Syllabus

KCS-401: Operating System

B.TECH. (COMPUTER SCIENCE AND ENGINEERING)

FOURTH SEMESTER (DETAILED SYLLABUS)

Course Outcome (CO) Bloom's Knowledge Level (KL)

At the end of course, the student will be able to understand

CO 1: Understand the structure and functions of OS	K1, K2
CO 2: Learn about Processes, Threads and Scheduling algorithms.	K1, K2
CO 3: Understand the principles of concurrency and Deadlocks	K2
CO 4: Learn various memory management scheme	K2
CO 5: Study I/O management and File systems.	K2,K4

Unit-I

Introduction : Operating system and functions, Classification of Operating systems- Batch, Interactive, Time sharing, Real Time System, Multiprocessor Systems, Multiuser Systems, Multiprocess Systems, Multithreaded Systems, Operating System Structure- Layered structure, System Components, Operating System services, Reentrant Kernels, Monolithic and Microkernel Systems.

Unit-II

Concurrent Processes: Process Concept, Principle of Concurrency, Producer / Consumer Problem, Mutual Exclusion, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Test and Set operation; Classical Problem in Concurrency- Dining Philosopher Problem, Sleeping Barber Problem; Inter Process Communication models and Schemes, Process generation.

Unit-III

CPU Scheduling: Scheduling Concepts, Performance Criteria, Process States, Process Transition Diagram, Schedulers, Process Control Block (PCB), Process address space, Process identification information, Threads and their management, Scheduling Algorithms, Multiprocessor Scheduling. Deadlock: System model, Deadlock characterization, Prevention, Avoidance and detection, Recovery from deadlock.

Unit-IV

Memory Management: Basic bare machine, Resident monitor, Multiprogramming with fixed partitions, Multiprogramming with variable partitions, Protection schemes, Paging, Segmentation, Paged segmentation, Virtual memory concepts, Demand paging, Performance of demand paging, Page replacement algorithms, Thrashing, Cache memory organization, Locality of reference.

Unit-V

I/O Management and Disk Scheduling: I/O devices, and I/O subsystems, I/O buffering, Disk storage and disk scheduling, RAID. File System: File concept, File organization and access mechanism, File directories, and File sharing, File system implementation issues, File system protection and security.

Text Book:

T1: Silberschatz, Galvin and Gagne, **"Operating Systems Concepts"**, Wiley

References:

R1: Andrew S. Tanenbaum **"Modern Operating System "**Pearson Education

R2: D M Dhamdhere, "**Operating Systems : A Concept based Approach**", 2nd Edition,TMH

R3: William Stallings, "**Operating Systems: Internals and Design Principles**", 6th Edition, Pearson Education

R4: Harvey M Dietel, " **An Introduction to Operating System**", Pearson Education

R5: Sibsankar Halder and Alex A Aravind, "**Operating Systems**", Pearson Education

Unit – II

Concurrent Processes: Process Concept, Principle of Concurrency, Producer / Consumer Problem, Mutual Exclusion, Critical Section Problem, Dekker's solution, Peterson's solution, Semaphores, Test and Set operation; Classical Problem in Concurrency- Dining Philosopher Problem, Sleeping Barber Problem; Inter Process Communication models and Schemes, Process generation.

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 count)  == 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
- *P1, P2,* and *P3* share mailbox A
- *P1,* sends; *P2* and *P3* 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

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 Pi

```
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 nSemaphore *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 locksnCan implement a counting semaphore S as a binary semaphore
- Provides mutual exclusionSemaphore mutex; // initialized to 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.

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

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	
.	.
.	.
signal (S);	
signal (Q);	
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 writenProblem – 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

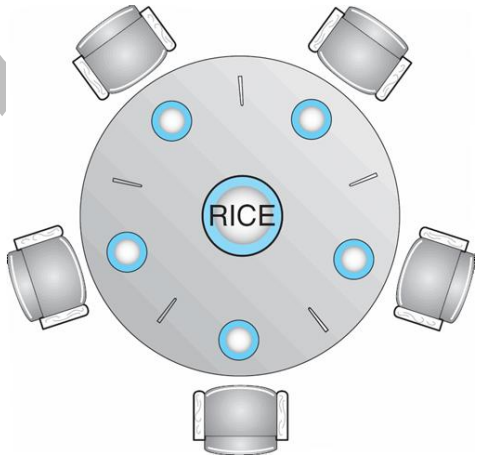
The structure of a writer process

