Operating Systems

1. Introduction

1.1 What is an Operating System

- There are two main tasks for operating systems
    1. Providing abstractions to user programs
    2. Managing the computer's resources

1.1.1 The Operating System as an Extended Machine

- Users do not want to be involved in the programming of storage devices.
- Moreover, Operating System provides a simple, high-level abstraction such that these devices contain a collection of named files.
- Such files consist of the useful piece of information like a digital photo, email messages, or web page.
- Operating System provides a set of basic commands or instructions to perform various operations such as read, write, modify, save or close.
- Operating Systems turn ugly hardware into beautiful abstractions.

1.1.2 The Operating System as a Resource Manager

- The concept of an operating system as primarily providing abstractions to application programs is a top-down view
- An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system.
- Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices.
- In the bottom-up view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs wanting them.

1.2 History of Operating Systems

1.2.1 The First Generation (1945-1955) : Vacuum Tubes

- First generation computers used vacuum tubes.
- All programming done in machine language or even worse yet, by wiring up electrical circuits by connecting thousands of cables to plugboards to control the machine's basic functions.
- There were no OS, programming language, anything but wires and operators(people).
- It weighted tens of tons and was very expensive

1.2.2 The Second Generation (1955-1965) : Transistors and Batch Systems

- Instead of vacuum tubes, transistors began to be used.**(Mainframe)**
- Languages like Assembly and FORTRAN came out so programming became more easy.
- Disadvantage of second generation computers was, they were producing a lot of heat
- Punch cards used to program.
- First anchestors of operating system seen in this generation

1.2.3 The Third Generation (1965-1980) : ICs and Multiprogramming

- With third generation computers integrated circuits(IC) came into our lifes.
- Instead of using punch cards or wiring cables, first keyboards showed up.
- For the first time with this generation, computers began to run multiple programs at the same time.**(Multiprogramming)** **(**operating system are evolving but still they are just batch systems**)**

1.2.4 The Fourth Generation (1980-Present) : Personal Computers

- CPU(microprocessor) invented and with this invention, computer size get significantly smaller and devices like mouse, cell phones, etc. showed up.
- The first home computers produced in this generation
- Macintosh can be an example for this generation’s computer

1.2.5 The Fifth Generation (1990-Present) : Mobile Computers

- Phones.
- Operating systems like Android, IOS, RIM(Blackberry OS), etc.

1.3 Computer Hardware Review

1.4 The Operating System Zoo

1.5 Operating System Concepts

1.6 System Calls

1.7 Operating System Structure

1.7.1 Monolithic Systems

- By far the most common organization, in the monolithic approach the entire operating system runs as a single program in kernel mode.
- The operating system is written as a collection of procedures, linked together into a single large executable binary program.
- When this technique is used, each procedure in the system is free to call any other one, if the latter provides some useful computation that the former needs.
- Being able to call any procedure you want is very efficient, but having thousands of procedures that can call each other without restriction may also lead to a system that is unwieldy and difficult to understand.
- Also, a crash in any of these procedures will take down the entire operating system

1.7.2 Layered Systems

- Archeology

1.7.3 Microkernels

- The idea is not to have one large kernel where one bug can crash the whole system.
- Instead, there is a lot of processes running in user-mode, where **microkernel** handles only the most crucial parts of the system (IPC/memory allocation/etc).
- They are mostly used in critical systems like rockets/satellites/etc in order to provide flawless operation whatever happens.
- Also an example of such system is **MINIX 3**

1.7.4 Client-Server Model

1.7.5 Virtual Machines

- A Virtual Machine (VM) is **a compute resource that uses software instead of a physical computer to run programs and deploy apps**.
- One or more virtual “guest” machines run on a physical “host” machine.

1.8 The World According to C

2. Processes & Threads

2.1 Processes

2.1.1 The Process Model

- **Process**, is just an abstraction(instance) of a running program
- **Multiprogramming**, is rapid switching between processes

2.1.2 Process Creation

- There are 4 ways to create a process
    1. System Initialization (Daemons created)
    2. Fork - Exec by a running process
    3. User request for new process
    4. Initialization of batch job
- **Daemon**, is a process that runs on background and idle otherwise.

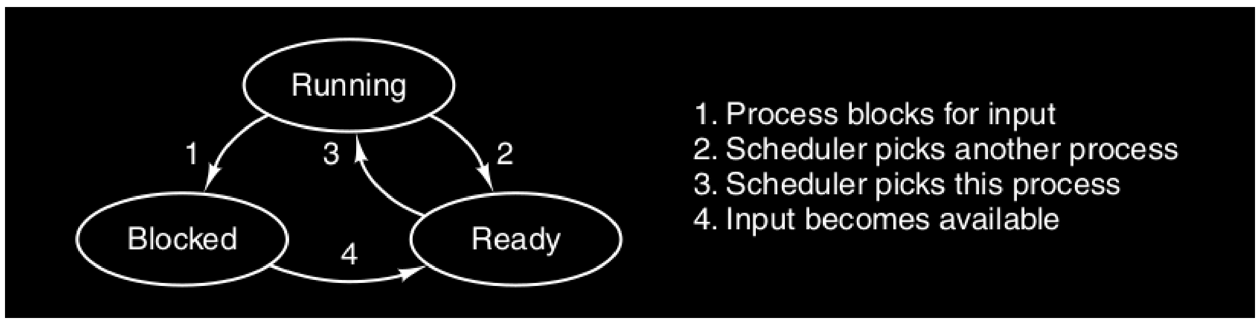
2.1.3 Process Termination

- There are 4 ways to terminate a process
    1. Normal Exit (Voluntary)
    2. Error Exit (Voluntary)
    3. Fatal Error (Involuntary)
    4. Killed By Another Process (Involuntary)

2.1.4 Process Hierarchies

- In UNIX, there is a parent-child hierarchy and OS starts with init
- In Windows, there is no hierarchy. Every process is same (uses a special token: **handle** instead)

2.1.5 Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- There are 3 states that a process may be in
    - Running (using the CPU at the moment)
    - Ready (can run, but CPU occupied by another process)
    - Blocked (unable to run until some external event happens)

2.1.6 Implementation of Processes

- **Process Table,** is simply an array of structure.
- Each **entry** in process table contains necessary informations about a process such as process' state, pc, sp, memory allocation, file descriptors, etc.

2.1.7 Modeling Multiprogramming

- CPU Utilaztion : $1 - p^n$

  Example

  Computer Memory : 8 gb
  OS and it's tables : 2 gb
  Each user program : 2 gb
  Average I/O wait : %80

  3 user program can fit, thus n = 3

  From the equation = $1 - (0.8)^3 = 0.49$

  If we add 8 more gb to the memory, 7 program can fit, thus n = 7

From the equation = $1 - (0.8)^7 = 0.79$

So adding 8gb memory increased throughput by %30

2.2 Threads

2.2.1 Thread Usage

- There are 3 reasons to explain necessity of threads
    1. (Main Reason) Most of the time multiple activities running at once in a process. By creating threads we can run these activities in quasi-parallel manner. Since threads share same address memory and data among themselves, it is possible to do parallelism. But it is impossible to do with processes.
    2. Threads are lighter in weight, faster(easier) to create/destroy than processes.(Generally 10-100x times faster)
    3. Threads yield to performance increase if there are no CPU bounds and there are substantial activities that can overlep.

2.2.2 The Classical Thread Model

- Different threads in a process are not as independent as different processes.
- All threads have exactly the same address space, which means that they also share the same global variables.
- Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary.
- Unlike different processes, which may be from different users, and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight.

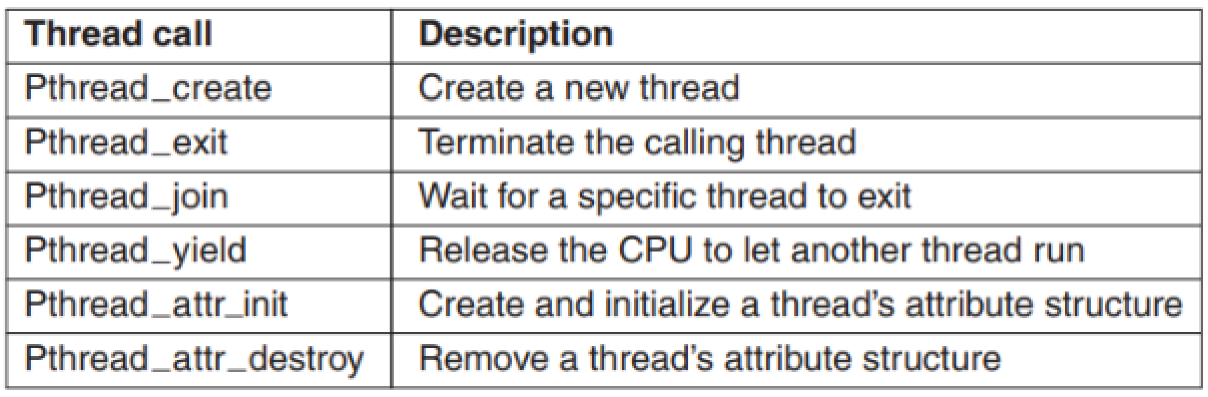
  Pre-Process Items (shared by all threads in a process)

    - Address Space
    - Global Variables
    - Open Files
    - Child Processes
    - Signals and Signal Handlers, etc.

  Pre-Thread Items (private to each thread)

    - Program Counter
    - Registers
    - Stack
    - State

2.2.3 POSIX Threads



Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

2.2.4 Implementing Threads in User Space

- When we implement threads in the user space, each process needs its **own private thread table**.
- Kernel thinks it's managing ordinary single-threaded processes

  Advantages

    - Can be implemented on an operating system that does not support threads.

- Faster (no system call, trapping and context switch to kernel needed)
- Each process can have its own optimized and customized scheduling algorithm.

Disadvantages

- When a blocking call made to process by the operating system (since operating system doesn't know about threads of the process), all threads are blocked instead of the related one. (This is a major issue since the purpose of the threads are exactly this blocking only one thread feature.)
- If a thread causes a page fault, the kernel, unaware of even the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.
- Threads can't get periodic clock interrupts from OS to schedule themselves (so they have to yield periodically).

2.2.5 Implementing Threads in The Kernel

- Kernel has the **thread table** which keeps track of all the threads in the system.
- When a thread wants to create or destroy a thread, it makes a system call. Kernel updates the thread table.

  Advantages

  - Eliminates all the disadvantages of the user space implementation.

  Disadvantages

  - Each thread operation is a relatively expensive system call.
  - Optimization for this disadvantage: **Thread Recycling**

    Thread Recycling

    - Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads.
    - When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected.
    - Later, when a new thread must be created, an old thread is reactivated, saving some overhead.
    - Thread recycling is also possible for user-level threads, but since the thread-management overhead is much smaller, there is less incentive to do this.

