Process Management

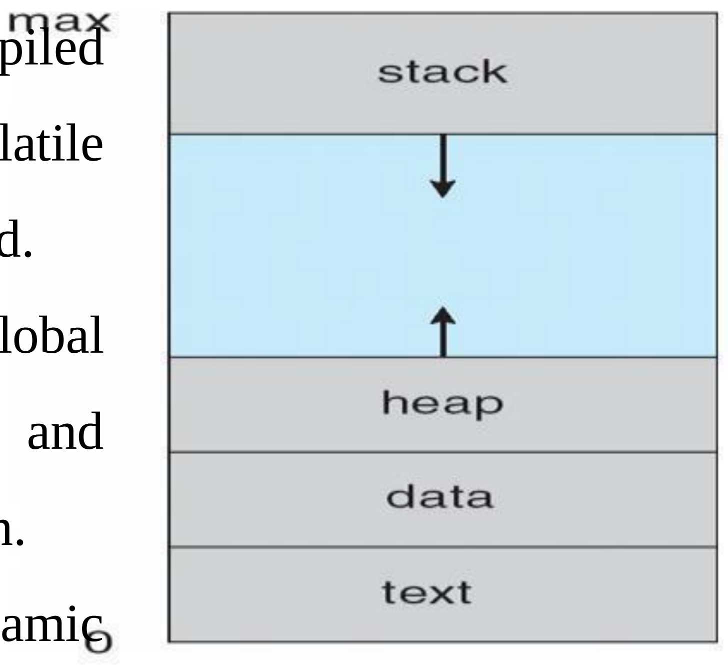
Process Concept

- An operating system executes a variety of programs
  - batch systems - jobs
  - time-shared systems - user programs or tasks
  - Job, task and program used interchangeably
- Process - a program in execution
  - process execution proceeds in a sequential fashion
- A process contains
  - program counter, stack and data section

Process vs Program

- program is a group of instructions to carry out a specified task.
- A program is a **passive entity,** for example, a file accommodating a group of instructions to be executed (executable file).
- Program is stored on disk in some file and does not require any other resources.
- When a program is executed, it is known as process.
- It is considered as an **active entity** and realizes the actions specified in a program.
- Process holds resources such as CPU, memory address, disk, I/O etc.
- Multiple processes can be related to the same program. It handles the operating system activities through **PCB (Process control Block)** which includes program counter, stack, state etc
- A web browser launches multiple processes, e.g., one per tab

- **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.



Process in Memory

The Process Model

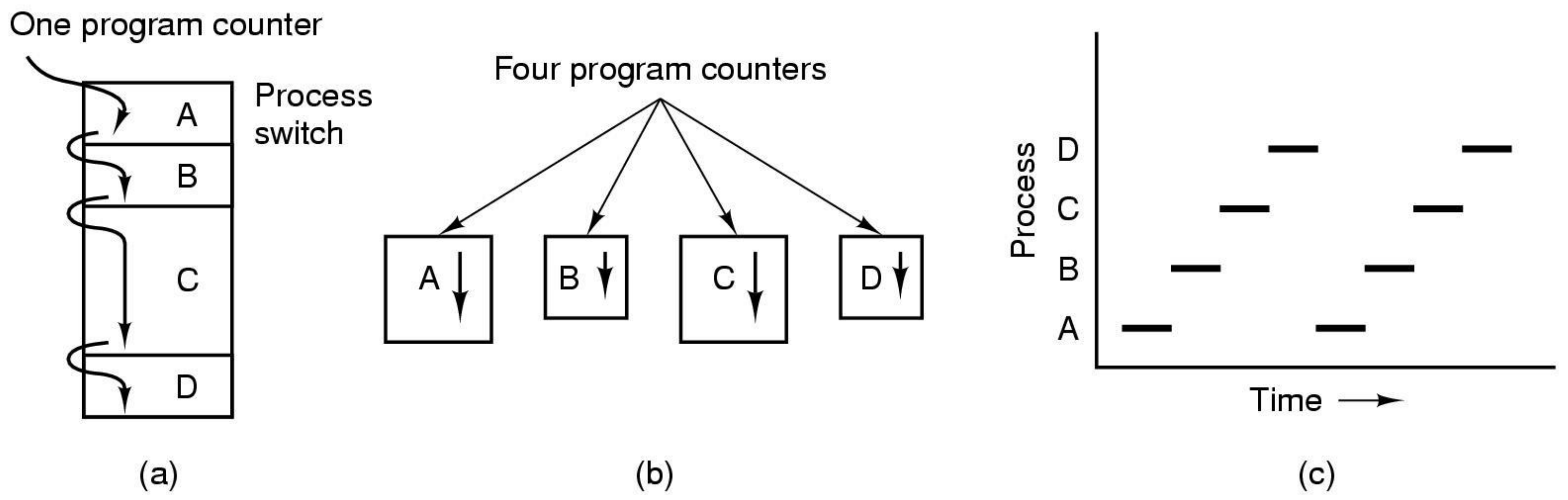


Figure 3-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

- Conceptually, every process has its own virtual CPU but in reality CPU switches back and forth from process to process.
- This Rapid switching back and forth is called **"Multi-Programming.**“

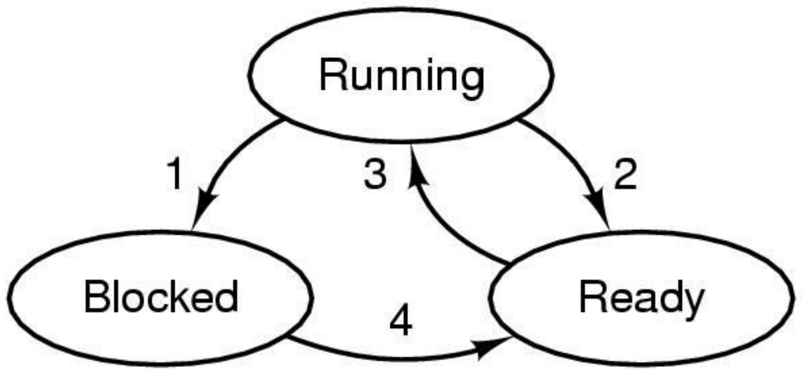
Multi-Programming

- In this there is 4 program in memory. We have single shared processor by all programs.
- There is only one program counter all programs in memory. In this program counter is initialized to execute the part of program A then with the help of process switch it transfer control to program B and execute some part of this.
- After that program counter for program C is active and execute some part of it and so onto program D. Then all the steps continuous may be randomly with the help of process switches until all program is not completed.

One Program At One Time

- At any given instant only one process run and other process after some interval of time.
- In other point of view all processes progress but only process actually runs at given instant of time.

Process States



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

Three-state process model is constituted of **READY**, **RUNNING** & **WAITING**.

Running: the process is currently executed by the CPU

Blocked: the process is waiting for a resource to come available

Ready: the process is ready to be selected.

Five state process models

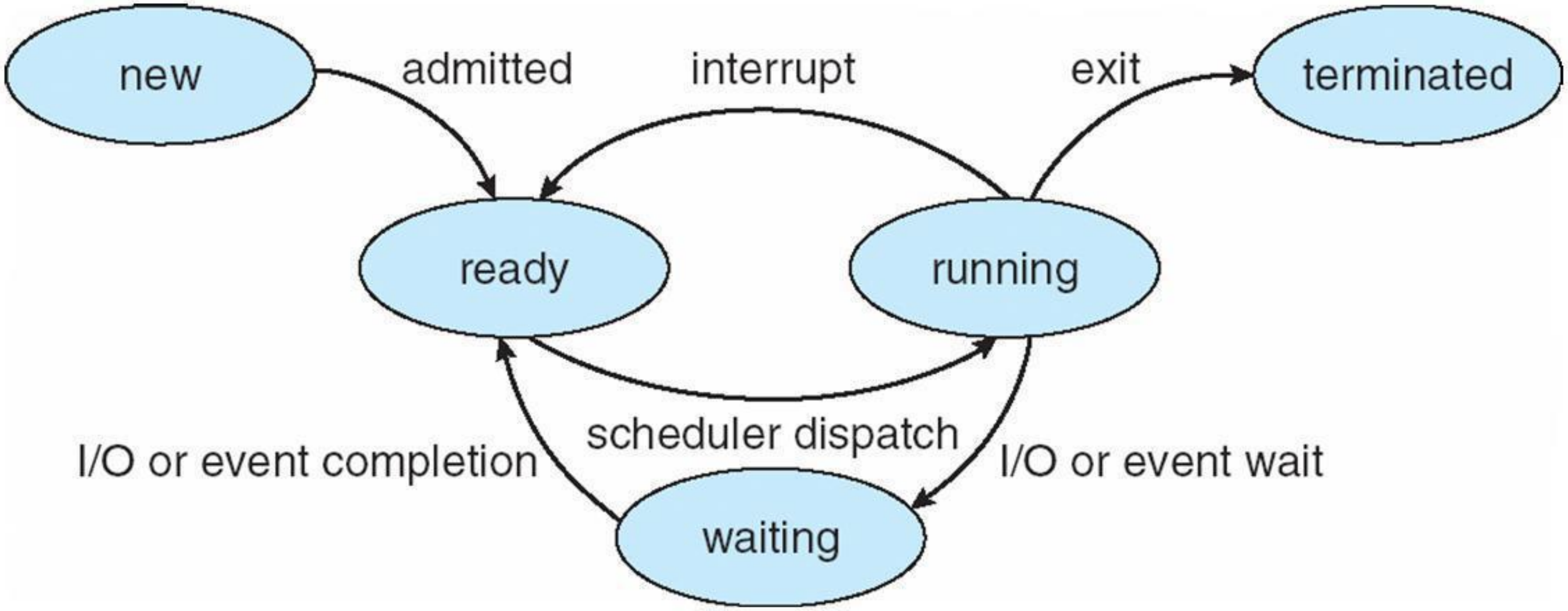


Figure 3-2. five-state Process State Transition Diagram

New : The process is being created.

Running : Instructions are being executed.

Waiting : Waiting for some event to occur(such as an I/O completion or reception of a signal).

Ready : Waiting to be assigned to a processor.

Terminated : Process has finished execution.

- **Preemptive** scheduling allows a running **process** to be interrupted by a high priority **process**.
- **Non-preemptive** scheduling, any new **process** has to wait until the running **process** finishes its CPU cycle.

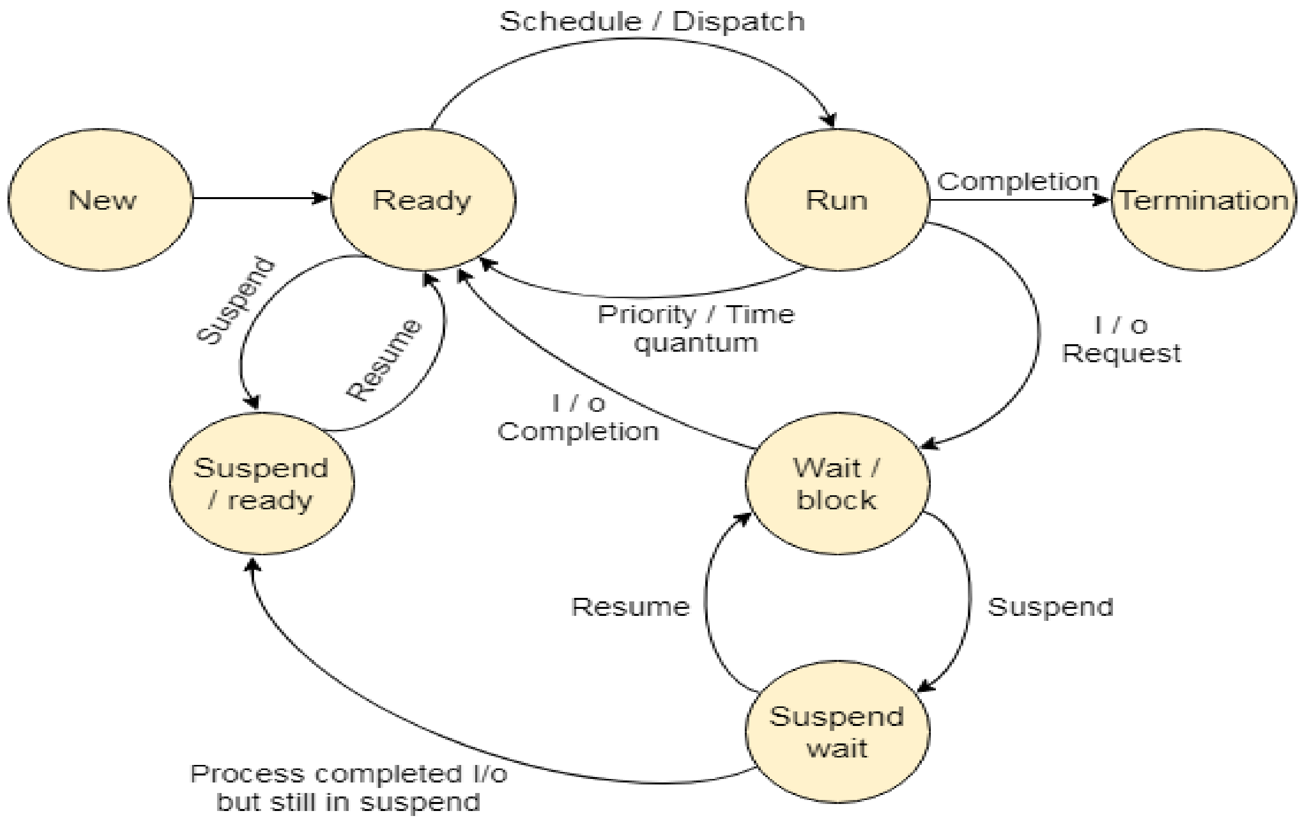
Process suspension

- Many OS are built around (Ready, Running, Blocked) states. But there is one more state that may aid in the operation of an OS - suspended state.
- When none of the processes occupying the main memory is in a Ready state, OS swaps one of the blocked processes out onto to the Suspend queue.

Suspend ready: A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

Suspend wait: Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory.

