BCA
Fourth Semester
" Operating System"

Unit III

Process

A process is a program in execution. The execution of a process must progress in a sequential fashion. Definition of process is following.

- A process is defined as an entity which represents the basic unit of work to be implemented in the system.

Components of a process are following.

S.N.	Component & Description
1	**Object Program** Code to be executed.
2	**Data** Data to be used for executing the program.
3	**Resources** While executing the program, it may require some resources.
4	**Status** Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

Program

A program by itself is not a process. It is a static entity made up of program statement while process is a dynamic entity. Program contains the instructions to be executed by processor.

A program takes a space at single place in main memory and continues to stay there. A program does not perform any action by itself.

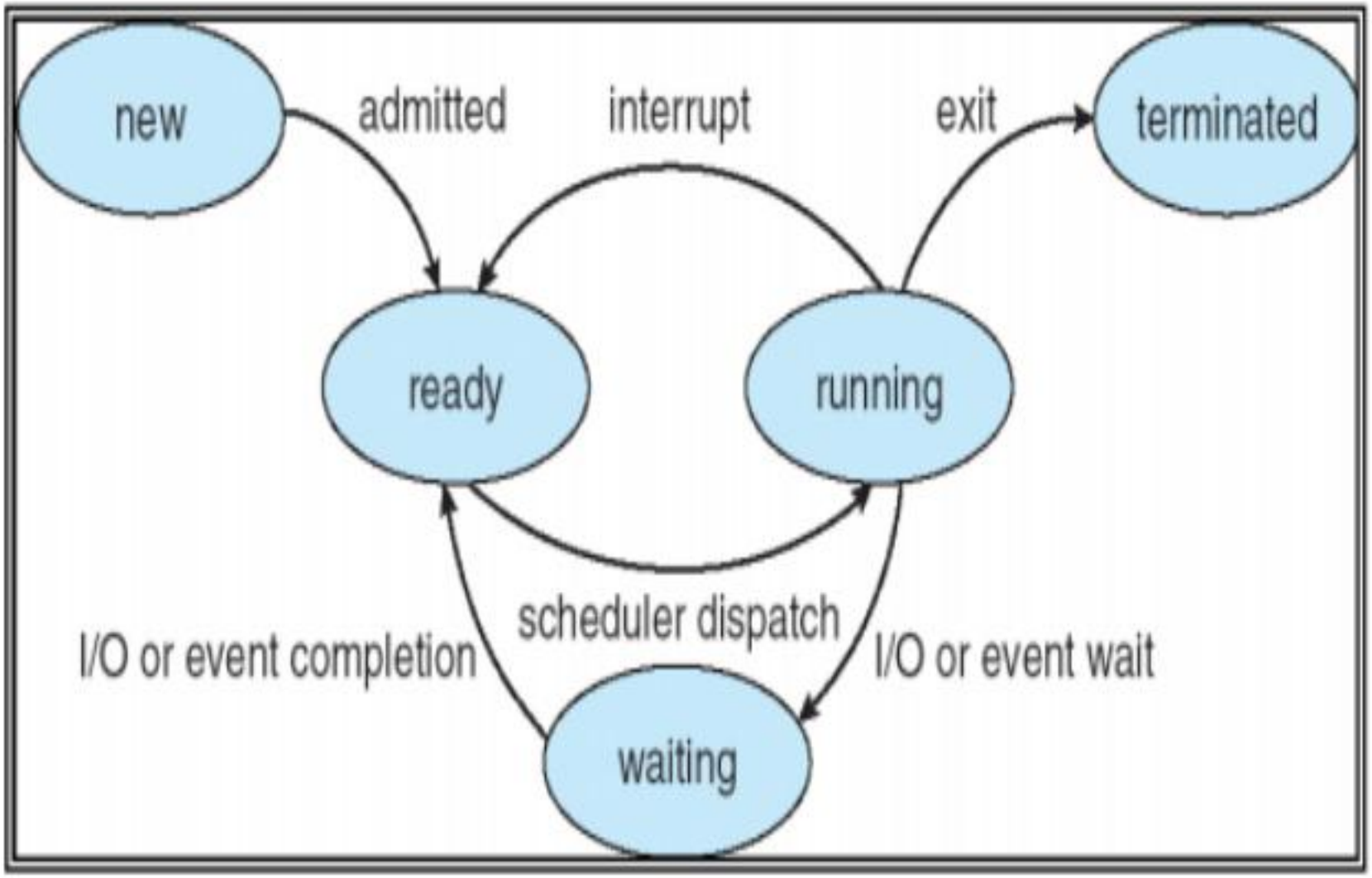
Process States

As a process executes, it changes state. The state of a process is defined as the current activity of the process.

Process can have one of the following five states at a time.

S.N.	State & Description
1	**New** The process is being created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
3	**Running** Process instructions are being executed (i.e. The process that is currently being executed).
4	**Waiting** The process is waiting for some event to occur (such as the completion of an I/O operation).
5	**Terminated** The process has finished execution.

Unit III

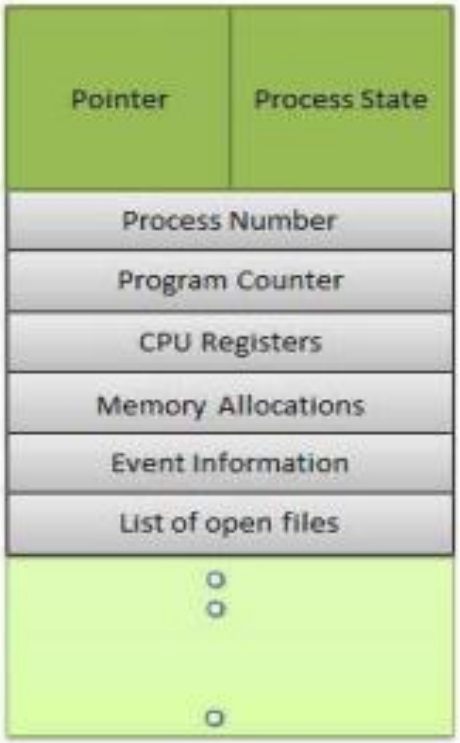

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process Control Block, PCB

Each process is represented in the operating system by a process control block (PCB) also called a task control block. PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process.

PCB contains many pieces of information associated with a specific process which is described below.

S.N.	Information & Description
1	**Pointer** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2	**Process State** Process state may be new, ready, running, waiting and so on.
3	**Program Counter** Program Counter indicates the address of the next instruction to be executed for this process.
4	**CPU registers** CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.
5	**Memory management information** This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for deallocating the memory when the process terminates.
6	**Accounting information** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.



Process control block includes CPU scheduling, I/O resource management, file management information etc. The PCB serves as the repository for any information which can vary from process to process. Loader/linker sets flags and registers when a process is created. If that process gets suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB. By this technique, the hardware state can be restored so that the process can be scheduled to run again.

Operation on Processes

- Several operations are possible on the process. Process must be created and deleted dynamically. Operating system must provide the environment for the process operation. We discuss the two main operations on processes.

1. Create a process
2. Terminate a process

Create Process

- Operating system creates a new process with the specified or default attributes and identifier. A process may create several new subprocesses. Syntax for creating new process is :

CREATE (processed, attributes)

- Two names are used in the process they are parent process and child process.

Parent process is a creating process. Child process is created by the parent process. Child process may create another subprocess. So it forms a tree of processes. When operating system issues a CREATE system call, it obtains a new process control block from the pool of free memory, fills the fields with provided and default parameters, and insert the PCB into the ready list. Thus it makes the specified process eligible to run the process.

- When a process is created, it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc. Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation. When process creates a subprocess, that subprocess may obtain its resources directly from the operating system. Otherwise it uses the resources of parent process.
- When a process creates a new process, two possibilities exist in terms of execution.

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

- For address space, two possibilities occur:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

Terminate a Process

DELETE system call is used for terminating a process. A process may delete itself or by another process. A process can cause the termination of another process via an appropriate system call. The operating system reacts by reclaiming all resources allocated to the specified process, closing files opened by or for the process. PCB is also removed from its place of residence in the list and is returned to the free pool. The DELETE service is normally invoked as a part of orderly program termination.

- Following are the resources for terminating the child process by parent process.

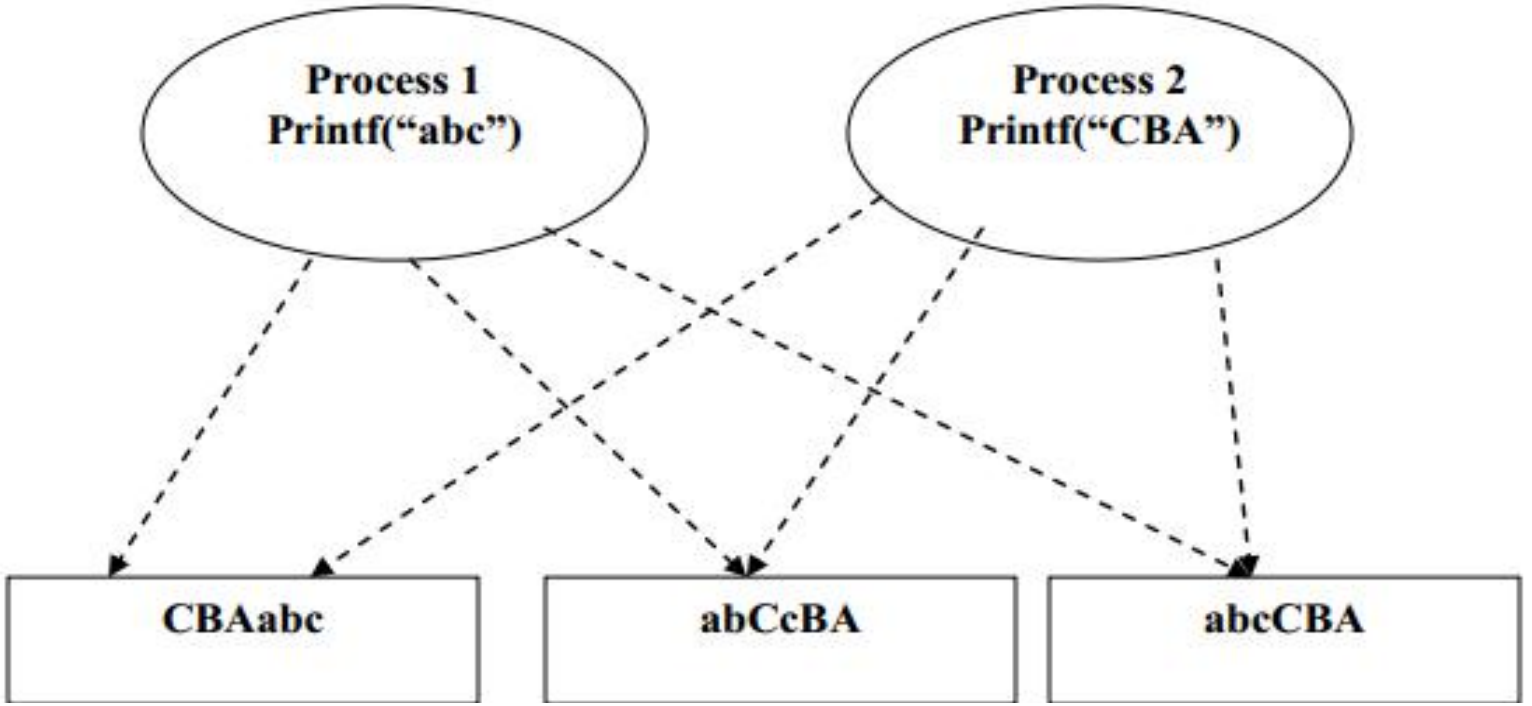
1. The task given to the child is no longer required.
2. Child has exceeded its usage of some of the resources that it has been allocated.
3. Operating system does not allow a child to continue if its parent terminates.

