CSE3221.3
**Operating System Fundamentals**

## No.2

# Process

*Prof. Hui Jiang*
*Dept of Computer Science and Engineering*
*York University*

## How OS manages CPU usage?

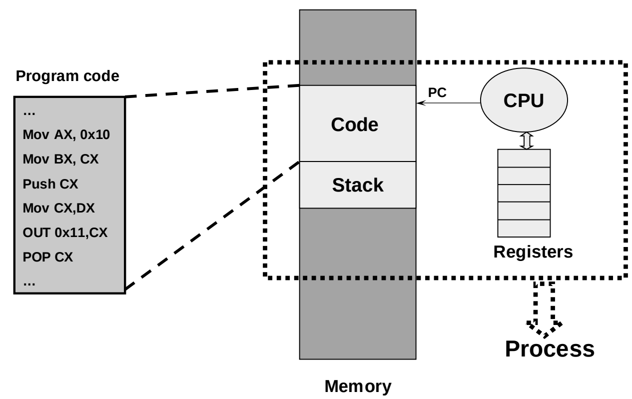
- How CPU is used?
  - Users run programs in CPU
- In a multiprogramming system, a CPU always has several jobs to run.
- How to define a CPU job?
  - The important concept:

**PROCESS**

## Process

- Process is a running program, a program in execution.
- Process is a basic unit of CPU activities, a process is a unit of work in a multiprogramming system.
- Many different processes in a multiprogramming system:
  - User processes executing user code
    - Word processor, Web browser, email editor, etc.
  - System processes executing operating system codes
    - CPU scheduling
    - Memory-management
    - I/O operation
- Multiple processes concurrently run in a CPU.

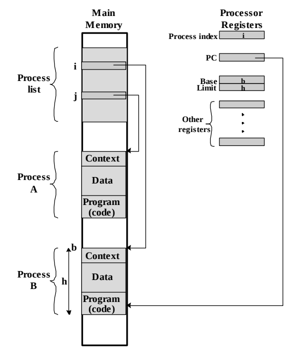
## Process vs. Program Code



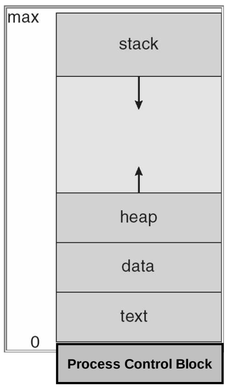
# Process

- **A Process includes:**
  - **Text Section:** memory segment including program codes.
  - **Program Counter (PC):** the address of the instruction to be executed next.
  - **All CPU's Registers**
  - **Stack:** memory segment to save temporary data, such as local variable, function parameters, return address, ...
  - **Data Section:** memory segment containing global and static variables.