Process State Transition Diagram



Process Control Block(PCB)

- Each process is represented in the operating system by a process control block (PCB)-also called a task control block.
  - Process State - e.g. new, ready, running etc.
  - Process Number – Process ID
  - Program Counter - address of next instruction to be executed

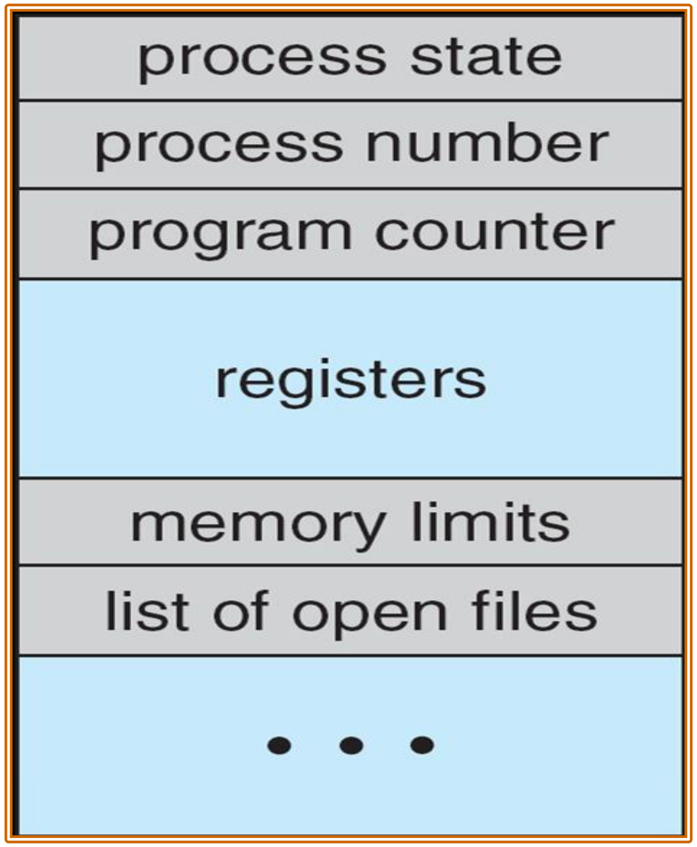


Figure 3.3. Process control block (PCB).

- CPU registers - general purpose registers, index registers, stack pointer etc.
- CPU scheduling information - This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory Management information - This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
- Accounting information - This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O Status information - This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Process Creation

Events which cause process creation:

- Initialization of operating system
- A user request to create a new process
- Initiation of a batch job
- Parent process create children process
- Address space
- Resource sharing
- Parent and child share no resources

Process Hierarchy

- Modern general purpose operating system used to create and destroy processes. A process may create several new processes during its time of execution.
- The creating process is called "Parent Process", while new processes are called "Child Processes".

There are different possibilities concerning creating new processes:-

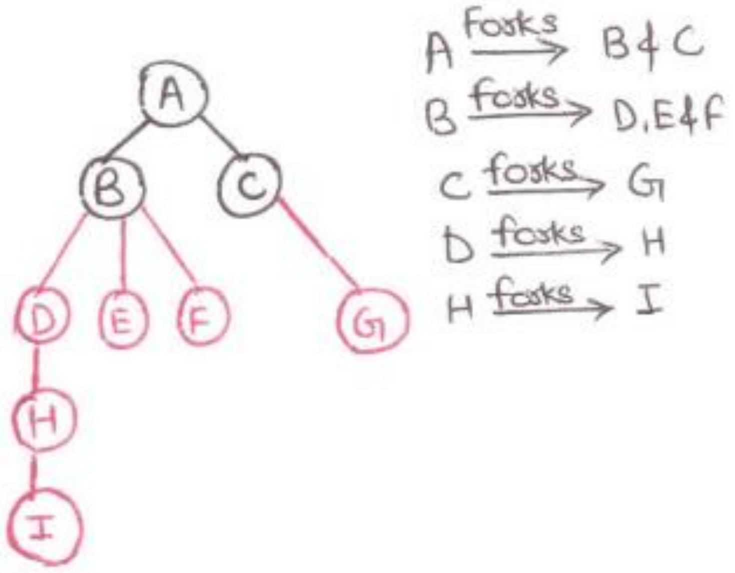
Execution: The parent process continues to execute concurrently with its children processes or it waits until all of its children processes have terminated (sequentially).

Sharing: Either the parent and children processes share all resources (like memory or files) or the children processes share only a subset of their parent's resources or the parent and children process share number of resources in common.

A parent process can terminate equation of one of its children for one of these reasons:

- The child process has exceeded its usage of the resources it has been allocated. For this a mechanism must be available to allow the parent process inspect the state of its children process.
- Task assigned to child process is no longer required.

In unix this is done by the **'Fork'** system call, which creates a **'child'** process and the **'exit system call'**, which terminates current process.



A
B
C
D
E
F
G
H
I
A $\xrightarrow{\text{Forks}}$ B & C
B $\xrightarrow{\text{forks}}$ D, E & F
C $\xrightarrow{\text{forks}}$ G
D $\xrightarrow{\text{forks}}$ H
H $\xrightarrow{\text{forks}}$ I

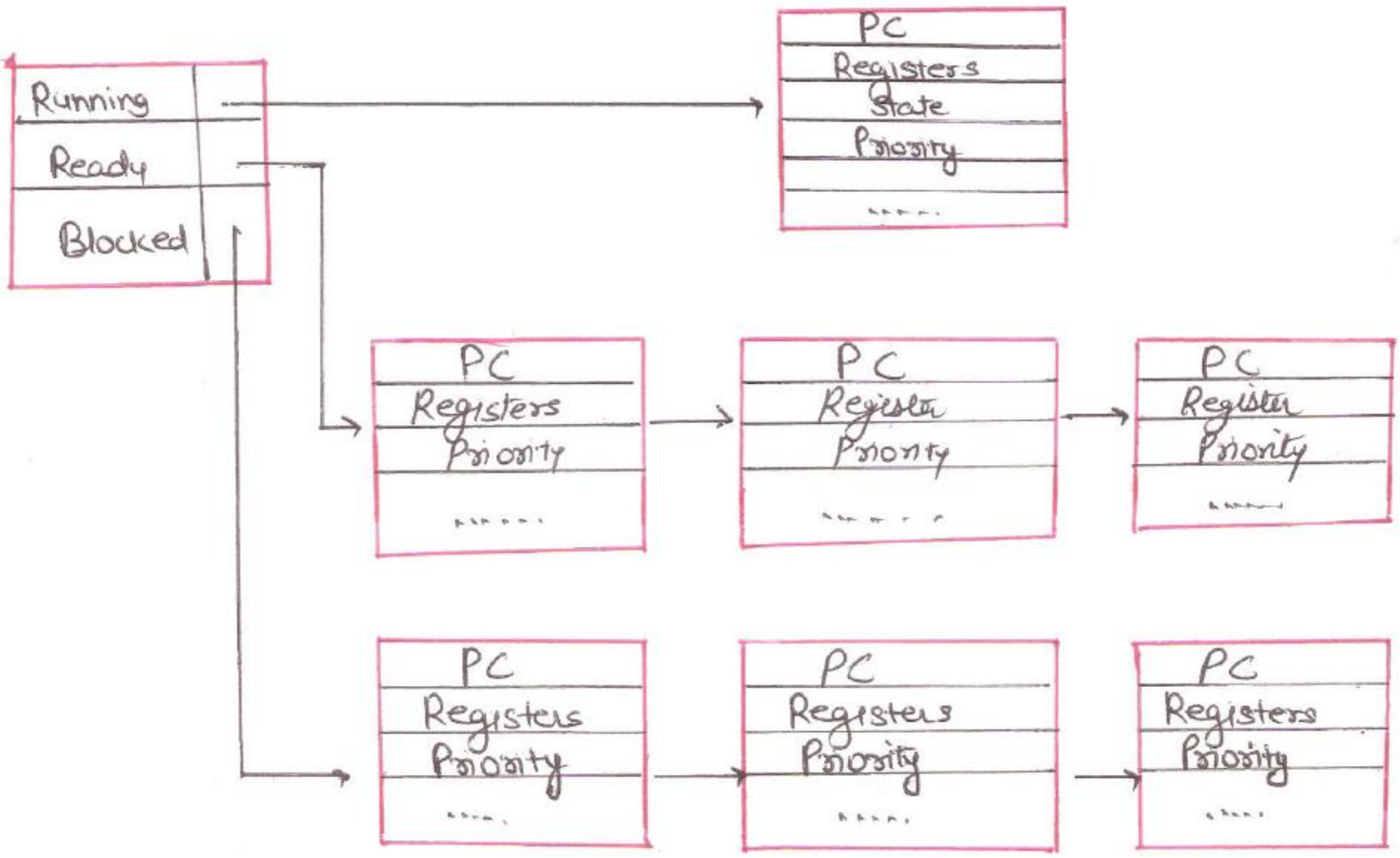
Process Termination

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).
- When a process is terminated, the resources that were being utilized by the process are released by the operating system.
- When a child process terminates, it sends the status information back to the parent process before terminating.

Implementation of Processes

- Process Model is implemented by **Process Table and Process Control Block** which keep track all information of process.
- At the time of creation of a new process, operating system allocates a memory for it loads a process code in the allocated memory and setup data space for it .
- The state of process is stored as ' new ' in its PCB and when this process move to ready state its state is also changes in PCB.
- When a running process needs to wait for an input output devices, its state is changed to 'blocked'. The various queues used for this which is implemented as linked list.


Running
Ready
Blocked
PC
Registers
State
Priority
PC
Registers
Priority
PC
Register
Priority
PC
Register
Priority
PC
Registers
Priority
PC
Registers
Priority
PC
Registers
Priority

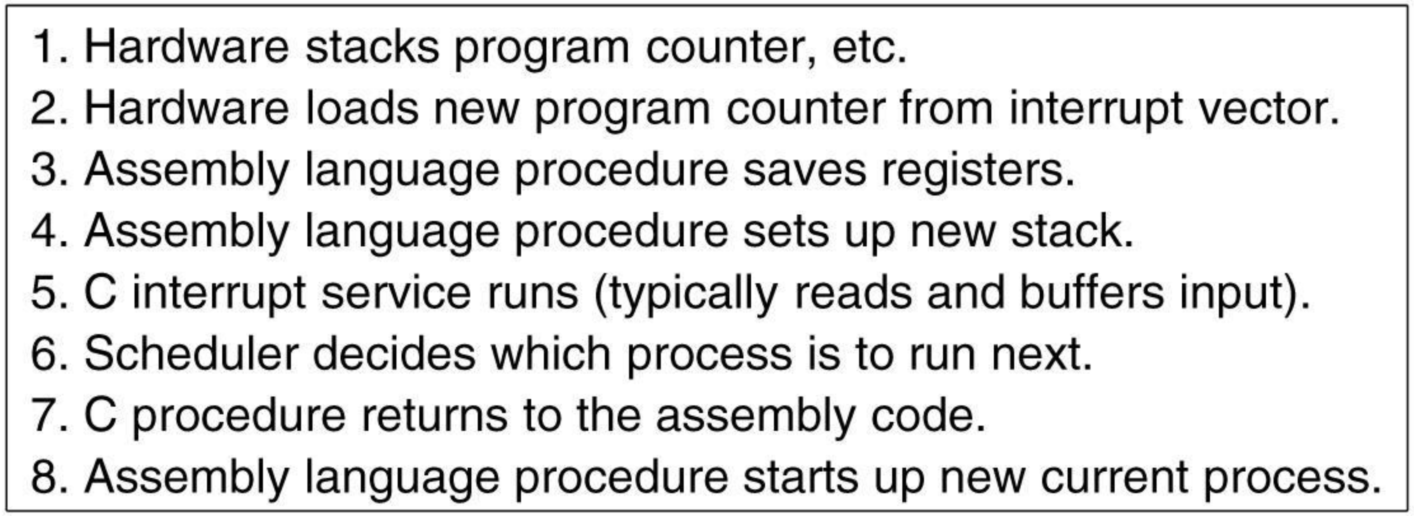
There are many following queues:

- **Read Queue:** This queue is used for storing the processes with state ready .
- **Blocked Queue:** it is used for storing the processes but need to wait for an input output device or resource.
- **Suspended Queue:** It is used for storing the blocked process that have been suspended.
- **Free process Queue:** it is used for the information of empty space in the memory where a new PCB can be created.

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure. Some of the fields of a typical process-table entry.

Implementation of Processes



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

Figure 3-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Process management

Compiled by, Er. Nabaraj Bahadur Negi

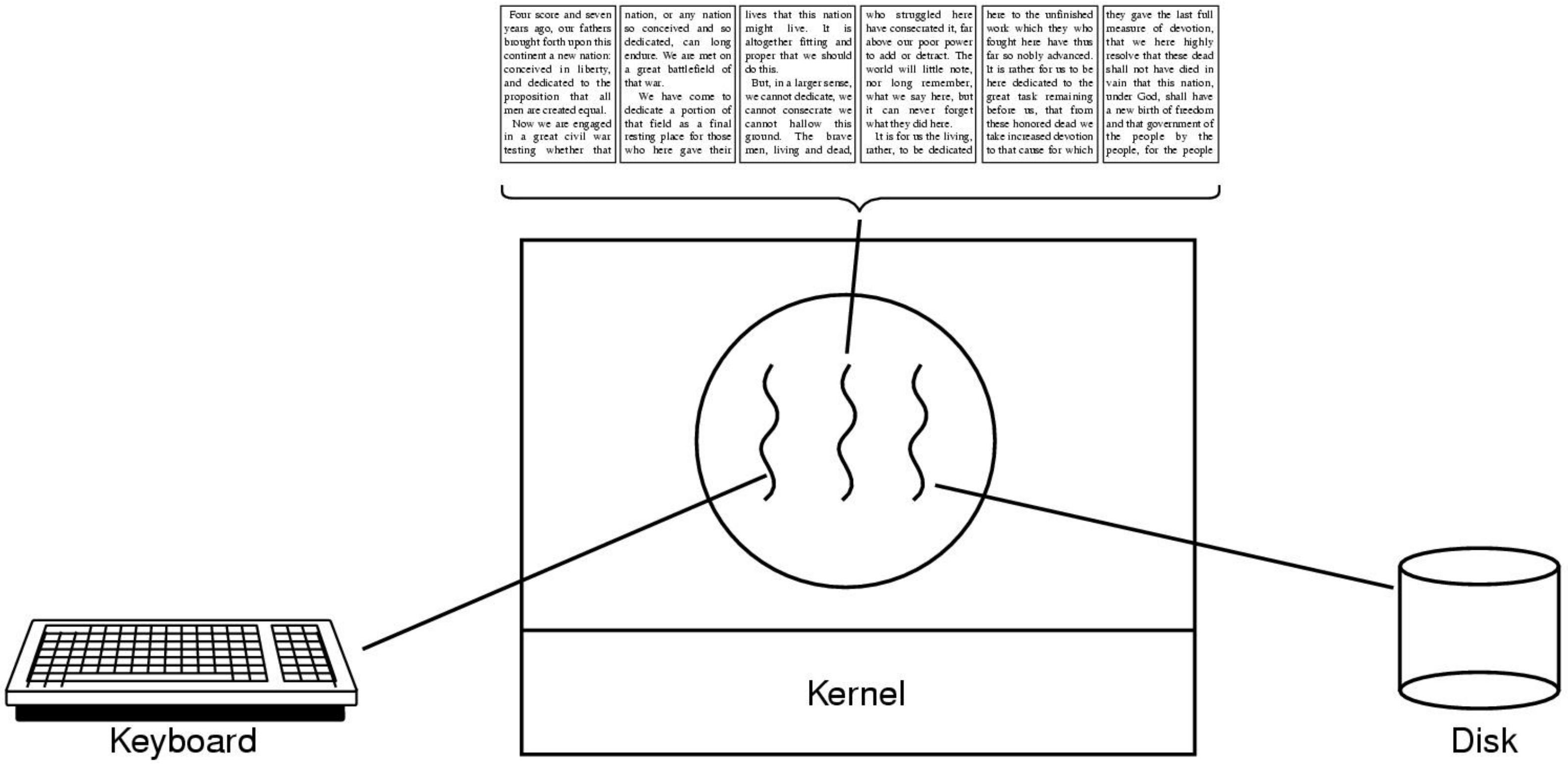
Topics Covered

- Threads
- Inter process communication(IPC)
- Implementing mutual exclusion
- Classical IPC problems

Thread

- A **thread (or lightweight process)**
  - It consists of : program counter, register set and stack space.
  - A **system registers** which hold its current working variables, a **stack** which contains the execution history and **counter** that keeps track of which instruction to execute next**.**
  - A thread shares the following with peer threads:
    - Code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
  - May have own **PBC**
- Traditional (heavyweight) processes have a single thread of control - there is one program counter, and one sequence of instructions that can be carried out at any given time.

