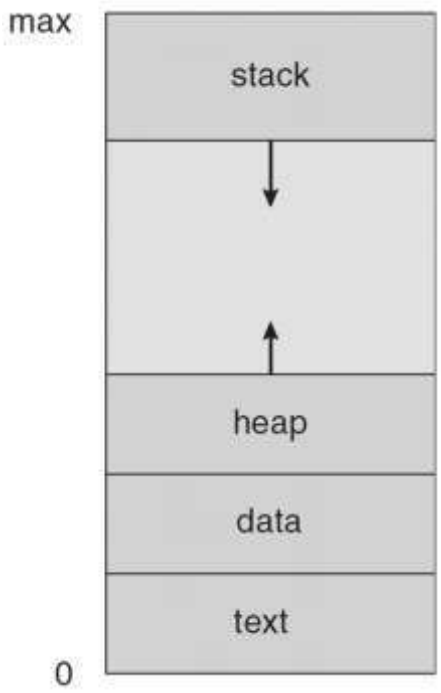
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

Process Management

- An operating system executes a variety of programs:
    - Batch system – jobs
    - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
    - program counter
    - stack
    - data section



Process State

As a process executes, it changes *state*

- new: The process is being created
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- ready: The process is waiting to be assigned to a processor
- terminated: The process has finished execution

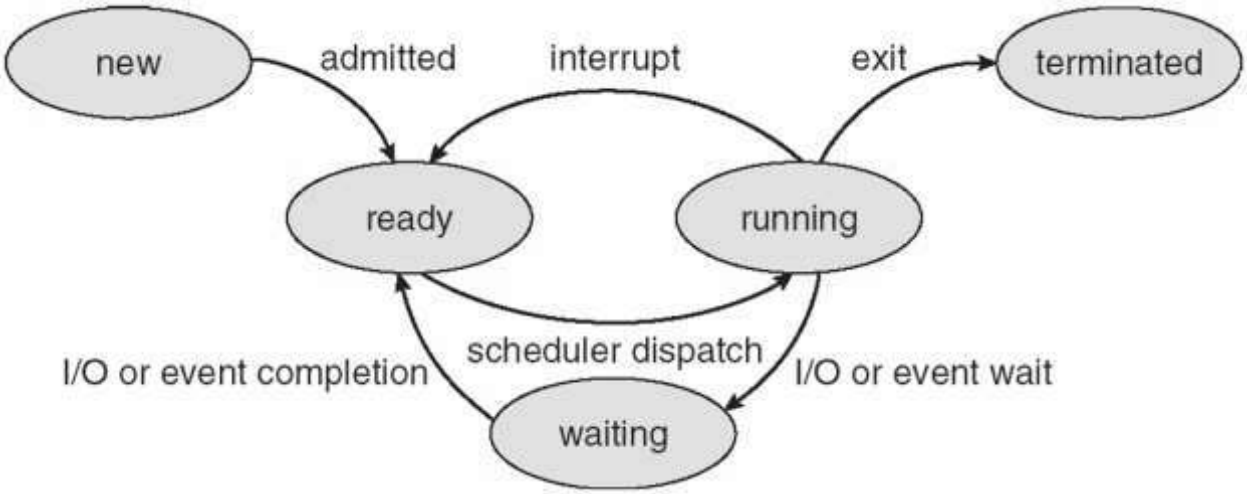


Fig: Process Transition Diagram

PCB: Process Control Block

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

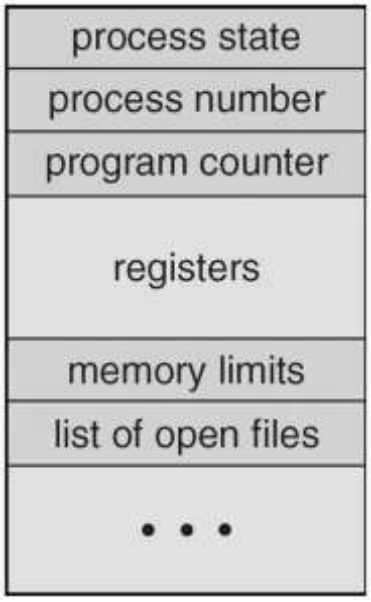


Fig: PCB

Context Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Process Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory, ready and waiting to execute
- Device queues – set of processes waiting for an I/O device
- Processes migrate among the various queues