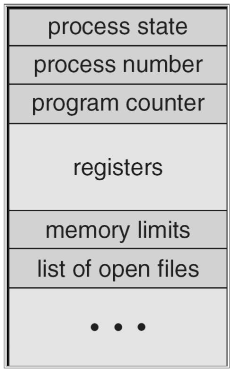
# Process in Memory




# Process in Memory

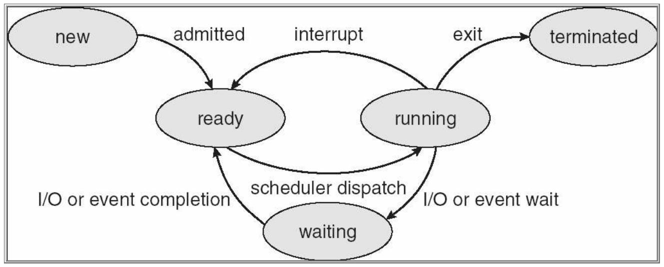



# Data Structure to represent a Process: Process Control Block (PCB)




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

## Process States

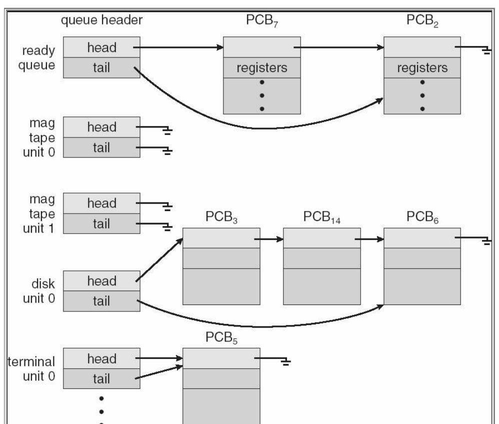



- **New:** the process is just being created
- **Running:** instructions are being executed by CPU
- **Waiting:** waiting for some event, I/O completion or a signal
- **Ready:** waiting to be assigned to CPU to run
- **Terminated:** it finished execution

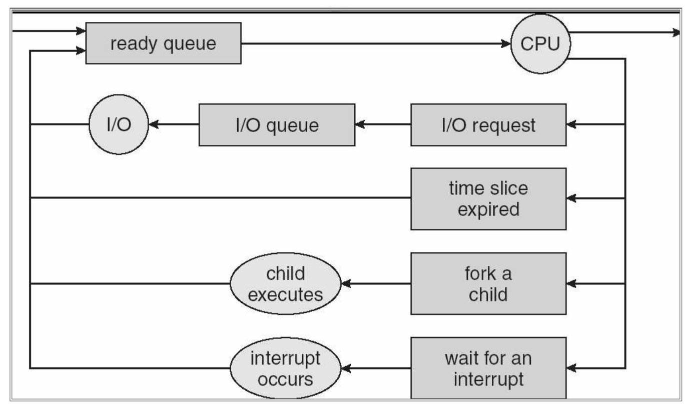
## Process Scheduling: scheduling queues

- **Scheduling Queues:**
  - **List of processes competing for the same resource.**
- **Queues is generally implemented as linked lists.**
- **Each item in the linked list is PCB of a process, we extend each PCB to include a pointer to point to next PCB in the queue.**
- **Examples of scheduling queues:**
  - **Ready Queue: all processes waiting for CPU**
  - **Device Queues: all processes waiting for a particular device; Each device has its own device queue.**

## Scheduling Queues



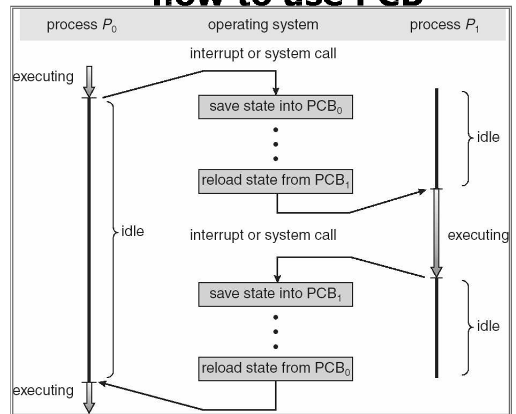

## Queuing Diagram



## Schedulers

- The scheduler's role
- Scheduler categories:
  - Long-term Scheduler (Job scheduler):
    - choose a job from job pool to load into memory to start.
    - Control the degree of multiprogramming – number of process in memory.
    - Select a good mix of I/O-bound processes and CPU-bound processes.
  - Short-term scheduler (CPU scheduler)
    - Select a process from ready queue to run once CPU is free.
    - Executed very frequently (once every 100 millisecond).
    - Must be fast for efficiency.
  - Medium-term scheduler: SWAPPING
    - Swap out / swap in.

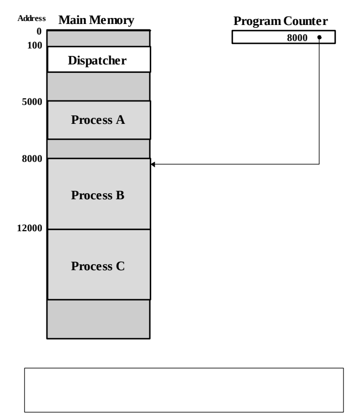
## CPU Switch from process to process: how to use PCB




## Context Switch

- Context Switch: switching the CPU from one process to another.
  - Saving the state of old process to its PCB.
  - CPU scheduling: select a new process.
  - Loading the saved state in its PCB for the new process.
- The context of a process is represented by its PCB.
- Context-switch time is pure overhead of the system, typically from 1–1000 microseconds, mainly depending on:
  - Memory speed.
  - Number of registers.
  - Existence of special instruction.
  - The more complex OS, the more to save.
- Context switch has become such a performance bottleneck in a large multi-programming system:
  - New structure to reduce the overhead: THREAD.