- **Multithreading:** a single program made up of a number of different concurrent activities .
- An application typically is implemented as a separate process with several threads of control.
- A Web browser might have one thread display images or text while another thread retrieves data from the network.
- For Example, **A word processor** may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.
- In certain situations, a single application may be required to perform several similar tasks. For example, a **Web server** accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several clients concurrently accessing it.



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Figure. A word processor with three threads

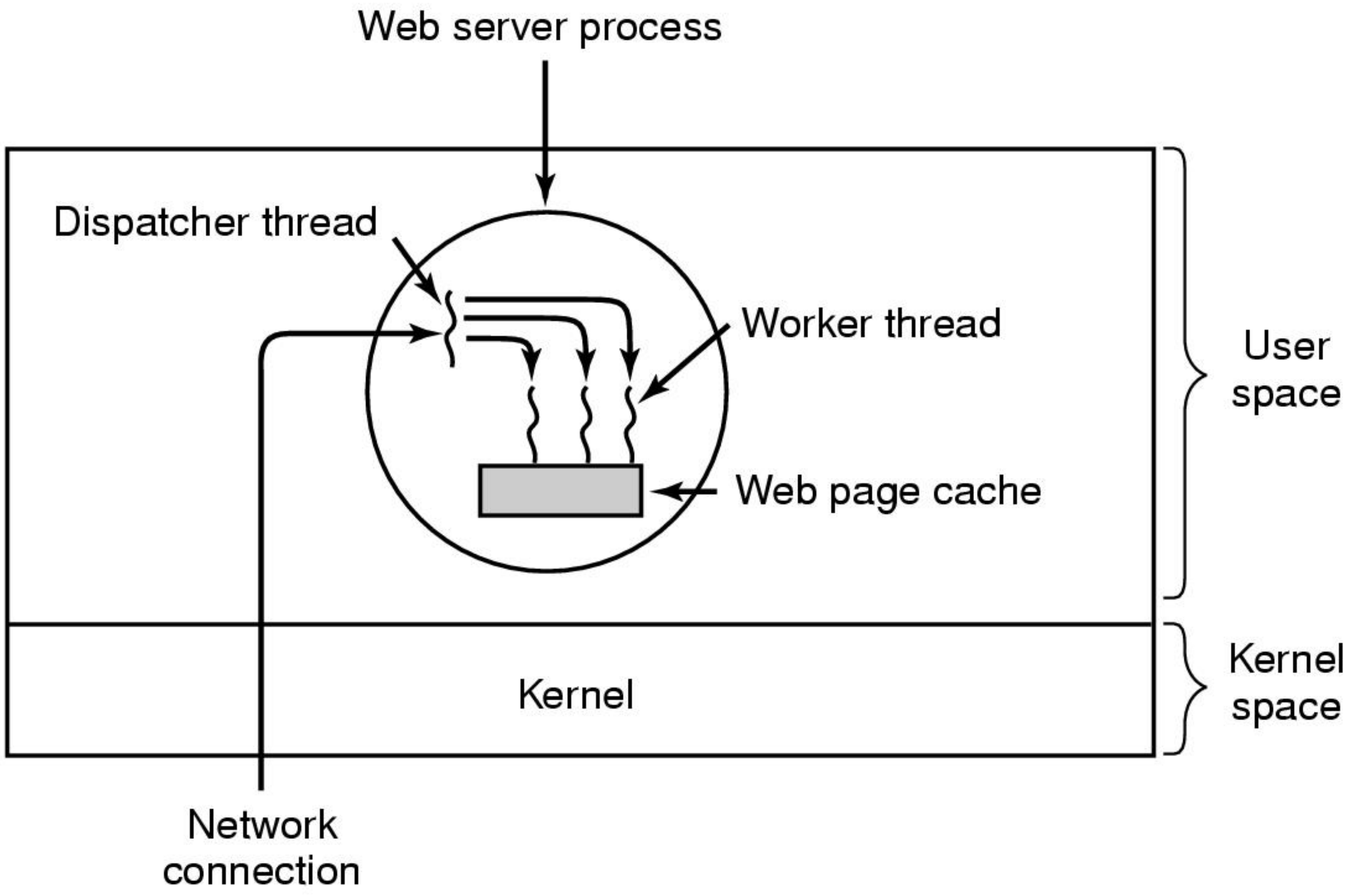


Figure. A multithreaded Web server

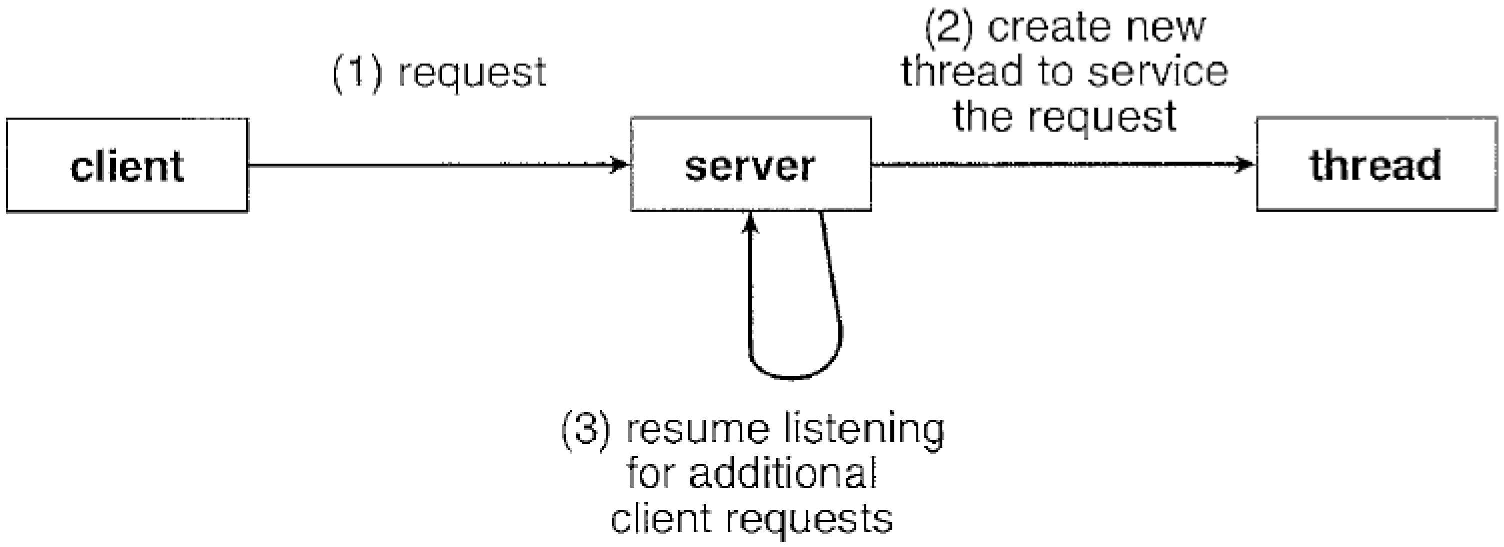

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Figure .Multithreaded server architecture.

- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

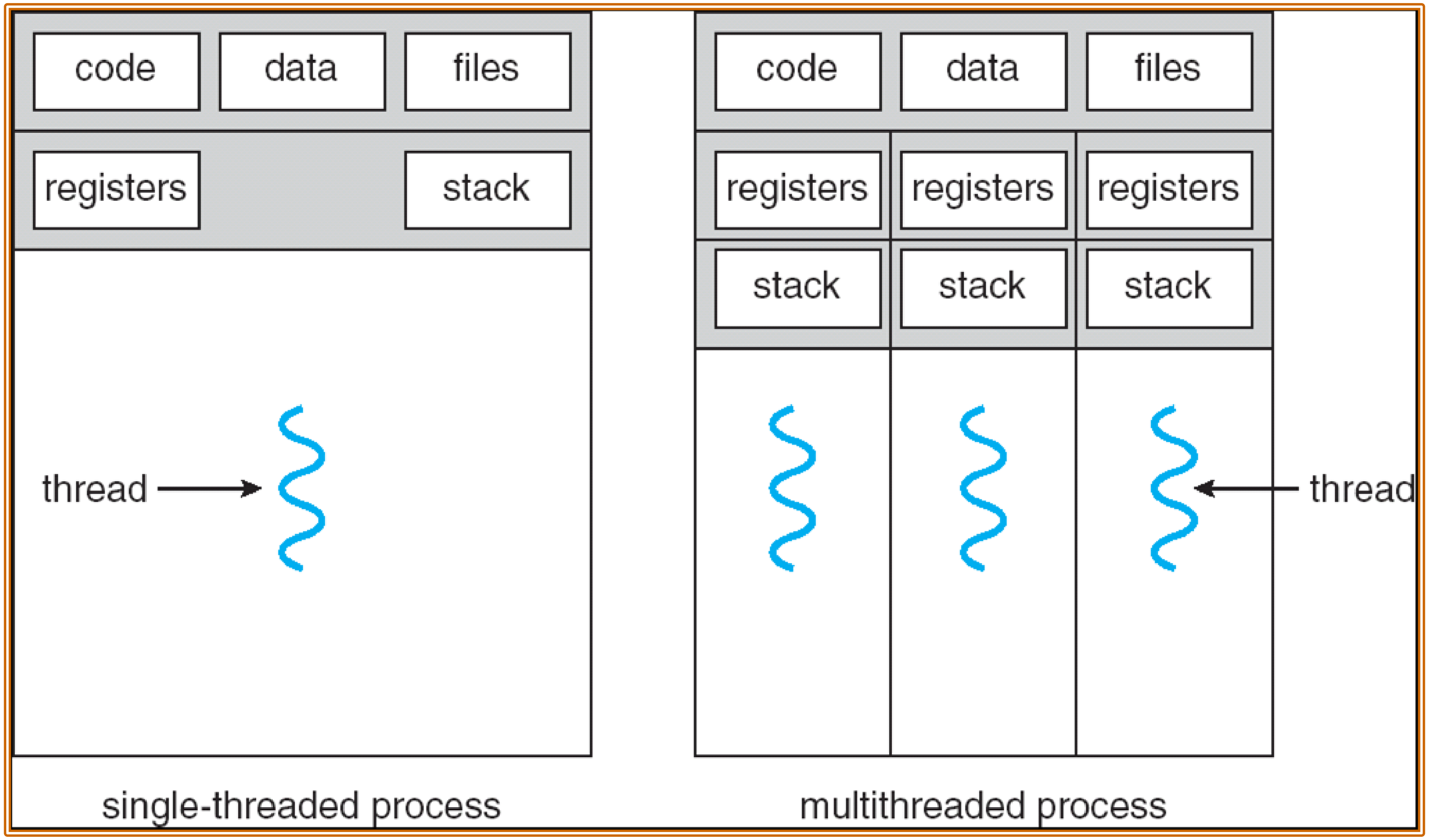


Figure .Single-threaded and multithreaded processes.

Multicore Programming

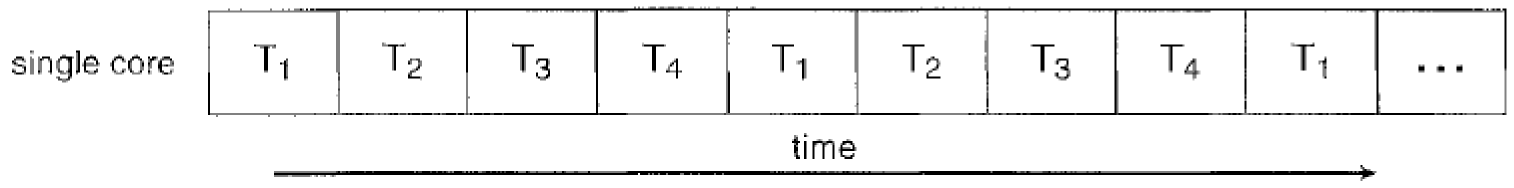


Figure .Concurrent execution on a single-core system.

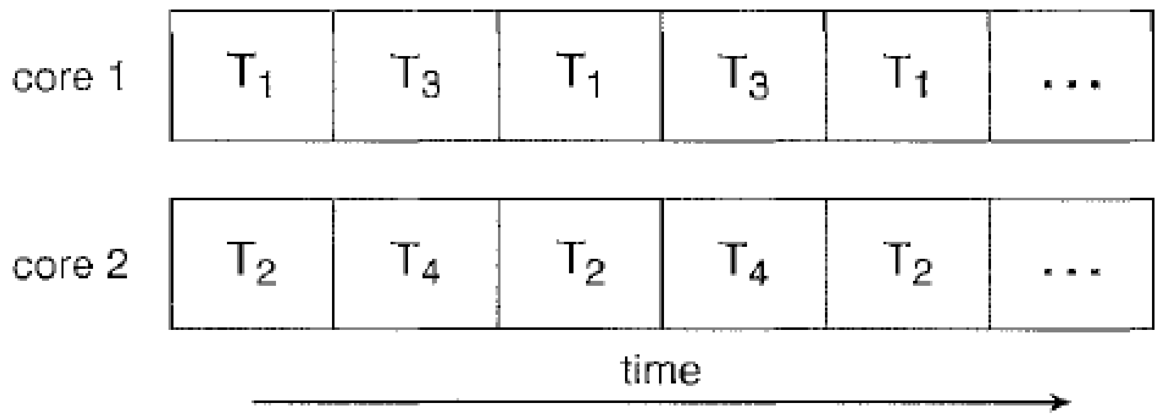


Figure . Parallel execution on a multicore system.

- A recent trend in computer architecture is to produce chips with multiple **cores**, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads,
- On a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing.