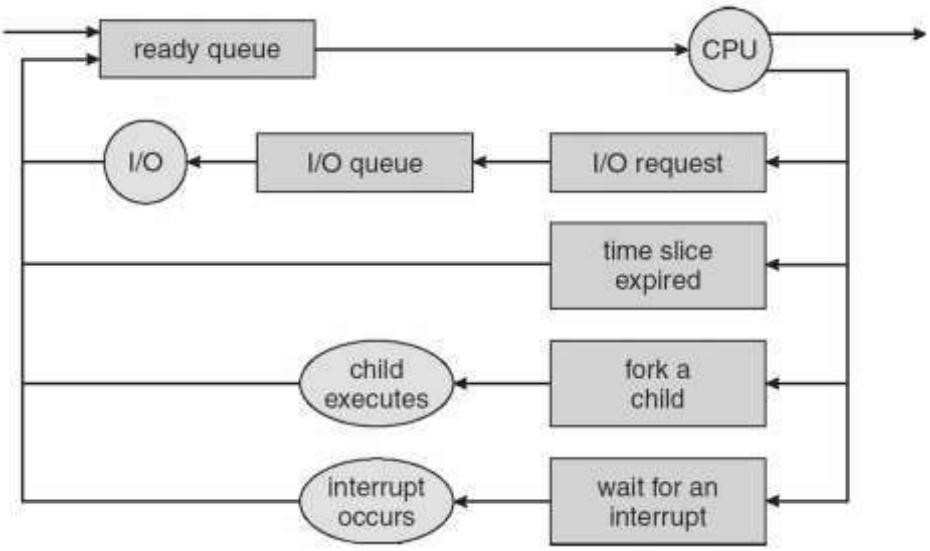
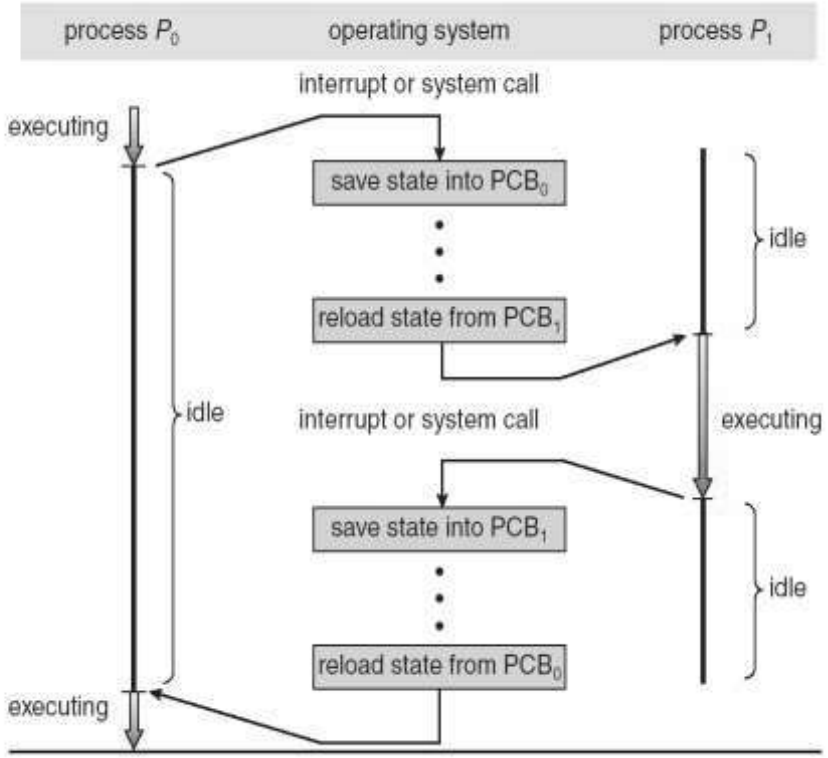


Fig: Process Scheduling



Schedulers

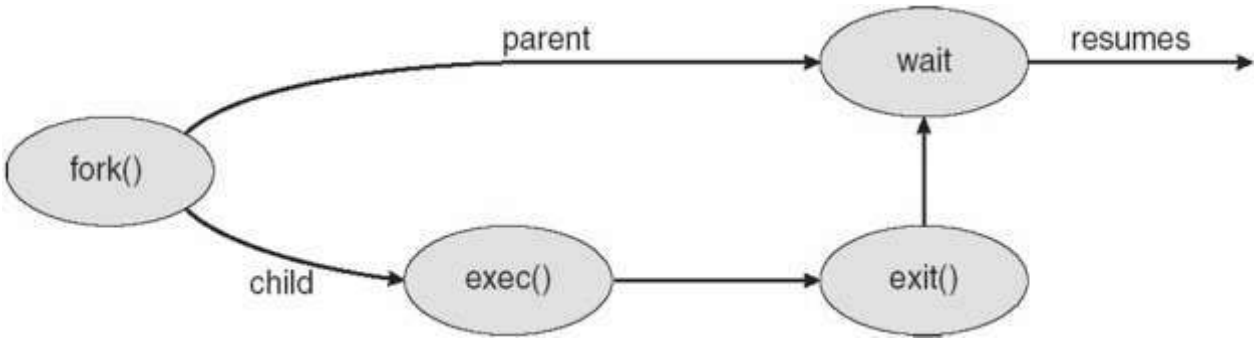
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
    - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
    - CPU-bound process – spends more time doing computations; few very long CPU bursts