## Context Switch: example



# Trace of Processes

| 5000 | 8000 | 12000 |
|---|---|---|
| 5001 | 8001 | 12001 |
| 5002 | 8002 | 12002 |
| 5003 | 8003 | 12003 |
| 5004 | | 12004 |
| 5005 | | 12005 |
| 5006 | | 12006 |
| 5007 | | 12007 |
| 5008 | | 12008 |
| 5009 | | 12009 |
| 5010 | | 12010 |
| 5011 | | 12011 |
| (a) Trace of Process A | (b) Trace of Process B | (c) Trace of Process C |

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

# Trace of Processes

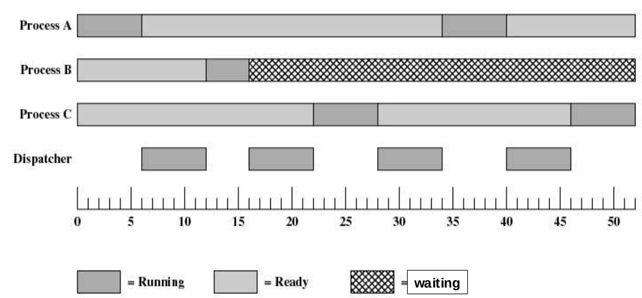
| | |
|---|---|
| 1 | 5000 |
| 2 | 5001 |
| 3 | 5002 |
| 4 | 5003 |
| 5 | 5004 |
| 6 | 5005 |
| ------------------ Time out | |
| 7 | 100 |
| 8 | 101 |
| 9 | 102 |
| 10 | 103 |
| 11 | 104 |
| 12 | 105 |
| 13 | 8000 |
| 14 | 8001 |
| 15 | 8002 |
| 16 | 8003 |
| --------------- I/O request | |
| 17 | 100 |
| 18 | 101 |
| 19 | 102 |
| 20 | 103 |
| 21 | 104 |
| 22 | 105 |
| 23 | 12000 |
| 24 | 12001 |
| 25 | 12002 |
| 26 | 12003 |

| | |
|---|---|
| 27 | 12004 |
| 28 | 12005 |
| ------------------ Time out | |
| 29 | 100 |
| 30 | 101 |
| 31 | 102 |
| 32 | 103 |
| 33 | 104 |
| 34 | 105 |
| 35 | 5006 |
| 36 | 5007 |
| 37 | 5008 |
| 38 | 5009 |
| 39 | 5010 |
| 40 | 5011 |
| ------------------ Time out | |
| 41 | 100 |
| 42 | 101 |
| 43 | 102 |
| 44 | 103 |
| 45 | 104 |
| 46 | 105 |
| 47 | 12006 |
| 48 | 12007 |
| 49 | 12008 |
| 50 | 12009 |
| 51 | 12010 |
| 52 | 12011 |
| ------------------ Time out | |

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

# Process State

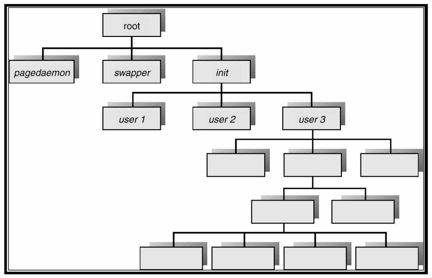



# Operations on Processes (UNIX as an example)

- Process creation.
- Process termination.
- Inter-process communication (IPC).
- Unix programming:
  - Multiple-process programming.
  - Cooperating process tasks.