2.2.6 Hybrid Implementations

- Kernel is aware of only the kernel level threads and schedules those.
- These threads have user-level threads multiplexed on top of them.

2.2.7 Schedular Activations

2.2.8 Pop-Up Threads

2.2.9 Making Single-Threaded Code Multithread

2.3 Interprocess Communication

2.3.1 Race Conditions

- Race condition occurs when two or more threads try to access or change **shared data/memory** at the same time.
- Result of the change in data is dependent on thread scheduling algorithm, thus no one knows which thread will reach to **shared data** first.
- To prevent race conditions, we can assign these **shared places** as **critical regions** and lock them.

2.3.2 Critical Regions

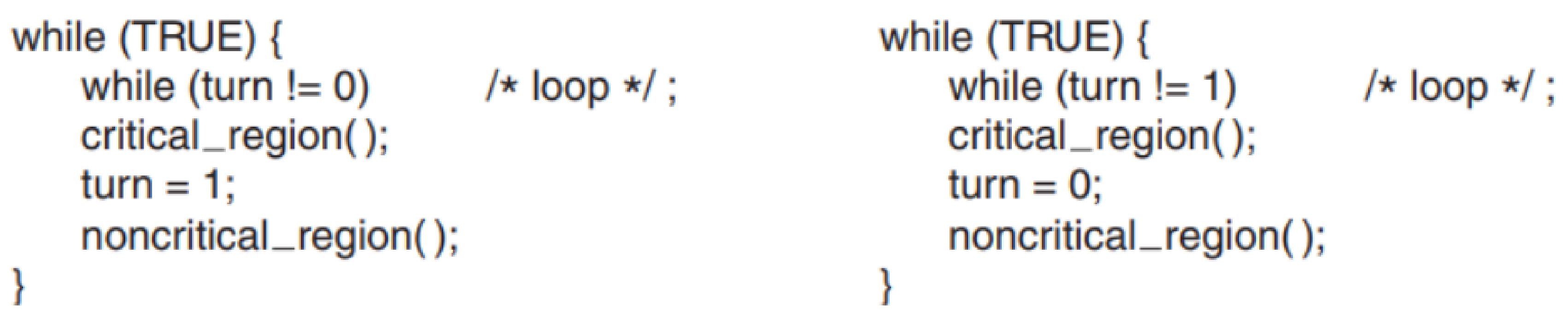
- Critical region, is a part of the program which tries to access **shared data/memory.**
- To prevent multiple access at the same time, we need to achieve **mutual exclusion.** There are 4 main requirements to achieve it:
  1. No two processes may be simultaneously inside their critical regions.
  2. No assumptions may be made about speeds or the number of CPUs.
  3. No process running outside its critical region may block any process.

4. No process should have to wait forever to enter its critical region.

2.3.3 Mutual Exclusion with Busy Waiting

- One way to satisfy these requirement is busy waiting.
- There are 5 different algorithm that achieves mutual exclusion with busy waiting:

1. **Disabling Interrupts(Bad)**
   - On a single-processor system you can disable all interrupts when a process enters its critical region.
   - This way no clock or other interrupts will happen, and the process will run until it’s out of its critical region so, CPU won’t be switched to other processes.
   - But giving the power to turn off interrupts to processes is unwise.
   - If they won’t return it or don’t exit their critical region for a long time, all the system will halt.
   - Also, if there are more than one processor, this won’t work since the interrupts will be blocked for only one CPU and others can change the shared data.
   - Operating system can disable the interrupts, but processes shouldn’t be able to do it.
2. **Lock Variables(Bad)**
   - There is a global lock variable, processes set it to 1 before entering their critical region and 0 when exiting.
   - If it is 1 already, they don’t enter to critical region (since it is already occupied by another process).
   - This won’t solve it too since after setting it to 1 and before entering to critical region, switch may happen. Meaning, this is not an atomic operation
   - A lock that uses busy waiting is called a **spin lock**.
3. **Strict Alternation(Bad)**
   - Strict Alternation Approach is the software mechanism implemented at user mode.
   - It is a busy waiting solution which can be implemented only for two processes.
   - In this approach, a turn variable is used which is actually a lock



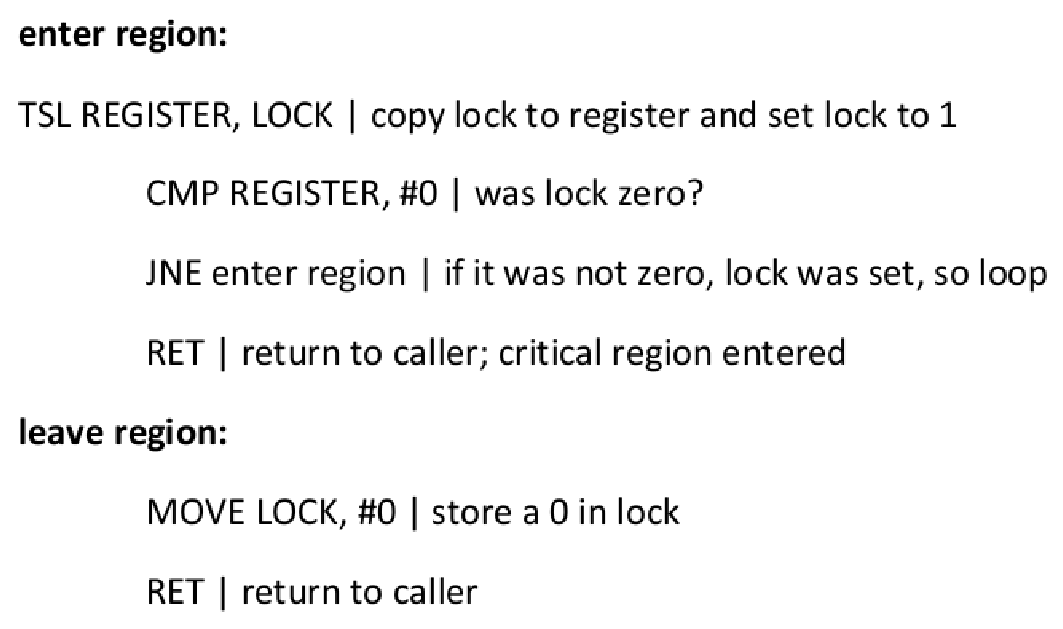
```
while (TRUE) {                                      while (TRUE) {
    while (turn != 0)       /* loop */ ;                while (turn != 1)       /* loop */ ;
    critical_region( );                                 critical_region( );
    turn = 1;                                           turn = 0;
    noncritical_region( );                              noncritical_region( );
}                                                   }
```

   - Violates condition 3(No process running outside its critical region may block any process.) when a process is faster than other.
4. **Peterson’s Solution**
   - Peterson’s Solution preserves all three conditions :
     - Mutual Exclusion is assured as only one process can access the critical section at any time.
     - Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
     - Bounded Waiting is preserved as every process gets a fair chance.

   Disadvantages

     - Peterson’s solution works for two processes, but this solution is best scheme in user mode for critical section.
     - This solution is also a busy waiting solution so CPU time is wasted. So that **“SPIN LOCK”** problem can come. And this problem can come in any of the busy waiting solution.
5. **The TSL Instruction(Test and Set Lock)**

- It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.
- The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.



```
enter region:
TSL REGISTER, LOCK | copy lock to register and set lock to 1
        CMP REGISTER, #0 | was lock zero?
        JNE enter region | if it was not zero, lock was set, so loop
        RET | return to caller; critical region entered
leave region:
        MOVE LOCK, #0 | store a 0 in lock
        RET | return to caller
```

- In both Peterson's solution and TSL Instruction depends on processes not cheating and calling the leave region after the critical region.

2.3.4 Sleep and Wakeup

- Unfortunately, even though busy waiting is a solution to achieve mutual exclusion, these methods waste a lot of CPU time.
- Also in some cases, these methods can have unexpected results such as **priority inversion problem.**
- Instead, we can use sleep and wakeup calls. By using these two function we can save a lot of CPU time while protecting critical region.
- To show this solution, we can use **Producer-Consumer problem.** In this example, even though we achieve mutual exclusion, there is a risk for fatal race condition where both producer and consumer goes indefinite sleep and never wake up.
- Solving this problem is possible with **wakeup waiting bit** but when we have more than two or three processes, it is likely to change **wakeup waiting bit** insufficiently. Instead, we can use Semaphores to solve this problem.

2.3.5 Semaphores

- Semaphore is an integer variable that is shared between threads to solve critical region problem by using two atomic operations. These operations are **wait** and **signal**.
    - **wait(),** decrements the semaphore variable and if it is 0 it doesn't decrement just waits until variable is not 0 (not busy wait)
    - **signal(),** increments the semaphore variable, saying that other processes can enter to critical region
- Semaphore means a signaling mechanism whereas Mutex is a locking mechanism
- There are two types of semaphores:
    1. Counting Semaphore
    2. Binary Semaphore
- Biggest issue in semaphore is **priority inversion problem.** Because if a low priority process is in the critical section, then no other higher priority process can get into the critical section.
- And second issue is, these wait and signal operations must be done in correct order, thus implementation of the semaphores become harder.
- You can think semaphores as bouncers at a nightclub. There are a dedicated number of people that are allowed in the club at once. If the club is full no one is allowed to enter, but as soon as one person leaves another person might enter.

2.3.6 Mutexes

- Mutexes best explained with this stackoverflow answer.

> When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk.
>
> Replace Chicken with Mutex and person with thread and you basically have the concept of a mutex.
>
> Of course, there is no such thing as a rubber mutex. Only rubber chicken. My cats once had a rubber mouse, but they ate it.
>
> Of course, before you use the rubber chicken, you need to ask yourself whether you actually need 5 people in one room and would it not just be easier with one person in the room on their own doing all the work. Actually, this is just extending the analogy, but you get the idea.

- Mutex is a **locking mechanism** used to synchronize access to a resource. Only one task can acquire the mutex. It means there is ownership associated with a mutex, and only the owner can release the lock (mutex).
- A **binary semaphore** is different from mutex because it can be signaled **by any thread** (or process)

2.3.7 Monitors

- A **Monitor** is an object designed to be accessed from multiple threads.
- The member functions or methods of a monitor object will enforce mutual exclusion, so only one thread may be performing any action on the object at a given time.
- If one thread is currently executing a member function of the object then any other thread that tries to call a member function of that object will have to wait until the first has finished.
- A monitor is like a public toilet. Only one person can enter at a time. They lock the door to prevent anyone else coming in, do their stuff, and then unlock it when they leave.