Co-operating Processes

- Co-operating process is a process that can affect or be affected by the other processes while executing. If suppose any process is sharing data with other processes, then it is called co-operating process. Benefit of the co-operating processes are :

1. Sharing of information
2. Increases computation speed
3. Modularity
4. Convenience

- Co-operating processes share the information : Such as a file, memory etc. System must provide an environment to allow concurrent access to these types of resources. Computation speed will increase if the computer has multiple processing elements are connected together. System is constructed in a modular fashion. System function is divided into number of modules.



- Behavior of co-operating processes is nondeterministic i.e. it depends on relative execution sequence and cannot be predicted a priori. Co-operating processes are also Reproducible. For example, suppose one process writes "ABC", another writes "CBA" can get different outputs, cannot tell what comes from which. Which process output first "C" in "ABCCBA". The subtle state sharing that occurs here via the terminal. Not just anything can happen, though. For example, "AABBCC" cannot occur.

Process Hierarchy

- In some computer systems when a process creates another process, then the parent process and child process continue to be associated in certain ways. The child process can itself creates more processes that forms a process hierarchy.
- In Unix system, a process group formed by a process and all of its children and further descendants.
- Whenever a computer user sends a signal from the keyboard, that signal is then delivered to all the members of the process group that are currently associated with the keyboard.
- Individually, each process can catch the signal, ignore the signal, or take the default action, windows systems doesn't have any concept of a process hierarchy.
- Since each process is an independent entity with its own program counter and internal state, processes sometime need to interact with other processes.
- Sometime, a process may generate some output that is used by some other process as their input.

Implementation of Process:

Operating system maintains a table (an array of structure) known as process table with one entry per process to implement the process. The entry contains detail about the process such as, *process state, program counter, stack pointer, memory allocation, the status of its open files, its accounting information, scheduling information and everything else* about the process that must be saved when the process is switched from running to ready or blocked state, so that it can be restarted later as if it had never been stopped.

Process Management	**Memory Management**	**File management**
Register program counter Program status word stack pointer process state priority Scheduling parameter Process ID parent process process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment pinter to data segment pinter to stack segment	Root directory Working directory File descriptors USER ID GROUP ID

Each I/O device class is associated with a location (often near the bottom of the memory) called the ***Interrupt Vector.*** It contains the address of interrupt service procedure. Suppose that user process 3 is running when a disk interrupt occurs. User process 3's program counter, program status word and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the disk interrupt vector. That is all the hardware does. From here on, it is up to the software in particular the interrupt service procedure.

Interrupt handling and scheduling are summarized below.

1. Hardware stack program counter etc.
2. Hardware loads new program counter from interrupt vector
3. Assembly languages procedures save registers.
4. Assembly language procedures sets up new stack.
5. C interrupt service runs typically reads and buffer input.
6. Scheduler decides which process is to run next.
7. C procedures returns to the assembly code.
8. Assembly language procedures starts up new current process.

System calls

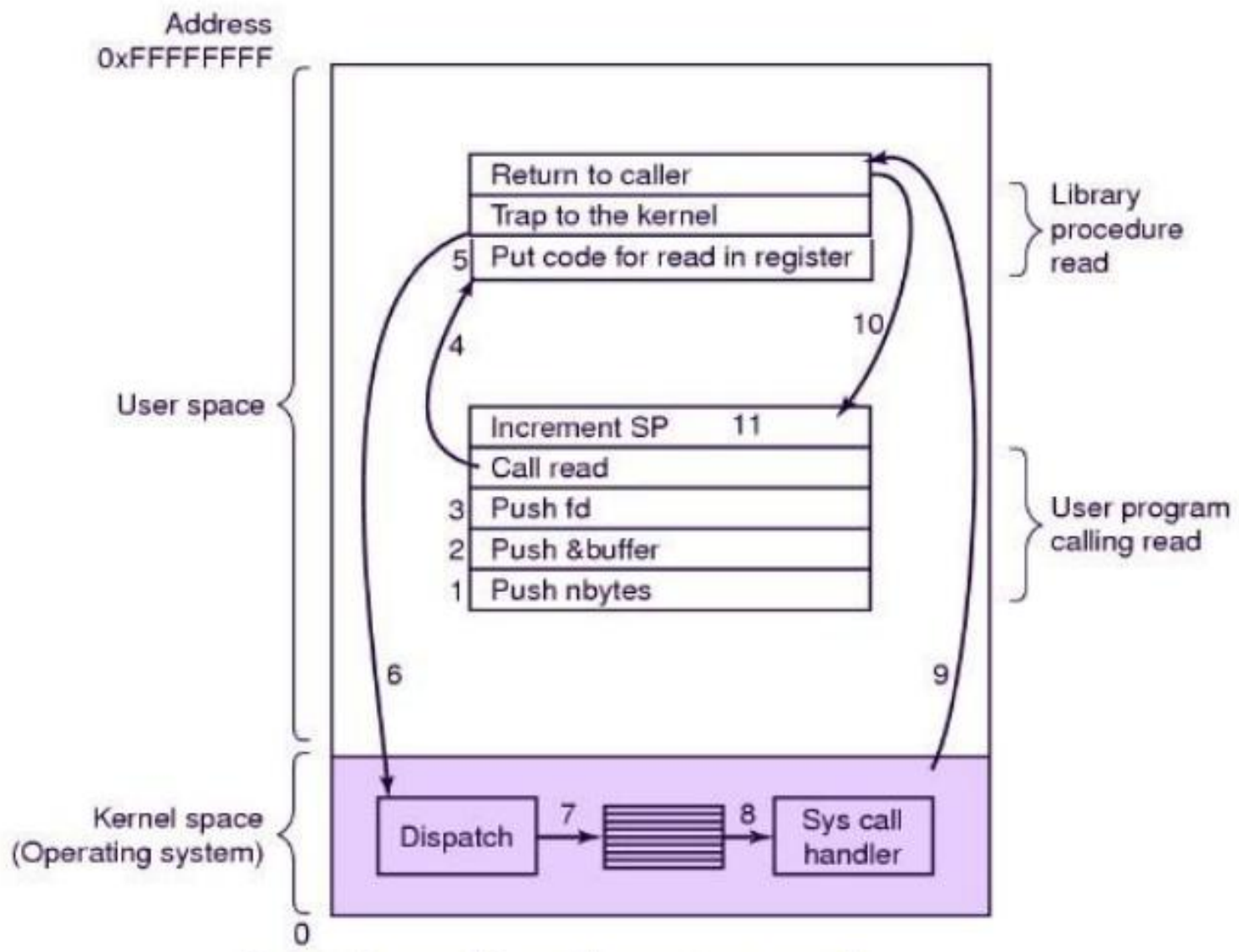
System calls provide the interface between a process and the operating system.

- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, C++)

System calls performed in a series of steps. Most of the system calls are invoked as the following example system call: read

count = read(fd, buffer, nbytes)

- Push parameters into the stack (1-3)
- Calls library procedure (4)
- Pass parameters in registers.(5)
- Switch from user mode to kernel mode and start to execute.(6)
- Examines the system call number and then dispatches to the correct system call handler via a table of pointer.(7)
- Runs system call handlers (8).
- Once the system call handler completed its work, control return to the library procedure.(9)
- This procedure then return to the user program in the usual way. (10) – Increments SP before call to finish the job.



Steps in making the system call read (fd, buffer, nbytes)

Types of System calls

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Types of System calls

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

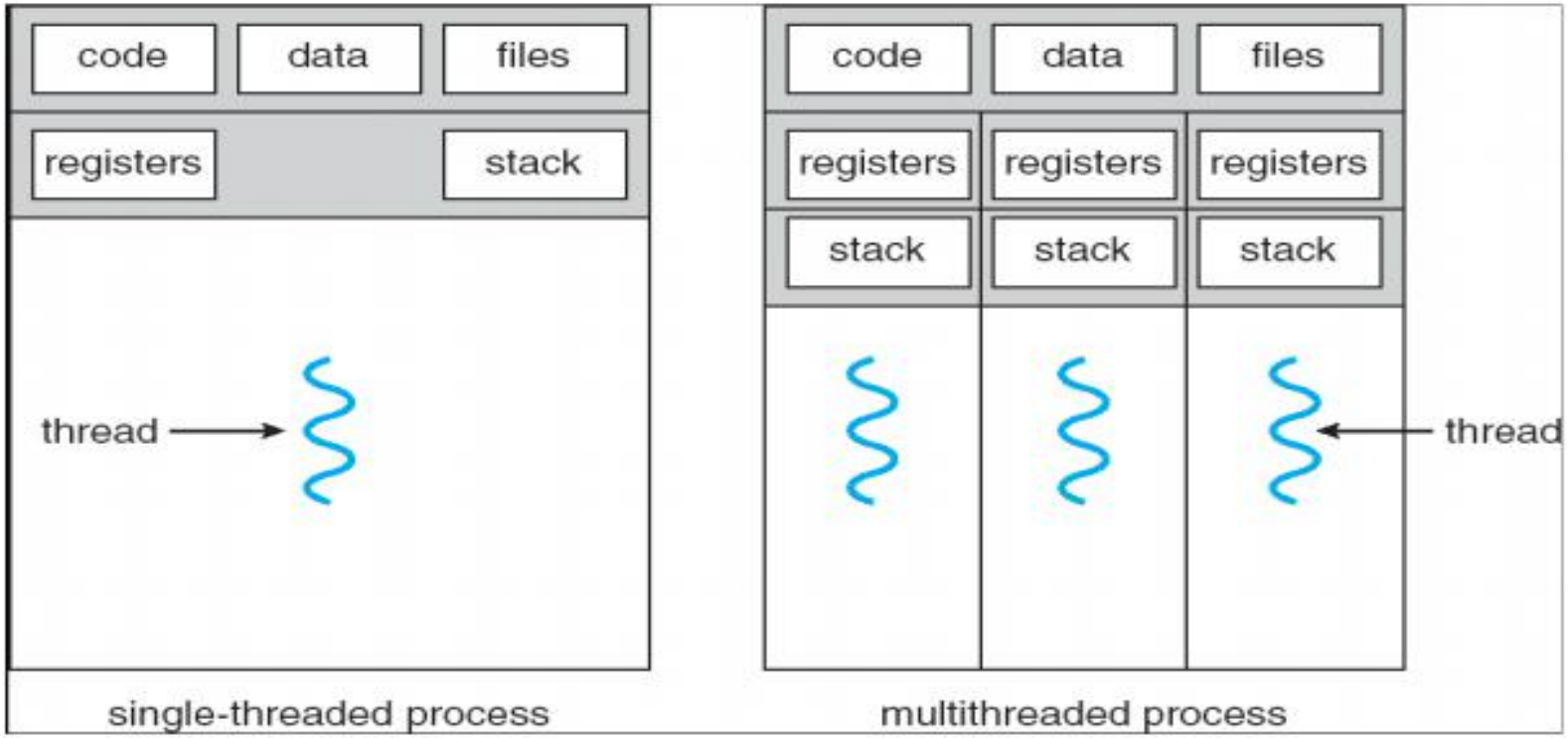
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Introduction of Thread

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack. Threads are a popular way to improve application performance through parallelism. A thread is sometimes called **a light weight process.**
- Threads represent a software approach to improving performance of operating system by reducing the over head thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread

can exist outside a process. Each thread represents a separate flow of control.