```
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:

l 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 (…) {……}

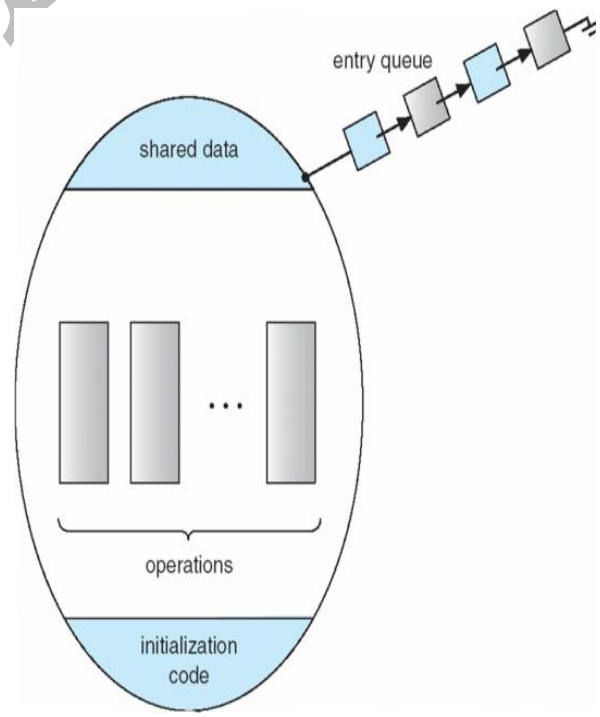
Initialization code (….) { … }

…

}

}

Schematic view of a Monitor



Condition Variables

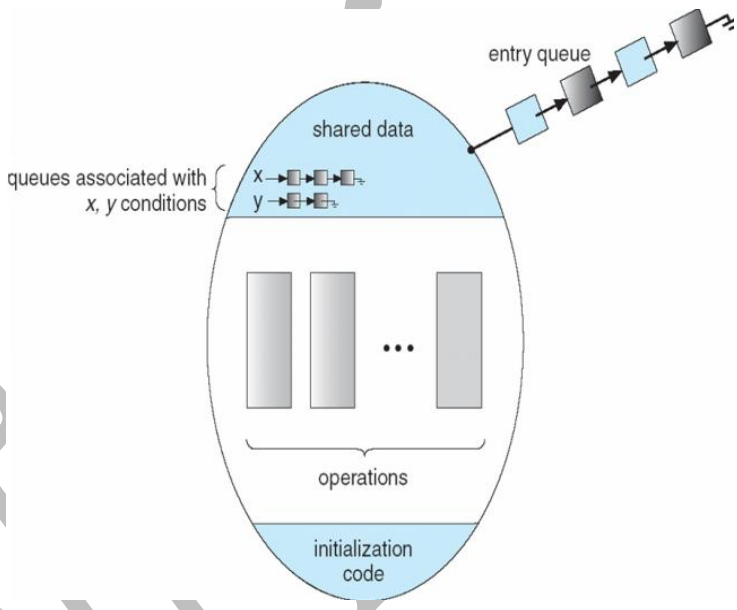
condition x, y;

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:

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

Monitor Implementation Using Semaphores

Variables

```
                                semaphore mutex;  // (initially  = 1)
                                semaphore next;     // (initially  = 0)
                                int next-count = 0;nEach procedure F  will be replaced by
                                   wait(mutex);
                                      …
```

body of *F*;

…

if (next_count > 0)

signal(next)

else

signal(mutex);nMutual exclusion within a monitor is ensured.

Monitor Implementation

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

semaphore x_sem; // (initially = 0)

int x-count = 0;nThe 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;

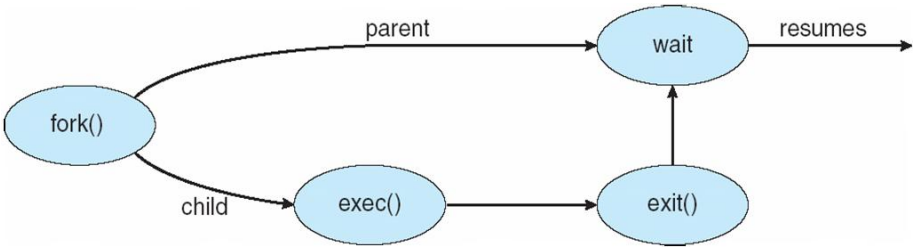
condition x;

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

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

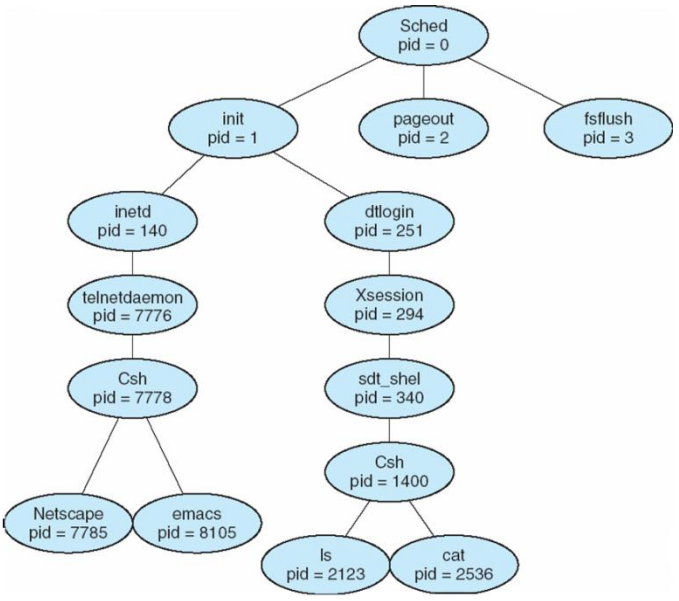
Process Creation



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