## Process Creation(1)

- A process can create some new processes via a *create-process* system call:
  - Parent process / children process.
- All process in Unix form a tree structure.




## Process Creation(2)

- Resource Allocation of child process
  - The child process get its resource from OS directly.
  - Constrain to its parent's resources.
- Parent status
  - The parent continues to execute concurrently with its children.
  - The parent waits until its children terminate.
- Initialization of child process address space
  - Child process is a duplicate of its parent process.
  - Child process has a program loaded into it.
- How to pass parameters (initialization data) from parent to child?

## UNIX Example: *fork()*

- In UNIX, each process is identified by its process number (*pid*).
- In UNIX, *fork()* is used to create a new process.
- Creating a new process with *fork():*
  - New child process is created by *fork().*
  - Parent process' address space is copied to new process' space (initially identical address space).
  - Both child and parent processes continue execution from the instruction after *fork().*
  - Return code of *fork()* is different: in child process, return code is zero, in parent process, return code is nonzero (it is the process number of the new child process)
  - If desirable, another system call *execlp()* can be used by one of these two processes to load a new program to replace its original memory space.

## Typical Usage of fork()

```
#include <stdio.h>
void main(int argc, char *argv[ ])
{
   int pid ;

   /* fork another process */
  pid = fork() ;

  if (pid < 0)  {  /* error occurred */
     fprintf(stderr, "Fork Failed!\n") ;
     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\n") ;
     exit(0) ;
   }
}
```

## Process Termination

- Normal termination:
  - Finishes executing its final instruction or call *exit()* system call.
- Abnormal termination: make system call *abort().*
  - The parent process can cause one of its child processes to terminate.
    - The child uses too much resources.
    - The task assigned to the child is no longer needed.
    - If the parent exits, all its children must be terminated in some systems.
- Process termination:
  - The process returns data (output) to its parent process.
    - In UNIX, the terminated child process number is return by *wait()* in parent process.
  - All its resources are de-allocated by OS

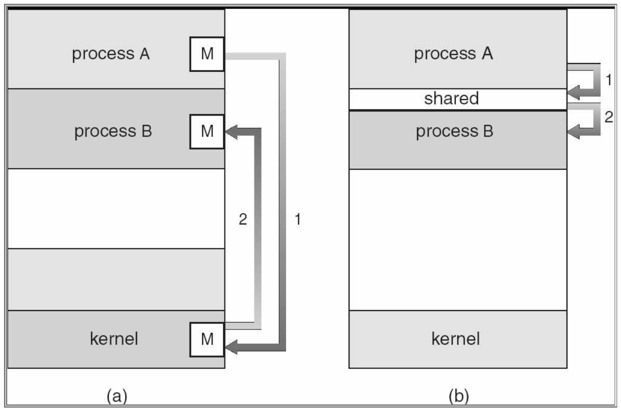
## Multiple-Process Programming in Unix

- Unix system calls for process control:
  - *getid*(): get process ID (*pid*) of calling process.
  - *fork*(): create a new process.
  - *exec()*: load a new program to run.
    - execl(char *pathname, char *arg0, …);
    - execv(char *pathname, char* argv[ ]) ;
    - execle(), execve(), execlp(), execvp()
  - *wait(), waitid*(): wait child process to terminate.
  - *exit(), abort()*: a process terminates.

## Cooperating Processes

- Concurrent processes executing in the operating system
  - Independent: runs alone
  - Cooperating: it can affect or be affected by other processes
- Why cooperating processes?
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Need inter-process communication (IPC) mechanism for cooperating processes:
  - Shared-memory
  - Message-passing

## IPC Approaches



## Inter-process Communication (IPC): Message Passing

- IPC with message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- IPC based on message-passing system:
  - Processes communication without sharing space.
  - Communication is done through the passing of messages.
  - At least two operations:
    - *send*(message)
    - *receive*(message)
  - Message size: fixed vs. variable
  - Logical communication link:
    - Direct vs. indirect communication
    - Symmetric vs. asymmetric communication
    - Automatic or explicit buffering

