

UNIT -2

PROCESS MANAGEMENT

Process Concept

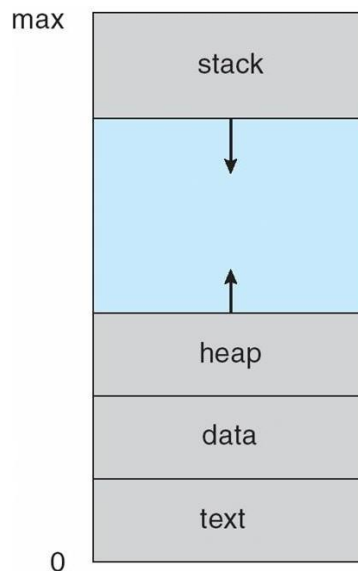
- An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

A process includes:

- program counter
- stack
- data section

Process in Memory

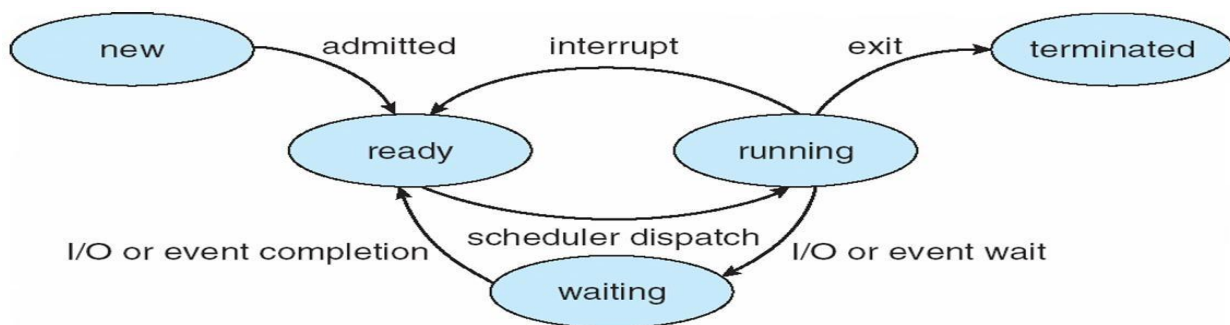


Process State

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

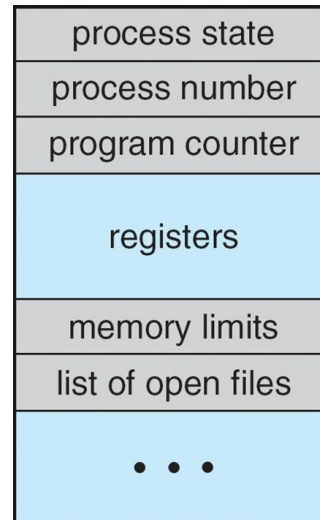
Diagram of Process State



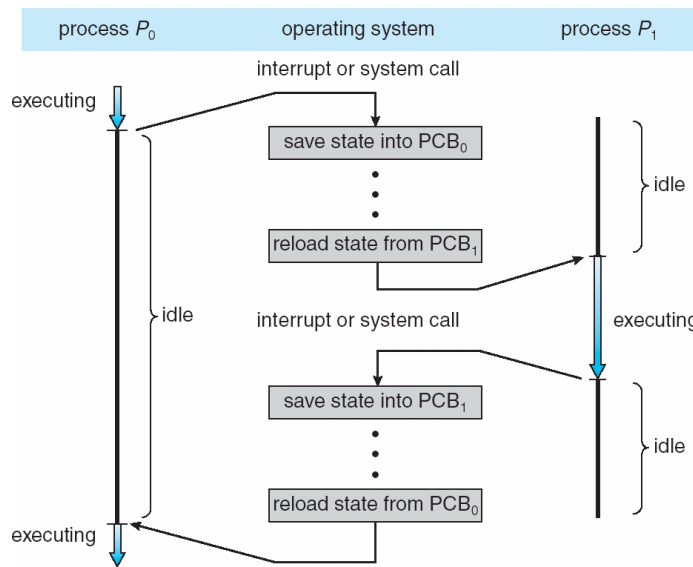
Process Control Block (PCB)

Information associated with each process

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



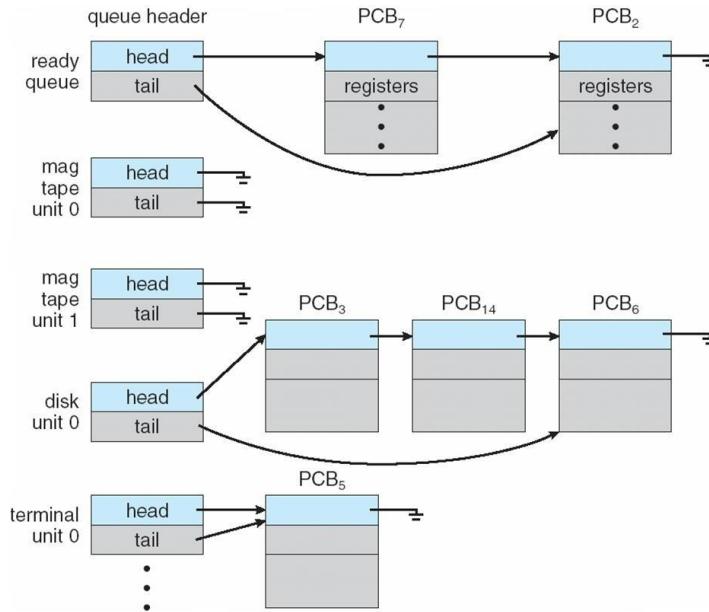
CPU Switch From Process to Process



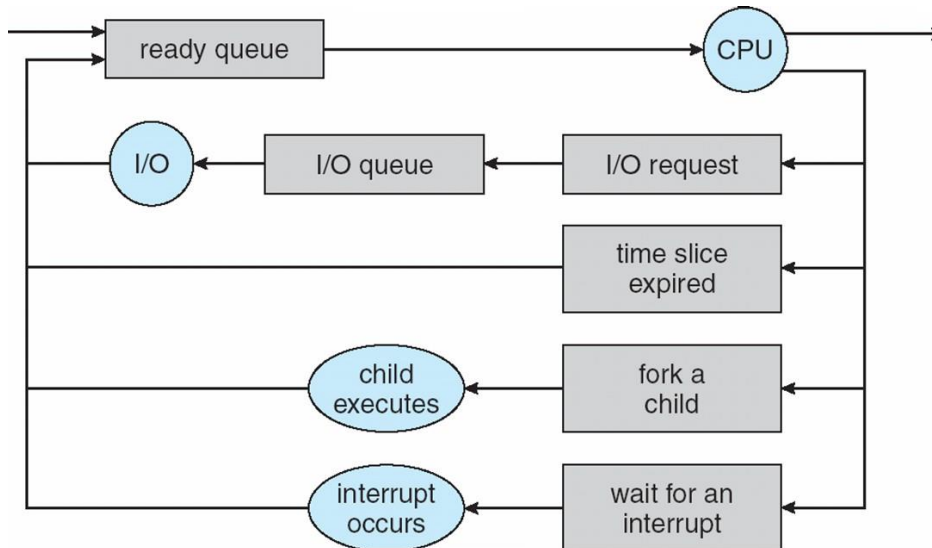
Process Scheduling Queues

- **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

Ready Queue and Various I/O Device Queues



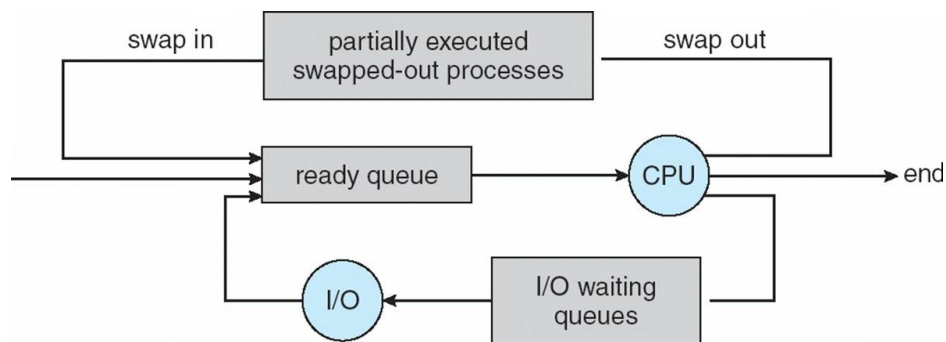
Representation of Process Scheduling



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

Addition of Medium Term Scheduling



- Short-term scheduler is invoked very frequently (milliseconds) \mathcal{P} (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) \mathcal{P} (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