Types of Threads

- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

User-level threads

- User-level threads are implemented by users and the kernel is not aware of the existence of these threads.
- User level thread is also called many-to-one mapping thread because the operating system maps all threads in a multithreaded process to a single execution context. The operating system considers each multithreaded processes as a single execution unit.
- User level thread uses space for thread scheduling. The threads are transparent to the operating system

- User threads are implemented by users and managed entirely by the run-time system.
- If one user level thread performs blocking operation then entire process will be blocked.
- Implementation is by a thread library at the user level
- User level threads are created by runtime libraries that cannot execute privileged instructions.
- User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, if one thread causes a page fault, the process blocks.
- Example : **Java thread, POSIX threads**.

Kernel-supported threads

- Supported by the kernel.
- Threads are constructed and controlled by system calls. The system knows the state of each thread
- Kernel level thread support one-to-one thread mapping. Mapping requires each user thread with kernel thread.
- Kernel level threads are implemented by Operating system. In the kernel level thread, thread management is done by kernel.
- Implementation of kernel thread is complicated.
- Since, kernel managing threads, kernel can schedule another thread if a given thread blocks rather than blocking the entire processes.
- Kernel level threads are slower than user level threads
- Examples: Windows XP/2000,Solaris 2,Linux,Mac OS X

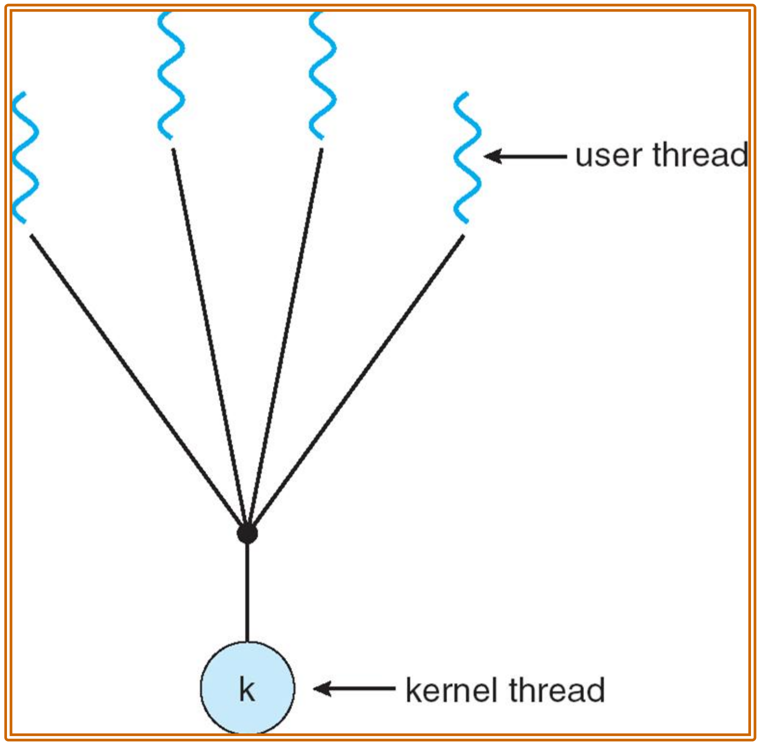
Thread libraries: Thread libraries provide programmers with an API for creating and managing threads.

- Thread libraries may be implemented either in user space or in kernel space. Library entirely in user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- Three primary thread libraries:
    - **POSIX Ptheads:** may be provided as either a user or kernel library, as an extension to the POSIX standard.
    - **Win32 threads:** provided as a kernel-level library on Windows systems.
    - **JAVA threads:** Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Multithreading Models

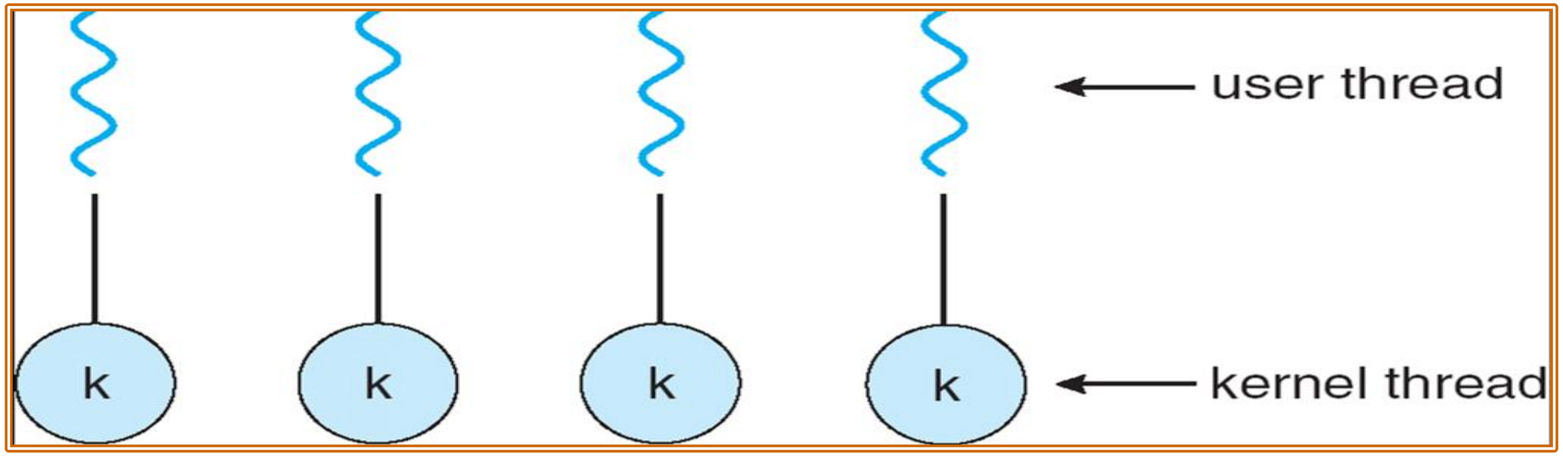
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- If one thread initiates a blocking system call, then no other thread in the same process can execute
- No parallelism.



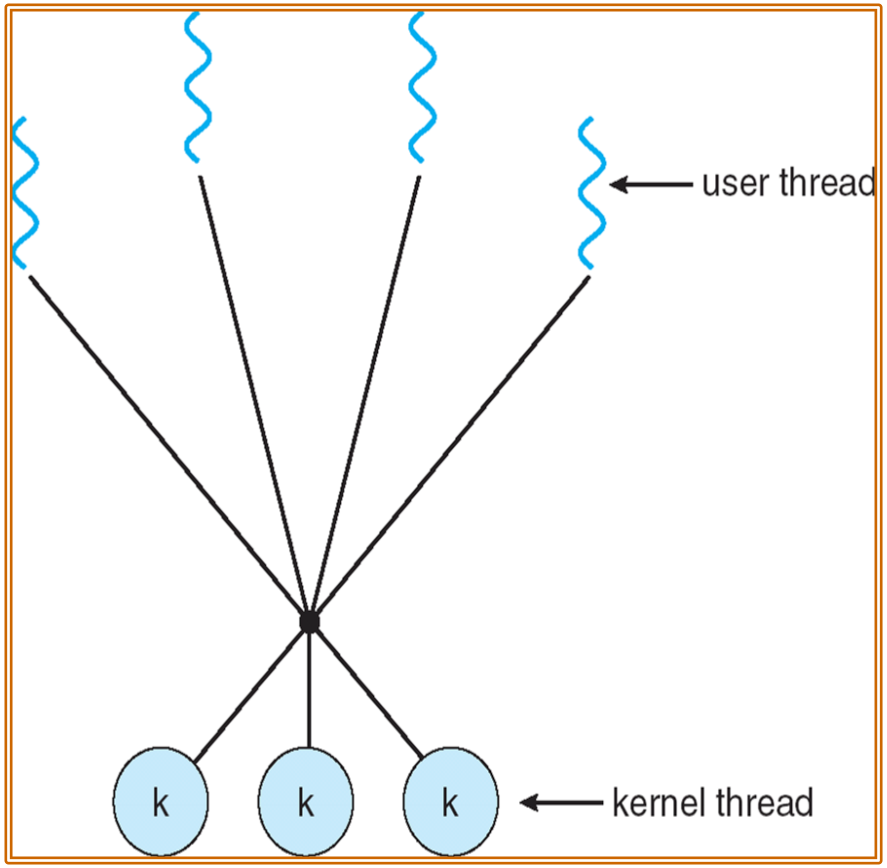
One-to-One

- Each user-level thread maps to kernel thread
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- Allows multiple threads to run in parallel on multiprocessors.
- Examples: Windows NT/XP/2000; Linux; Solaris 9 and later
- Drawback: Creating a user thread requires creating the corresponding kernel thread



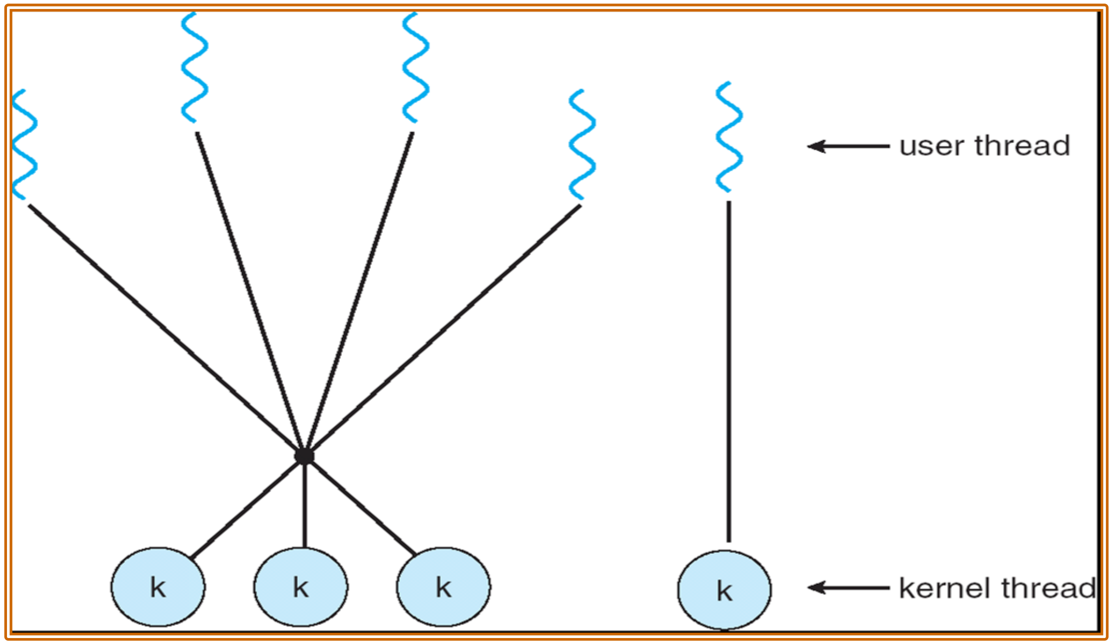
Many-to-Many Model

- Allows many user level threads to a smaller or equal number of kernel threads.
- Allows the operating system to create a sufficient number of kernel threads
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Solaris prior to version 9,Windows NT/2000 with the *ThreadFiber* package



Two-level Model

- One popular variation on the many-to-many model , which allows either many-to-many or one-to-one operation
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier



Threads: benefits

User responsiveness:

- When one thread is blocked (e.g., waiting for I/O), another thread in the same process can continue to handle user input/output.
- This improves overall application responsiveness, especially in interactive programs (e.g., GUIs, web servers).

Resource sharing:

- Memory is shared (i.e., address space shared)
- Open files, sockets shared

Economy (Efficiency):

- Creating a new thread is much faster and requires less system overhead than creating a new process.
- Context switching between threads is typically faster than between processes, as threads share process resources.

Speed: E.g., Solaris: thread creation about 30x faster than heavyweight process creation; context switch about 5x faster with thread

Hardware Parallelism:

- Threads, like heavyweight processes, can utilize multiprocessor (multi-core) architectures.
- Different threads of the same process can run in parallel on multiple CPUs, boosting performance for compute-intensive tasks.

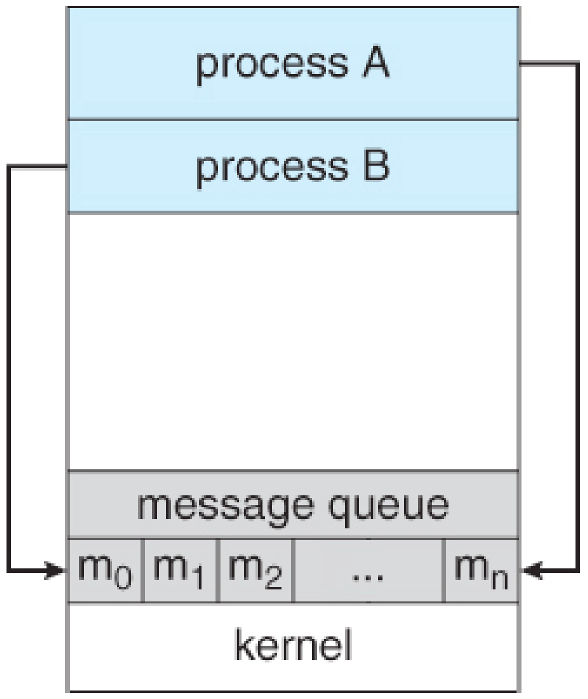
Inter-process Communication(IPC)

- Inter-process communication (IPC) refers to mechanisms that allow processes and threads to exchange data and synchronize their actions. IPC is essential for building applications where multiple processes or threads need to work together.

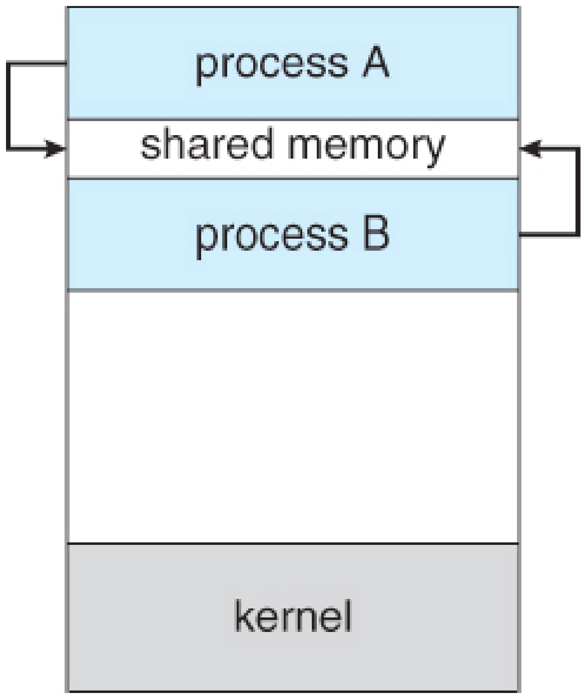
Reasons for Process Co-operation:

- **Information sharing:** Allows processes or threads to exchange data.
- **Computation speedup:** Tasks can be divided into smaller subtasks that run in parallel.
- **Modularity:** System functions are split into separate, smaller processes or threads. Each module handles a specific part of the system, making it easier to design, test, and maintain.