## Direct Communication

- Each process must explicitly name the recipient or sender of the communication.
  - *send*(P,message)
  - *Receive*(Q,message)
- A link is established between each pair of processes
- A link is associated with exactly two processes
- Asymmetric direct communication: no need for recipient to name the sender
  - *send*(P,message)
  - *receive*(&id,message): id return the sender identity
- Disadvantage of direct communication:
  - Limited modularity due to explicit process naming

## Indirect Communication

- The messages are sent to and received from *mailbox.*
- *Mailbox* is a logical unit where message can be placed or removed by processes. (each mailbox has a unique id)
  - *send*(A,message): A is mailbox ID
  - *receive*(A,message)
- A link is established in two processes which share mailbox.
- A link may be associated with more than two processes.
- A number of different link may exist between each pair of processes.
- OS provides some operations on mailbox
  - Create a new mailbox
  - Send and receive message through the mailbox
  - Delete a mailbox

## Synchronization in message-passing

- Message passing may be either blocking or non-blocking.
- Blocking is considered synchronous
- Non-blocking is considered asynchronous
- *send*() and *receive*() primitives may be either blocking or non-blocking.
  - Blocking send
  - Non-blocking send
  - Blocking receive
  - Non-blocking receive
- When both the *send* and *receive* are blocking, we have a *rendezvous* between the sender and the receiver.

## Buffering in message-passing

- The buffering provided by the logical link:
  - Zero capacity: the sender must block until the recipient receives the message (no buffering).
  - Bounded capacity: the buffer has finite length. The sender doesn't block unless the buffer is full.
  - Unbounded capacity: the sender never blocks.

## IPC in UNIX

- ★ Signals
- ★ Pipes
- ★ Message queues
- Shared memory
- Sockets
- others

## Signal function in Unix

- Signal is a technique to notify a process that some events have occurred.
- The process has three choices to deal with the signal:
  - Ignore the signal
  - Let the default action occur.
  - Provide a function that is called when the signals occurs.
- *signal*() function: change the action function for a signal

```
#include <signal.h>
void (*signal(int signo, void (*func) (int )   ) ;
```

- *kill()* function: send a signal to another process

```
#include <sys/types.h>
#include <signal.h>
int kill (int pid, int signo) ;
```

## Unix Signals

| Name | Description | ANSI C | POSIX.1 | SVR4 | 4.3+BSD | Default action |
|---|---|---|---|---|---|---|
| SIGABRT | abnormal termination (abort) | • | • | • | • | terminate w/core |
| SIGALRM | time out (alarm) | | • | • | • | terminate |
| SIGBUS | hardware fault | | | • | • | terminate w/core |
| SIGCHLD | change in status of child | | job | • | • | ignore |
| SIGCONT | continue stopped process | | job | • | • | continue/ignore |
| SIGEMT | hardware fault | | | • | • | terminate w/core |
| SIGFPE | arithmetic exception | • | • | • | • | terminate w/core |
| SIGHUP | hangup | | • | • | • | terminate |
| SIGILL | illegal hardware instruction | • | • | • | • | terminate w/core |
| SIGINFO | status request from keyboard | | | | • | ignore |
| SIGINT | terminal interrupt character | • | • | • | • | terminate |
| SIGIO | asynchronous I/O | | | • | • | terminate/ignore |
| SIGIOT | hardware fault | | | • | • | terminate w/core |
| SIGKILL | termination | | • | • | • | terminate |
| SIGPIPE | write to pipe with no readers | | • | • | • | terminate |
| SIGPOLL | pollable event (poll) | | | • | | terminate |
| SIGPROF | profiling time alarm (setitimer) | | | • | • | terminate |
| SIGPWR | power fail/restart | | | • | | ignore |
| SIGQUIT | terminal quit character | | • | • | • | terminate w/core |
| SIGSEGV | invalid memory reference | • | • | • | • | terminate w/core |
| SIGSTOP | stop | | job | • | • | stop process |
| SIGSYS | invalid system call | | | • | • | terminate w/core |
| SIGTERM | termination | • | • | • | • | terminate |
| SIGTRAP | hardware fault | | | • | • | terminate w/core |
| SIGTSTP | terminal stop character | | job | • | • | stop process |
| SIGTTIN | background read from control tty | | job | • | • | stop process |
| SIGTTOU | background write to control tty | | job | • | • | stop process |
| SIGURG | urgent condition | | | • | • | ignore |
| SIGUSR1 | user-defined signal | | • | • | • | terminate |
| SIGUSR2 | user-defined signal | | • | • | • | terminate |
| SIGVTALRM | virtual time alarm (setitimer) | | | • | • | terminate |
| SIGWINCH | terminal window size change | | | • | • | ignore |
| SIGXCPU | CPU limit exceeded (setrlimit) | | | • | • | terminate w/core |
| SIGXFSZ | file size limit exceeded (setrlimit) | | | • | • | terminate w/core |