2.3.8 Message Passing

2.3.9 Barriers

2.3.10 Avoiding Locks: Read-Copy Update

2.4 Scheduling

2.4.1 Introduction to Scheduling

2.4.2 Scheduling in Batch Systems

First-Come, First-Served (FCFS)

- Jobs are executed on first come, first serve basis
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Shortest Job First/Next (SJF or SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Shortest Remaining Time Next (SRT)

- Shortest remaining time is the preemptive version of the **Shortest Job First** algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

2.4.3 Scheduling in the Interactive Systems

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Priority Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Multiple Queues

- Has priority classes
- Highest priority job runs for 1 quantum, second 2 quanta 4,8,16…
- Processes start as highest priority and reduced to lower priority class each time it is called (Less context switches each time)

Shortest Process Next

- (Problem: How to know running times, (assumption on past behavior))

Guaranteed Scheduling

- Gives promises to users and lives up to them.
- First gives 1/n of the CPU cycles to each process. Then checks the used time/given time ratios and updates the given time accordingly.

Lottery Scheduling

- Processes are given lottery tickets.
- More important processes can be given extra tickets. Processes can exchange tickets

Fair-Share Scheduling

- Normally, If there are 2 users, first with 8 processes and second with 2 processes, first will have the 80% of the CPU time.
- Fair-Share scheduling prevents this and gives them both 50% of the CPU time.

2.4.4 Scheduling in Real Time Systems

- Behavior of real-time processes are known in advance by the system
- Can be non-preemptive since they are not general purpose.
- Processes blocks themselves when needed so no need for preemption.
- **Hard real time**: absolute deadlines,
- **Soft real time:** missing deadlines occasionally is undesired but tolerable.

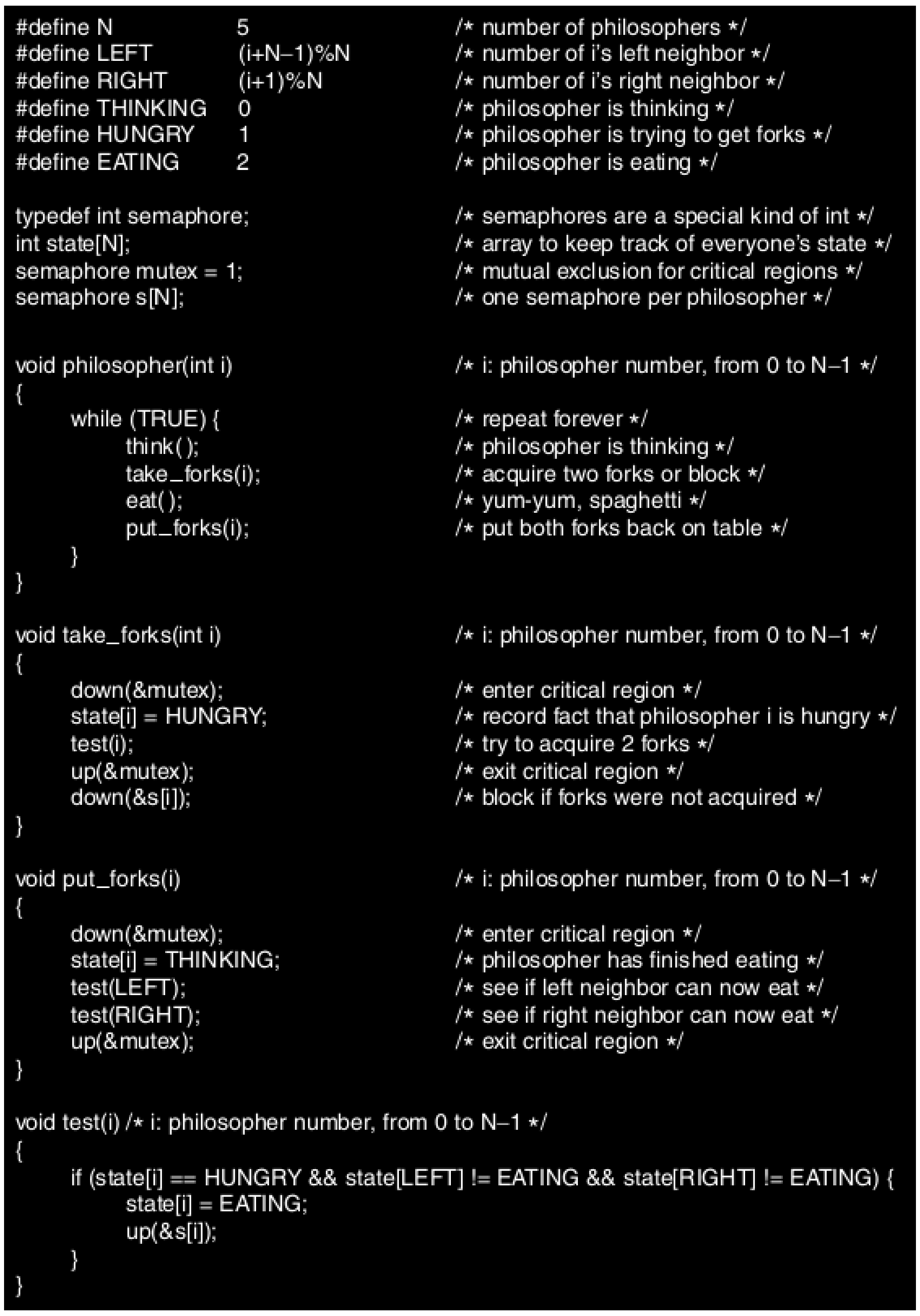
2.4.5 Policy vs Mechanism

2.4.6 Thread Scheduling

2.5 Classical IPC Problems

2.5.1 The Dining Philosophers Problem

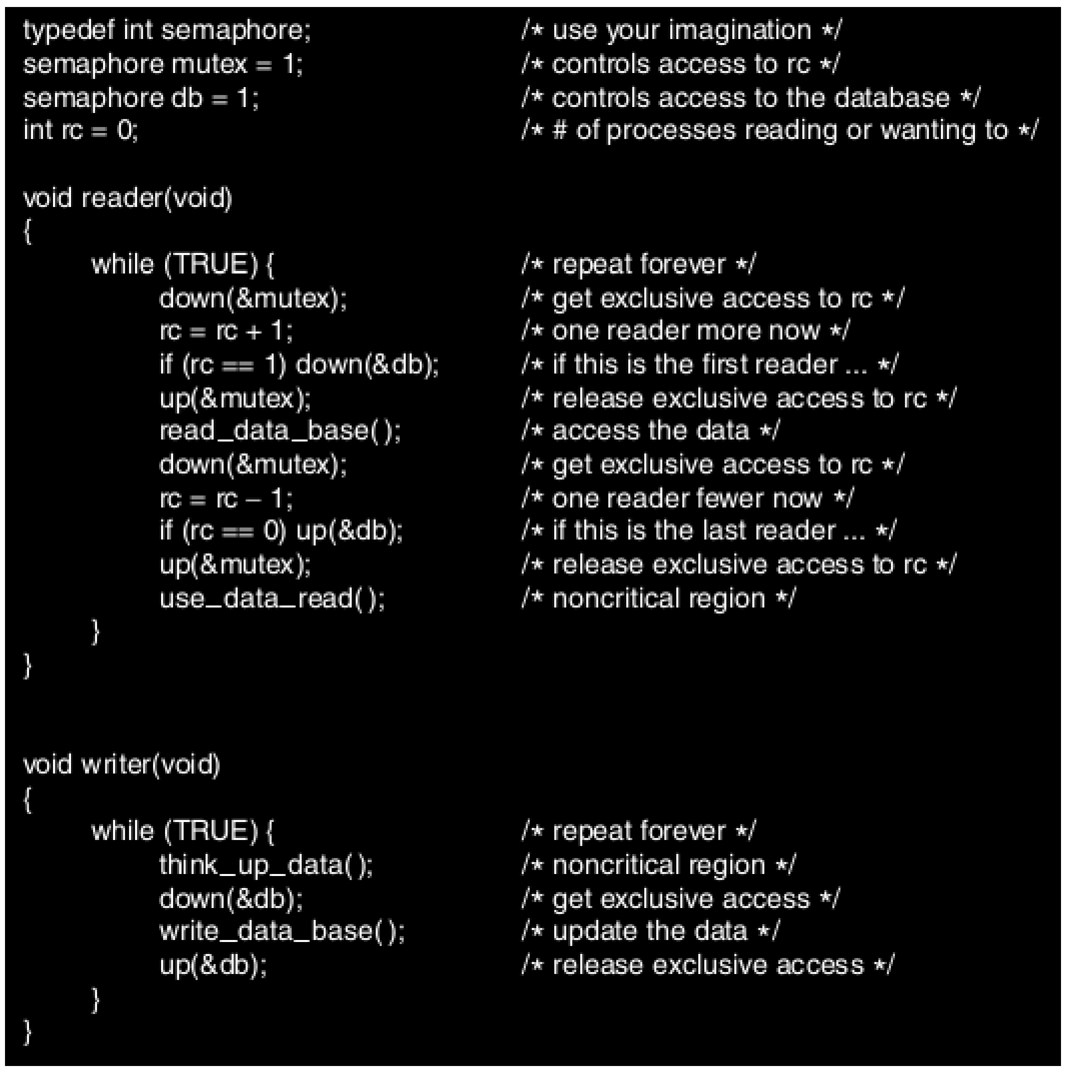
- Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti.
- The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.
- When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.
- The question is : Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?
- The answer to that question is **yes.** Easiest way to solve this question is by using semaphore.



```
#define N           5               /* number of philosophers */
#define LEFT        (i+N−1)%N       /* number of i's left neighbor */
#define RIGHT       (i+1)%N         /* number of i's right neighbor */
#define THINKING    0               /* philosopher is thinking */
#define HUNGRY      1               /* philosopher is trying to get forks */
#define EATING      2               /* philosopher is eating */

typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */


void philosopher(int i)             /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}

void take_forks(int i)              /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
    test(i);                        /* try to acquire 2 forks */
    up(&mutex);                     /* exit critical region */
    down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                   /* enter critical region */
    state[i] = THINKING;            /* philosopher has finished eating */
    test(LEFT);                     /* see if left neighbor can now eat */
    test(RIGHT);                    /* see if right neighbor can now eat */
    up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

2.5.1 The Readers and Writers Problem

- Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.
- The question is how do you program the readers and the writers?
- Answer is simple: by using semaphores. But this time we need to be careful.
- When a writer suspended, next reader shouldn't be admitted to the critical region. Instead writer should be admitted immediately.