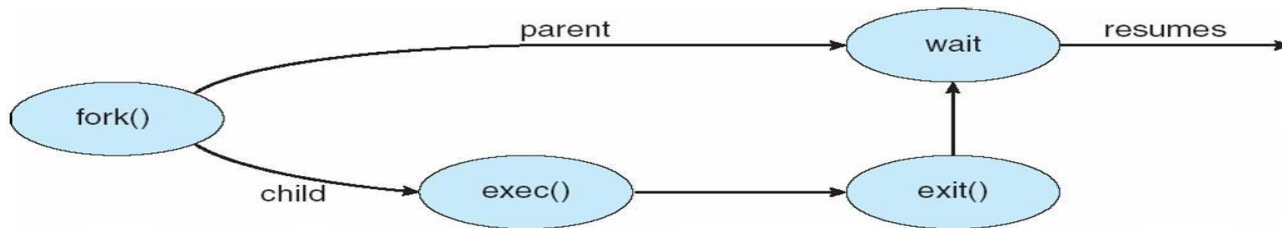
Context Switch

- 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 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

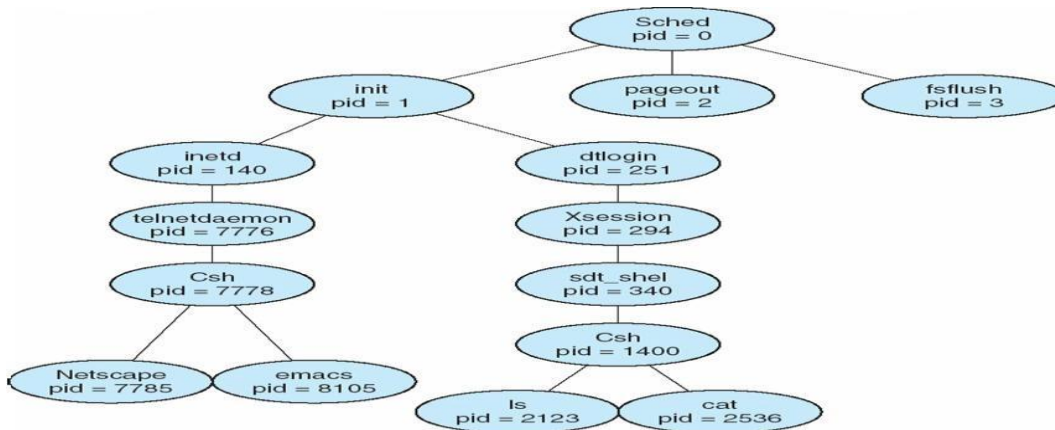
Process Creation



C Program Forking Separate Process

```
int main()
{
pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```

A tree of processes on a typical Solaris



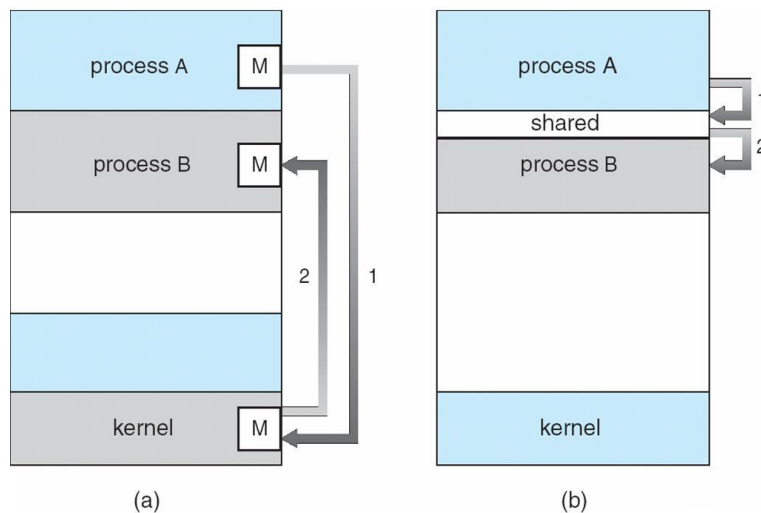
Process Termination

- 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**

Interprocess 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

Communications Models



Cooperating Processes

- **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

Bounded-Buffer – Shared-Memory Solution

Shared data

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    ...
```

```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
while (true) {
```

```
/* Produce an item */
```

```
while (((in = (in + 1) % BUFFER_SIZE) == out)
```

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

```
    buffer[in] = item;
```

```
    in = (in + 1) % BUFFER_SIZE;
```

```
}
```

Bounded Buffer – Consumer

```
while (true) {
```

```
    while (in == out)
```

```
        ; // do nothing -- nothing to consume
```

```
        // remove an item from the buffer
```

```
        item = buffer[out];
```

```
        out = (out + 1) % BUFFER_SIZE;
```

```
        return item;
```

```
}
```

Interprocess Communication – 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
- Mailbox sharing
- $P_1, P_2,$ and P_3 share mailbox A
- $P_1,$ sends; P_2 and P_3 receive
- Who gets the message?
- Solutions
- 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.

Synchronization

- 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

Examples of IPC Systems - POSIX

- POSIX Shared Memory
- Process first creates shared memory segment
- `segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- Process wanting access to that shared memory must attach to it
- `shared memory = (char *) shmat(id, NULL, 0);`
- Now the process could write to the shared memory
- `printf(shared memory, "Writing to shared memory");`
- When done a process can detach the shared memory from its address space
- `shmdt(shared memory);`

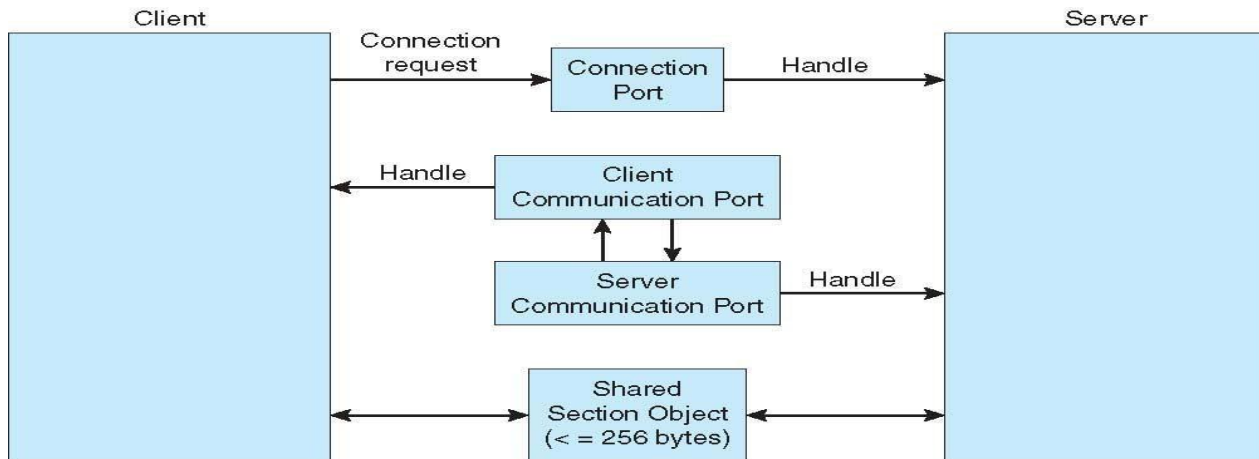
Examples of IPC Systems - Mach

- Mach communication is message based
- Even system calls are messages
- Each task gets two mailboxes at creation- Kernel and Notify
- Only three system calls needed for message transfer
- `msg_send()`, `msg_receive()`, `msg_rpc()`
- Mailboxes needed for communication, created via
- `port_allocate()`

Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
- Only works between processes on the same system
- Uses ports (like mailboxes) to establish and maintain communication channels
- Communication works as follows:
 - The client opens a handle to the subsystem's connection port object
 - The client sends a connection request
 - The server creates two private communication ports and returns the handle to one of them to the client
 - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

Local Procedure Calls in Windows XP



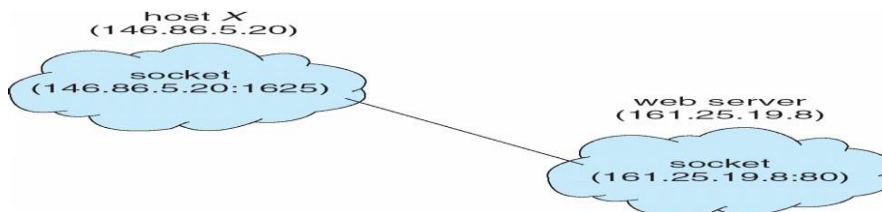
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

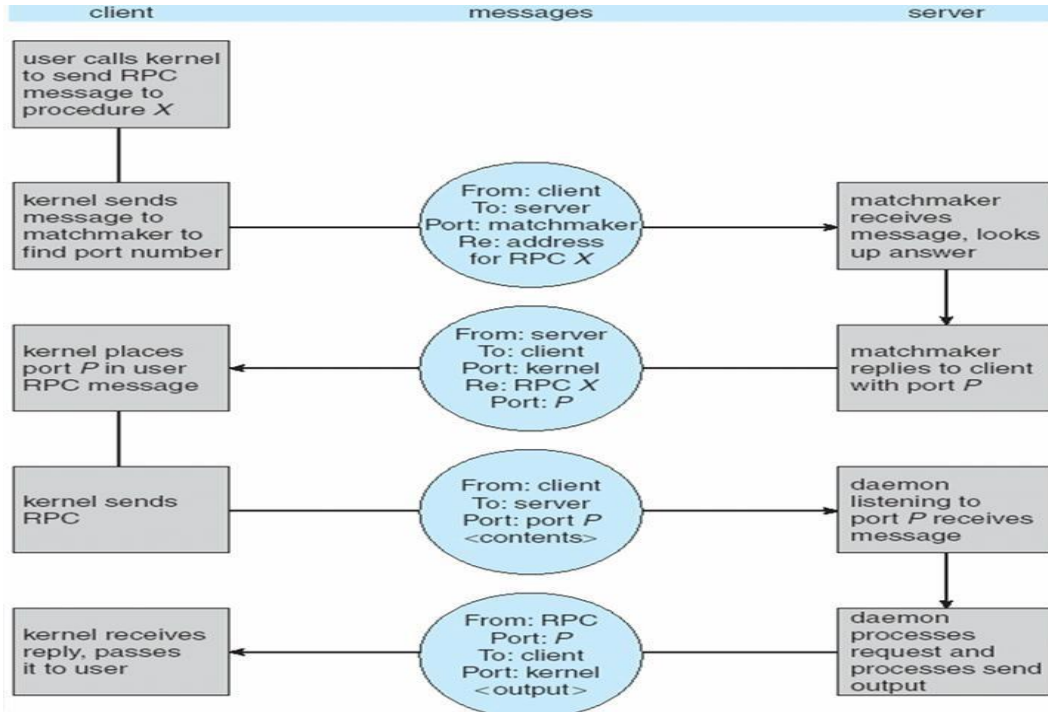
Socket Communication



Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

Execution of RPC

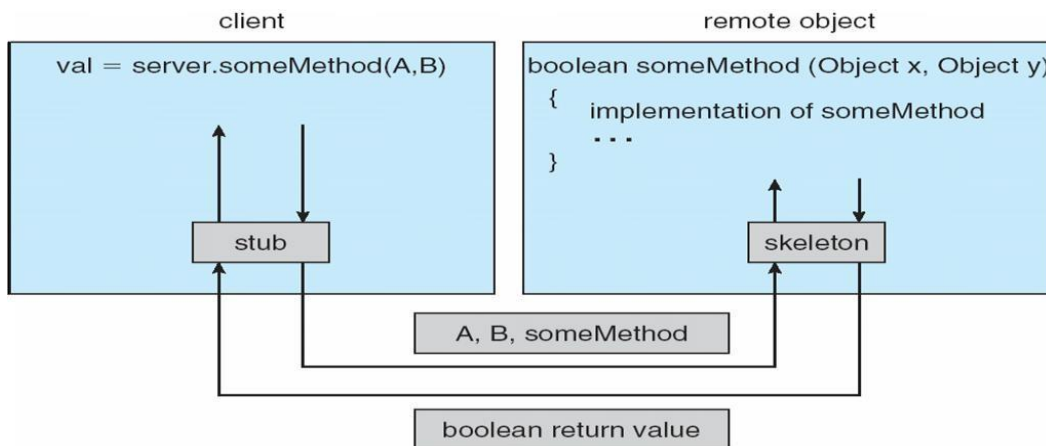


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object



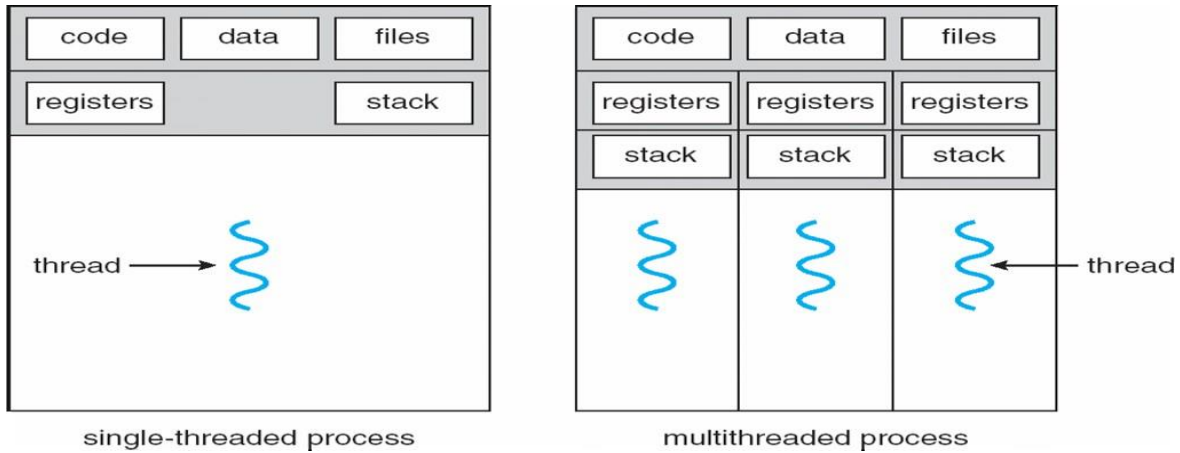
Marshalling Parameter



Threads

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Single and Multithreaded Processes