# Example: signal in UNIX

```
#include <signal.h>

static void sig_int(int) ;

int main() {

   if(signal(SIGINT,sig_int)==SIG_ERR)
      err_sys("signal error") ;

  sleep(100) ;
}

void sig_int(int signo)
{
  printf("Interrupt\n") ;
}
```

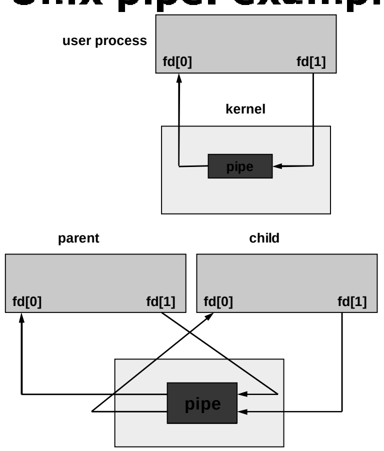
- Event SIGINT: type the interrupt key (Ctrl+C)
- The default action is to terminate the process.
- Now we change the default action into printing a message to screen.

# Unix Pipe

- Half-duplex; only between parent and child.
- Creating a pipe:
  - Call pipe();
  - Then call fork();
  - Close some ends to be a half-duplex pipe.

```
#include <unistd.h>

int pipe( int filedes[2] ) ;
```

# Unix pipe: example




# Unix Pipe: example

```
int main() {

   int n, fd[2] ;
   int  pid ;
   char  line[200] ;

  if( pipe(fd) < 0 )   err_sys("pipe error") ;

  if ( (pid = fork()) < 0 ) err_sys("fork error") ;
  else if ( pid > 0 )  {
      close(fd[0]) ;
      write(fd[1], "hello word\n", 12) ;
 } else  {
   close(fd[1]) ;
   n = read(fd[0], line, 200) ;
   write(STDOUT_FILENO, line, n) ;
}
 exit(0) ;
}
```

## Message Queues in Unix

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int flag) ;
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag) ;
int msgrcv(int msqid, void *ptr, size_t nbytes, int flag) ;
```

## msgget() in UNIX

**int msgget(key_t key, int flag) ;**

- key ➔ an integer to identify the message queue. Should be unique in a system
- msgflg ➔ 0 : access to an existing queue
  IPC_CREAT bit set : create a queue
- return value
  - -1 on error
  - non-negative integer on success: message id

## msgsnd() in UNIX

int *msgsnd* (int msgid, const void *msgp, int msgsz, int msgflg) ;

- msgid ➔ msg id returned by *msgget()*
- msgp ➔ ptr to a structure