- Fig. 4.1shows the single and multithreaded process.



- Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

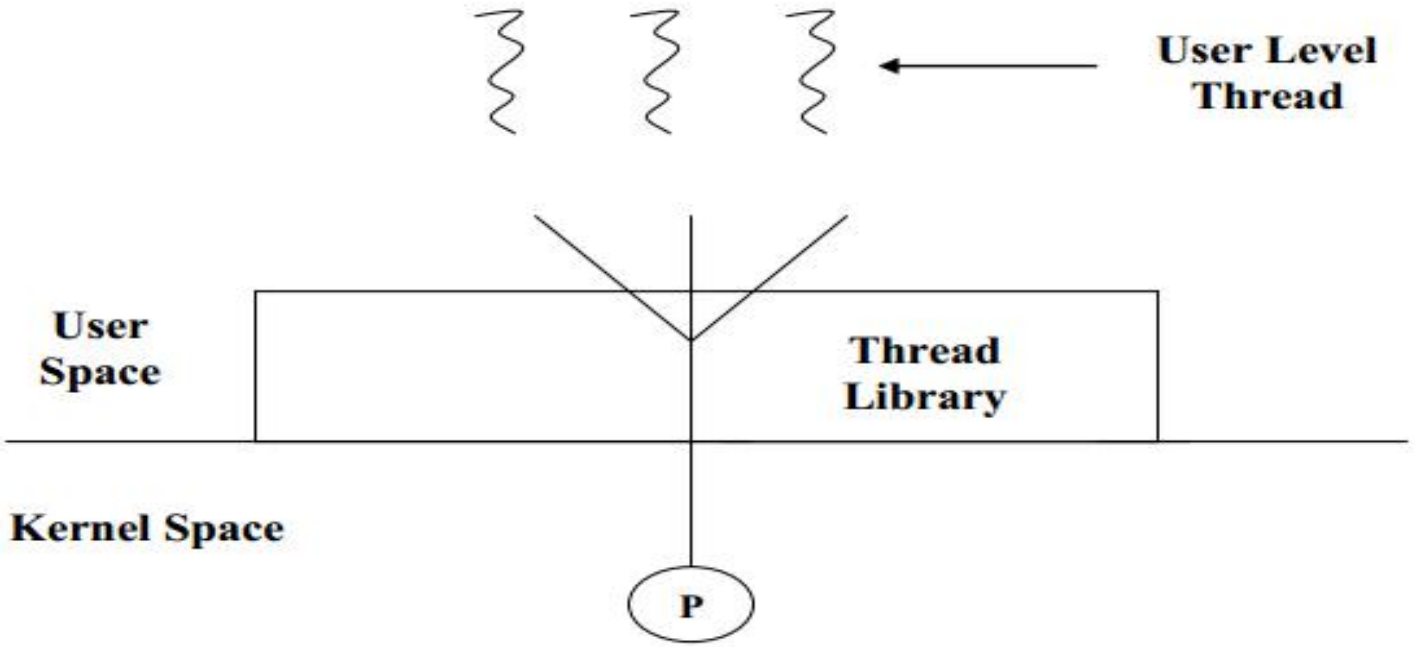
Types of Thread

Threads is implemented in two ways :

1. User Level
2. Kernel Level

.1 User Level Thread

- In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.
- Fig. 4.2 shows the user level thread.



Advantage of user level thread over Kernel level thread :

1. Thread switching does not require Kernel mode privileges.
2. User level thread can run on any operating system.
3. Scheduling can be application specific.
4. User level threads are fast to create and manage.

Disadvantages of user level thread :

1. In a typical operating system, most system calls are blocking.
2. Multithreaded application cannot take advantage of multiprocessing.

2 Kernel Level Threads

- In Kernel level thread, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individuals threads within the process.

- Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages of Kernel level thread:

1. Kernel can simultaneously schedule multiple threads from the same process on multiple process.
2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
3. Kernel routines themselves can multithreaded.

Disadvantages:

1. Kernel threads are generally slower to create and manage than the user threads.

2. Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Sr. No	User Level Threads	Kernel Level Thread
1	User level thread are faster to create and manage.	Kernel level thread are slower to create and manage.
2	Implemented by a thread library at the user level.	Operating system support directly to Kernel threads.
3	User level thread can run on any operating system.	Kernel level threads are specific to the operating system.
4	Support provided at the user level called user level thread.	Support may be provided by kernel is called Kernel level threads.
5	Multithread application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Advantages of Thread

1. Thread minimize context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures –

The benefits of multithreading can be greatly increased in a multiprocessor architecture.

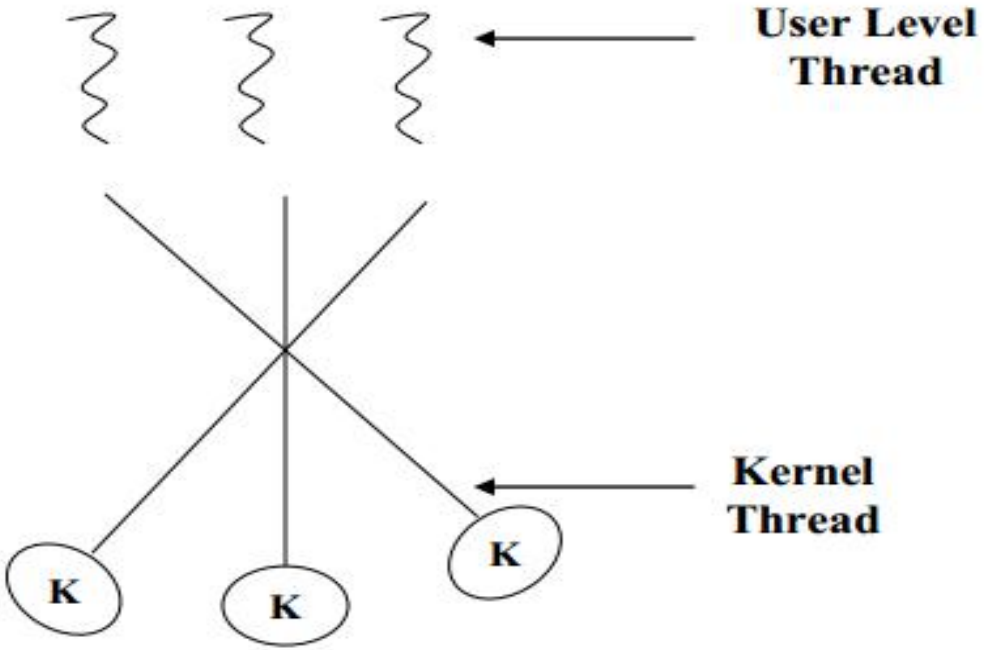
Multithreading Models

- Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types:

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

Many to Many Model

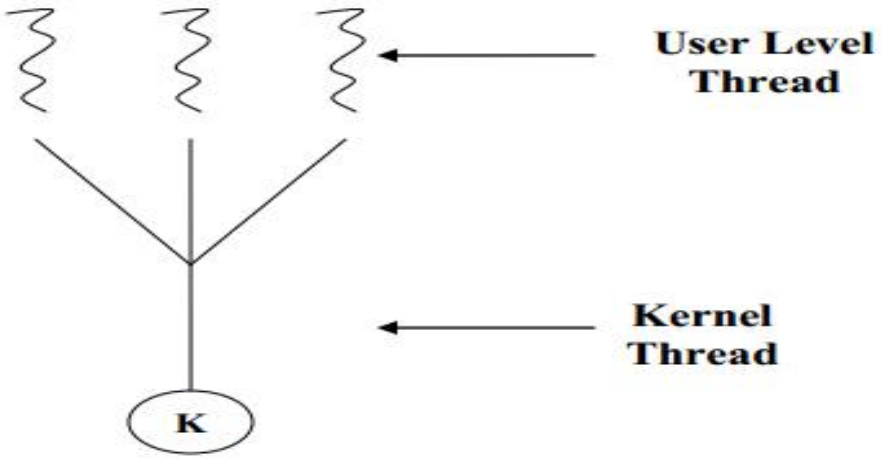
- In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.
- Fig. 4.3 shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

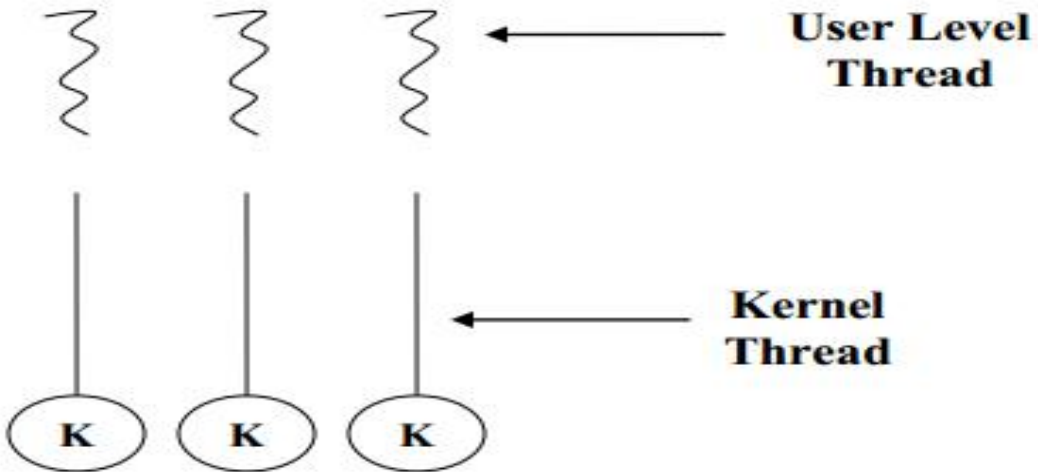
Fig.4.4 shows the many to one model.