Benefits

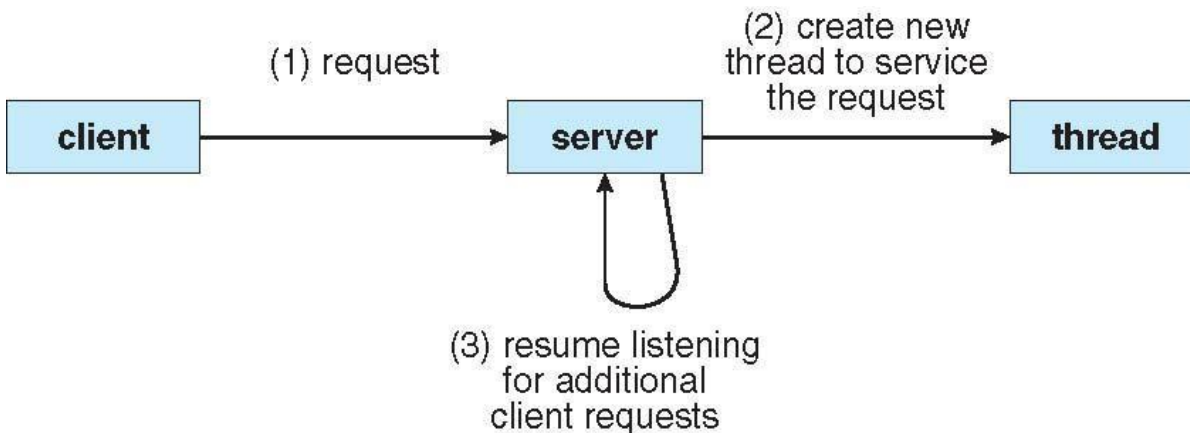
- Responsiveness
- Resource Sharing
- Economy
- Scalability

Multicore Programming

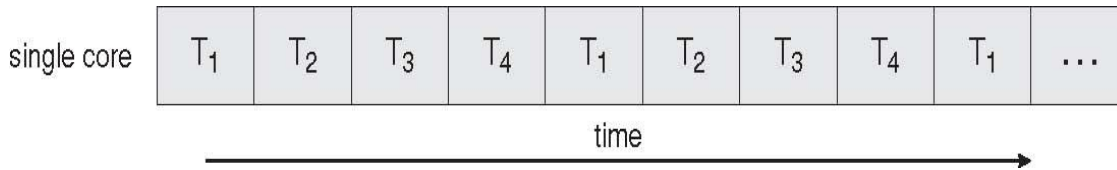
Multicore systems putting pressure on programmers, challenges include

- **Dividing activities**
- **Balance**
- **Data splitting**
- **Data dependency**
- **Testing and debugging**

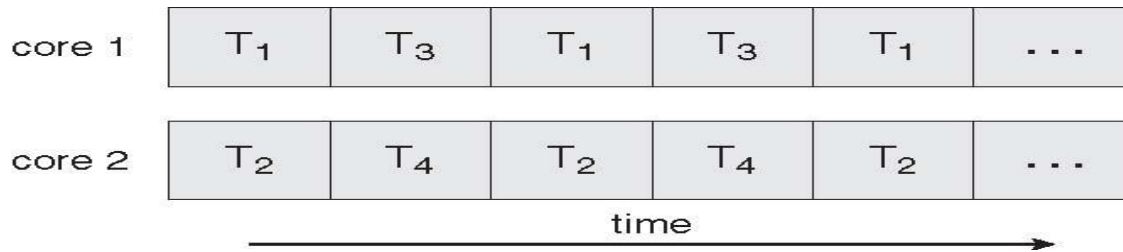
Multithreaded Server Architecture



Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



User Threads

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

Kernel Threads

Supported by the Kernel

Examples

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

Many user-level threads mapped to single kernel thread

Examples:

- Solaris Green Threads
- GNU Portable Threads

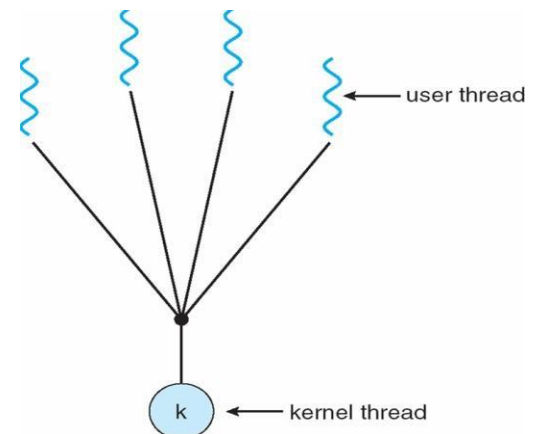
One-to-One

Each user-level thread maps to kernel thread

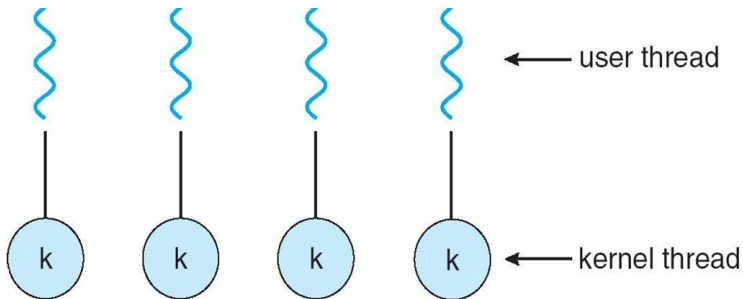
Examples

Windows NT/XP/2000

Linux



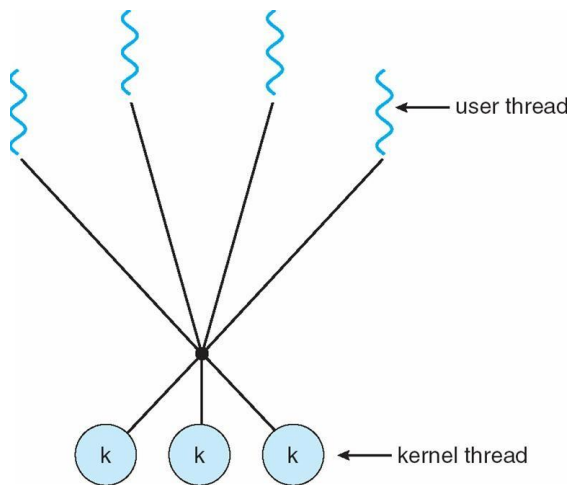
Solaris 9 and later



Many-to-Many Model

- 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

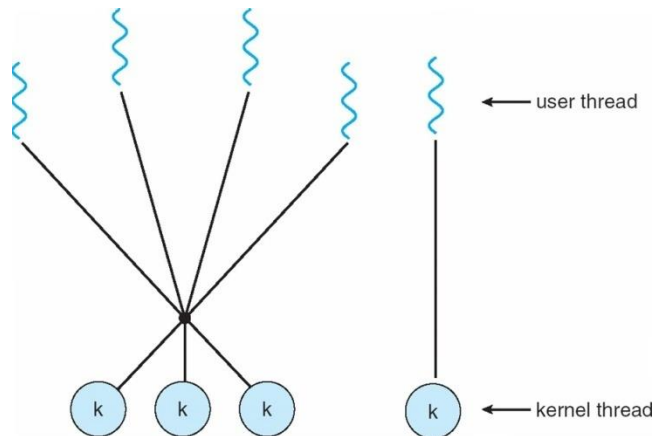


Two-level Model

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



Thread Libraries

- 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

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
- 1. Signal is generated by particular event
- 2. Signal is delivered to a process
- 3. Signal is handled
- Options:
- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

