed
1. Directory has file name and location index block. Index block is a sequential list of the blocks in the file.

v. Free Space – linked into on file. Bit Vector (1 per block indicates if the block is free or not.

vi. Directories – simplest way is to make linear table, some systems use a hash table.

20. I/O Systems
    a. Lots of OS deals with files (70%)
        i. Hardware devices:
            1. polling
            2. interrupt
            3. interrupt with DMA
        ii. Device types
            1. Block
                a. Disk
                b. Tape
            2. Character
                a. Keyboard
                b. Serial i/o (COMM port)
    b. Disks
        i. Structure
        ii. Head Scheduling
            1. FCFS – First Come First Serve
            2. SSTF – Shortest Seek Time First

3. SCAN – (elevator algorithm) – serve all requests going one way, then go the other.
    a. Cyclic elevator – goes one direction, then head moves back to start.

21. Quiz 3 review
22. Quiz 3