- If the user level thread libraries are implemented in the operating system, that system does not support Kernel threads use the many to one relationship modes.

One to One Model

There is one to one relationship of user level thread to the kernel level thread. Fig. 4.5 shows one to one relationship model. This model provides more concurrency than the many to one model.



- It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.

Benefits of Multi-threading:

Responsiveness: Mutlithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.

Resource Sharing: By default, threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity withing the same address space.

Economy: Allocating memory and resources for process creation is costly. Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads. It is more time consuming to create and manage process than threads.

Utilization of multiprocessor architecture: The benefits of multi threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors. Mutlithreading on a multi-CPU increases concurrency.

Sr. No	Process	Thread
1	Process is called heavy weight process.	Thread is called light weight process.
2	Process switching needs interface with operating system.	Thread switching does not need to call a operating system and cause an interrupt to the Kernel.
3	In multiple process implementation each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.
5	Multiple redundant process uses more resources than multiple threaded.	Multiple threaded process uses fewer resources than multiple redundant process.
6	In multiple process each process operates independently of the others.	One thread can read, write or even completely wipe out

Interprocess Communication:

Processes frequently needs to communicate with each other. For example in a shell pipeline, the output of the first process must be passed to the second process and so on down the line. Thus there is a need for communication between the process, preferably in a well-structured way not using the interrupts.

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for **message passing, synchronization, shared memory, and remote procedure calls (RPC).**

co-operating Process: A process is independent if it can't affect or be affected by another process. A process is co-operating if it can affects other or be affected by the other process. Any process that shares data with other process is called co-operating process. There are many reasons for providing an environment for process co-operation.

1.Information sharing: Several users may be interested to access the same piece of information(for instance a shared file). We must allow concurrent access to such information.

2.Computation Speedup: Breakup tasks into sub-tasks.

3.Modularity: construct a system in a modular fashion.

4.convenience:

co-operating process requires IPC. There are two fundamental ways of IPC.

a. Shared Memory

b. Message Passing

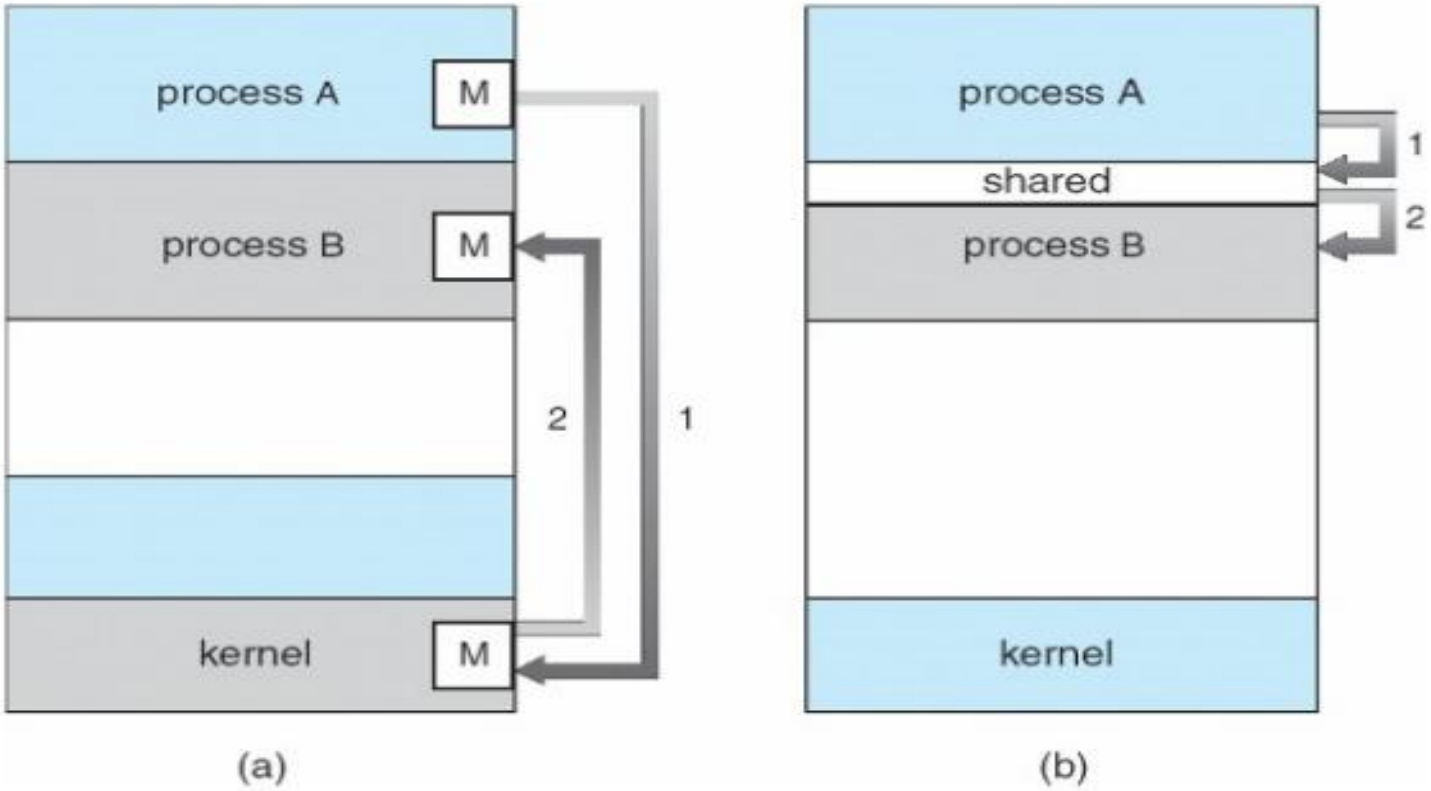


Fig: Communication Model a. Message Passing b. Shared Memory

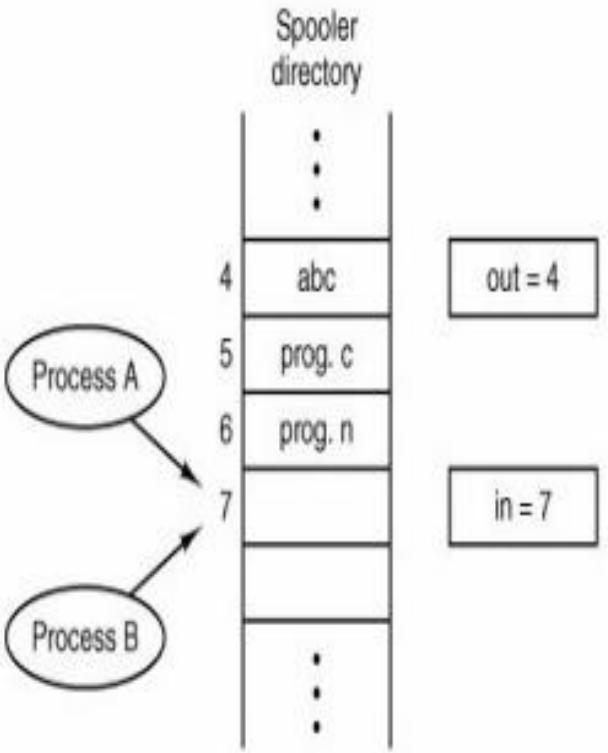
Shared Memory:

- Here a region of memory that is shared by co-operating process is established.
- Process can exchange the information by reading and writing data to the shared region.
- Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.
- System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required,all access are treated as routine memory access.

Message Passing:

- communication takes place by means of messages exchanged between the co-operating process
- Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
- Easier to implement than shared memory.
- Slower than that of Shared memory as message passing system are typically implemented using system call which requires more time consuming task of Kernel intervnetion.

Race Condition:



The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when are called **race conditions.**

To see how interprocess communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and removes their names from the directory.

Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,

out: which points to the next file to be printed

in: which points to the next free slot in the directory.

At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.

Process A reads in and stores the value, 7, in a local variable called **next_free_slot**. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.

Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks at **next_free_slot**, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes **next_free_slot + 1**, which is 8, and sets **in** to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Avoiding Race Conditions:

Critical Section:

To avoid race condition we need **Mutual Exclusion. Mutual Exclusion** is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

The difficulty above in the printer spooler occurs because process B started using one of the shared variables before process A was finished with it.

That part of the program where the shared memory is accessed is called the **critical region or critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

(Rules for avoiding Race Condition) Solution to Critical section problem:

1. **No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)**
2. **No assumptions may be made about speeds or the number of CPUs.**
3. **No process running outside its critical region may block other processes.**
4. **No process should have to wait forever to enter its critical region.**