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

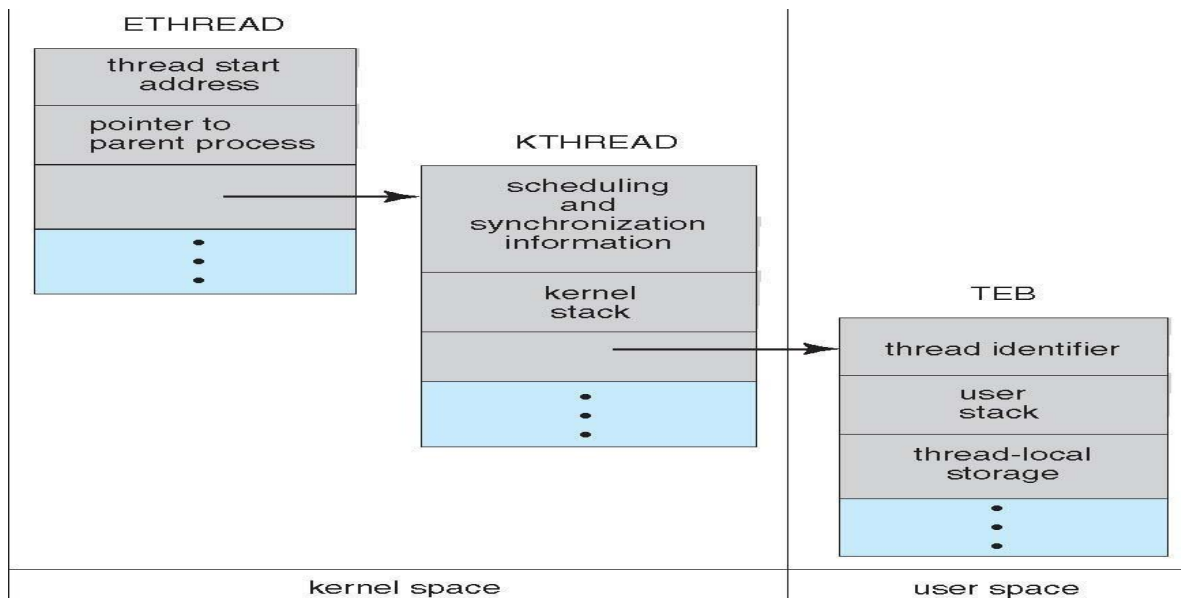
Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Windows XP Threads



Implements the one-to-one mapping, kernel-level

- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
 - ETHREAD (executive thread block)
 - KTHREAD (kernel thread block)
 - TEB (thread environment block)

Linux Threads

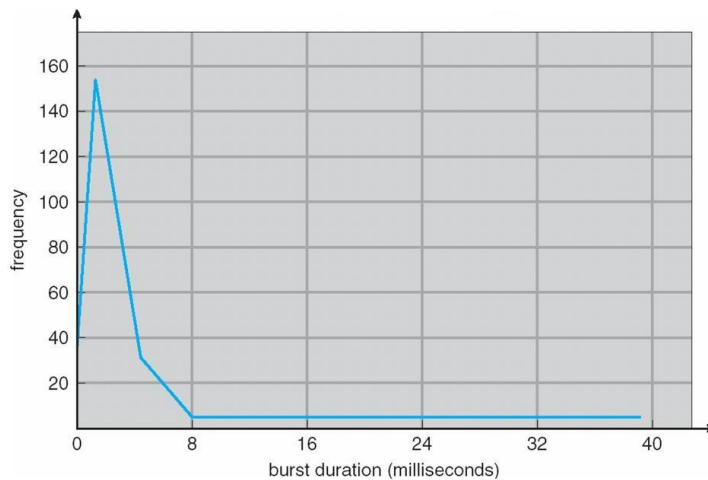
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

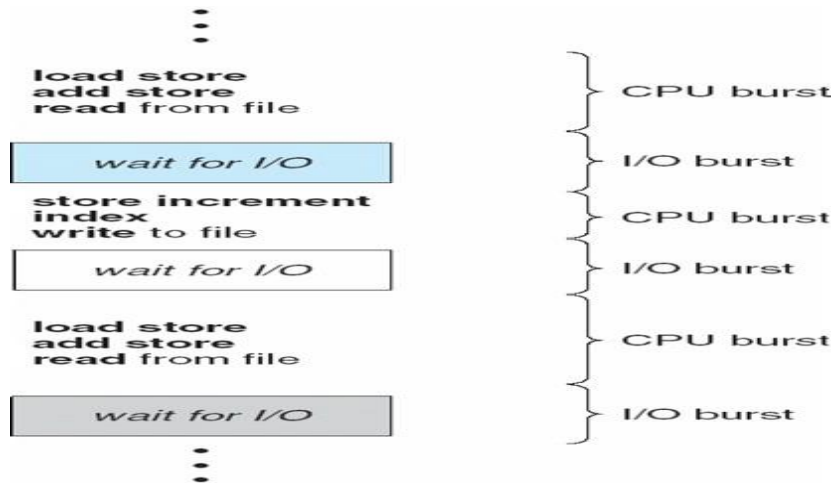
CPU Scheduling

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- 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

Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



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

Scheduling Criteria

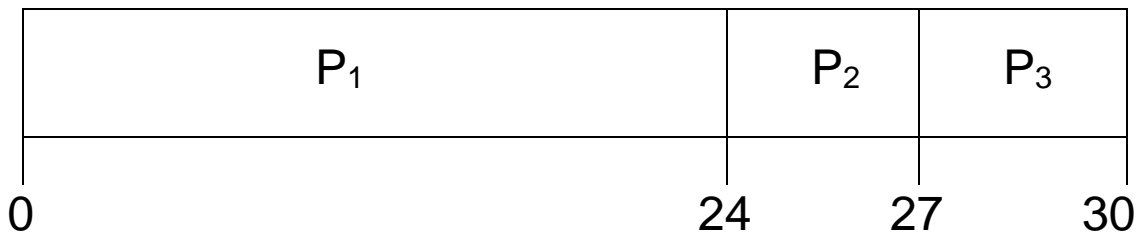
- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1, P_2, P_3

The Gantt Chart for the schedule is:



Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

Average waiting time: $(0 + 24 + 27)/3 = 17$

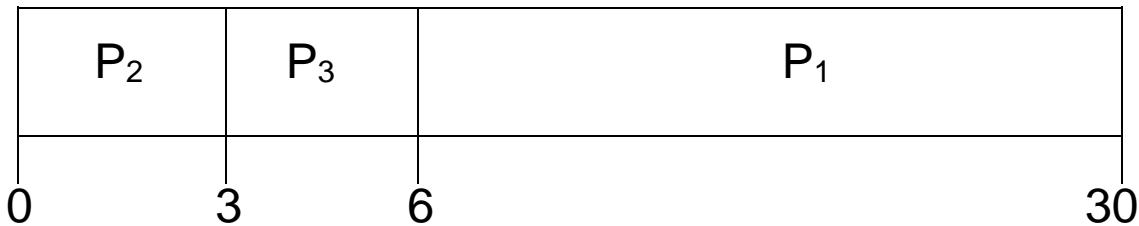
Suppose that the processes arrive in the order

P_2, P_3, P_1

The Gantt chart for the schedule is: P_2, P_3, P_1
 Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
 Average waiting time: $(6 + 0 + 3)/3 = 3$

Much better than previous case

Convoy effect short process behind long process



Shortest-Job-First (SJF) Scheduling

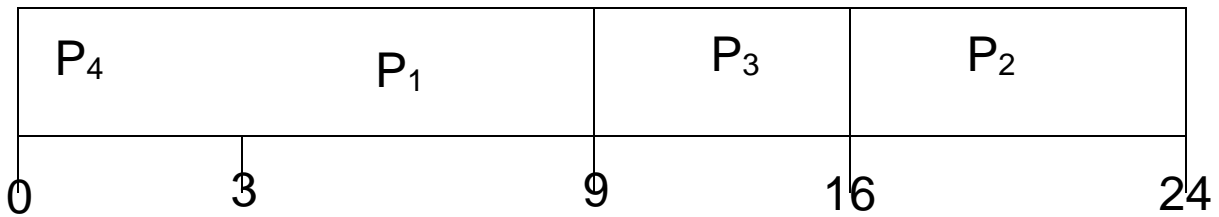
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

The difficulty is knowing

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

SJF scheduling chart

average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$ the length of the next CPU request