```
struct msgStruct{
    long mType ; //type of the message
    char mText[MAX_LEN]; //actual data
};
```

- msgsz ➔size of data in msg
- msgflg ➔ always 0 in our cases
- return value
  - -1 on failure
  - 0 on success

## msgrcv() in UNIX

int *msgrcv*(int msgid, const void *mshp, int msgsz, long msgtype, int msgflg) ;

- msgid ➔ msg id returned by *msgget()*
- msgp ➔ ptr to a msg structure (same as above)
- msgsz ➔size of buffer in msg
- msgflg ➔ always 0 in our cases
- msgtype ➔ 0: get first message in the queue
  >0 : get first message of type msgtype
  <0 : beyond our consideration
- return value
  - -1 on failure
  - No. of bytes in the message on success

## Example: create a msg queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY  32894  /* your CS log in number */

int main() {
   int msgid ;

   msgid = msgget(KEY,0) ;

   if( msgid < 0) {
        msgid = msgget(KEY, IPC_CREAT|0666) ;
        if(msgid < 0 )
            printf("Error in creating message queue!\n");
   }
}
```

## Example: sending a message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY  32894
#define MAX_LEN 100

typedef struct {
            long mType ;
            char mText[MAX_LEN] ;

} Message ;

int main() {
   int msgid ;
   Message msg ;

   strcpy(msg.mText, "Hello world!") ;
   msg.mType = 1 ;

   msgid = msgget(KEY,0) ;

   if( msgid < 0) {
            printf("Error in creating message queue!\n");
            return -1 ;
            }

    if(msgsnd(msgid, &msg,MAX_LEN,0) < 0 )
       printf("Error in sending message\n") ;
      else
       printf("sent message successfully\n") ;
}
```

## Example: receiving a message

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define KEY  32894
#define MAX_LEN 100

typedef struct {
            long mType ;
            char mText[MAX_LEN] ;

} Message ;

int main() {
   int msgid ;
   Message msg ;

   msgid = msgget(KEY,0) ;

   if( msgid < 0) {
            printf("Error in creating message queue!\n");
            return -1 ;
            }

            if(msgrcv(msgid, &msg,MAX_LEN,0,0) < 0 )
             printf("Error in receiving message\n") ;
            else
             printf("Received message: %s\n",msg.mText) ;

   if(msgctl(msgid,IPC_RMID,NULL)<0)     // Remove the message queue from system
     printf("Error in removing message queue!\n") ;
   else
     printf("Removed message queue successfully!\n) ;
}
```

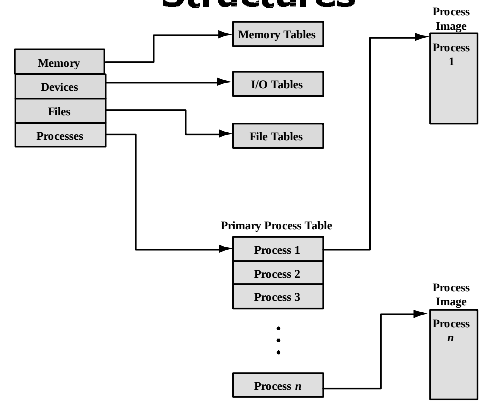
## Shared Memory in Unix

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(const void *shmaddr);

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

## Overall OS Control Structures

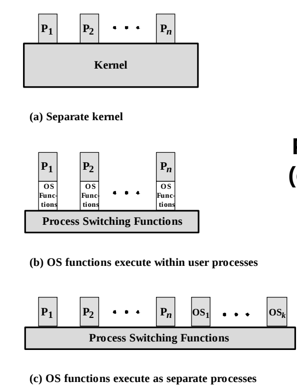
- Tables are constructed for each entity the operating system manages.
  - Process table: PCBs and process images.
  - Memory table: Allocation of main memory to processes; Protection attributes for access to shared memory regions.
  - File table: all opened files; location on hardware; Current Status.
  - I/O table: all I/O devices being used; status of I/O operations.

## Operating System Control Structures




## Execution of Operating System

- Non-process Kernel
  - Execute kernel outside of any process
  - Operating system code is executed as a separate entity that operates in privileged mode
- Execution Within User Processes
  - Operating system software within context of a user process
  - Process executes in privileged mode when executing operating system code
- Process-Based Operating System
  - Implement operating system as a collection of system processes
  - Useful in multi-processor or multi-computer environment




Relationship Between Operatin System and User Processes

**Mode switch**

**vs.**

**Process switch (context switch)**