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- Resource sharing
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Parent and child share no resources
- Execution
    - Parent and children execute concurrently
    - Parent waits until children terminate
- Address space
    - Child duplicate of parent
    - Child has a program loaded into it
- UNIX examples
    - **fork** system call creates new process

- **exec** system call used after a **fork** to replace the process' memory space with a new program



Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination

Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity

- Convenience

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

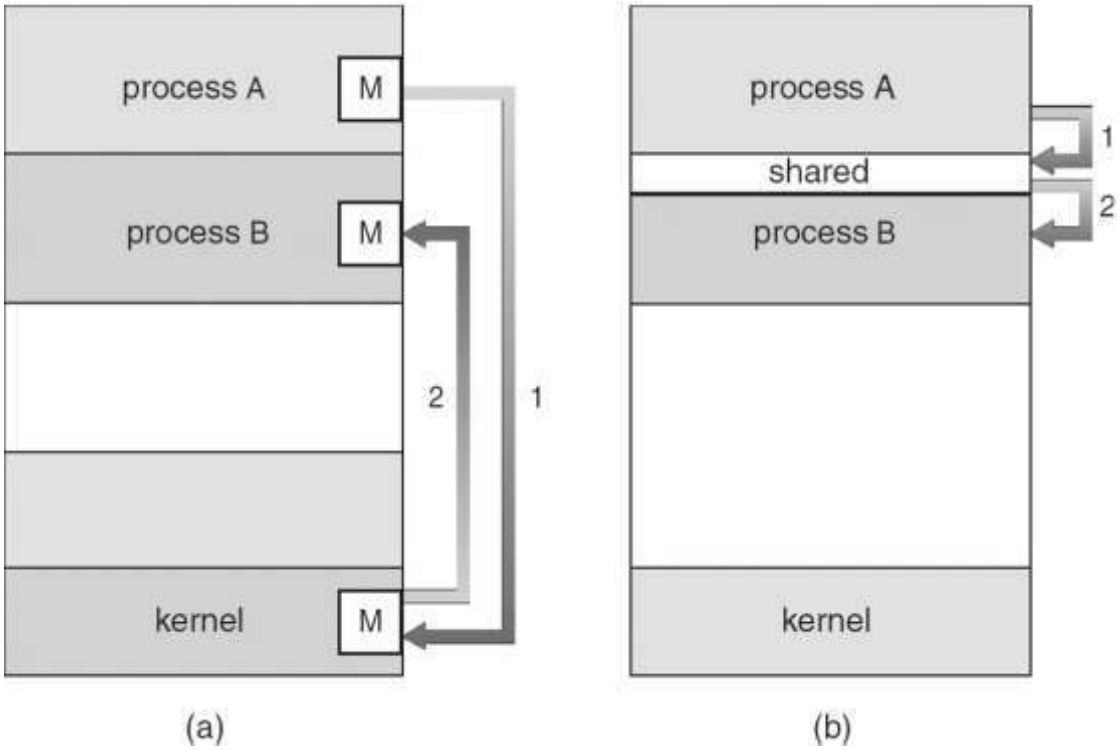


Fig:a- Message Passing, b- Shared Memory

Cooperating Process

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity

- Convenience

Producer Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

```
        while (true) {
/* Produce an item */

   while (((in = (in + 1) % BUFFER SIZE count)  == out)

          ;   /* do nothing -- no free buffers */

          buffer[in] = item;

          in = (in + 1) % BUFFER SIZE;

 }
```

Fig: Producer Process

```
                  while (true) {

      while (in == out)

          ; // do nothing -- nothing to consume

            // remove an item from the buffer

            item = buffer[out];

            out = (out + 1) % BUFFER SIZE;
```

IPC-Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:

    - send(*message*) – message size fixed or variable
    - receive(*message*)
- If *P* and *Q* wish to communicate, they need to:
    - establish a *communication link* between them
    - exchange messages via send/receive
- Implementation of communication link
    - physical (e.g., shared memory, hardware bus)
    - logical (e.g., logical properties)

Direct Communication

- Processes must name each other explicitly:
    - **send** (*P, message*) – send a message to process P
    - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
- The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
    - Each mailbox has a unique id
    - Processes can communicate only if they share a mailbox
- Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links

- Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A, message*) – receive a message from mailbox A
- Allow a link to be associated with at most two processes
- Allow only one process at a time to execute a receive operation
- Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronisation

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full

3. Unbounded capacity – infinite length
Sender never waits

Thread

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack.
- A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