- **Convenience:** Enables users to perform multiple tasks simultaneously (multitasking). Example: A user can edit a document, compile code, print, and browse the web all at once.
- There are two primary models of Interprocess Communication:



(a) Message Passing



(b) Shared Memory

- In the **shared memory system**, the cooperating process which wants to initiate the communication establishes a region of shared memory in its address space. The other cooperative process which requires the shared data has to attach itself to the address space of the initiating process.
- The process A has some data, to share with process B. Process A has to take the initiative and establish a shared memory region in its own address space and store the data or information to be shared in its shared memory region.
- Process B requires the information stored in the shared segment of the A. So, process B has to attach itself to the shared address space of A. Now, B can read out the data from there.
- Process A and B can exchange information or data by reading and writing data in the shared segment of the process.
- The **message-passing system** shares the data by passing the messages. If a process needs to share the data, it writes the data in the message and passes it on to the kernel. The process that requires the shared data read it out from the kernel.

Race Conditions

- **Process Synchronization** is a mechanism that deals with the synchronization of processes. It controls the execution of processes running concurrently to ensure that consistent results are produced.
- The situation where two or more processes reading or writing some share data and the final results depends on who runs precisely when are called **Race conditions**.
- A **race condition** occurs when two processes can interact and the outcome depends on the order in which the processes execute.
- When a process wants to print a file, it enters the file name in a special **spooler directory**.
- Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name.
- 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 queued for printing).

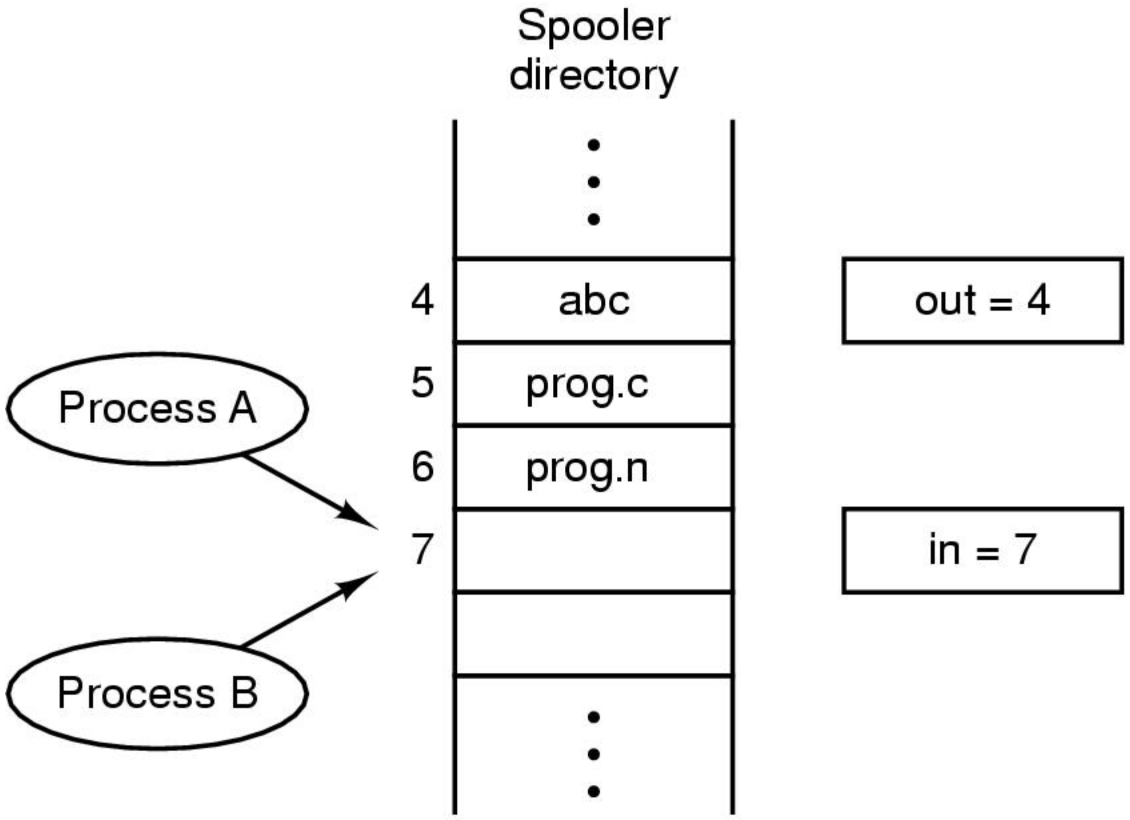


Figure .Two processes want to access shared memory at same time

- An operating system is providing printing services, “printer system process ” picks a print job from spooler. Each job gets a job number, and this is done by the spooler using two global variables “in” and “out”.
- 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 in slot 7 and update into be an 8. Then it goes off and does other things.
- Eventually, process A runs again, starting from the place it left off. 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.

Critical Regions

- That part of the program where the shared memory is accessed is called the **critical region** or **critical section**.
- Atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.
- To avoid race condition, we need mutual exclusion.
- **Mutual exclusion**: Mechanism which makes sure that two or more processes do not access a common resource at the same time.
- The difficulty in printer spooler occurs because process B started using one of the shared variables before process A was finished with it. If we could arrange matter such that no two processes were ever in there critical regions at the same time, we could avoid race conditions.

Solution to Critical Section Problem:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.

Requirements of Synchronization mechanisms

Mutual Exclusion:

- This means that if one process is executing inside the critical section, no other process should be allowed to enter the critical section at the same time.
- This prevents simultaneous access to shared resources, thereby avoiding inconsistent results or data corruption.

Progress:

- Progress states that if no process is currently in the critical section and some processes wish to enter it, then the selection of the processes that will enter next cannot be postponed indefinitely.
- In other words, processes not wishing to enter the critical section should not prevent others from entering.

Bounded Waiting:

- There must be a limit on the number of times that other processes are allowed to enter the critical section after a process has made a request to enter and before the request is granted.
- This ensures bounded waiting, so no process waits forever (prevents starvation).

Architectural Neutrality:

- The synchronization mechanism should be architecturally neutral (portable).
- This means that if the solution works correctly on one system architecture, it should also work on other architectures without requiring changes.

Example :

- Process *A* enters its critical region at time *T*1. A little later, at time *T*2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time.
- Process *B* is temporarily suspended until time *T*3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at *T*4) and we are back to the original situation with no processes in their critical regions.

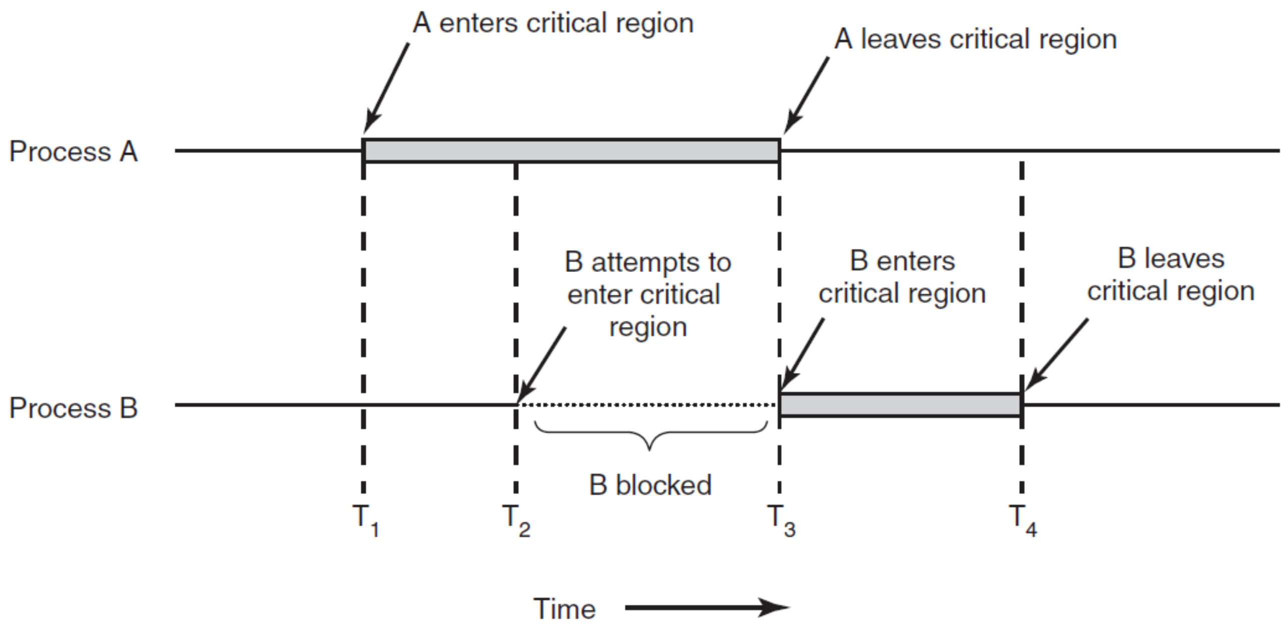


Figure . Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling interrupts

- Simplest solution, After entering critical region, disable all interrupts
- 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.
- May be used inside operating system kernel when some system Structures are to be updated, but is not recommended for Implementation of mutual exclusion in user space.
- Advantage: process inside critical region may update shared resources Without any risk of races
- Disadvantage: if after leaving the region interrupts are not reenabled There will be a crash of the system. Moreover: useless in multiprocessor Architectures.

Lock variables

- Software solution at user level, multiprogram med solution.
- A software solution. Considering having a single, shared lock variable, initially 0.
- When a process wants to enter critical region, it first tests the lock.
- If the lock = 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.
- The pseudo code of the mechanism looks like following.

```
Entry Section →
While (lock! = 0);
Lock = 1;
//Critical Section
Exit Section →
Lock =0;
```

Strict Alternation/Turn Variable

```
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)

Figure. A proposed solution to the critical region problem. (a) Process P0. (b) Process P1. In both cases, be sure to note the semicolons terminating the while statements.

- Turn Variable or Strict Alternation Approach is the software mechanism implemented at user mode.
- It keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process P0 inspects turn, finds it to be 0, and enters its critical region. Process P1 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. A lock that uses busy waiting is called a **spin lock**.
- When process P0 leaves the critical region, it sets turn to 1, to allow process P1 to enter its critical region. This way no two process can enters critical region simultaneously.

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 */
}
```

Figure . Peterson's solution for achieving mutual exclusion.

- Peterson's solution is used for mutual exclusion and allowed to processes to share a single use resource without conflict. It uses only shared memory for communication.
- Peterson's solution originally worked only with two processes, but has been generalized for more than two.
- Initially neither process is in its 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 makes a call to enter region, it will hang there until interested[0] goes to FALSE, an event that happens only when process 0 calls leave region to exit the critical region.
- Now consider the case that both process call **"Enter Region"** almost simultaneously. Then both will store the process number in 'turn'. Which ever store is done last is the one that counts the first one is overwritten and 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 a critical region until process 0 exit its critical region.

TSL Instruction

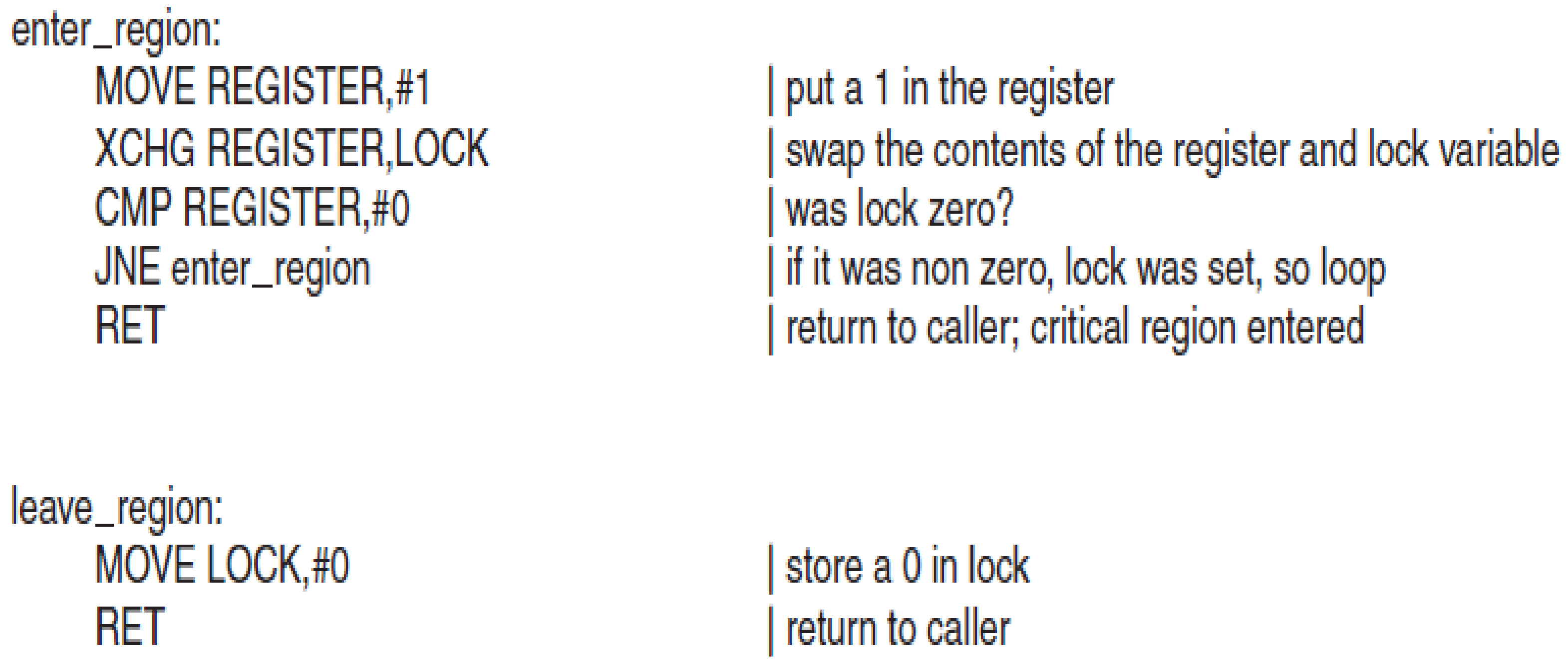
```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Figure . Entering and leaving a critical region using the TSL instruction.

- Software solution has many problem like, high processing overhead, logical errors etc.
- Hardware solution, multiprocess solution(n>=2), some computer architectures offer an instruction **TEST AND SET LOCK** (**TSL**).
- 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 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.

- An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically, for example, a register and a memory word.
- All Intel x86 CPUs use XCHG instruction for low-level synchronization.