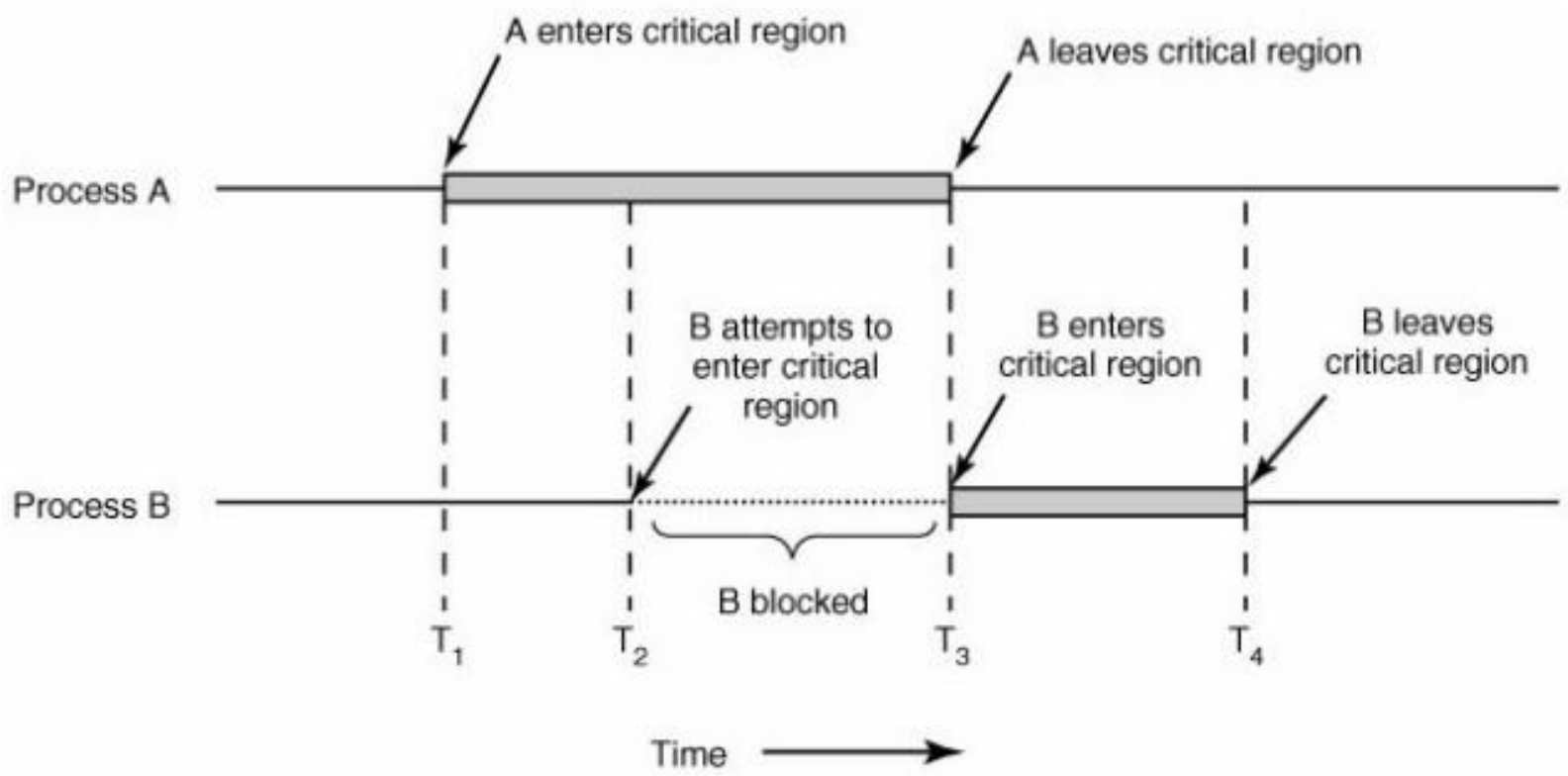


Fig:Mutual Exclusion using Critical Region

Techniques for avoiding Race Condition:

1. Disabling Interrupts
2. Lock Variables
3. Strict Alteration
4. Peterson's Solution
5. TSL instruction
6. Sleep and Wakeup
7. Semaphores
8. Monitors
9. Message Passing

1.Disabling Interrupts:

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur.

The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Disadvantages:

1. It is unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did, and then never turned them on again?
2. Furthermore, if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Advantages:

it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

2.Lock Variables

- a single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Drawbacks:

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3.Strict Alteration:

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

(a)

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

(b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Integer variable turn is initially 0.

It keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.

Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a spin lock. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region. This way no two process can enters critical region simultaneously.

Drawbacks:

Taking turn is is not a good idea when one of the process is much slower than other. This situation requires that two processes strictly alternate in entering their critical region.

Example:

- Process 0 finishes the critical region it sets turn to 1 to allow process 1 to enter critical region.
- Suppose that process 1 finishes its critical region quickly so both process are in their non critical region with turn sets to 0.
- Process 0 executes its whole loop quickly, exiting its critical region & setting turn to 1. At this point turn is 1 and both processes are executing in their noncritical regions.
- Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now since turn is 1 and process 1 is busy with its noncritical region.

This situation violates the condition 3 set above: No process running outside the critical region may block other process. In fact the solution requires that the two processes strictly alternate in entering their critical region.

4.Peterson's Solution:

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */
int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE)*/
void enter_region(int process)        /* process is 0 or 1 */
{
    int other;                        /* number of the other process */
    other = 1 - process;              /* the opposite of process */
    interested[process] = TRUE;       /* show that you are interested */
    turn = process;                   /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)        /* process: who is leaving */
{
    interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Initially neither process is in critical region. Now process 0 calls enter_region. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, enter_region returns immediately. If process 1 now calls enter_region, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls leave_region to exit the critical region.

Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

5. The TSL Instruction

TSL RX,LOCK

(Test and Set Lock) that works as follows: 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 indivisibleno 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 non 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
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in LOCK. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

Problems with mutual Exclusion:

The above techniques achieves the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

1. This approach waste CPU time
2. There can be an unexpected problem called priority inversion problem.

Priority Inversion Problem:

Consider a computer with two processes, H, with high priority and L, with low priority, which share a critical region. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the priority inversion problem.

Let us now look at some IPC primitives that blocks instead of wasting CPU time when they are not allowed to enter their critical regions. Using blocking constructs greatly improves the CPU utilization

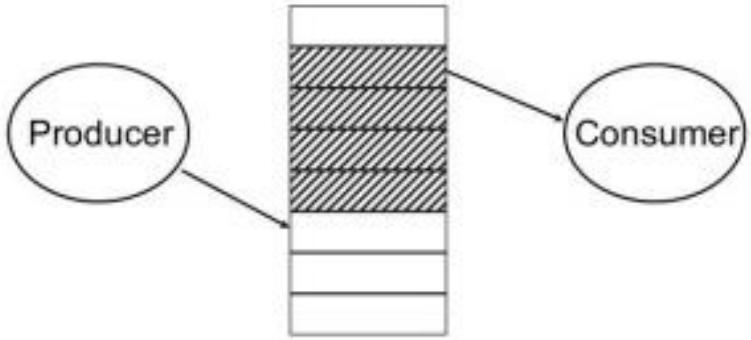
Sleep and Wakeup:

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

Examples to use Sleep and Wakeup primitives:

Producer-consumer problem (Bounded Buffer):

Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.



Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */
void producer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        item = produce_item();                  /* generate next item */
        if (count == N) sleep();                /* if buffer is full, go to sleep */
        insert_item(item);                      /* put item in buffer */
        count = count + 1;                      /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);       /* was buffer empty? */
    }
}
void consumer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        if (count == 0) sleep();                /* if buffer is empty, got to sleep */
        item = remove_item();                   /* take item out of buffer */
        count = count - 1;                      /* decrement count of items in
 buffer */
        if (count ==N - 1) wakeup(producer);    /* was buffer full? */
        consume_item(item);                     /* print item */
    }
}
```

Fig:The producer-consumer problem with a fatal race condition.

N → Size of Buffer

Count--> a variable to keep track of the no. of items in the buffer.

Producers code:
The producers code is first test to see if count is N. If it is, the producer will go to sleep ; if it is not the producer will add an item and increment count.

Consumer code:
It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
Each of the process also tests to see if the other should be awakened and if so wakes it up.
This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)
3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

Semaphore:

In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment . Synchronization tool that does not require busy waiting . **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations. P and V. If S is the semaphore variable, then,**

P operation: Wait for semaphore to become positive and then decrement **P(S):** **while(S<=0)** **do no-op;** **S=S-1**	**V Operation:** Increment semaphore by 1 **V(S):** S=S+1;

Atomic operations: When one process modifies the semaphore value, no other process can simultaneously modify that same semaphores value. In addition, in case of the P(S) operation the testing of the integer value of S (S<=0) and its possible modification (S=S-1), must also be executed without interruption.

Semaphore operations:
P or Down, or Wait: P stands for *proberen* for "to test
V or Up or Signal: Dutch words. **V** stands for *verhogen* ("increase")

wait(sem) -- decrement the semaphore value. if negative, suspend the process and place in queue. (Also referred to as *P(), down* in literature.)

signal(sem) -- increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V(), up* in literature.)

Counting semaphore – integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement Also known as mutex locks

Provides mutual exclusion

```
Semaphore S;   // initialized to 1
wait (S);
        Critical Section
signal (S);
```

Semaphore implementation without busy waiting:
Implementation of wait:

```
wait (S){
        value - -;
        if (value < 0) {
                add this process to waiting queue
                block(); }
        }
```

Implementation of signal:

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P); }
        }
```

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */
void producer(void)
{
    int item;
    while (TRUE){                   /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}
```

```
void consumer(void)
{
    int item;
    while (TRUE){                   /* infinite loop */
        down(&full);                /* decrement full count */
        down(&mutex);               /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                 /* leave critical region */
        up(&empty);                 /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

Fig: The producer-consumer problem using semaphores.

This solution uses three semaphore.

1. Full: For counting the number of slots that are full, initially 0
2. Empty: For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.
3. Mutex: To make sure that the producer and consumer do not access the buffer at the same time, initially 1.

Here in this example seamphores are used in two different ways.

1.**For mutual Exclusion**: The mutex semaphore is for mutual exclusion. It is designed to guarantee that only one process at at time will be reading or writing the buffer and the associated variable.

2.**For synchronization:** The full and empty semaphores are needed to guarantee that certain certain event sequences do or do not occur. In this case, they ensure that producer stops running when the buffer is full and the consumer stops running when it is empty.

The above definition of the semaphore suffer the problem of busy wait. To overcome the need for busy waiting, we can modify the definition of the P and V operation of the semaphore. When a Process executes the P operation and finds that the semaphores value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places into a waiting queue associated with the semaphore, and the state of the process is switched to the

Advantages of semaphores:

- Processes do not busy wait while waiting for resources. While waiting, they are in a "suspended" state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantages of semaphores:

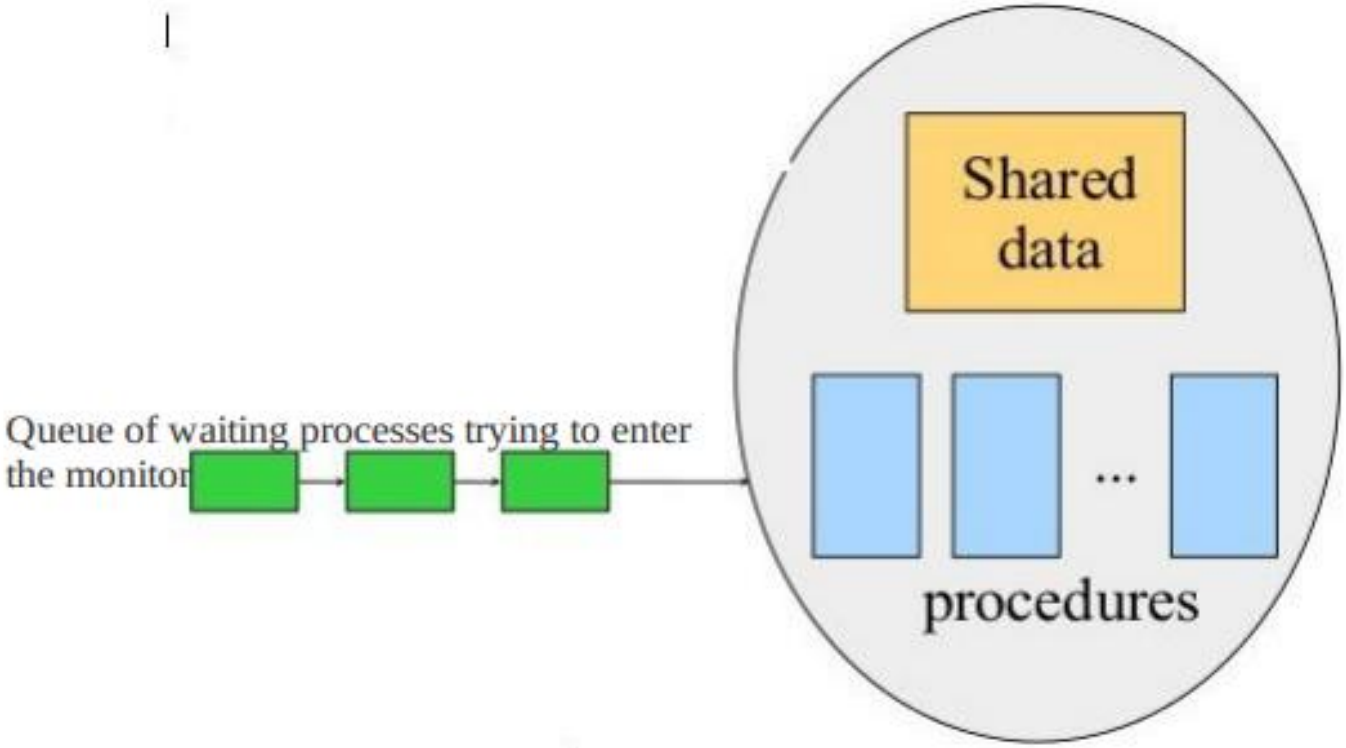
•can only be invoked by processes--not interrupt service routines because interrupt routines cannot block

•user controls synchronization--could mess up.

Monitors:

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.



With semaphores IPC seems easy, but Suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- This rule, which is common in modern object-oriented languages such as Java, was relatively unusual for its time,
- Figure below illustrates a monitor written in an imaginary language, Pidgin Pascal.

```
monitor example
 integer i;
 condition c;
 procedure producer (x);
 .
 .
 .
 end;
 procedure consumer (x);
 .
 .
 .
 end;
end monitor;
```

Fig: A monitor

Message Passing:

Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Message passing is a method of communication where messages are sent from a sender to one or more recipients. Forms of messages include **(remote) method invocation, signals, and data packets**. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

Synchronous message passing systems requires the sender and receiver to wait for each other to transfer the message

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready.

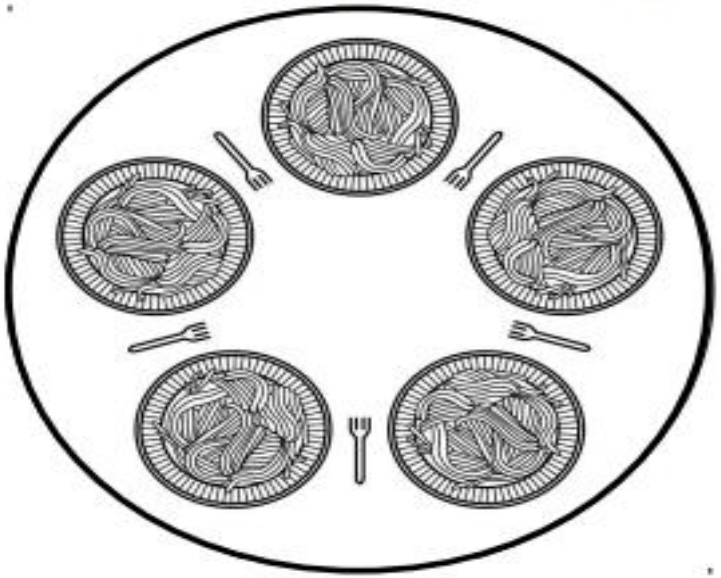
Classical IPC Problems

1. Dining Philosophers Problem
2. The Readers and Writers Problem
3. The Sleeping Barber Problem

1. Dining philosophers problems:

There are N philosphers sitting around a circular table eating spaghetti and discussing philosphy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.

One idea is to instruct each philosopher to behave as follows:

- think until the left fork is available; when it is, pick it up
- think until the right fork is available; when it is, pick it up
- eat
- put the left fork down
- put the right fork down
- repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a