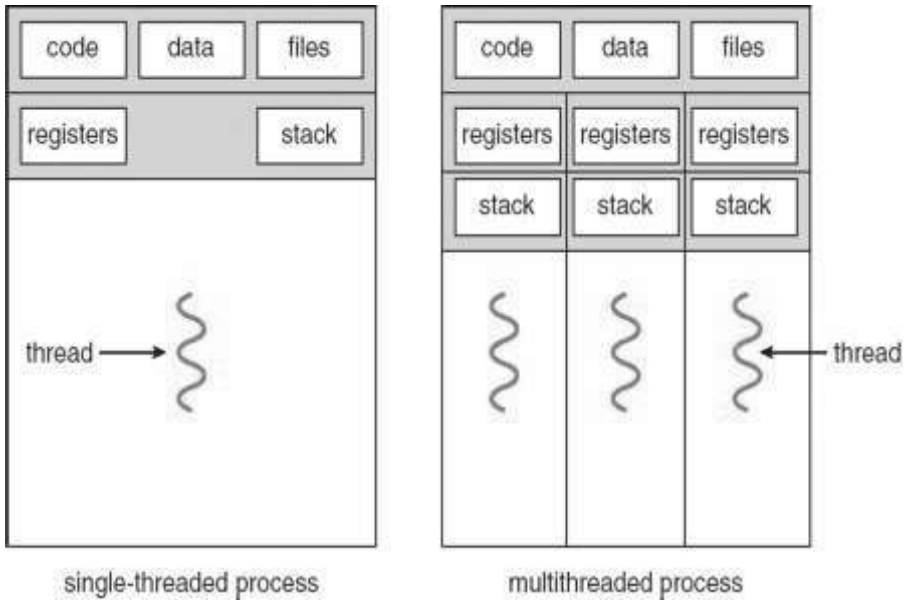


Fig: Single threaded vs multithreaded process

Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

User Threads

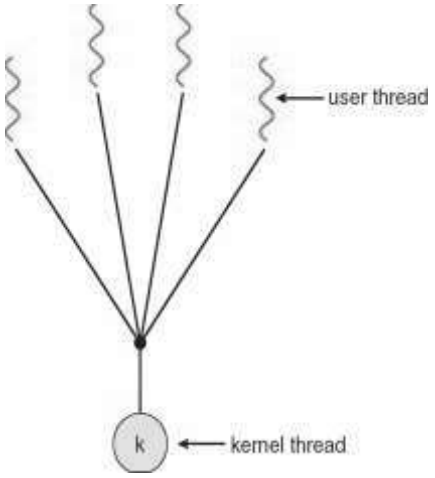
- Thread management done by user-level threads library
- Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Kernel Thread

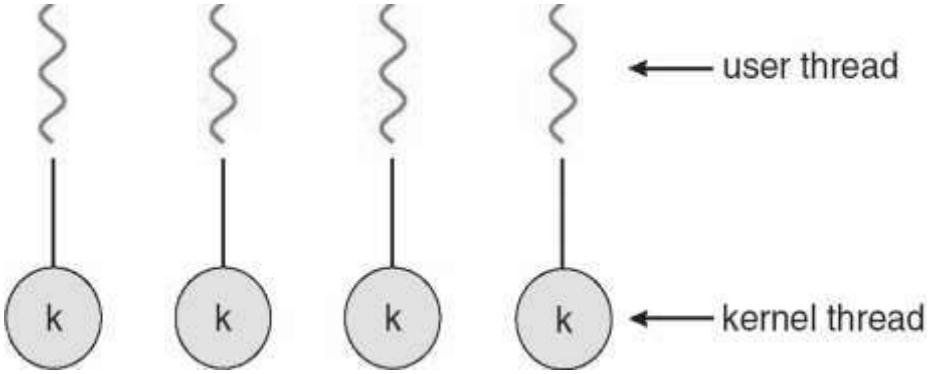
- Supported by the Kernel
- Examples
    - Windows XP/2000
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X

Multithreading Models

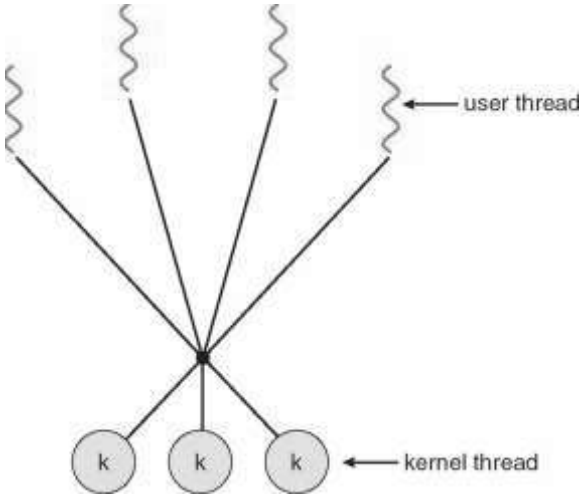
- Many-to-One
    - Many user-level threads mapped to single kernel thread
    - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