```
enter_region:
        MOVE REGISTER,#1                | put a 1 in the register
        XCHG REGISTER,LOCK              | swap the contents of the register and lock variable
        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
```

Figure . Entering and leaving a critical region using the XCHG instruction.

Sleep and Wakeup

- When a process is not permitted to access its critical section, it uses a system call known as **sleep**, which causes that process to block. The process will not be scheduled to run again, until another process uses the **wakeup** system call.
- In most cases, **wakeup** is called by a process when it leaves its critical section if any other processes have blocked.
- There is a popular example called **producer consumer problem** which is the most popular problem simulating **sleep and wake** mechanism.
- The problem describe two processes, **User and Consumer**, who share a common **fixed size buffer.** Producer consumer problem also known as the **"Bounded Buffer Problem"** is a multi-process synchronization problem.
- Disadvantage: wakeup signal may be lost, which leads to deadlock.

Producer: The producer's job is to generate a bit of data, put it into the buffer and start again.

Consumer: The consumer is consuming the data (i.e remaining it from the buffer) one piece at a time.

- If the buffer is empty, then a consumer should not try to access the data item from it. Similarly, a producer should not produce any data item if the buffer is full.
- Counter: It counts the data items in the buffer. or to track whether the buffer is empty or full. Counter is shared between two processes and updated by both.

How it works?

• Counter value is checked by consumer before consuming it.

• If counter is 1 or greater than 1 then start executing the process and updates the counters.

• Similarly producer check the buffer for the value of Counter for adding data.

• If the counter is less than its maximum values, it means that there is some space in Buffer.

```
#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 */
    }
}
```

Figure. The producer-consumer problem with a fatal race condition.

Types of mutual exclusion

- Semaphores
- Monitors
- Locks (mutexes)
- Message passing
- Bounded Buffer(Producer consumer)

Semaphores

- In 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use.
- Proposed variable or abstract datatype called **semaphore ,**A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
- Dijkstra proposed having two operations on semaphores, now usually called down and up (generalizations of sleep and wakeup). he used the names P and V instead of down/wait and up/signal.
- The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible **atomic action**.

Two operations which can be used to access and change the value of semaphore variable.

```
P(semaphore s){
While(s==0);       /* wait until s=0*/
S=s-1;
}
V(semaphore s){
S=s+1;
}
```

```
#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 */
        }
}
```

Figure. The producer-consumer problem using semaphores.

- This solution uses three semaphores: one called full for counting the number of slots that are full, one called empty for counting the number of slots that are empty, and one called mutex to make sure the producer and consumer do not access the buffer at the same time.
- Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**, only takes the values 0 and 1 during execution of a program. Hence it is often called a **mutex**
- **Counting semaphore** can be used when we need to have more than one process in the critical section at the same time.
- Counting = $-\infty$ to $+\infty$, Binary=0,1
- It is a very popular tool used for **process synchronization**.

Monitors

- To make it easier to write correct programs, brinch hansen (1973) and hoare (1974) proposed a higher-level synchronization primitive called a **monitor**.
- A monitor is a collection of procedures,

 Variables, and data structures that are all

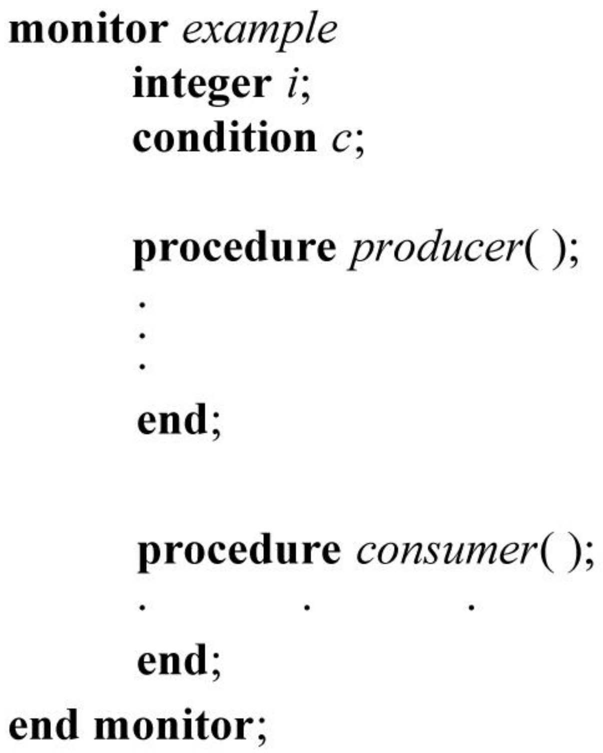
Grouped together in a special kind of module

Or package.

- Monitor is same as a class type: like

object of class are created, the variable

of monitor type are defined.



```
monitor example
     integer i;
     condition c;

     procedure producer( );
     .
     .
     .
     end;

     procedure consumer( );
     .       .       .
     end;
end monitor;
```

Figure. A monitor.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;

    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Figure . An outline of the producer-consumer problem with monitors

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again
ok:     RET                         | return to caller; critical region entered


mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET                         | return to caller
```

Figure. Implementation of *mutex lock* and *mutex unlock*.

- A **mutex** is a shared variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked.
- Tw o procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls mutex lock. If the mutex is currently unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.
- On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls mutex unlock. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure . Some of the Pthreads calls relating to mutexes.

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure . Some of the Pthreads calls relating to condition variables.

Message Passing

- This method of interprocess communication uses two primitives, **send** and **receive,** which, like semaphores and unlike monitors, are system calls rather than language constructs.

```
send(destination, &message);
and
receive(source, &message);
```

two methods of message addressing:

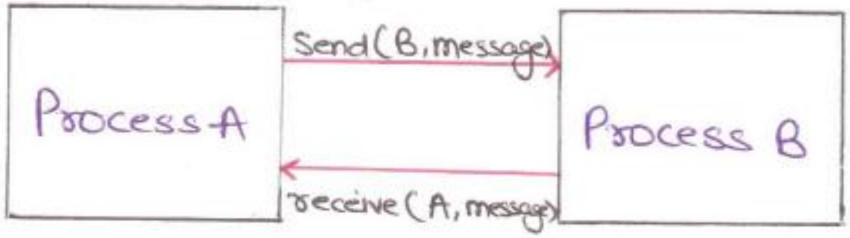
- **Direct addressing**, each process contains unique address. The following rendezvous mechanism may be used:
    - If **send** called before receive, the sending process suspended till the moment of the very message sending after receive call,
    - If **receive** called before send, the receiving process suspended till the moment of the very message sending after send call.

Example:

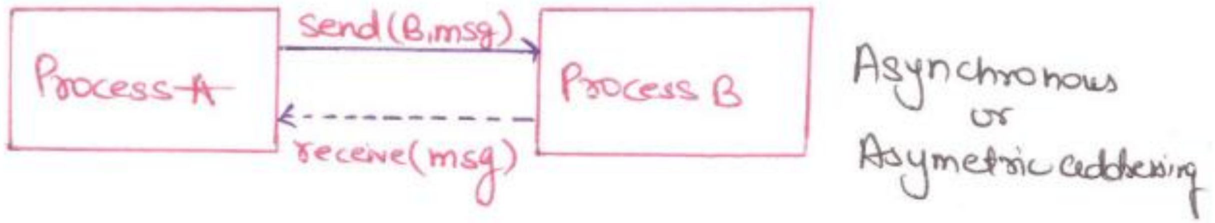
If process A sends a message to process B, then

send(B, message);
Receive(A, message);

- By message passing a link is established between A and B. Here the receiver knows the Identity of sender message destination. This type of arrangement in direct communication is known as **Symmetric Addressing.**



- Another type of addressing known as asymmetric addressing where receiver does not know the ID of the sending process in advance.



- **Indirect addressing**, via some mailbox playing the role of the Intermediate buffer. Send and receive has as an argument mailbox Address, not the address of any particular process.
- The sender and receiver processes should share a mailbox to communicate.

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                           /* message buffer */

        while (TRUE) {
                item = produce_item( );      /* generate something to put in buffer */
                receive(consumer, &m);       /* wait for an empty to arrive */
                build_message(&m, item);     /* construct a message to send */
                send(consumer, &m);          /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);       /* get message containing item */
                item = extract_item(&m);     /* extract item from message */
                send(producer, &m);          /* send back empty reply */
                consume_item(item);          /* do something with the item */
        }
}
```

Figure. The producer-consumer problem with *N* messages

Classical IPC Problems

The Dining philosopher problem

- In 1965, Dijkstra posed and then solved a **synchronization** problem he called 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.
- A philosopher either eat or think
- 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.

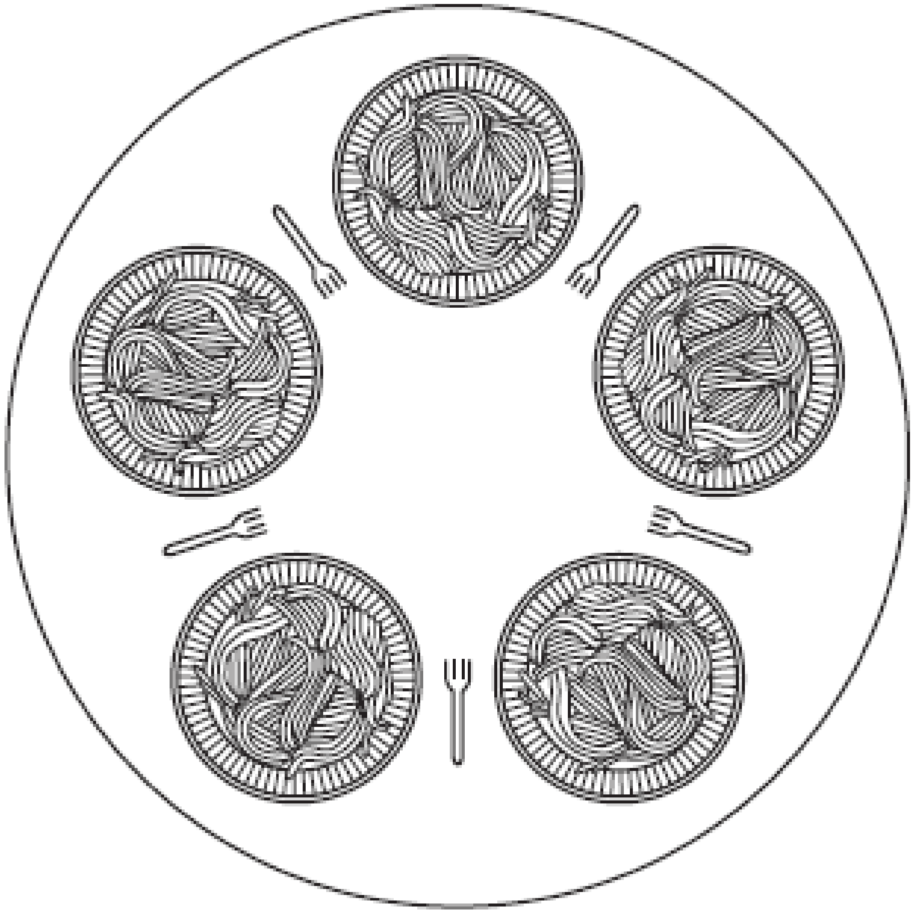


Figure . Lunch time in the Philosophy Department.

```
#define N 5                    /* number of philosophers */
void philosopher(int i)        /* i: philosopher number, from 0 to 4 */
{
while (TRUE) {
think( );                      /* philosopher is thinking */
take fork(i);                  /* take left for k */
take fork((i+1) % N);          /* take right for k; % is modulo operator */
eat( );                        /* yum-yum, spaghetti */
put fork(i);                   /* put left for k back on the table */
put fork((i+1) % N);           /* put right for k back on the table */
}
}
```

Figure. A nonsolution to the dining philosophers problem.

- The obvious solution is wrong. Suppose that all five philosophers take their left forks ,then waiting for right fork (None will be able to take their right forks), and there will be a **deadlock**.
- **Deadlock,** the ultimate form of starvation, occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are each waiting for the other(s) to do something.
- **Starvation** occurs when one or more threads in your program are blocked from gaining access to a resource and, as a result, cannot make progress.
- A **solution** of the **Dining Philosophers Problem** is to use a semaphore to represent a forks. A forks can be picked up by executing a wait operation on the semaphore and released by executing a signal 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 for ks */
#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 for ks or block */
eat( );                                  /* yum-yum, spaghetti */
put forks(i);                            /* put both for ks 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);                               /* tr y to acquire 2 for ks */
up(&mutex);                           /* exit critical region */
down(&s[i]);                           /* block if for ks 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]);
}
}
```

Figure . A solution to the dining philosophers problem.

The readers and writers problem

- Reader-writer problem in operating system which is used for process synchronization, which access the database.
- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state.
- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read.
- 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.

- Special type of mutual exclusion problem. A special solution can do better than a general solution.

Solution one:

- Readers have priority. Unless a writer is currently writing, readers can always read the data.

Solution two:

- Writers have priority. Guarantee no new readers are allowed when a writer wants to write.

Other possible solutions:

Weak reader's priority or weak writer's priority.

Weak reader's priority: An arriving reader still has priority over waiting writers. However, when a writer departs, both waiting readers and waiting writers have equal priority.