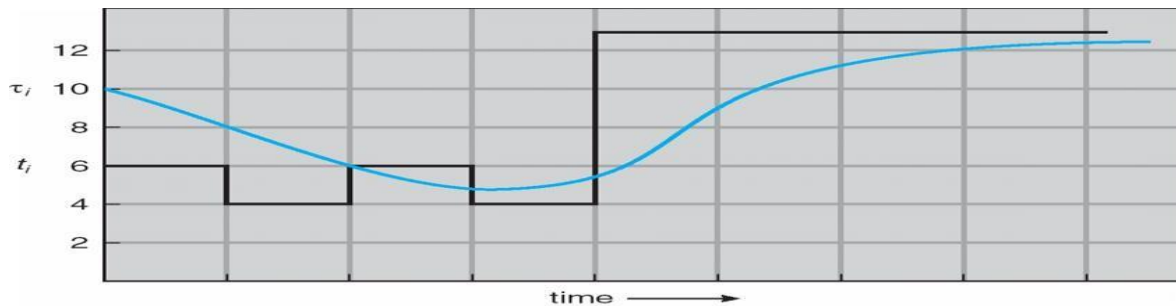


Determining Length of Next CPU Burst

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define :

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

$\alpha = 0$

$t_{n+1} = t_n$

Recent history does not count

$\alpha = 1$

$t_{n+1} = \alpha t_n$

Only the actual last CPU burst counts

If we expand the formula, we get:

$t_{n+1} = \alpha t_n + (1 - \alpha) t_{n-1} + \dots$

$+ (1 - \alpha)^j t_{n-j} + \dots$

$$+(1 - a)^{n+1} t_0$$

Since both a and $(1 - a)$ are less than or equal to 1, each successive term has less weight than its predecessor

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

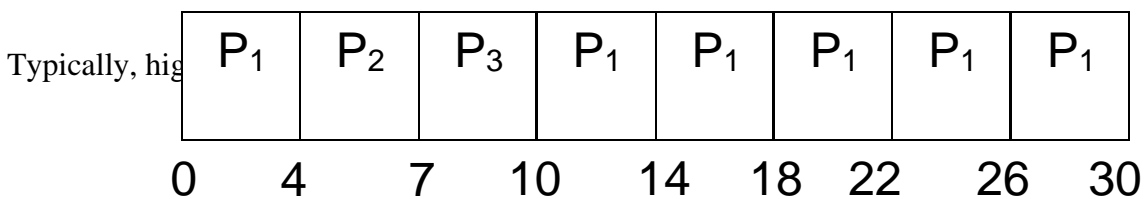
Round Robin (RR)

- 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 P FIFO
- q small P q must be large with respect to context switch, otherwise overhead is too high

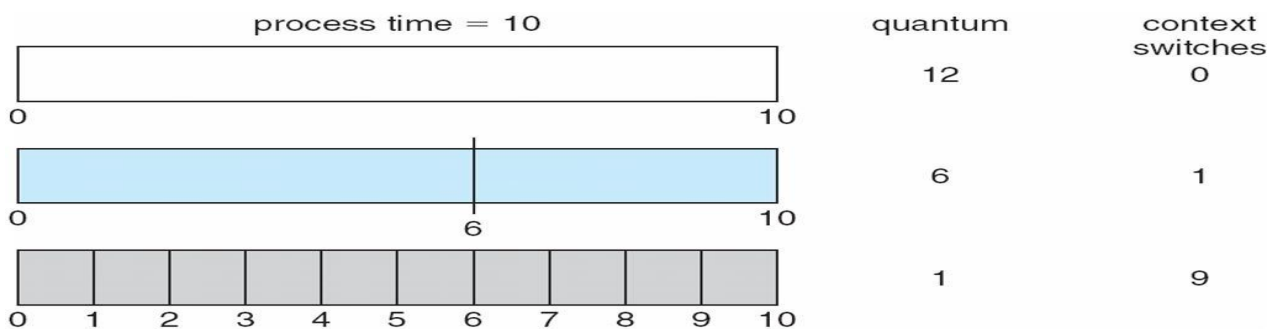
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

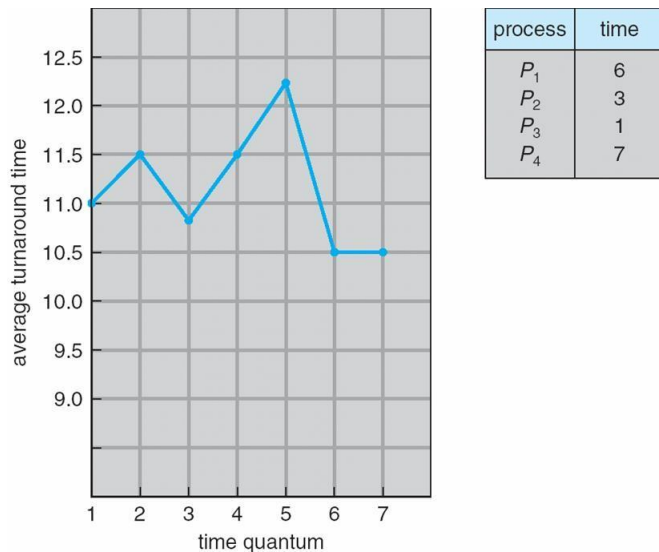
The Gantt chart is:



Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

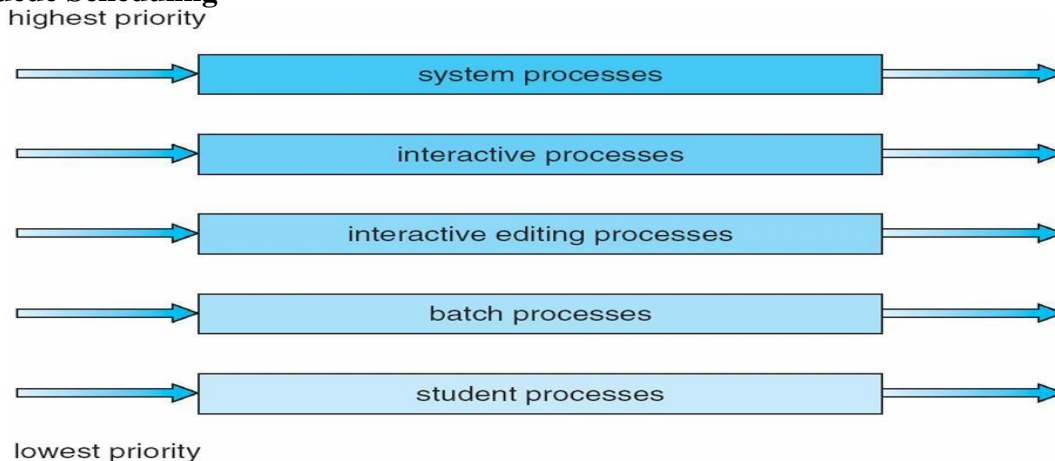


Multilevel Queue

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

20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

- 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

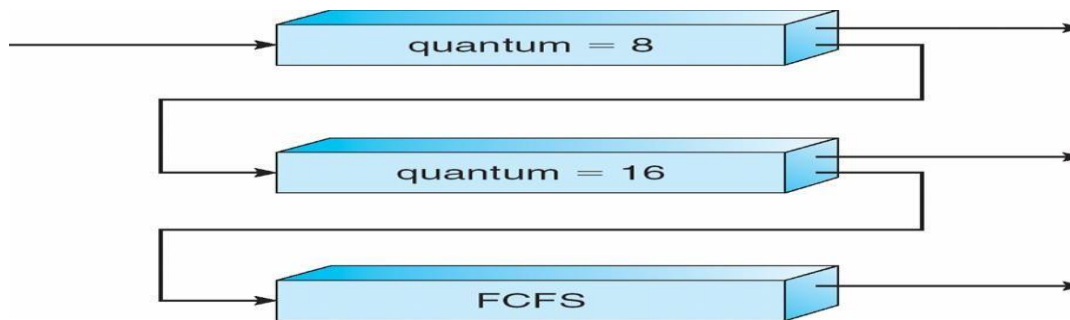
method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS
- Scheduling
- A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
- At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