- One-to-One
    - Each user-level thread maps to kernel thread
    - Examples
    - Windows NT/XP/2000
    - Linux
    - Solaris 9 and later



- Many-to-Many
    - Allows many user level threads to be mapped to many kernel threads
    - Allows the operating system to create a sufficient number of kernel threads
    - Solaris prior to version 9
    - Windows NT/2000 with the *ThreadFiber* package



Thread Library

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
    - Library entirely in user space
    - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
    - Extending Thread class
    - Implementing the Runnable interface

Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation of target thread
    - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
    - Asynchronous cancellation terminates the target thread immediately
    - Deferred cancellation allows the target thread to periodically check if it should be cancelled

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
    - Usually slightly faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
    - Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS) –** competition among all threads in system

Difference between Process and Thread

Process	**Thread**
Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments 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.
If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Process Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution

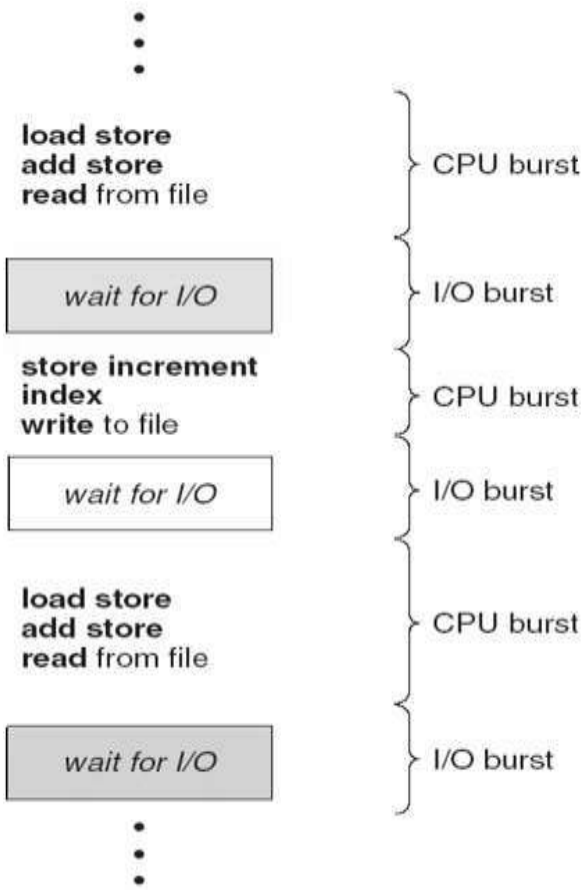


Fig: CPU burst and I/O burst

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
    1. Switches from running to waiting state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
    - switching context
    - switching to user mode
    - jumping to the proper location in the user program to restart that program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running

CPU Scheduling Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

CPU Scheduling Algorithms

A. First Come First Serve Scheduling

- Schedule the task first which arrives first
- Non preemptive In nature

B. Shortest Job First Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
    - The difficulty is knowing the length of the next CPU request

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

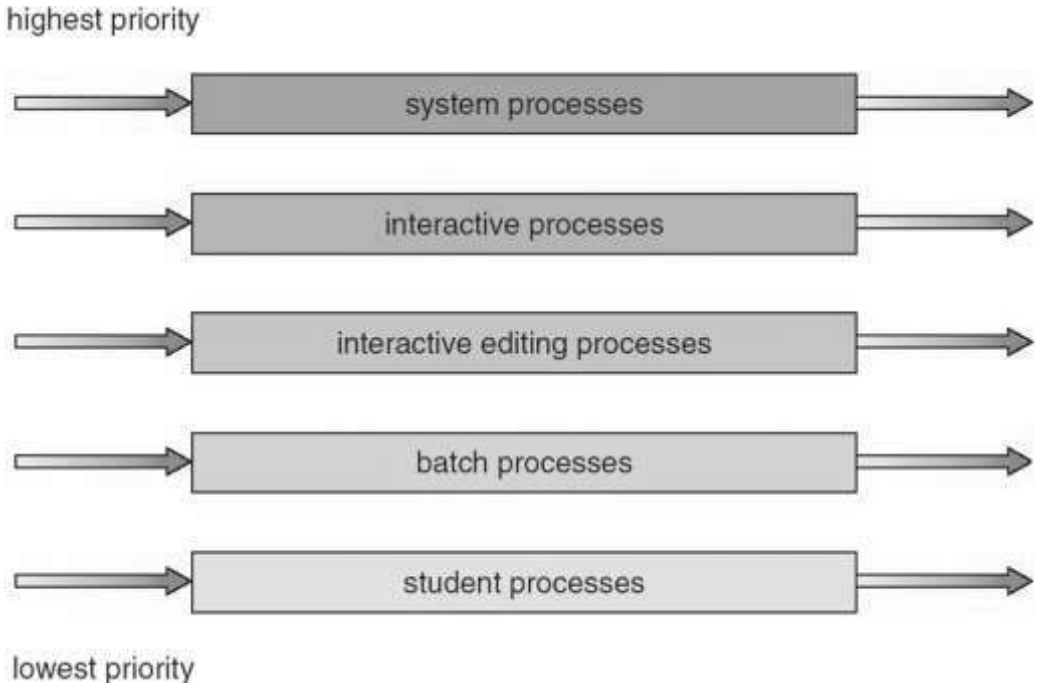
Round Robin Scheduling

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
    - q large $\Rightarrow$ FIFO
    - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Multilevel Queue Scheduling