```
typedef int semaphore;            /* use your imagination */
semaphore mutex = 1;              /* controls access to rc */
semaphore db = 1;                 /* controls access to the database */
int rc = 0;                       /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                /* repeat forever */
        down(&mutex);             /* get exclusive access to rc */
        rc = rc + 1;              /* one reader more now */
        if (rc == 1) down(&db);   /* if this is the first reader ... */
        up(&mutex);               /* release exclusive access to rc */
        read_data_base();         /* access the data */
        down(&mutex);             /* get exclusive access to rc */
        rc = rc – 1;              /* one reader fewer now */
        if (rc == 0) up(&db);     /* if this is the last reader ... */
        up(&mutex);               /* release exclusive access to rc */
        use_data_read();          /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                /* repeat forever */
        think_up_data();          /* noncritical region */
        down(&db);                /* get exclusive access */
        write_data_base();        /* update the data */
        up(&db);                  /* release exclusive access */
    }
}
```

3. Memory Management

- In this chapter, we will discuss how to manage RAM.
- Main memory(RAM) is a resource and operating systems have to manage it

3.1 No Memory Abstraction

- In early main-frame computers (1960, 1970 and 1980) there was no memory abstraction.
- When there is no abstraction it is very risky to have more than one processes running inside main memory. Because a bug in a program can wipe out entire system since they access pyhsical memory addresses directly
- To avoid a program to write another program’s address space, they added static relocation. Thus, making a call which access a memory address can be changed during execution by adding the offset

3.2 Address Spaces

3.2.1 The Notion of a Address Space

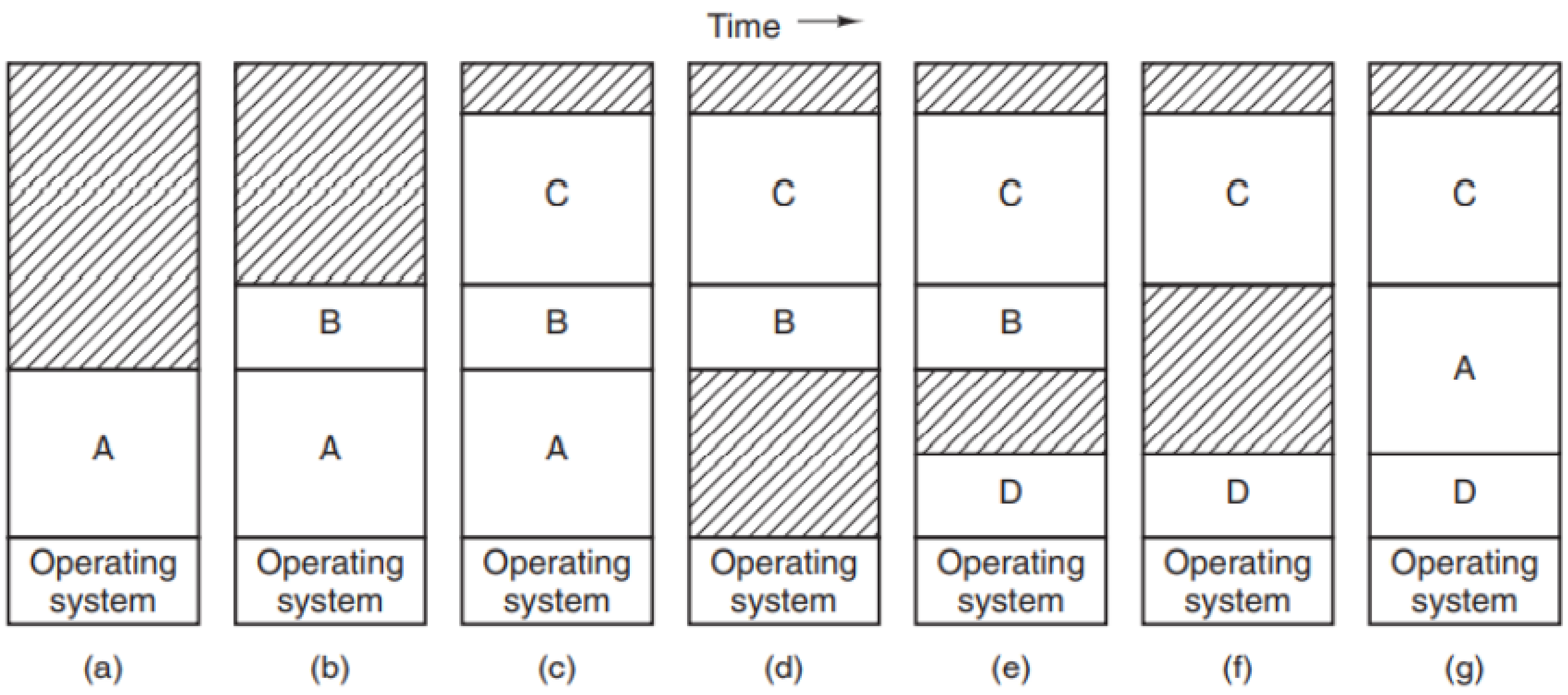
- Multiprogramming and memory protection are not easy without some kind of memory abstraction.
- An address space is the set of addresses that a process can use to address memory.
- Each process has its own address space, independent of those belonging to other processes.

  Base and Limit Registers

  - Base – Starting address of the program
  - Limit – Length of the program
  - It is a simple version of **dynamic relocation.**
  - These method is an easy way to give each process its own private address space.
  - Base and limit registers are protected and only the OS can change them during swapping processes
  - A disadvantage of relocation using base and limit registers is the need to perform an addition and a comparison on every memory reference
  - Comparisons can be done fast, but additions are slow due to carry-propagation time unless special addition circuits are used.

3.2.2 Swapping

- Main idea in swapping is bringing in each process in its entirety, running it for a while, then putting it back on the disk.
- Idle processes are mostly stored on disk, so they do not take up any memory when they are not running

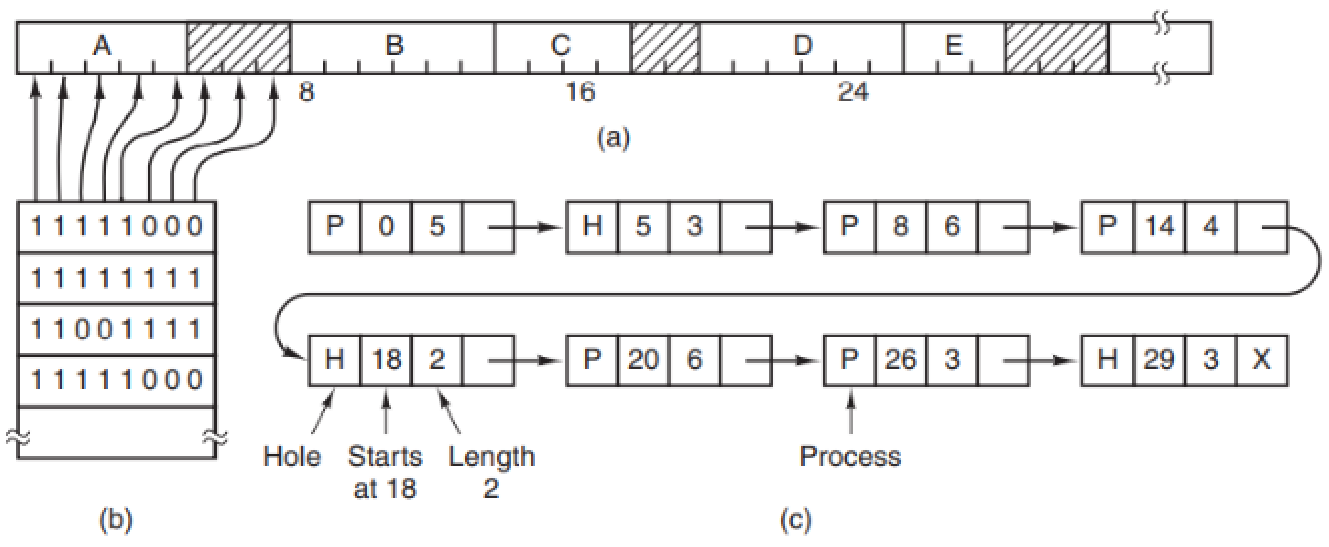


- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.
- This technique is known as **memory compaction**
- It is usually not done because it requires a lot of CPU time.

3.2.3 Managing Free Memory

- While managing free memory, memory is divided into small segments (units)
- There are 2 ways to manage free memory:

1. **Bitmap**
    - A bitmap is kept in memory to address occupied and unoccupied segments of the memory.
    - Each bit corresponds to a memory segment.
    - If the segments are small, bit-map is too big and If the segments are big, memory is wasted since segments won't be fully occupied.
    - Problem with the bitmap is, when a k-unit process is decided to loaded to memory, a search on bitmap should be made to find k adjacent 0 bits. This is slow.
2. **Linked List**
    - Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next item.



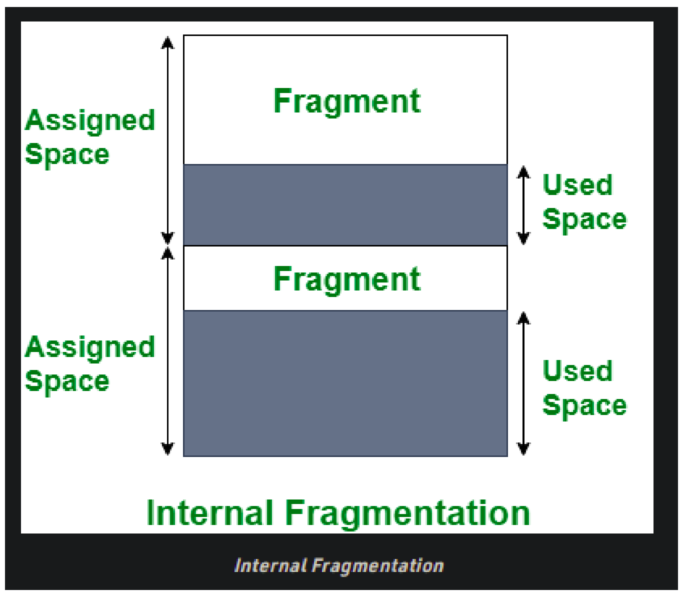
- While managing free memory, there are 5 different algorithms for finding an available space inside a memory:

1. First Fit
    - The memory manager scans the list of segments and finds a hole big enough, breaks it into two pieces if the hole is not an exact fit. (Fast)
2. Next Fit
    - Same with first fit but starts to scan from where it left. (Simulation Result: slightly worse than first fit.)
3. Best Fit
    - Searches the entire list and takes the smallest hole which is adequate.
    - Faster and wastes more memory since it breaks the small portions into two to place to process and remaining portion is too small to place a new process (Too many unused, small memory segments)