Multilevel Feedback Queues



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

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
```

```

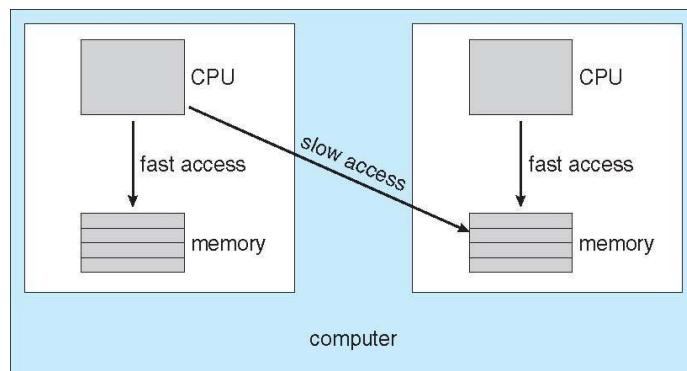
{
    int i; pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_t init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_t scope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_t schedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread_exit(0);
}

```

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

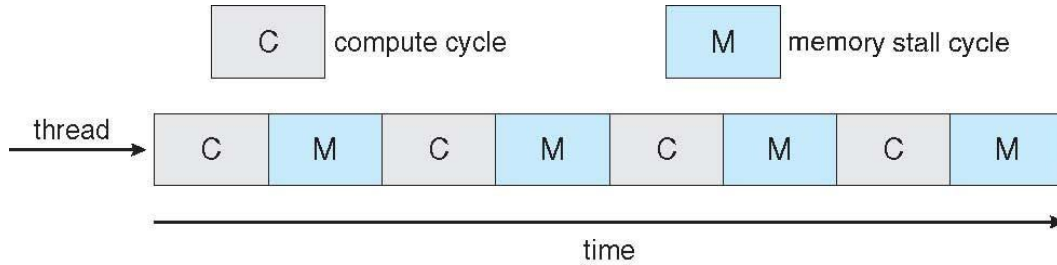
NUMA and CPU Scheduling



Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System



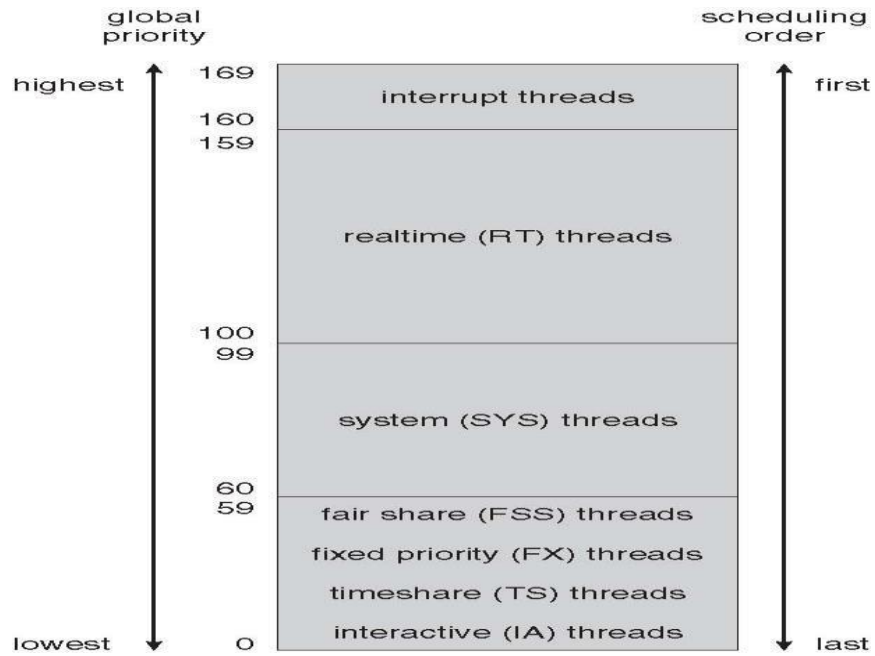
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling



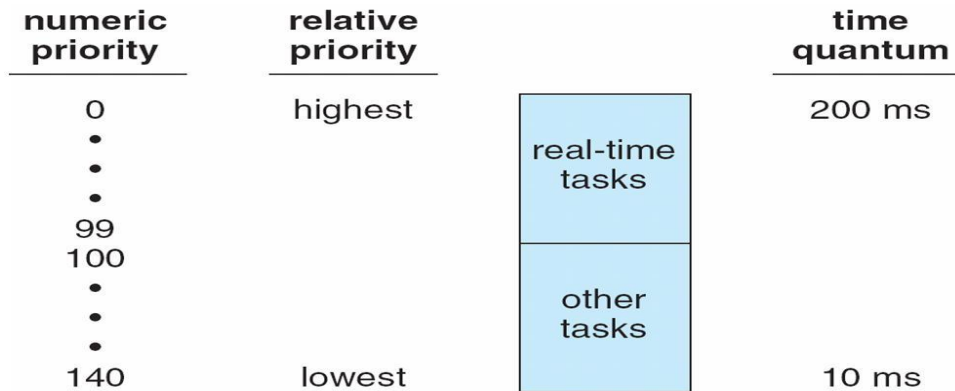
Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

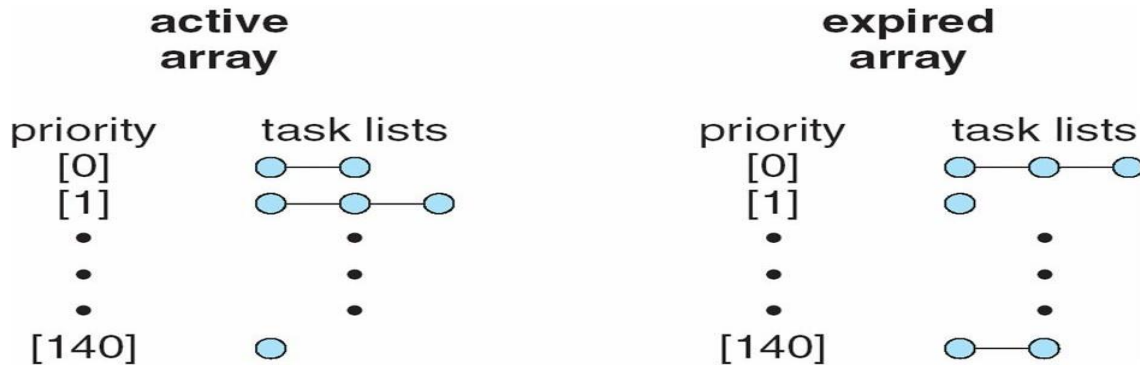
Linux Scheduling

- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140

Priorities and Time-slice length



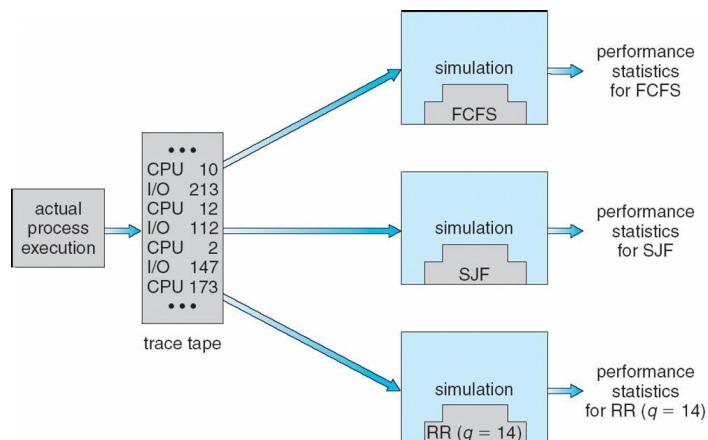
List of Tasks Indexed According to Priorities



Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queuing models
- Implementation

Evaluation of CPU schedulers by Simulation



CONCURRENCY

Process Synchronization

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity
- 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

Producer

```
while (true) {  
  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
  
}
```

Consumer

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

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}

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!

Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- 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 Locks

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

TestAndSet Instruction

Definition:

```
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);
```

Swap Instruction

Definition:

```
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++;
}

```

Semaphore as General Synchronization Tool

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

```

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

```

// 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 critical 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.

Semaphore Implementation with no Busy waiting

- 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);
    }
}
```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1


```

P0
wait (S);
wait (Q);
.
.
.
signal (S);
signal (Q);

P1
wait (Q);
wait (S);
.
.
.
signal (Q);
signal (S);

```

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- 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

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);
```

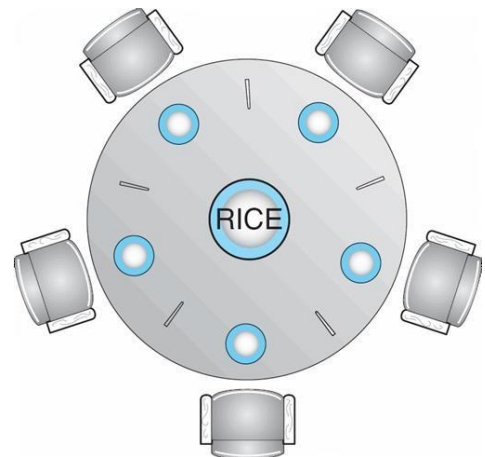
Dining-Philosophers Problem

- 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);
```



Problems with Semaphores

Incorrect use of semaphore operations:

! signal (mutex)

....

wait (mutex)

wait (mutex) ...

wait (mutex)

Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

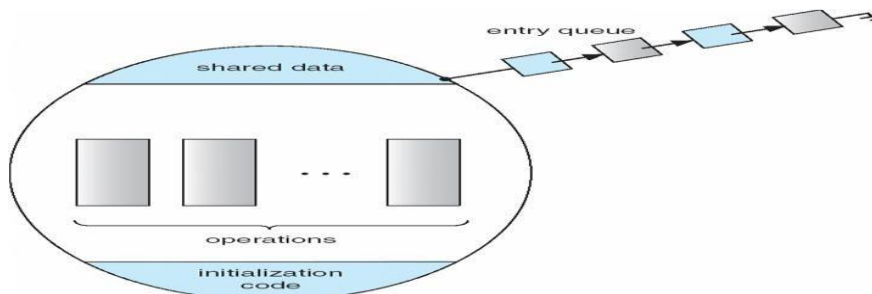
A high-level abstraction that provides a convenient and effective mechanism for process synchronization

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

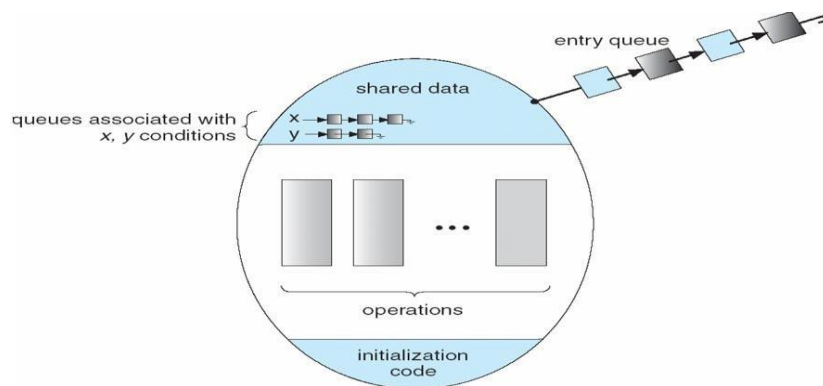
condition x, y;

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



Solution to Dining Philosophers

monitor DP

```
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];
    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ( ( state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }
    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Each philosopher I invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);
    EAT
DiningPhilosophers.putdown (i);
```

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--;
}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
```

```

        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
}
initialization code() {
    busy = FALSE;
}
}

```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- **Implements a variety of locks to support** multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
- An event acts much like a condition variable

Linux Synchronization

- Linux: Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
- semaphores
- spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
- mutex locks
- condition variables
- Non-portable extensions include:
- read-write locks

- spin locks

Atomic Transactions

- System Model
- Log-based Recovery
- Checkpoints
- Concurrent Atomic Transactions

System Model

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
- Here we are concerned with changes to stable storage – disk
- Transaction is series of read and write operations
- Terminated by commit (transaction successful) or abort (transaction failed) operation Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
- Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
- Example: disk and tape
- Stable storage – Information never lost
- Not actually possible, so approximated via replication or RAID to devices with independent failure modes
- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
- Log on stable storage, each log record describes single transaction write operation, including
Transaction name
Data item name
Old value
New value
- $\langle T_i \text{ starts} \rangle$ written to log when transaction T_i starts
- $\langle T_i \text{ commits} \rangle$ written when T_i commits
- Log entry must reach stable storage before operation on data occurs

Log-Based Recovery Algorithm

Using the log, system can handle any volatile memory errors

- $\text{Undo}(T_i)$ restores value of all data updated by T_i
- $\text{Redo}(T_i)$ sets values of all data in transaction T_i to new values
- $\text{Undo}(T_i)$ and $\text{redo}(T_i)$ must be idempotent
- Multiple executions must have the same result as one execution

- If system fails, restore state of all updated data via log
- If log contains $\langle T_i \text{ starts} \rangle$ without $\langle T_i \text{ commits} \rangle$, $\text{undo}(T_i)$
- If log contains $\langle T_i \text{ starts} \rangle$ and $\langle T_i \text{ commits} \rangle$, $\text{redo}(T_i)$

Checkpoints

Log could become long, and recovery could take long

Checkpoints shorten log and recovery time.

Checkpoint scheme:

1. Output all log records currently in volatile storage to stable storage

2. Output all modified data from volatile to stable storage

3. Output a log record $\langle \text{checkpoint} \rangle$ to the log on stable storage

Now recovery only includes T_i , such that T_i started executing before the most recent checkpoint, and all transactions after T_i . All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
- Inefficient, too restrictive
- Concurrency-control algorithms provide serializability

Serializability

- Consider two data items A and B
- Consider Transactions T_0 and T_1
- Execute T_0, T_1 atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are $N!$ valid serial schedules

Schedule 1: T_0 then T_1

	T_0	T_1	
N	read(A)		
	write(A)		
	read(B)		overlapped execute
	write(B)		cessarily incorrect
		read(A)	itions O_i, O_j
		write(A)	a item, with at least one write
		read(B)	perations of different transactions & O_i and O_j don't conflict
		write(B)	er O_j O_i equivalent to S
			pping nonconflicting operations
			• S IS CONFLICT SERIALIZABLE

Schedule 2: Concurrent Serializable Schedule

T_0	T_1
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
- Follow locking protocol for access control
- Locks
- Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
- Exclusive – T_i has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
- Similar to readers-writers algorithm

Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
- Growing – obtaining locks
- Shrinking – releasing locks
- Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp $TS(T_i)$ before T_i starts
- $TS(T_i) < TS(T_j)$ if T_i entered system before T_j
- TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
- If $TS(T_i) < TS(T_j)$, system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j

Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
- W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
- R-timestamp(Q) – largest timestamp of successful read(Q)

- Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order

Suppose T_i executes read(Q)

If $TS(T_i) < W\text{-timestamp}(Q)$, T_i needs to read value of Q that was already overwritten read operation rejected and T_i rolled back

If $TS(T_i) \geq W\text{-timestamp}(Q)$ read executed, $R\text{-timestamp}(Q)$ set to $\max(R\text{-timestamp}(Q), TS(T_i))$

Timestamp-ordering Protocol

Suppose T_i executes write (Q)

If $TS(T_i) < R\text{-timestamp}(Q)$, value Q produced by T_i was needed previously and T_i assumed it would never be produced Write operation rejected, T_i rolled back If $TS(T_i) < W\text{-timestamp}(Q)$, T_i attempting to write obsolete value of Q Write operation rejected and T_i rolled back Otherwise, write executed Any rolled back transaction T_i is assigned new timestamp and restarted Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible Under Timestamp Protocol

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)