- 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 Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
    - number of queues
    - scheduling algorithms for each queue
    - method used to determine when to upgrade a process
    - method used to determine when to demote a process
    - method used to determine which queue a process will enter when that process needs service

MODULE-II

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Pseudocode for Producer Process	Pseudocode for Consumer Process

while (true) {
/* produce an item and put in nextProduced */
while (count == BUFFER_SIZE)
; // do nothing
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
count++;
	while (true) {
	while (count == 0)
	; // do nothing
	nextConsumed = buffer[out];
	out = (out + 1) % BUFFER_SIZE;
	count--;
	/* consume the item in nextConsumed
	}

data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

- count++ could be implemented as

register1 = count

register1 = register1 + 1
count = register1

- count-- could be implemented as

register2 = count
register2 = register2 - 1
count = register2

- Consider this execution interleaving with “count = 5” initially:

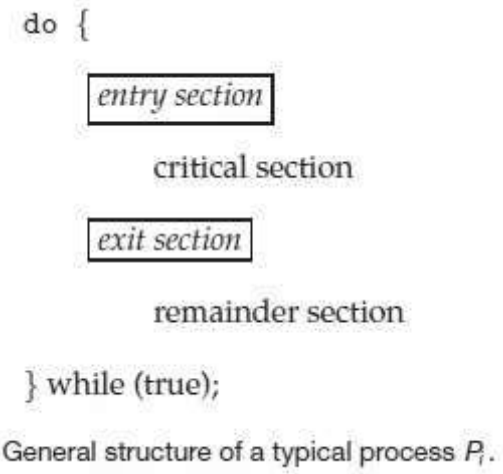
S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

Critical Section Problem

A section of code, common to n cooperating processes, in which the processes may be accessing common variables.

A Critical Section Environment contains:

- Entry Section Code requesting entry into the critical section.
- Critical Section Code in which only one process can execute at any one time.
- Exit Section The end of the critical section, releasing or allowing others in.
- Remainder Section Rest of the code AFTER the critical se



General structure of a typical process P_i.

- Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this
- request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

Solution to Critical Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the N processes

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
    - int turn;
    - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Hardware Synchronization

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
    - Currently running code would execute without preemption
    - Generally too inefficient on multiprocessor systems
        - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
        - Atomic = non-interruptable

- Either test memory word and set value
- Or swap contents of two memory words

Solution to Critical Section Problem using Lock

```
do {
        acquire lock
            critical section
        release lock
            remainder section
    } while (TRUE);
```

TestAndndSet Instruction

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {
while ( TestAndSet (&lock ))
        ;   // do nothing
    //    critical section
lock = FALSE;
        //      remainder section
} while (TRUE);
```

Sawp Instruction

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section
        lock = FALSE;
                //      remainder section
} while (TRUE);
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
```

```
            key = TestAndSet(&lock);
        waiting[i] = FALSE;
            // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
            j = (j + 1) % n;
        if (j == i)
            lock = FALSE;
        else
            waiting[j] = FALSE;
            // remainder section
} while (TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
    - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
        while S <= 0
            ; // no-op
        S--;
    }
```

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

- 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
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
        wait (mutex);
      // Critical Section
    signal (mutex);
                // remainder section
} while (TRUE);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
    - Could now have busy waiting in critical section implementation
        - But implementation code is short
        - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.
- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list
- Two operations:

- block – place the process invoking the operation on the appropriate waiting queue.
- wakeup – remove one of processes in the waiting queue and place it in the ready queue.

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

The pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- The structure of the producer process

```
do  {
        //   produce an item in nextp
    wait (empty);
    wait (mutex);
        //  add the item to the  buffer
     signal (mutex);
     signal (full);
} while (TRUE);
```

- The structure of the consumer process

```
do {
     wait (full);
     wait (mutex);
          //  remove an item from  buffer to nextc
     signal (mutex);
     signal (empty);
```

```
        //  consume the item in nextc
    } while (TRUE);
```

Readers-Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as ***readers*** *and to the latter as* ***writers****. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously,*
chaos may ensue.