4. Worst Fit
    - Put the new process to largest segment (Simulation Result: It is worse than first fit)
5. Quick Fit
    - Maintain separate lists for more commonly requested sizes. A table holds pointers to these separate lists.
    - Finding holes is extremely fast but merging has to be done time to time in order to avoid having unusable small segments.

- You can keep the processes and holes in 2 separate lists and make the holes list sorted.
- This makes best fit and first fit algorithms to be same. But it is slow since when deallocation happens, you have to remove an element from processes list and put it into sorted holes list.
- Before starting to virtual memory section, internal and external fragmantation must be covered.
- These fragmantations create problem since it means we didn't use main memory efficiently

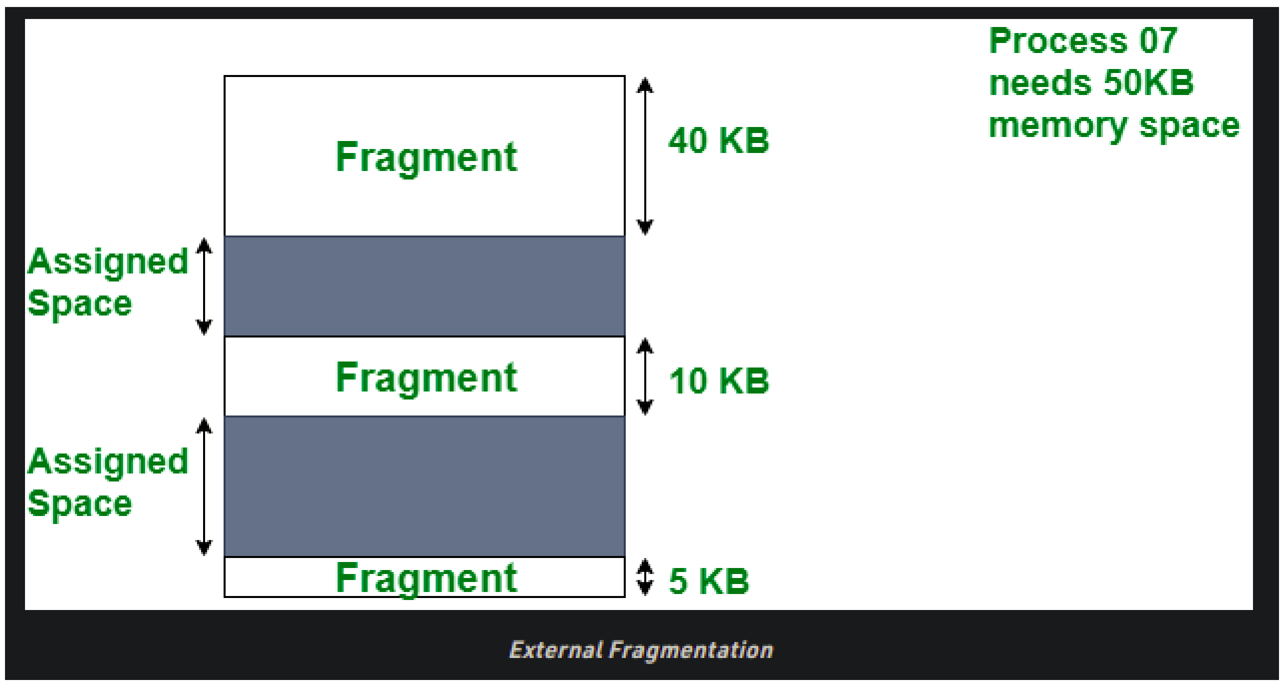
  Internal Fragmentation

    - Internal fragmentation happens when the method or process is larger than the memory.
    - Where the memory allotted(allocated) to the method is somewhat larger than the memory requested, then the difference between allotted and requested memory is called internal fragmentation.
    - The solution of internal fragmentation is the best-fit block.
    - Internal fragmentation occurs with paging and fixed partitioning.



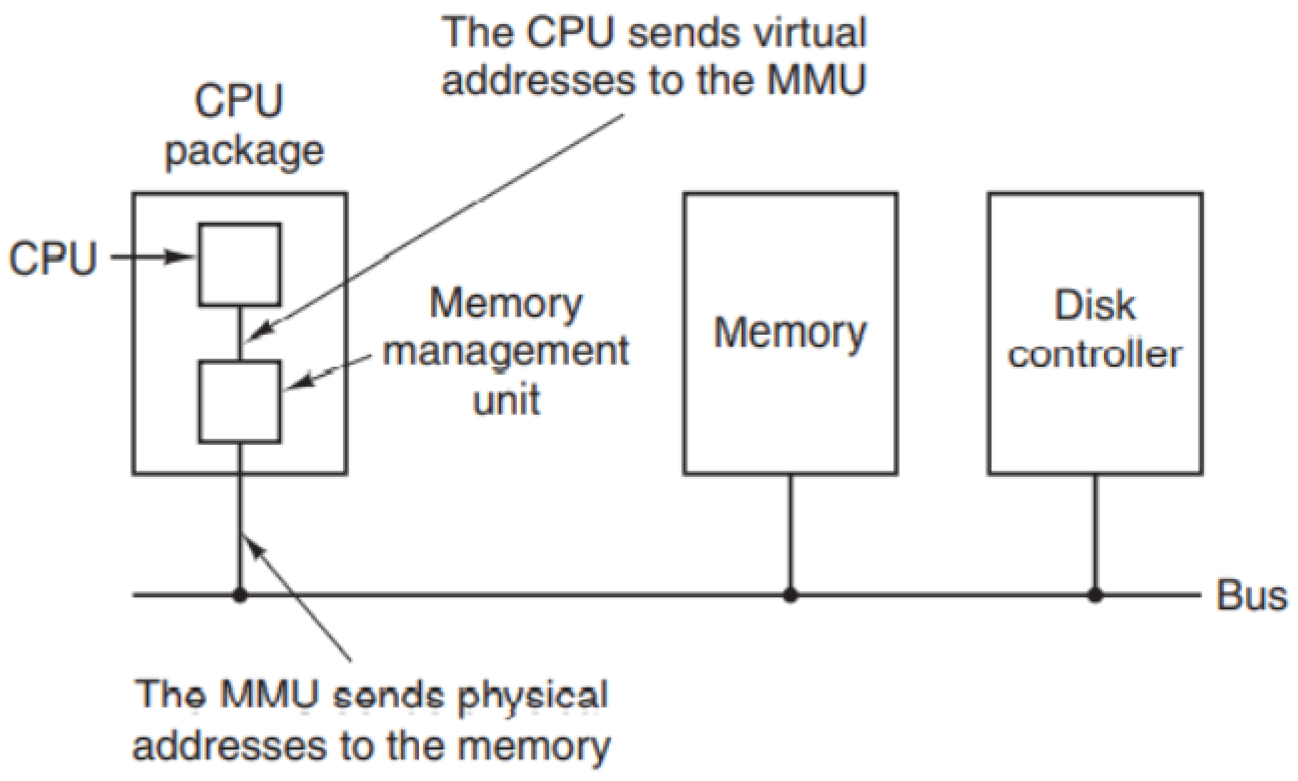
  External Fragmentation

    - External fragmentation happens when the method or process is removed.
    - Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation.
    - The solution to external fragmentation is compaction and paging.
    - External fragmentation occurs with segmentation and dynamic partitioning.



3.3 Virtual Memory

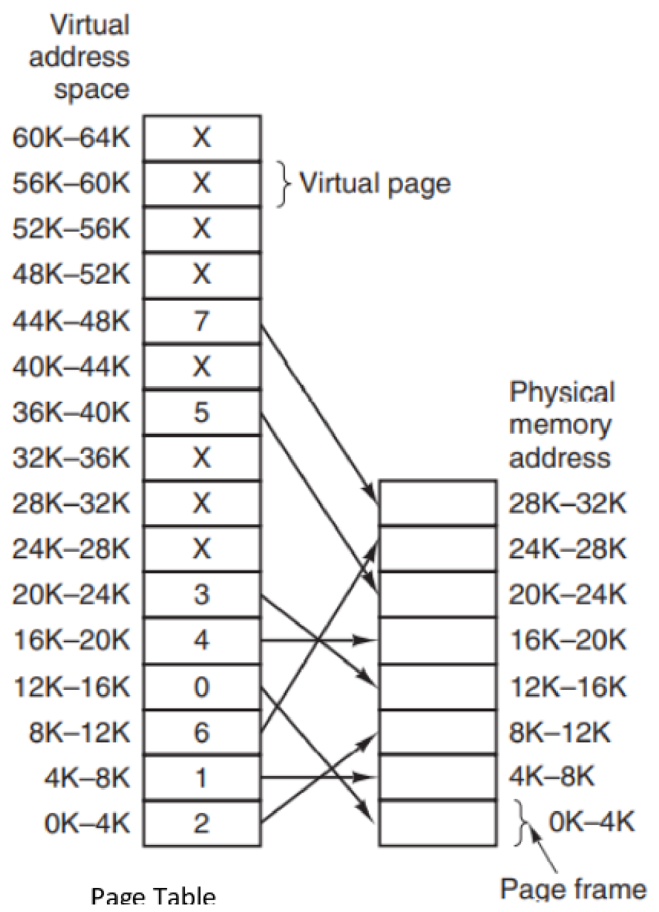
- Before virtual memory there was the concept of **overlays**.
- Processes were cut into pieces of overlays and loaded to memory one by one (if there is space above the previous overlay, if not on top of it)
- This “cutting processes into pieces” was done by programmers and wasn’t an easy job. So **Virtual Memory** is found.
- Each program has its own **address space**, which is broken up into chunks called **pages**.
- Each page is a contiguous range of addresses.
- These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program.
- When the program references a part of its address space that is in physical memory, the hardware performs the necessary mapping on the fly.
- When the program references a part of its address space that is **not** in physical memory, the operating system is alerted to go get the missing piece and re-execute the instruction that failed
- The programs need not to be an contigiues areas of pyshical memory anymore. This eliminates all problems of allocation inefficiency, compaction, relocation, memory protection, etc. No fragmantation occurs with virtual memory.
- Program-generated addresses are called **virtual addresses** and form the **virtual address space**.
- **MMU(Memory Management Unit)** maps the virtual addresses onto the physical memory addresses.


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

- Virtual Memory uses a technique called **paging**.

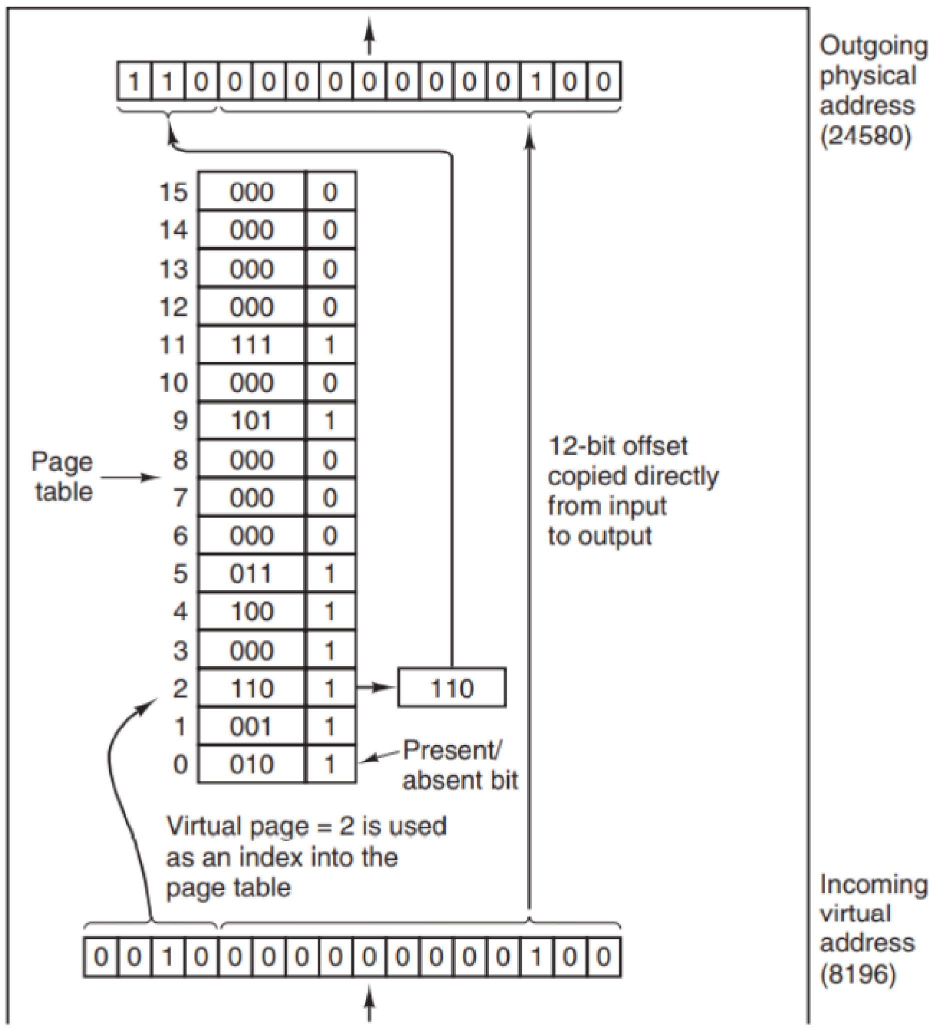
3.3.1 Paging

- The virtual address space consists of **fixed-size units** called **pages**.
- The corresponding units in the **physical memory** are called **page frames**



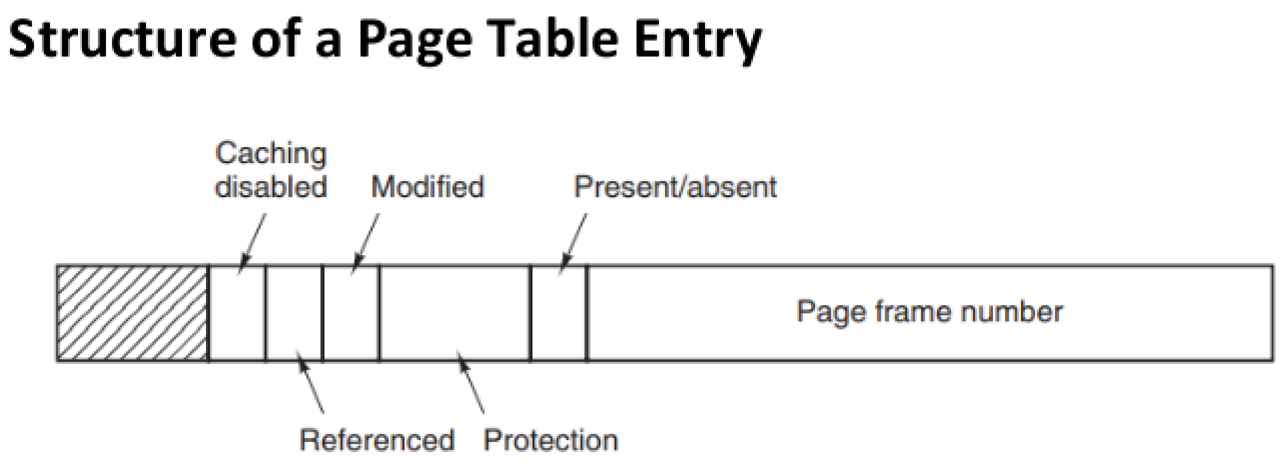
- When the program tries to access address 0, for example, using the instruction MOV REG,0 virtual address 0 is sent to the MMU.
- The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287).
- It thus transforms the address to 8192 and outputs address 8192 onto the bus.
- The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors.
- Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.
- Since we have only eight physical page frames, only eight of the virtual pages are mapped onto physical memory.
- The others, shown as a cross in the figure, are not mapped
- In the actual hardware, a Present / Absent bit keeps track of which pages are physically present in memory.
- When the unmapped addresses (present bit 0) are referenced, MMU notices it and causes the CPU to trap to the operating system.
- This trap is called **page fault**.
- Operating system picks a little-used page frame and if it is changed, writes it contents back to disk and then fetches the referenced page from disk into the page frame just freed, changes the map and restarts the trapped instruction.

  How does MMU work?



- 16-bit Virtual address 8196 comes.
- 0010000000000100 split into 4 (because there are 16 pages) and 12(because each page is 4KB (2^12)) bytes 4bit page number, 12bit offset.
- 4bit part is index of the page table.