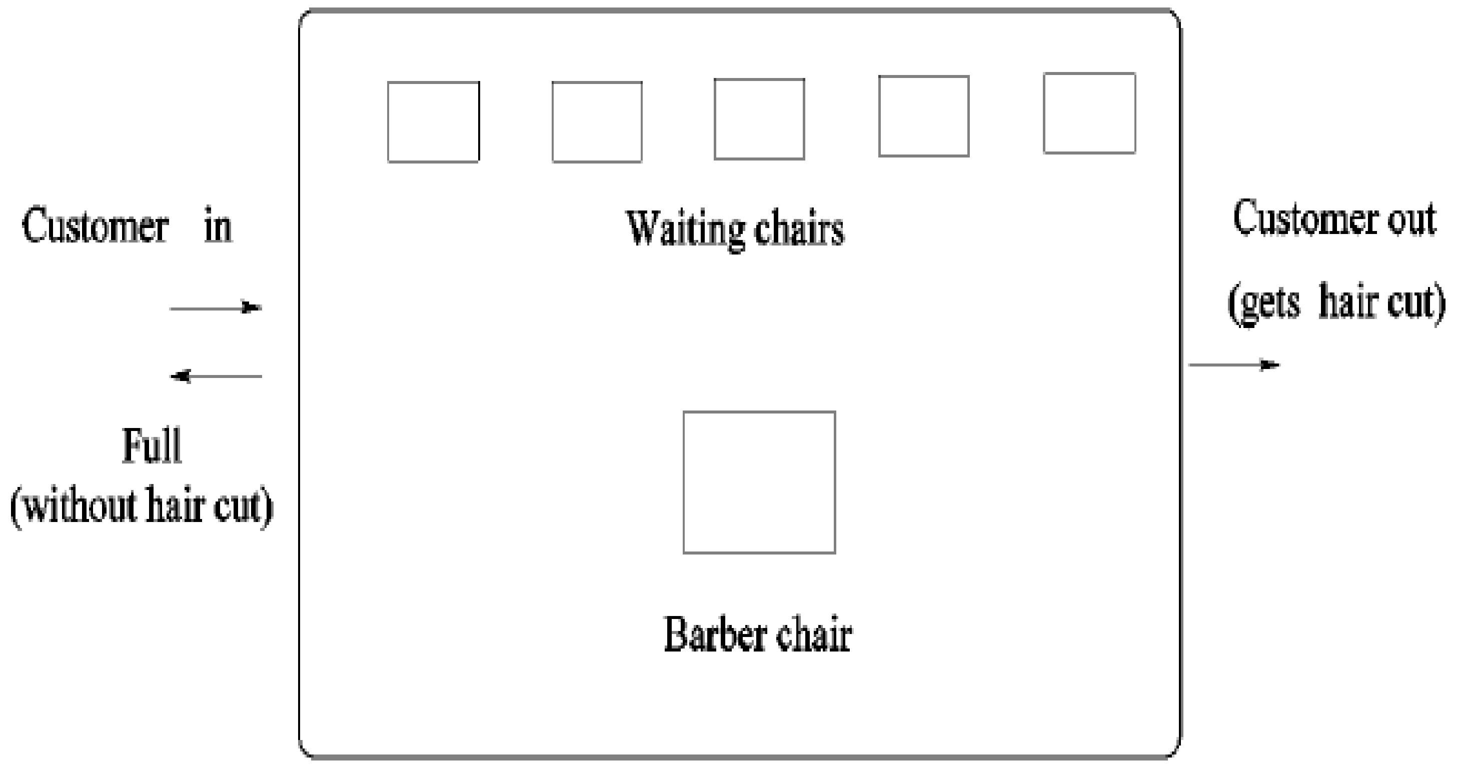
```
typedef int semaphore;                /* use your imagination */
semaphore mutex = 1;                   /* controls access to rc */
semaphore db = 1;                     /* controls access to the database */
int rc = 0;                           /* # of processes reading or wanting to */
void reader(void) {
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 few er 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 */
}
}
```

Figure. A solution to the readers and writers problem.

The Sleeping Barber Problem

Problem description:



- The barber shop has one barber, one barber chair, and n chairs for waiting customers, if any, to sit on.
- If there are no customers present, the barber sits down in the barber chair and falls asleep, as shown in the figure in the previous slide
- When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full).

The solution uses three semaphores:

- **customers** , which counts waiting customers (excluding the customer in the barber chair, who is not waiting),
- **barbers** , no. of barbers (0 or 1) based on barber is idle/waiting for customers,
- **mutex** , which is used for mutual exclusion.

- We also need a variable, waiting , which also counts the waiting customers. It is essentially a copy of customers .

```
#define CHAIRS 5                              /* number of chairs for waiting customers */
typedef int semaphore;
semaphore customers = 0;                       /* number of waiting customers */
semaphore barbers = 0;                        /* number of barbers waiting for customers */
semaphore mutex = 1;                          /* for mutual exclusion */
int waiting = 0;                              /* customers are waiting not being haircut */
void Barber(void){
 while (TRUE){
 down(customers);                             /* go to sleep if number of customers is 0 */
 down(mutex);                                  /* acquire access to ‘waiting’ */
 waiting = waiting – 1;                       /* decrement count of waiting customers */
 up(barbers);                                 /* one barber is now ready to cut hair */
 up(mutex);                                   /* release ‘waiting’ */
 cut_hair();                                  /* cut hair, non-CS */
 }}
```

```
void customer(void){
 down(mutex);                          /* enter CS */
 if (waiting < CHAIRS){
 waiting = waiting + 1;                /* increment count of waiting customers */
 up(customers);                        /* wake up barber if necessary */
 up(mutex);                            /* release access to ‘waiting’ */
 down(barbers);                        /* wait if no free barbers */
 get_haircut();                        /* non-CS */
 }else{
 up(mutex);                            /* shop is full, do not wait */
 }
}
```

Figure. A solution to the Sleeping Barber Problem

Process Scheduling

Categories of Scheduling Algorithms

Different environments need different scheduling algorithms

1. Batch Systems

- **Still widely used** in business and data processing.
- **Non-preemptive algorithms** (like FCFS, SJF) are preferred.

Why? Fewer process switches, which reduces overhead and is suitable for jobs that do not require user interaction.

2. Interactive Systems

- **Preemptive algorithms** are necessary (like Round Robin, Priority Scheduling).

Why? The system needs to switch between processes quickly to ensure responsive user interaction (e.g., multiple users or programs running at the same time).

3. Real-Time Systems

- Processes must **run quickly** and may **block** waiting for events.
- Require algorithms that guarantee deadlines and **predictable execution** (e.g., Rate Monotonic, Earliest Deadline First).

Scheduling Algorithm Goals

All Systems (General Scheduling Goals)

- **Fairness:** Every process should get a fair share of CPU time.
- **Policy Enforcement:** Ensure the scheduler follows the intended scheduling policy.
- **Balance:** Keep all system components (CPU, memory, I/O devices) as busy as possible.

1. Batch Systems:

- **Throughput:** Maximize the number of jobs completed per hour.
- **Turnaround Time:** Minimize the total time between job submission and completion.
- **CPU Utilization:** Keep the CPU busy all the time (avoid idle CPU).

2. Interactive Systems:

- **Response Time:** Respond to user requests as quickly as possible.
- **Proportionality:** Deliver results consistent with user expectations.

3. Real-Time Systems:

- **Meeting Deadlines:** Complete tasks within required time limits to avoid data loss or system failure.
- **Predictability:** Ensure consistent response times and quality (e.g., no jitter in multimedia applications).

Scheduling Criteria

- **Arrival time**: Time at which a process enters the ready queue or system.
- **Burst time**: Time required by a process to execute on the CPU.
- **Completion time**: Time at which a process completes its execution.
- **Turn Around Time**: Total time taken from arrival to completion. In simple words, it is the difference between the Completion time and the Arrival time.

Turn Around Time = Completion Time - Arrival Time

- **Waiting Time**: Time a process spends waiting in the ready queue. It is the difference between the Turn Around time and the Burst time of the process.

Waiting Time = Turn around time - Burst Time

- **Response Time:** The time interval between a process's arrival in the ready queue and its first execution on the CPU.

Response Time=Start Time-Arrival Time

- **Response Ratio :** Ratio used to prioritise processes based on waiting and burst time.

Response Ratio= (Waiting Time + Burst time) / Burst time

- **CPU utilization :** It measures how effectively the CPU is being used

  CPU utilization= (CPU Busy time/ Total Time)*100

Example: Total execution time (all burst times) = 20 ms, CPU idle time = 5 ms then, Total time = 25 ms.

CPU utilization=(20/25)*100=80%

- **Throughput :** Number of completed jobs per time unit**.**

  Throughput= Number of process completed/ Total time

Example: 5 processes completed, Total time from start to end = 20 ms.

Throughput=5/20=0/25 process per ms.

Scheduling in Batch Systems

- First-come first-served
- Shortest job first
- Shortest Remaining Time Next

First-Come First-Serve(FCFS) Scheduling

- Non-preemptive: Once a process starts executing, it runs to completion.
- Queue Type: FIFO (First-In, First-Out)- like waiting in line at a bank, post office, or for a printer.
- Runnable processes are added to the end of the ready queue.

Advantages:

- Simple and easy to implement.
- Fair in the sense that it serves in the order of arrival.

Disadvantages:

- Long average waiting times, especially if:

✓ A long job arrives before short jobs ("convoy effect").

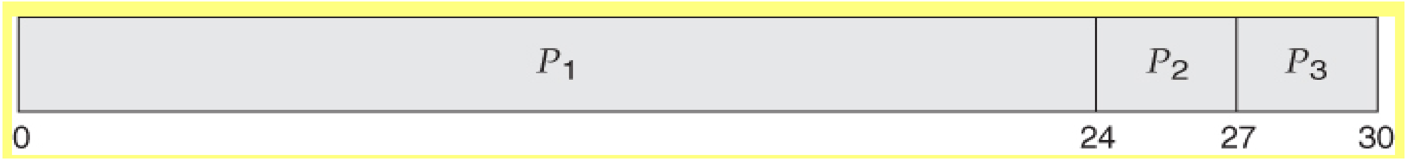
✓ The first process takes a long time, delaying others.

- Poor response time for short processes.

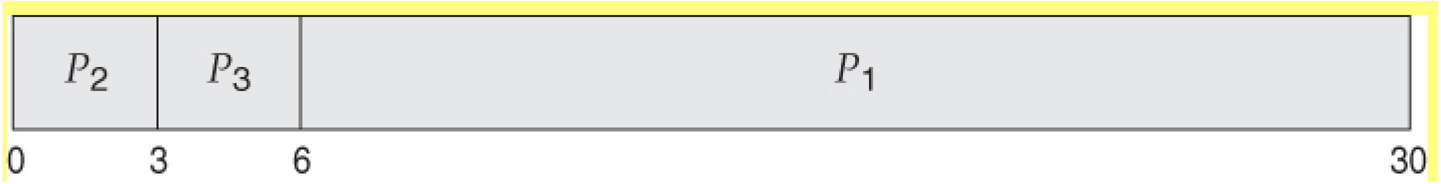
Example :1

Process	CPU Burst Time
P1	24
P2	3
P3	3

- In the first Gantt chart below, process P1 arrives first. Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
- The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17 ms.



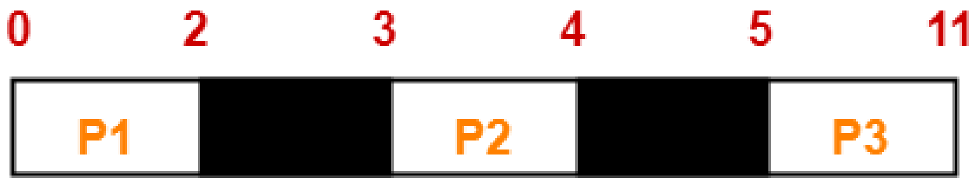
- Suppose that the processes arrive in the order *P2* , *P3* , *P1* ,Waiting time for *P1* = 6*; P2* = 0*; P3* = *3,*In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3 ms.



Example :2

Process Id	Arrival time	Burst time
P1	0	2
P2	3	1
P3	5	6

Gantt Chart: Here, black box represents the idle time of CPU.



Gantt Chart

Turn Around time = (Exit time/Completion time) – Arrival time

Waiting time = Turn Around time – (Burst time/CPU Execution time)

Process Id	Exit time	Turn Around time	Waiting time
P1	2	2 – 0 = 2	2 – 2 = 0
P2	4	4 – 3 = 1	1 – 1 = 0
P3	11	11- 5 = 6	6 – 6 = 0

Now,

Average Turn Around time = (2 + 1 + 6) / 3 = 9 / 3 = 3 unit

Average waiting time = (0 + 0 + 0) / 3 = 0 / 3 = 0 unit

Shortest-Job-First Scheduling (SJF)

- **Shortest Job First (SJF) / Shortest Job Next (SJN) / Shortest Process Next (SPN)**
- SJF is a CPU scheduling algorithm in which the **waiting process with the smallest execution (burst) time** is selected for the next execution.
- **Non-Preemptive SJF:** Once a process starts executing, it runs to completion. The scheduler always chooses the process with the **shortest next CPU burst** from the ready queue.
- **The shortest process executes first.**

Advantages:

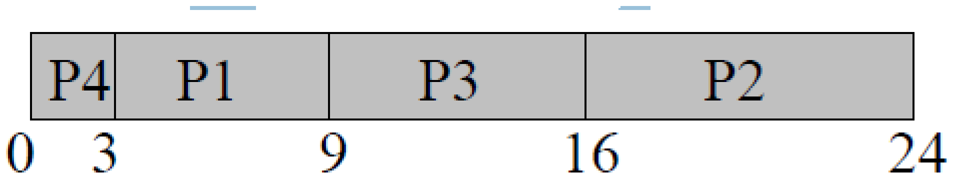
- Provides **minimum average waiting time** for a given set of processes.
- More efficient than FCFS for processes of varying lengths.

Disadvantages:

- **Difficult to predict the length** of the next CPU request.
- Can lead to **starvation** (long jobs might never get CPU if short jobs keep arriving).

Example :1

Process	CPU burst time
P1	6
P2	8
P3	7
P4	3



Average waiting time= (0+3+9+16)/4 = 7

Example : 2

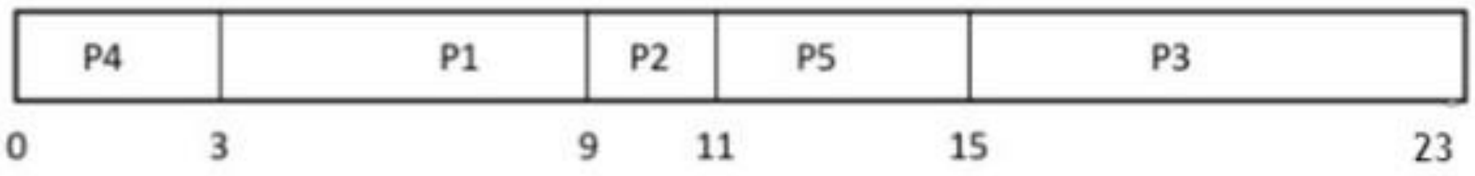
Process	Burst time/Execution time	Arrival time
P1	6	2
P2	2	5
P3	8	1
P4	3	0
P5	4	4

- ✓ **Completion time:** p1=9,p2=11,p3=23,p4=3, p5=15
- ✓ **Turn Around Time = (Completion Time - Arrival Time)**
- P1=9-2=7,p2=11-5=6,p3=23-1=22,p4=3-0=3,p5=15-4=11

Average Turn around time =7+6+22+3+11/5=49/5

- ✓ **Waiting Time = (Turn around time - Burst Time)**
- P4= 3-3=0 P1= 7-6=1 P2= 6-2=4 P5= 11-4=7 P3= 22-8=14

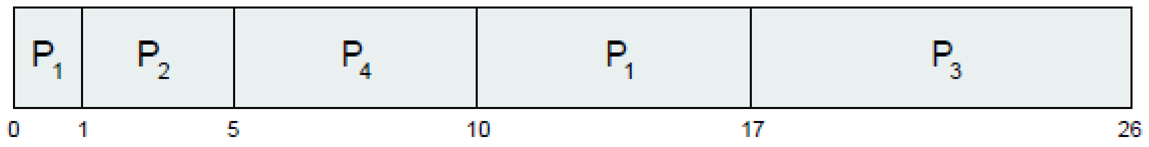
Average Waiting Time= 0+1+4+7+14/5 = 26/5 = 5.2



Shortest remaining time next

- A **preemptive** version of shortest job first is **shortest remaining time next**.
- A process with shortest burst time begins execution.
- If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.