- A data set is shared among a number of concurrent processes
    - Readers – only read the data set; they do **not** perform any updates
    - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
    - Data set
    - Semaphore mutex initialized to 1
    - Semaphore wrt initialized to 1
    - Integer readcount initialized to 0
- The structure of a writer process

```
do {
  wait (wrt) ;
```

```
            //    writing is performed
            signal (wrt) ;
        } while (TRUE);
```

- The structure of a reader process

```
do {
            wait (mutex) ;
            readcount ++ ;
            if (readcount == 1)
                        wait (wrt) ;
            signal (mutex)

            // reading is performed
            wait (mutex) ;
            readcount  - - ;
            if (readcount  == 0)
                        signal (wrt) ;
            signal (mutex) ;
        } while (TRUE);
```

Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a

chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
- The structure of Philosopher *i*:

```
do  {
        wait ( chopstick[i] );
            wait ( chopStick[ (i + 1) % 5] );

                //  eat
            signal ( chopstick[i] );
            signal (chopstick[ (i + 1) % 5] );

             //  think
} while (TRUE);
```

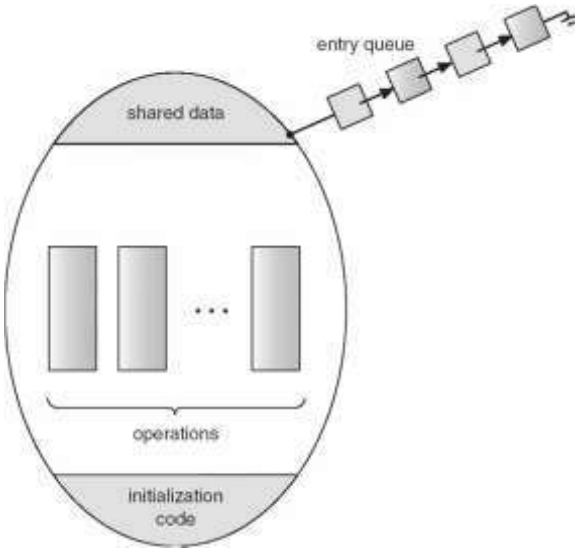
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
                …
        }
```

}

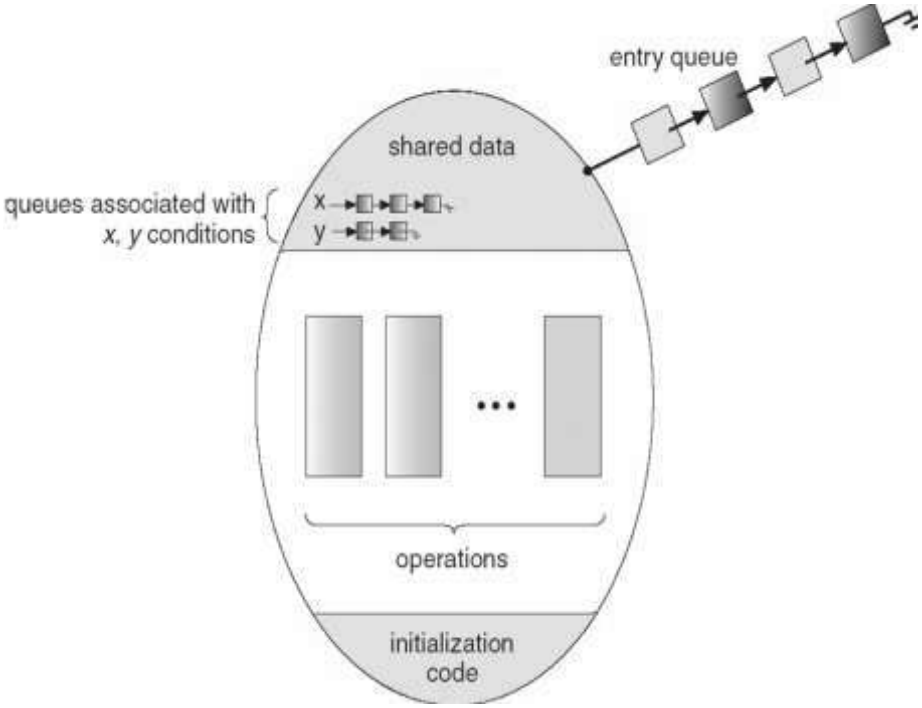
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
    - x.wait () – a process that invokes the operation is suspended.
        - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
            body of F;
    …
if (next_count > 0)
        signal(next)
else
        signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x-count--;
```

The operation x.signal can be implemented as:

```
if (x-count > 0) {
```

```
            next_count++;
            signal(x_sem);
            wait(next);
            next_count--;
        }
```

Deadlock

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
    - System has 2 disk drives
    - P_1 and P_2 each hold one disk drive and each needs another one
- Example
    - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

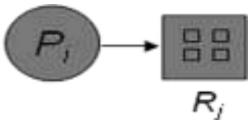
System Model

- Resource types R_1, R_2, . . ., R_m *(CPU cycles, memory space, I/O devices)*
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**