- 3-bit number which page table holds at that index is added to offset and it is the physical address.

3.3.2 Page Tables



- The purpose of the page table is to map virtual pages onto page frames.
- The **Protection bits** tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.
- **The Modified and Referenced bits** keep track of page usage.
- When a page is written to, the hardware automatically sets the **Modified bit**. This bit is of value when the operating system decides to reclaim a page frame.
- If the page in it has been modified (i.e., is ‘‘dirty’’), it must be written back to the disk. If it has not been modified (i.e., is ‘‘clean’’), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the dirty bit, since it reflects the page’s state.
- **The Referenced bit** is set whenever a page is referenced, either for reading or for writing.
- Its value is used to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are far better candidates than pages that are.

- **Caching disabled bit:** If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off.

3.3.3 Speeding Up Paging

- There are 2 major problem that need to be considered when using Virtual Memory
    1. The mapping from virtual address to physical address must be fast. Since it is done at every memory reference.
    2. If the virtual address space is large, the page table will be large. For example 2- bit address space has 1 million pages and each process has its own page table.

    #### Translation Lookaside Buffers (TLB)

    - TLB is kind of cache that keeps the most frequently used page table address in MMU
    - It is usually the MMU and consists of a small number of entries.
    - Each entry contains info about one page, including virtual page number, modified bit, protection bits, and the physical page frame in which the page is located.
    - These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table.
    - When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). Doing so requires special hardware, which all MMUs with TLBs have.
    - If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table.
    - If the virtual page number is present in the TLB, but the instruction is trying to write on a read-only page, a protection fault is generated.
    - The interesting case is what happens when the virtual page number is not in the TLB.
    - The MMU detects the miss and does an ordinary page table lookup.
    - It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus, if that page is used again soon, the second time it will result in a TLB hit rather than a miss.
    - When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there, except the reference bit.
    - When the TLB is loaded from the page table, all the fields are taken from memory.
    - There are 2 miss types in TLB:
        - **Soft miss:** page is not in TLB but in memory.
        - **Hard Miss:** page is not in TLB and memory, Disk access is required. Million times slower than soft miss.
    - There are 2 types of **page fault** for page tables:
        - **Minor Page Fault:** Page is not in the processes page table, but in some other processes page table. No need for disk access.
        - **Major Page Fault:** Page is not in memory.

    #### Difference Between A Standart Lookup and Cache Lookup

    - With **Standart lookup,** you give a entry number 7 and it gives address back(for example 000)
    - With **Cache lookup,** you don't say give me a number 7, you say that you have the virtual page number 19 and it will return page frame number

3.3.4 Page Tables for Large Memories

3.4 Page Replacement Algorithms

3.4.1 The Optimal Page Replacement Algorithm (impossible)

- The page which won't be used for the longest time should be replaced.

- This is not possible since OS can’t know when each of the pages will be referenced next.
- This algorithm is used on simulations to compare other algorithms time with this one.

3.4.2 The Not Recently Used Page Replacement Algorithm

3.4.3 FIFO Page Replacement Algorithm

3.4.4 The Second Chance Page Replacement Algorithm

3.4.5 The Clock Page Replacement Algorithm

3.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

3.4.7 Simulating LRU in Software

3.4.8 The Working Set Page Replacement Algorithm

3.4.9 The WSClock Page Replacement Algorithm

3.4.10 Summary of Page Replacement Algorithms

3.5 Design Issues For Paging Systems

3.5.1 Local vs Global Allocation Policies

3.5.2 Load Control

3.5.3 Page Size

3.5.4 Separate Instruction and Data Spaces

3.5.5 Shared Pages

3.5.6 Shared Libraries

3.5.7 Mapped Files

3.5.8 Cleaning Policy

3.5.9 Virtual Memory Interface

4. File Systems

4.1 Files

4.1.1 File Naming

- FAT-16, FAT-32, NTFS, ReFS, exFAT
- File Extension

4.1.2 File Structure