deadlock.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

Solution:

```
#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 N1 */
{
    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 N1 */
{
    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 N1 */
{
    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 N1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Readers Writer problems:

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). 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 process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

Solution to Readers Writer problems

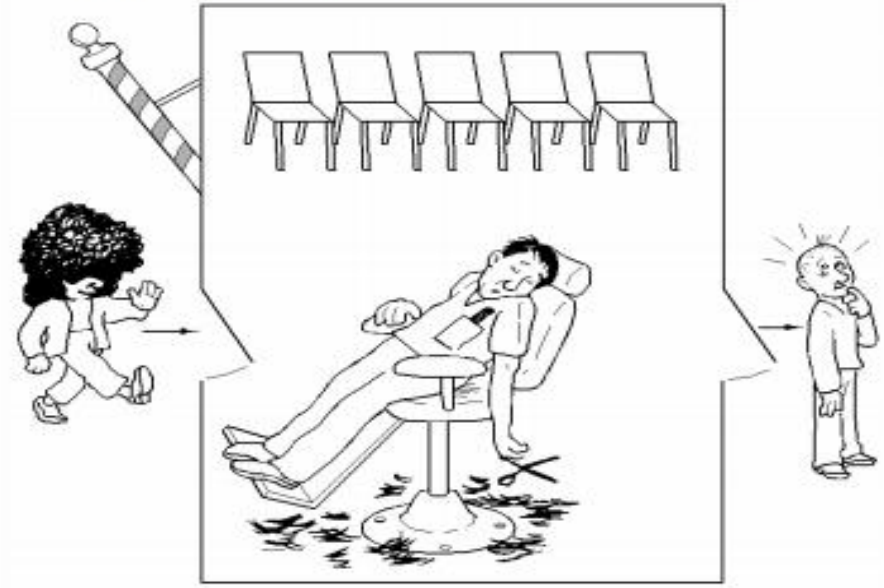
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */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

In this solution, the first reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Sleeping Barber Problem

customers arrive to a barber, if there are no customers the barber sleeps in his chair. If the barber is asleep then the customers must wake him up.



The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves. Based on a naive analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a

number of problems that can occur that are illustrative of general scheduling problems.

The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber. In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Solution:

Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.

A multiple sleeping barbers problem has the additional complexity of coordinating several barbers among the waiting customers

Process Scheduling

Reading: Chapter 5 of textbook

Which process is given control of the CPU and how long?

By switching the processor among the processes, the OS can make the computer more productive – basic objective for multiprogrammed operating systems

Process Scheduling Introduction

- Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Types of Scheduling

- Preemptive
- Non-preemptive
- Batch
- Interactive
- Real time

Types of Scheduling Contd..

Non-preemptive

- Once a process has been given the CPU, it runs until blocks for I/O or termination.
- Treatment of all processes is fair.
- Response times are more predictable.
- Useful in real-time system.
- Shorts jobs are made to wait by longer jobs - no priority

Types of Scheduling Contd..

Preemptive

- Processes are allowed to run for a maximum of some fixed time.
- Useful in systems in which high-priority processes requires rapid attention.
- In timesharing systems, preemptive scheduling is important in guaranteeing acceptable response times.
- High overhead.

Types of Scheduling Contd..
Batch System

Batch systems are still in widespread use in the business world for doing payroll, inventory, accounts receivable, accounts payable, interest calculation (at banks), claims processing (at insurance companies), and other periodic tasks. In batch systems, there are no users impatiently waiting at their terminals for a quick response to a short request. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance. The batch algorithms are actually fairly general and often applicable to other situations as well, which makes them worth studying, even for people not involved in corporate mainframe computing.

Types of Scheduling Contd.. Interactive

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug. Preemption is needed to prevent this behavior. Servers also fall into this category, since they normally serve multiple (remote) users, all of whom are in a big hurry.

Types of Scheduling Contd.. Real Time

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious.

Some goals of Scheduling algorithm

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Criteria

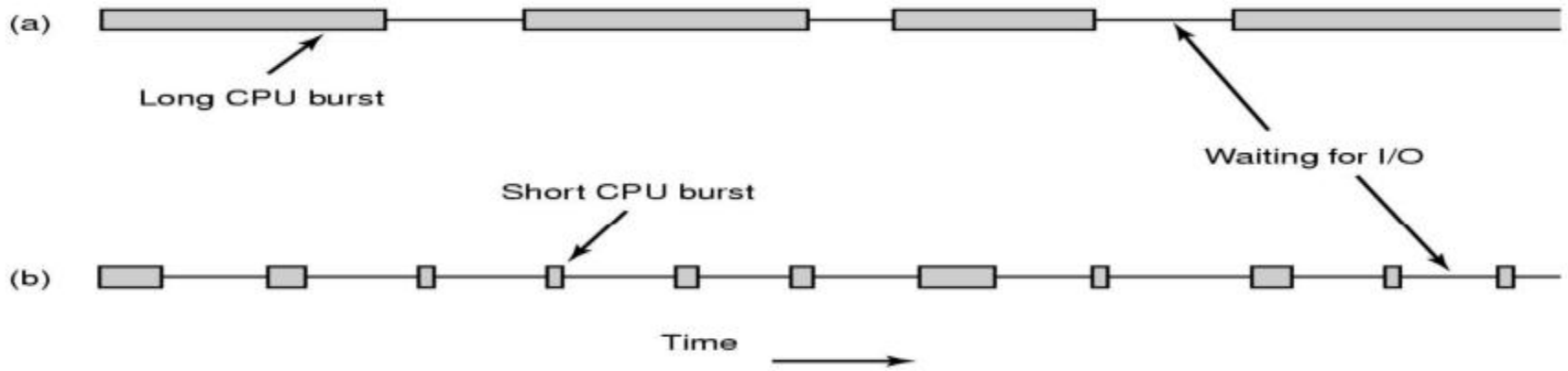
The scheduler is to identify the process whose selection will results the best possible system performance.

Criteria for comparing scheduling algorithms:

CPU Utilization *Balance Utilization*
Throughput *Turnaround Time Waiting Time*
Response Time
Predictability *Fairness Priorities*

The scheduling policy determine the important of each criteria.
The scheduling algorithms should designed to optimizes maximum possible criteria.

CPU-I/O Burst



Process execution consists of a cycle of CPU execution and I/O wait.

Process execution begins with a CPU burst that is followed by I/O burst, then another CPU burst, then another I/O burst.....

CPU-bound: Processes that use CPU until the quantum expire.

I/O-bound: Processes that use CPU briefly and generate I/O request.

CPU-bound processes have a long CPU-burst while I/O-bound processes have short CPU burst.

Key idea: when I/O bound process wants to run, it should get a chance quickly.

When to Schedule

1. When a new process is created.
2. When a process terminates.
3. When a process blocks on I/O, on semaphore, waiting for child termination etc.
4. When an I/O interrupt occurs.
5. When quantum expires.

Once a process has been given the CPU, it runs until blocks for I/O or termination is known as *nonpreemptive*, otherwise it is *preemptive*.

Dispatcher vs. Scheduler

Dispatcher

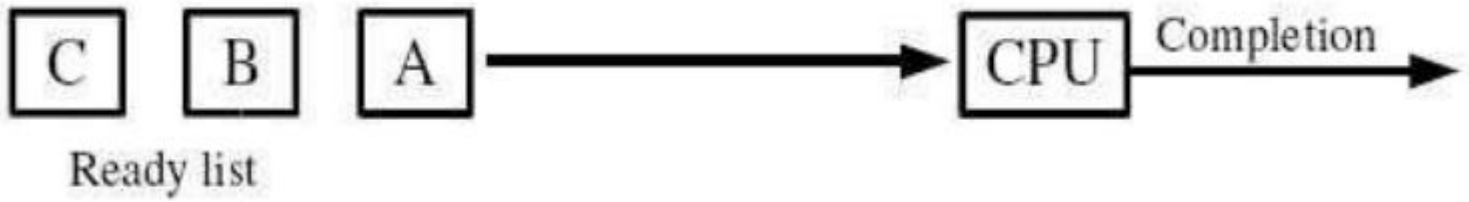
- Low level mechanism.
- Responsibility: Context switch
  - Save execution state of old process in PCB.
  - Load execution state of new process from PCB to registers.
  - Change the scheduling state of the process (running, ready, blocked)
  - Switch from kernel to user mode.

Scheduler

- Higher-level policy.
- Responsibility: Which process to run next.

Scheduling Algorithms

First-Come First-Serve (FCFS)



Processes are scheduled in the order they are received.

Once the process has the CPU, it runs to completion -Nonpreemptive.

Easily implemented, by managing a simple queue or by storing time the process was received.

Fair to all processes. Problems:

- No guarantee of good response time.
- Large average waiting time.
- Not applicable for interactive system.

Scheduling Algorithms

Shortest Job First (SJF)

The processing times are known in advanced.

SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.

The decision policies are based on the CPU burst time. Advantages:

Reduces the average waiting time over FCFS.

Favors shorts jobs at the cost of long jobs.

Problems:

Estimation of run time to completion. Accuracy?

Not applicable in timesharing system.

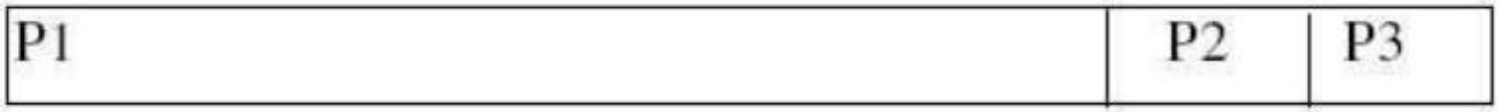
Scheduling Algorithms

SJF -Performance

Scenario: Consider the following set of processes that arrive at time 0, with length of CPU-bust time in milliseconds.

Processes	Burst time
P1	24
P2	3
P3	3

if the processes arrive in the order P1, P2, P3 and are served in FCFS order,



The average waiting time is (0 + 24 + 27)/3 = 17.

if the processes are served in SJF

The average waiting time is (6 + 0 + 3)/3 = 3.

Scheduling Algorithms

Shortest-Remaining-Time-First (SRTF)

Preemptive version of SJF.

Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled. If the new process's time is less, the currently scheduled process is preempted.

Merits:

Low average waiting time than SJF.

Useful in timesharing. Demerits:

Very high overhead than SJF.

Requires additional computation.

Favors short jobs, longs jobs can be victims of starvation.

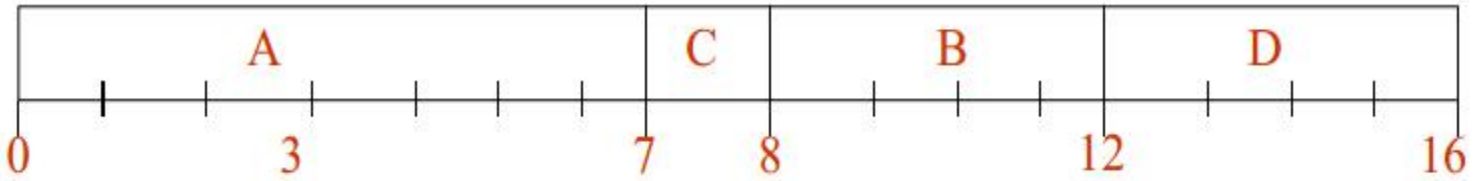
Scheduling Algorithms

SRTF-Performance

Scenario: Consider the following four processes with the length of CPU-burst time given in milliseconds:

Processes	Arrival Time	Burst Time
A	0.0	7
B	2.0	4
C	4.0	1
D	5.0	4

SJF:

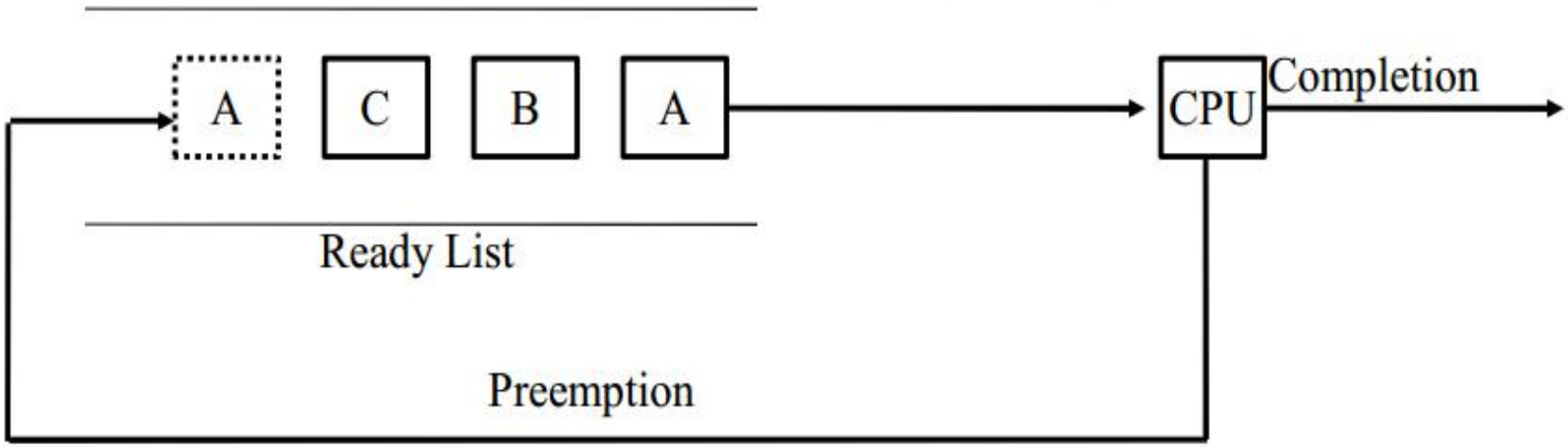


Average waiting time = (0 + 6 + 3 + 7)/4 = 4

SRTF:

Average waiting time = (9 + 1 + 0 +2)/4 = 3

Scheduling Algorithms
Round-Robin (RR)



Preemptive FCFS.

Each process is assigned a time interval (quantum), after the specified quantum, the running process is preempted and a new process is allowed to run.

Preempted process is placed at the back of the ready list.

Advantages:

Fair allocation of CPU across the process.

Used in timesharing system.

Low average waiting time when process lengths very widely.

Scheduling Algorithms
RR-Performance

- Poor average waiting time when process lengths are identical.

  Imagine 10 processes each requiring 10 msec burst time and 1msec quantum is assigned.

  RR: All complete after about 100 times.

  FCFS is better! (About 20% time wastages in context-switching).
- Performance depends on quantum size.

Quantum size:

If the quantum is very large, each process is given as much time as needs for completion; RR degenerate to FCFS policy.

If quantum is very small, system busy at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Optimal quantum size?

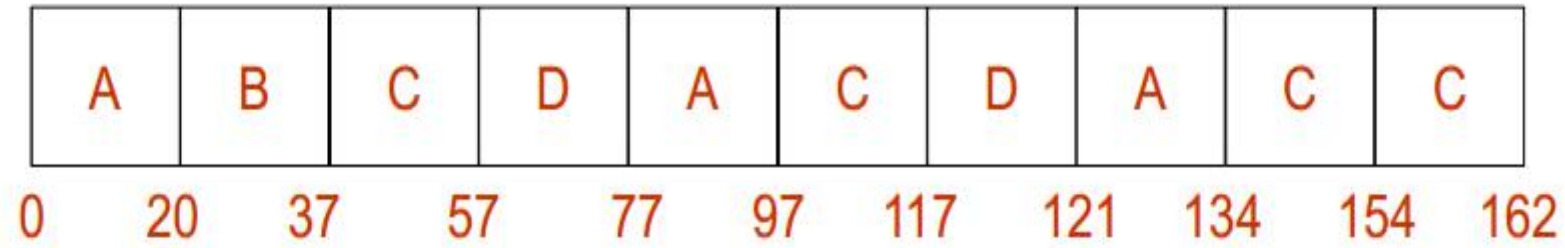
Key idea: 80% of the CPU bursts should be shorter than the quantum.

20-50 msec reasonable for many general processes.

Scheduling Algorithms
Example of RR with Quantum = 20

Process	Burst Time
A	53
B	17
C	68
D	24

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*.

Scheduling Algorithms
Priority