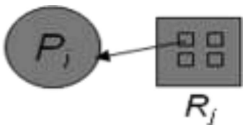
Deadlock Characterization

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j



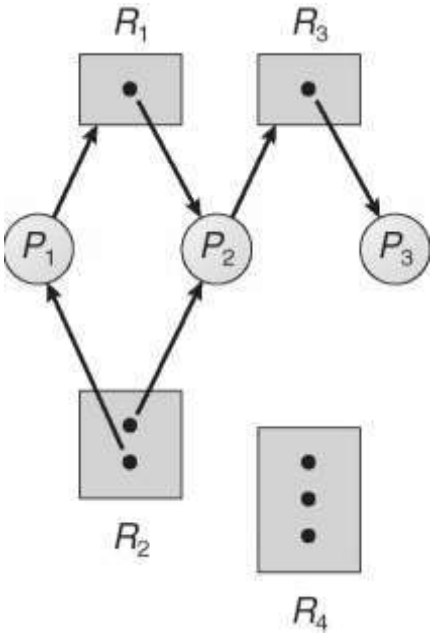


Fig: RAG

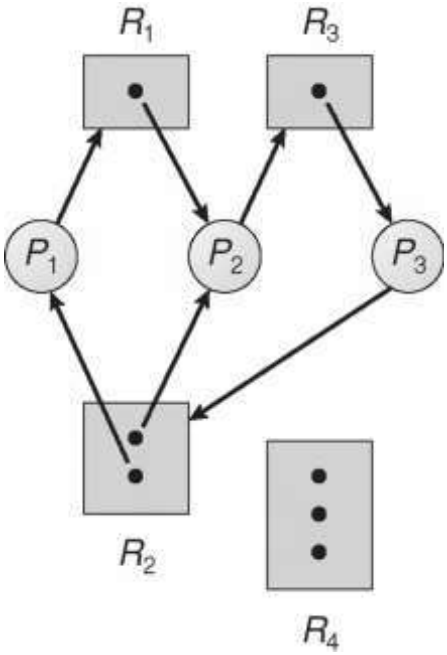


Fig: RAG with a deadlock

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock
    - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlock

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- **Mutual Exclusion –** not required for sharable resources; must hold for nonsharable resources

- **Hold and Wait –** must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
    - Low resource utilization; starvation possible
- **No Preemption –**
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
    - Preempted resources are added to the list of resources for which the process is waiting
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait –** impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

- Requires that the system has some additional *a priori* information available
- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe state

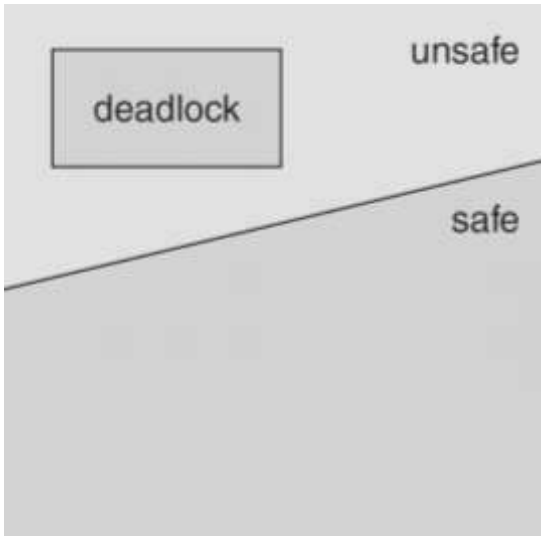
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still

request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

- That is:
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Facts

- **If a system is in safe state ⇒ no deadlocks**
- **If a system is in unsafe state ⇒ possibility of deadlock**
- **Avoidance ⇒ ensure that a system will never enter an unsafe state.**

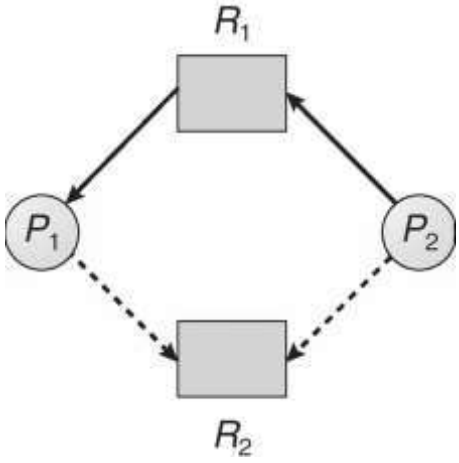


Deadlock Avoidance Algorithm

- Single instance of a resource type
    - Use a resource-allocation graph
- Multiple instances of a resource type
    - Use the banker's algorithm

RAG Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



Banker's Algorithm

Assumptions

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structure for Bankers' Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max***: n x m* matrix. If *Max* [*i,j*] = k, then process P_i may request at most k instances of resource type R_j