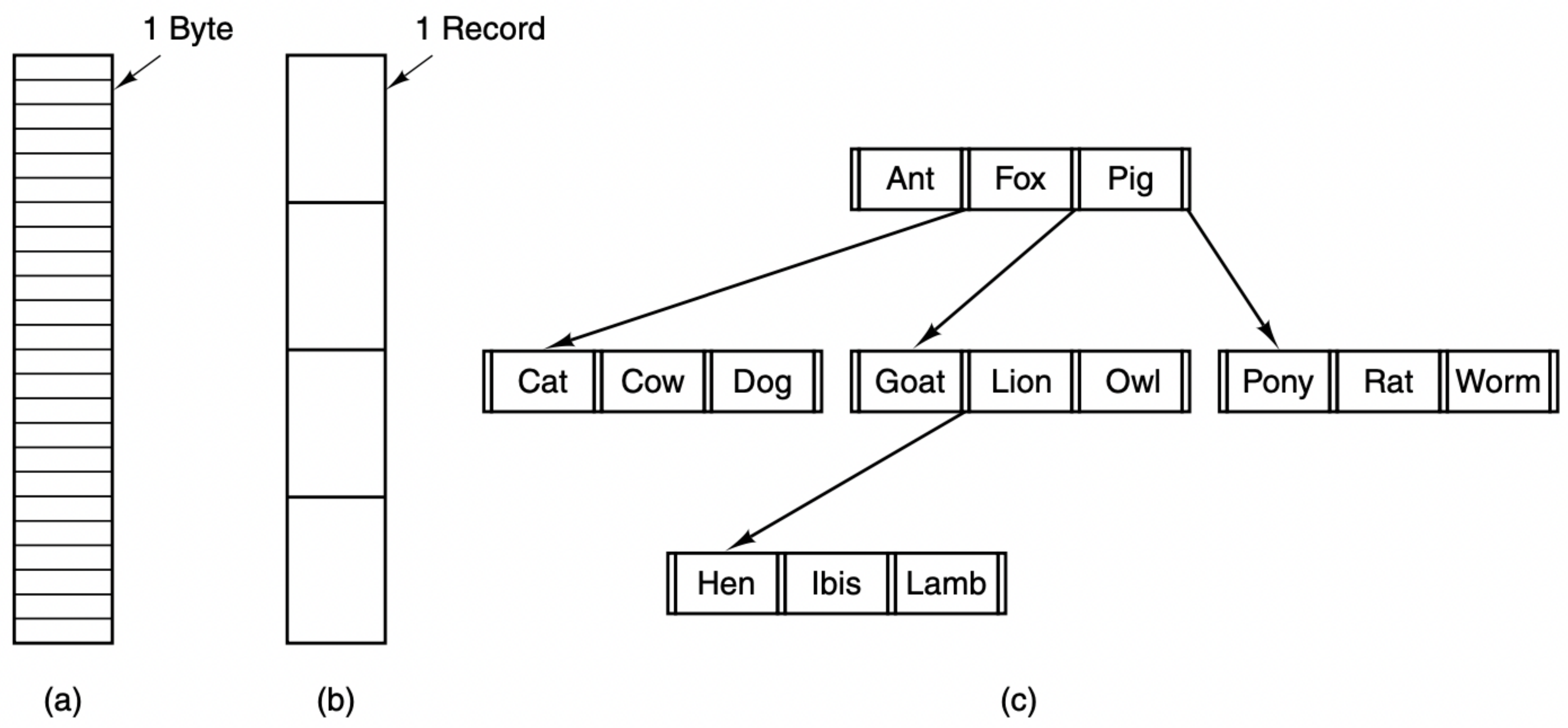


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

4.1.3 File Types

- Regular Files (Windows, UNIX)
    - ASCII Files
    - Binary Files
- Directories (Windows, UNIX)
- Character Special Files (UNIX)
- Block Special Files (UNIX)

4.1.4 File Access

- Sequential Access
- Random Access

4.1.5 File Attributes (Metadata)

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

4.1.6 File Operations

- Create, delete, open, close, read, write, append, seek, get attributes, set attributes, rename

4.2 Directories

4.2.1 Single Level Directory Systems

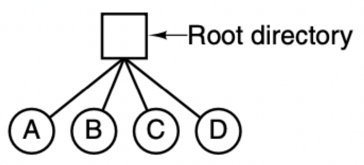


Figure 4-6. A single-level directory system containing four files.

4.2.2 Hierarchical Directory Systems

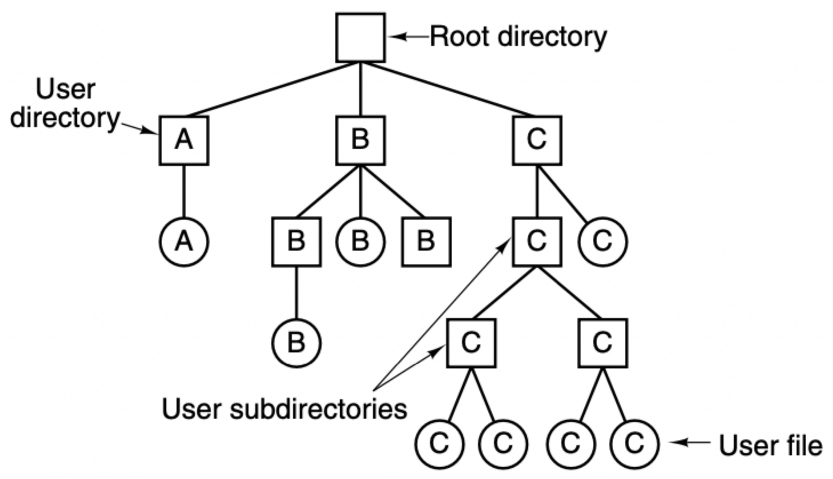


Figure 4-7. A hierarchical directory system.

4.2.3 Path Names

- Absolute path name, relative path name, working directory(current directory)
- “.”(dot) refers to current directory
- “..”(dotdot) refers to parent directory (refers to itself if it is root directory)

4.2.4 Directory Operations

- Create, delete, opendir, closedir, readdir, rename, link, unlink

4.3 File System Implementation

4.3.1 File System Layout

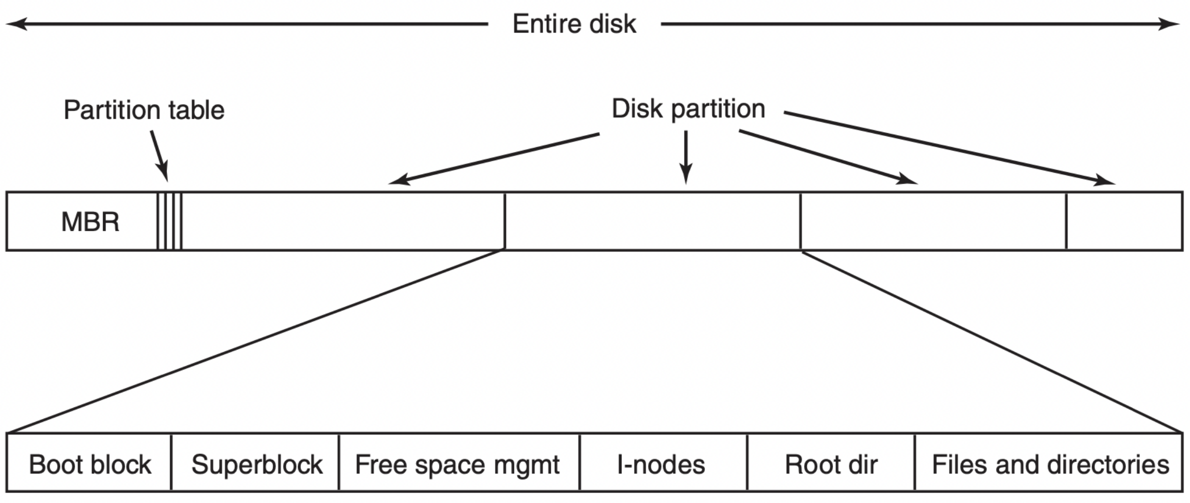
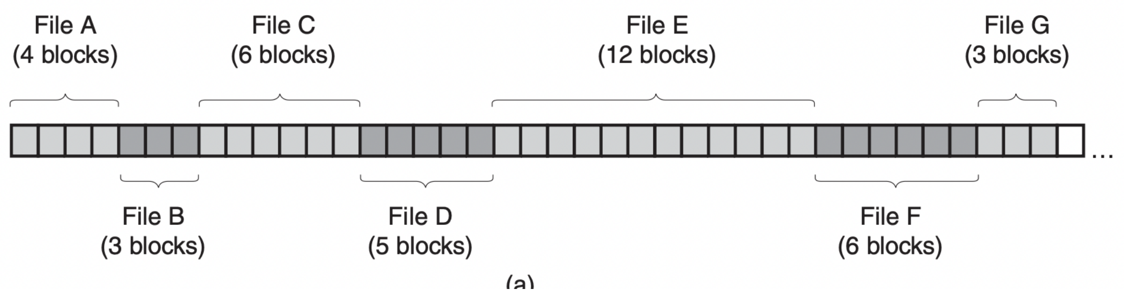


Figure 4-9. A possible file-system layout.

- **MBR** → Master Boot Record (used to boot computer)
- After power on computer : BIOS → MBR → Active Partition → Boot Block → Boot Loader

4.3.2 Implementing Files

- **Contiguous Allocation :** Simple to implement, high performance, external fragmentation, internal fragmentation



- **Linked List Allocation :** No external fragmentation, really slow random acces

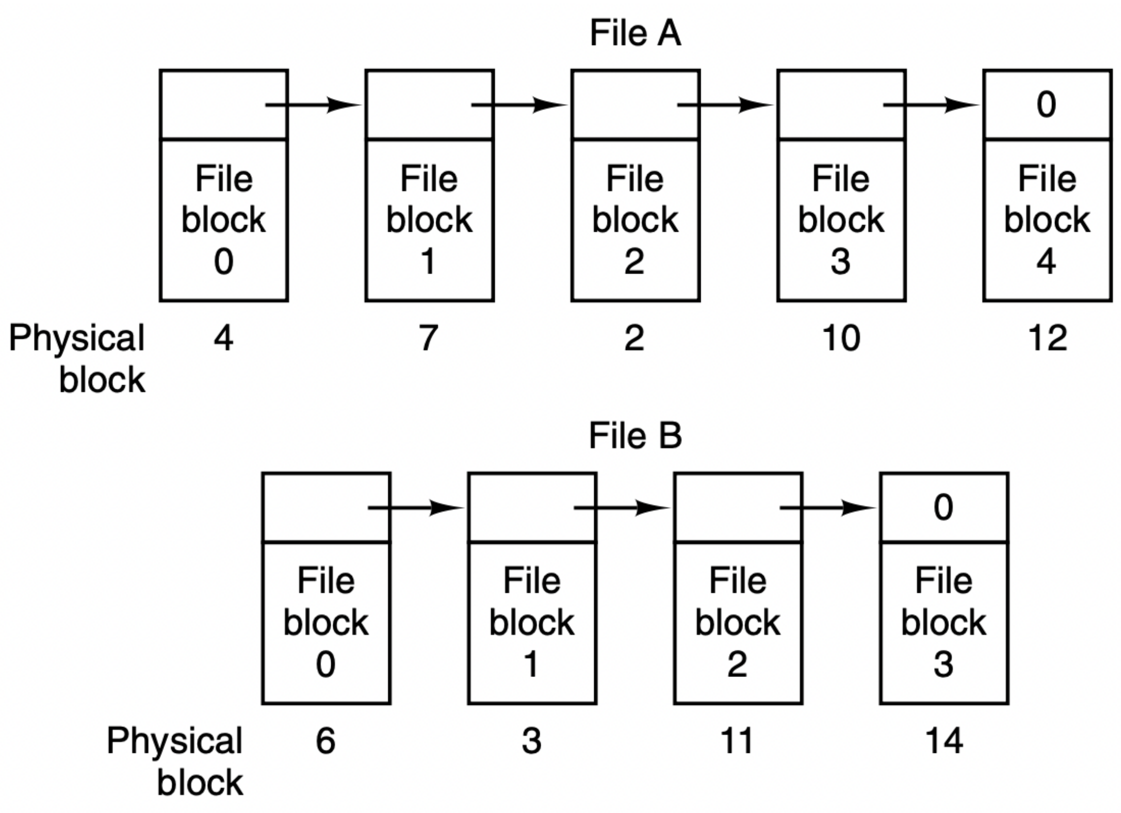


Figure 4-11. Storing a file as a linked list of disk blocks.

- **Linked List Allocation Using a Table in Memory :** Fast random access, no external fragmentation, very high main memory(RAM) consumption

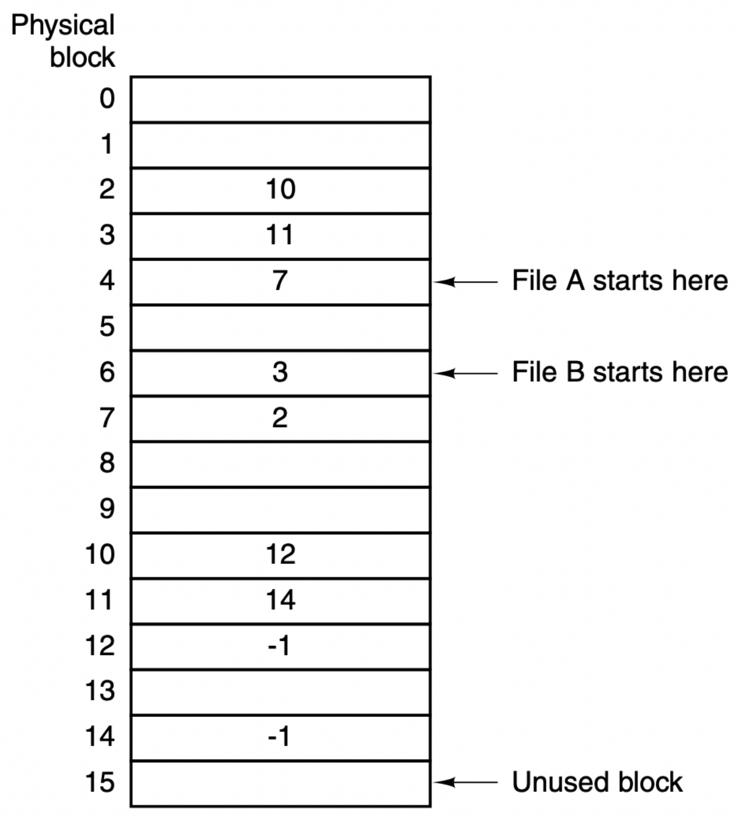


Figure 4-12. Linked-list allocation using a file-allocation table in main memory.

- **I-nodes :** Only in memory when corresponding file is open, i-node capacity might be reached before disk capacity full

4.3.3 Implementing Directories

4.3.4 Shared Files

4.3.5 Log-Structured File Systems

4.3.6 Journaling File Systems

4.3.7 Virtual File Systems

4.4 File System Management and Optimization

4.4.1 Disk-Space Management

4.4.2 File-System Backups

4.4.3 File-System Consistency

4.4.4 File-System Performance

4.4.5 Defragmenting Disks

4.5 Example File Systems

4.5.1 The MS-DOS File System

4.5.2 The UNIX V7 File System

4.5.3 CD-ROM File System

5. Input / Output

5.1 Principles of I/O Hardware

5.1.1 Input/Output Devices

- There are two types of I/O devices(roughly)

  1. Block Devices

     - Stores information in fixed-size blocks
     - Each one has their own address
     - Common block sizes range from 512 byte to 65536 byte
     - Most important thing about block devices is, they can be read or writen independently from each other

        - A hard-disk can be an example

    2. Character Devices
        - Delivers or accepts a stream of characters
        - Doesn’t have any block structure
        - It is not addressable
        - Printers, mouse can be an example

- Boundaries between block and character devices are not well defined
- Clocks for example, they are not addressable but in the same time they don’t generate or accept a stream of characters
- The file system, only deals with abstract block devices

5.1.2 Device Controllers

- I/O units consist of **mechanical component** and **electronic component**
- Electronic components are also called **device controllers** or **adapter**
- Interface between device controller and device is often very low level
- We can think a disk formatted with 2,000,000 sectors of 512 bytes per track.
- From drive it is read as :
    - **Serial Bit Stream**(starting with a **preamble**) → **4095 bits in a sector** → **Checksum** or **ECC**(Error-Correcting Code)
- Device controller’s job is to convert the serial bit stream into a block of bytes and perform any error correction necessary

5.1.3 Memory-Mapped I/O

- Controllers have their registers to communicate with the CPU
- By writing to these registers, OS can command to device:
    - Deliver data
    - Accept data
    - Switch itself on/off, etc.
- By reading from these registers OS can learn:
    - The device’s state
    - Is it prepared to accept a new command, etc.
- But how do **CPU** communicates with the **controller registers**? There is 2 alternative for that:
    1. In first approach, each control register is assigned to an **I/O port** number. Every port number is stored in **I/O Port Space(**protected from user space**).** Address spaces for memory and I/O is separete in this approach.