Each process is assigned a priority value, and runnable process with the highest priority is allowed to run.

FCFS or RR can be used in case of tie.

To prevent high-priority process from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick.

Assigning Priority

Static:

Some processes have higher priority than others.

Problem: Starvation.

Dynamic:

Priority chosen by system.

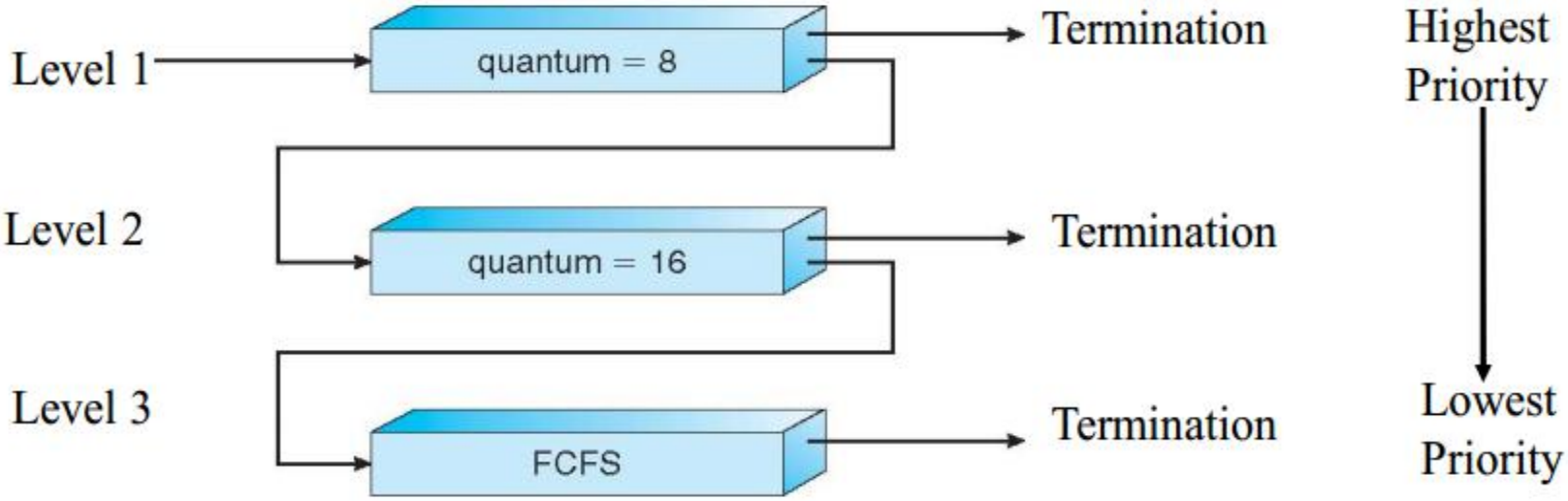
Decrease priority of CPU-bound processes.

Increase priority of I/O-bound processes.

Many different policies possible…….

E. g.: priority = (time waiting + processing time)/processing time.

Scheduling Algorithms
Multilevel-Feedback-Queues (MFQ)



MFQ implements multilevel queues having different priority to each level (here lower level higher priority), and allows a process to move between the queues. If the process use too much CPU time, it will be moved to a lower-priority queue.

Each lower-priority queue larger the quantum size.

Each queue may have its own scheduling algorithm

This leaves the I/O-bound and interactive processes in the high priority queue.

Scheduling Algorithms
MFQ-Example

Consider a MFQ scheduler with three queues numbered 1 to 3, with quantum size 8, 16 and 32 msec respectively.

The scheduler execute all process in queue 1, only when queue 1 is empty it execute process in queue 2 and process in queue 3 will execute only if queue 1 and queue 2 are empty.

A process first enters in queue 1 and execute for 8 msec. If it does not finish, it moves to the tail of queue 2.

If queue 1 is empty the processes of queue 2 start to execute in FCFS manner with 16 msec quantum. If it still does not complete, it is preempted and move to the queue 3.

If the process blocks before using its entire quantum, it is moved to the next higher level queue.

Priority Scheduling

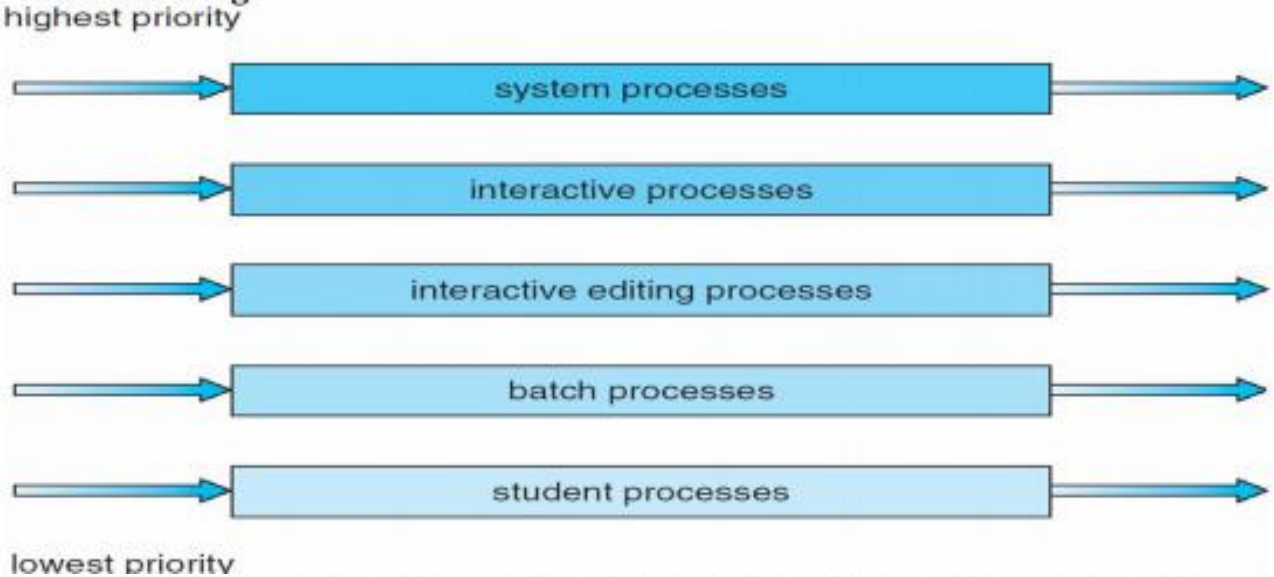
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ° highest priority)
- Preemptive
- nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ° **Starvation** – low priority processes may never execute
- Solution ° **Aging** – as time progresses increase the priority of the process

Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
- foreground – RR
- background – FCFS
- Scheduling must be done between the queues
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR

20% to background in FCFS

Multilevel Queue Scheduling



Real-Time Scheduling

- Hard Real-time Computing -
  - required to complete a critical task within a guaranteed amount of time.
- Soft Real-time Computing -
  - requires that critical processes receive priority over less important ones.
- Types of real-time Schedulers
  - Periodic Schedulers - Fixed Arrival Rate
  - Demand-Driven Schedulers - Variable Arrival Rate
  - Deadline Schedulers - Priority determined by deadline
  - ...

Issues in Real-time Scheduling

- Dispatch Latency
  - Problem - Need to keep dispatch latency small, OS may enforce process to wait for system call or I/O to complete.
  - Solution - Make system calls preemptible, determine safe criteria such that kernel can be interrupted.
- Priority Inversion and Inheritance
  - Problem: Priority Inversion
    - Higher Priority Process P0 needs kernel resource currently being used by another lower priority process P1.
    - P0 must wait for P1, but P1 isn't getting scheduled in CPU!
  - Solution: Priority Inheritance
    - Low priority process now inherits high priority until it has completed use of the resource in question.

Lottery Scheduling

- **Lottery scheduling** is a probabilistic scheduling algorithm for processes in an operating system.
- Processes are each assigned some number of lottery tickets, and the scheduler draws a random ticket to select the next process.
- The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as Shortest Job next and Fair Share Scheduling.
- Lottery scheduling solves the problem of Starvation.
- Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

Shortest Job Next Scheduling

- **Shortest job next** (**SJN**), also known as **shortest job first** (**SJF**) or **shortest process next** (**SPN**), is a scheduling policy that selects for execution the waiting process with the smallest execution time.
- SJN is a non-preemptive algorithm. Shortest Remaining Time is a preemptive variant of SJN.
- Shortest job next is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete.
- However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added. Highest Response Ration Next is similar but provides a solution to this problem using a technique called aging.

Shortest Job Next Scheduling Contd..

- Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution. While it is impossible to predict execution time perfectly, several methods can be used to estimate it, such as a weighted average of previous execution times.[3]
- Shortest job next can be effectively used with interactive processes which generally follow a pattern of alternating between waiting for a command and executing it. If the execution burst of a process is regarded as a separate "job", past behavior can indicate which process to run next, based on an estimate of its running time.

Fair Share Scheduling

- Fair-share scheduling is a scheduling algorithm for computer Operating systems in which the CPU usage is equally distributed among system users or groups, as opposed to equal distribution among processes.
- One common method of logically implementing the fair-share scheduling strategy is to recursively apply the round robin scheduling strategy at each level of abstraction (processes, users, groups, etc.) The time quantum required by round-robin is arbitrary, as any equal division of time will produce the same results.

Fair Share Scheduling

- For example, if four users (A,B,C,D) are concurrently executing one process each, the scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole (100% / 4 = 25%). If user B starts a second process, each user will still receive 25% of the total cycles, but each of user B's processes will now be attributed 12.5% of the total CPU cycles each, totaling user B's fair share of 25%. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% / 5 = 20%).
- Another layer of abstraction allows us to partition users into groups, and apply the fair share algorithm to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles will be distributed as follows:
- 100% / 3 groups = 33.3% per group Group 1: (33.3% / 3 users) = 11.1% per user Group 2: (33.3% / 2 users) = 16.7% per user Group 3: (33.3% / 4 users) = 8.3% per user

Highest Response Ratio Next (HRRN) Scheduling

- Highest Response Ratio Next (HRNN) is one of the most optimal scheduling algorithms.
- This is a non-preemptive algorithm in which, the scheduling is done on the basis of an extra parameter called Response Ratio.
- A Response Ratio is calculated for each of the available jobs and the Job with the highest response ratio is given priority over the others.
- Response Ratio is calculated by the given formula.

  Response Ratio = (W+S)/S Where
  - W → Waiting Time
  - S → Service Time or Burst Time

Highest Response Ratio Next (HRRN) Scheduling Contd..

- This algorithm not only favors shorter job but it also concern the waiting time of the longer jobs.
- Its mode is non preemptive hence context switching is minimal in this algorithm.

- Example: see the file provided in google classroom.

Guaranteed Scheduling:

- Make real promises to the users about performance.
- If there are n users logged in while you are working, you will receive about 1 /n of the CPU power.
- Similarly, on a single-user system with n processes running, all things being equal, each one should get 1 /n of the CPU cycles.
- To make good on this promise, the system must keep track of how much CPU each process has had since its creation.
- CPU Time entitled= (Time Since Creation)/n
- Then compute the ratio of Actual CPU time consumed to the CPU time entitled.
- A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to.
- The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

Two-Level Scheduling:

Performs process scheduling that involves swapped out processes. Two-level scheduling is needed when memory is too small to hold all the ready processes .

Consider this problem: A system contains 50 running processes all with equal priority. However, the system's memory can only hold 10 processes in memory simultaneously. Therefore, there will always be 40 processes swapped out written on virtual memory on the hard disk

It uses two different schedulers, one lower-level scheduler which can only select among those processes in memory to run. That scheduler could be a Round-robin scheduler. The other scheduler is the higher-level scheduler whose only concern is to swap in and swap out processes from memory. It does its scheduling much less often than the lower-level scheduler since swapping takes so much time. the higher-level scheduler selects among those processes in memory that have run for a long time and swaps them out

End