Example :1



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

Average Waiting Time = ((10-1) +(1-1) + (17-2) +(5-3))/4 = 26/4= 6.5 ms

Scheduling in Interactive Systems

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Fair-Share Scheduling
- Lottery Scheduling

Round-Robin Scheduling

- One of the Most widely used, oldest, fairest, and easiest algorithms is **round robin**.
- Round robin is a preemptive algorithm
- The CPU is shifted to the next process after fixed interval time, which is called time quantum/time slice.
- Process runs until it blocks or time quantum exceeded
- If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course.

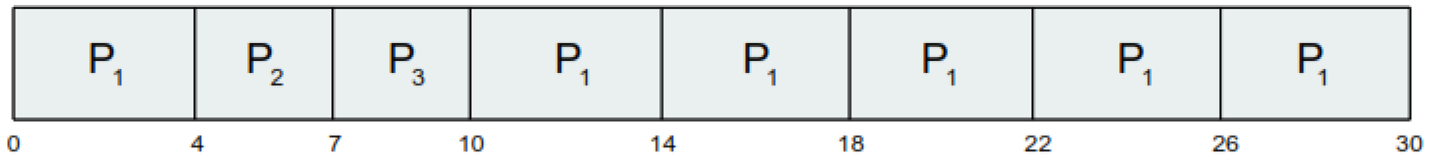
Example :1

Process	Burst time
P1	24
P2	3
P3	3

Time Slice=4.

Ready Queue: P1,P2,P3,P1

The Gantt chart assuming all processes arrive at time 0 is:



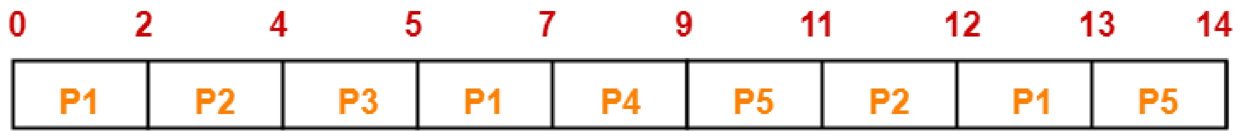
Average Waiting Time={(10-4)+(4-0)+(7-0)}/3=17/3=5.66 ms.

Example :2

If the CPU scheduling policy is Round Robin with time quantum = 2 unit, calculate the average waiting time and average turn around time.

Process	Arrival time	Burst time
P1	0	5
P2	1	3
P3	2	1
P4	3	2
P5	4	3

Ready Queue: P1, P2,P3,P1, P4, P5,P2, P1,P5



Gantt Chart

Process	Turn Around time	Waiting time
P1	13 – 0 = 13	13 – 5 = 8
P2	12 – 1 = 11	11 – 3 = 8
P3	5 – 2 = 3	3 – 1 = 2
P4	9 – 3 = 6	6 – 2 = 4
P5	14 – 4 = 10	10 – 3 = 7

Now,

Average Turn Around time = (13 + 11 + 3 + 6 + 10) / 5 = 43 / 5 = 8.6 unit

Average waiting time = (8 + 8 + 2 + 4 + 7) / 5 = 29 / 5 = 5.8 unit

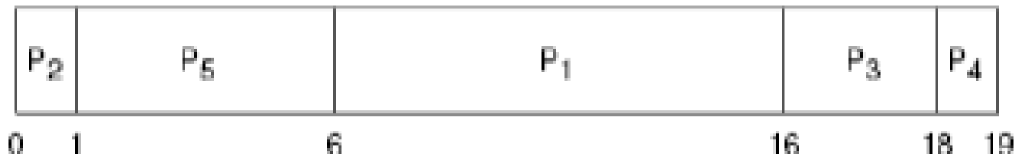
Priority Scheduling

- Jobs are run based on their priority, Always run the job with the highest priority
- Priorities can be **externally defined** (e.g., by user) or based on some **process-specific metrics** (e.g., their expected CPU burst)
- Can be preemptive
- Can be no preemptive
- Priorities can be **static**(i.e. they don’t change) or **dynamic** (they may change during execution)
- Problem =**Starvation** – low priority processes may never execute
- Solution = **Aging** – as time progresses increase the priority of the process

Example :1 If the CPU scheduling policy is priority non-preemptive, calculate the average waiting time and average turn around time. (lower number represents higher priority)

Process	Burst Time (ms)	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

- Priority scheduling Gantt Chart assuming all arrive at time 0



Turn Around time = Exit time/completion time – Arrival time

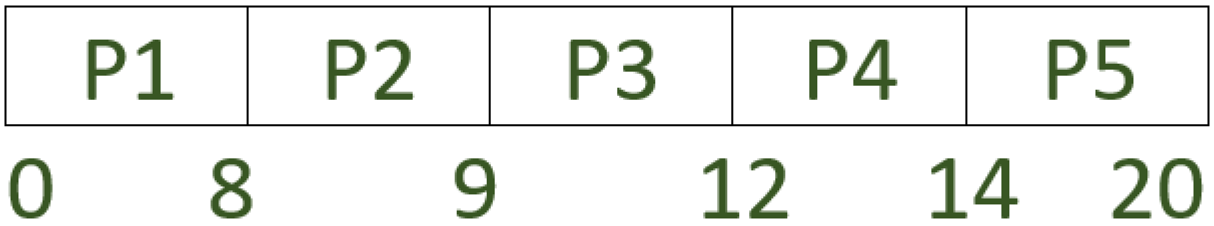
Waiting time = Turn Around time – Burst time/Execution time

Average waiting time = (0+1+6+16+18)/5 = 8.2 msec

Example: 2

PROCESS	ARRIVAL TIME	BURST TIME	PRIORITY
P1	0	8	3
P2	1	1	1
P3	2	3	2
P4	3	2	3
P5	4	6	4

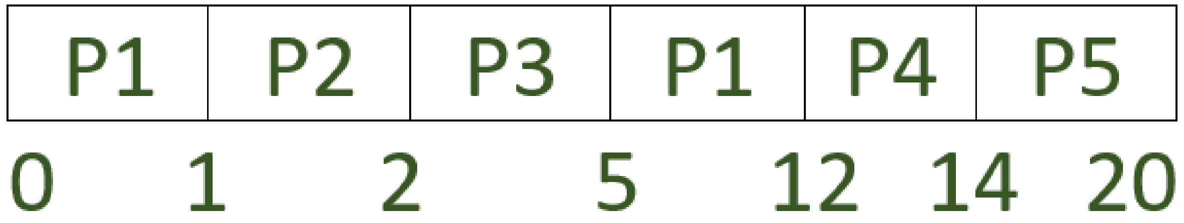
Priority Non-preemptive scheduling :



Average waiting time (AWT)= ((0-0) + (8-1) + (9-2) + (12-3) + (14-4)) / 5 = 33 / 5 = 6.6

Average turnaround time (TAT)= ((8-0) + (9-1) + (12-2) + (14-3) + (20-4)) / 5= 53 / 5 = 10.6

Priority Preemptive scheduling :



Average waiting time (AWT)= ((5-1) + (1-1) + (2-2) + (12-3) + (14-4)) / 5 = 23/5 = 4.6

Average turnaround time (TAT)= ((12-0) + (2-1) + (5-2) + (14-3) + (20-4)) / 5 = 43 / 5 = 8.5

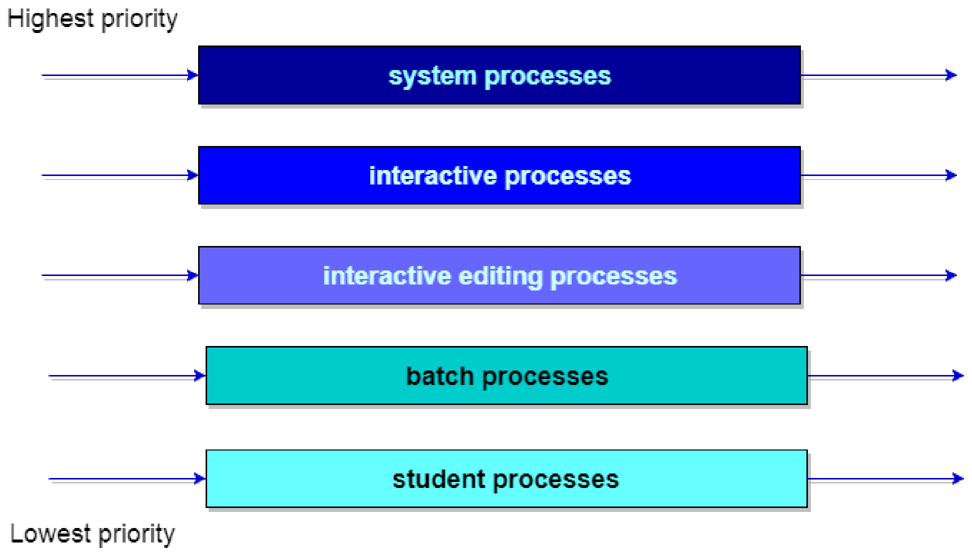
Multi-level queue

- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type

For Example:

- Separate queues might be used for foreground and background processes.
- The foreground queue might be scheduled by Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

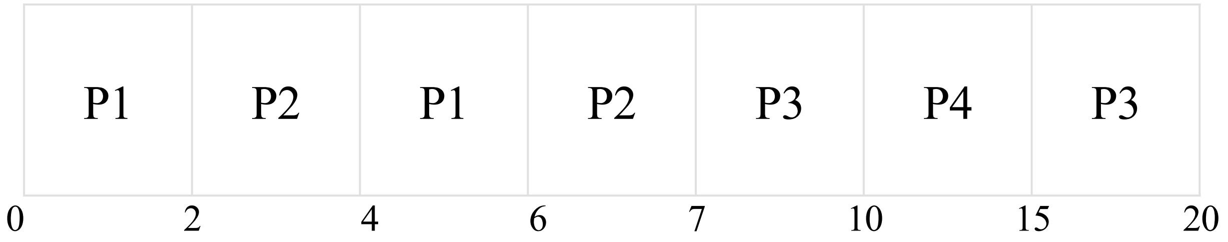
- Let us consider an example of a multilevel queue-scheduling algorithm with five queues:
  - System Processes
  - Interactive Processes
  - Interactive Editing Processes
  - Batch Processes
  - Student Processes



Example 1: Consider below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process. Priority of queue 1 is greater than queue 2. queue 1 uses Round Robin (Time Quantum = 2) and queue 2 uses FCFS.

Process	Burst time	Arrival time	Queue number
P1	4	0	1
P2	3	0	1
P3	8	0	2
P4	5	10	1

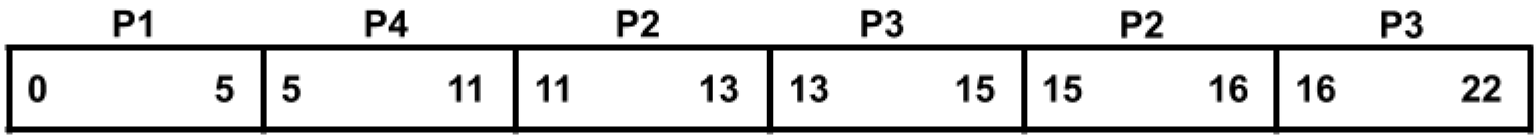
Solution: **Gantt chart** of the problem



Example 2: Consider below table of four processes under Multilevel queue scheduling. Queue number denotes the queue of the process. Queue1 is having higher priority and queue1 is using the FCFS approach and queue2 is using the round-robin approach(time quantum = 2ms).

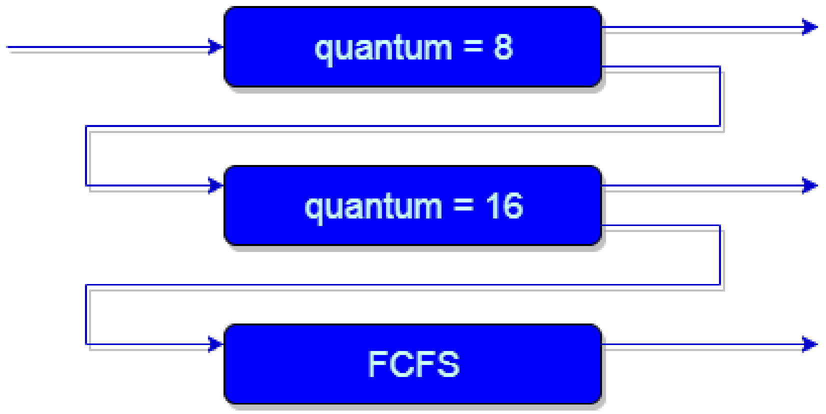
Process	Arrival time	Burst time	Queue
P1	0 ms	5 ms	1
P2	0 ms	3 ms	2
P3	0 ms	8 ms	2
P4	0 ms	6 ms	1

Gantt Chart



Multilevel Feedback Queue

- Multilevel feedback queue scheduling, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.



- Multilevel feedback queue scheduler is defined by the following parameters:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The method used to determine when to upgrade a process to a higher-priority queue.
  - The method used to determine when to demote a process to a lower-priority queue.
  - The method used to determine which queue a process will enter when that process needs service.

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.

Fair-Share Scheduling

- Users are assigned some fraction of the CPU
  - Scheduler takes into account who owns a process before scheduling it
- E.g., two users each with 50% CPU share
  - User 1 has 4 processes: A, B, C, D
  - User 2 has 2 processes: E, F
- If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

- On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Numerous other possibilities exist, of course, and can be exploited, depending on what the notion of fairness is.

Lottery Scheduling [Waldspurger and Weihl 1994]

- Jobs receive lottery tickets for various resources
    - E.g., CPU time
- At each scheduling decision, one ticket is chosen at random and the job holding that ticket wins
- If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20% chance of winning each lottery. In the long run, it will get about 20% of the CPU.
- Lottery scheduling can be used to solve problems that are difficult to handle with other methods.
- One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates.
- Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

Scheduling in Real-Time Systems

- A **real-time scheduling System** is composed of the **scheduler**, clock and the processing hardware elements.
- In a **real-time system**, a process or task has Schedulability ; tasks are accepted by a **real-time system** and completed as specified by the task deadline depending on the characteristic of the **scheduling** algorithm.
- Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met—or else!— and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable.
- The events that a real-time system may have to respond to can be further categorized as **periodic** (meaning they occur at regular intervals) or **aperiodic** (meaning they occur unpredictably).

- For example, if there are *m* periodic events and event *i* occurs with period *Pi* and requires *Ci* sec of CPU time to handle each event, then the load can be handled only if A real-time system that meets this criterion is said to be **schedulable**.

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A process that fails to meet this test cannot be scheduled because the total amount of CPU time the processes want collectively is more than the CPU can deliver.