    2. In second approach, the solution is to map all the control registers into the memory space. Each control register is assigned to an unique memory address. This approach is called **memory-mapped I/O**.

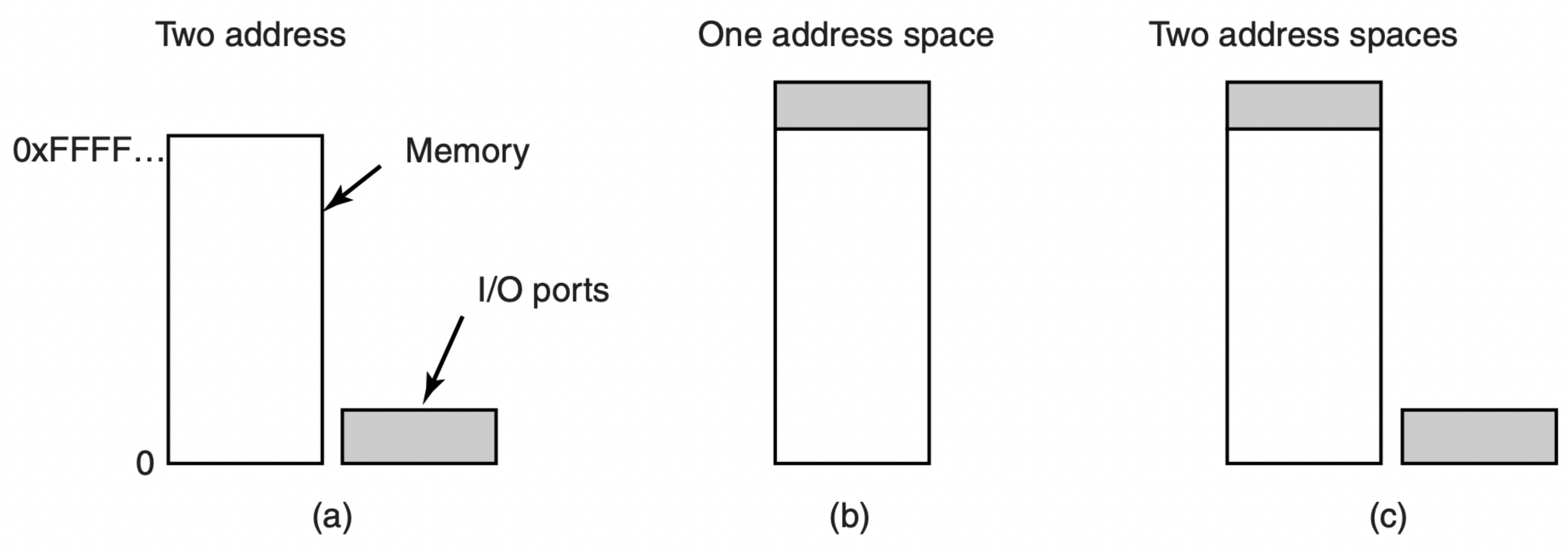


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

- Advantages of memory-mapped I/O:
    1. When special I/O instructions are needed to read and write the device control registers, we need to use **Assembly** since **C/C++** doesn't have any **In / Out** instructions. But with **memory-mapped I/O** these device control registers are just variables in memory and we can access them using C.
    2. There is no need to put special protection mechanism, all you have to do is not putting address space which is containing device control registers into any user's virtual memory
    3. Every instruction that can reference memory can also reference control registers.
- Disadvantages of memory-mapped I/O:
    1. Caching memory words is a common thing for nowadays computers. But this caching has disastrous affects on device control registers and can cause infinite loops. To prevent this, selective caching can be used but this brings extra complexity to the hardware and operating system

5.1.4 Direct Memory Access

5.1.5 Interrupts Revisited

5.2 Principles of I/O Software

5.2.1 Goals of the I/O Software

5.2.2 Programmed I/O

5.2.3 Interrupt-Driven I/O

5.2.4 I/O Using DMA

5.3 I/O Software Layers

5.3.1 Interrupt Handlers

5.3.2 Device Drivers

5.3.3 Device Independent I/O Software

5.3.4 User-Space I/O Software

5.4 Disks

5.4.1 Disk Hardware

5.4.2 Disk Formatting

5.4.3 Disk Arm Scheduling Algorithm

5.4.4 Error Handling

5.4.5 Stable Storage

5.5 Clocks

5.5.1 Clock Hardware

5.5.2 Clock Software

5.5.3 Soft Timers

5.6 User Interfaces : Keyboard, Mouse, Monitor

5.6.1 Input Software

5.6.2 Output Software

5.7 Thin Clients

5.8 Power Management

5.8.1 Hardware Issues

5.8.2 Operating System Issues

5.8.3 Application Program Issues

Important Keywords & Questions

Ontogeny Recapitulates Phylogeny

- Each new generation of computer systems seem to goes through the same phases of its ancestor.
- Mainframes-mini computer - … - smart card

Inverted Page Table

- Inverted Page Table is the global page table which is maintained by the Operating System for all the processes.
- In inverted page table, the number of entries is equal to the number of frames in the main memory.
- It can be used to overcome the drawbacks of page table.

What is BIOS

- BIOS is a program, stands for basic input/output system, which is stored in nonvolatile memory like ROM (Read Only Memory) or flash memory that allows you to set up and access your computer system at the greatest basic level.
- The main function of BIOS is to set up hardware and start an OS, and it contains generic code that is needed to control display screens, the keyboard, and other functions.

Preemptive and Non-Preemptive

- In preemptive scheduling, the CPU is allocated to the processes for a limited time and processes can be interrupted.
- In non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state and the processes can not be interrupted.

Is it important for a user to know if a library function call would make a system call? Explain why or why not ?

- Yes it is important, because if we know that a function call makes a system call, then we know that we can get blocked or it may take too much time.

What is an upcall? How does it address the user level thread problems?

- Upcall is a call from a lower-level subsystem, such as a kernel or framework, to a higher-level subsystem, such as user code.
- Every time a kernel thread blocks on I/O or suffers a page fault, the kernel "activates" the user level thread scheduler and tells it what happened.
- This way, the user level thread scheduler can continue to use the processor by deciding to run some other thread.
- This solves the problems associated with both kernel and user level threads: user level threads have fast context switch times, but suffer badly when blocks or page faults occur, whereas kernel threads handle blocking and faulting very well, but have slow context switch times (which get worse on modern CPU's).

Difference between the Kernel Space and User Space

- **The user space**, which is a set of locations where normal user processes run (i.e everything other than the kernel).
- The role of the kernel is to manage applications running in this space from messing with each other, and the machine.
- Processes running under the user space have access only to a limited part of memory, whereas the kernel has access to all of the memory.

PREVIOUS FINAL QUESTIONS

Is it important for a user to know if a library function