UNIT-2

Chapter-1: Process Management

Introduction

Process

A *process* is a program under execution.

- Process will need certain resources-such as CPU time, memory, files, and I/O devices to accomplish its task.
- Resources are allocated to the process either when it is created or while it is executing.
- A process is the unit of work in most systems. Systems consist of a collection of processes: Operating system processes execute system code, and user processes execute user code. All these processes may execute concurrently.
- The operating system is responsible for the following activities in connection with process management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

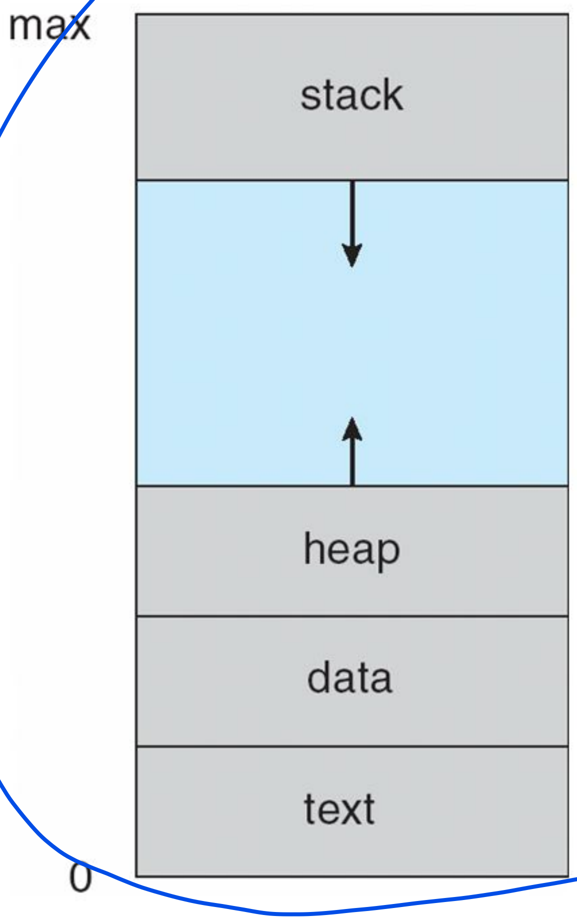
Process in Memory/Memory Layout of a Process

A process is stored across different segments of main memory.

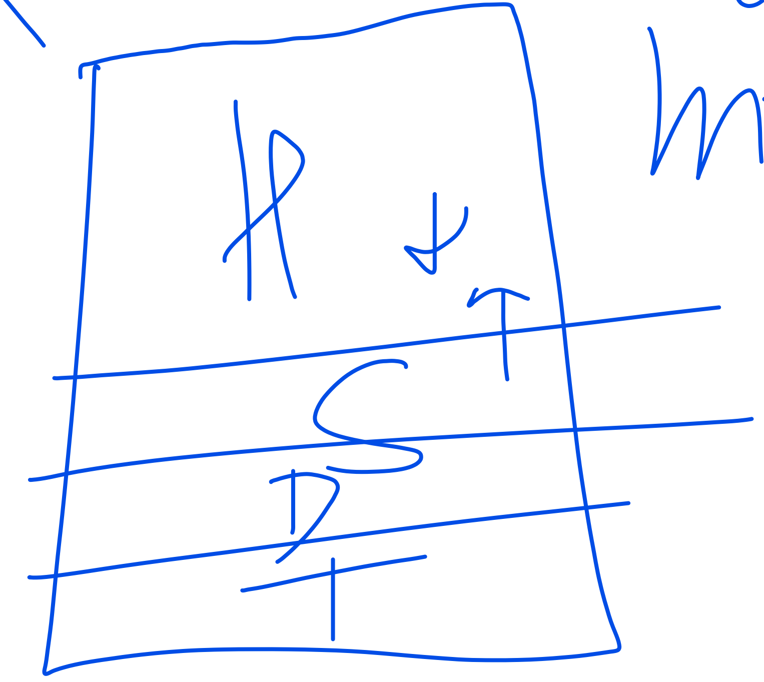
- The program code, is known as **text section.**
- It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
- Process also includes the **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables).
- Process includes a **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

Process in Memory

The structure of a process in memory is shown in Figure:



max



Process in mem

Program v/s Process

Program

A program is a *passive entity*, such as a file containing a list of instructions stored on disk (often called an executable file).

Process

Process is an *active entity*, with a program counter specifying the next instruction to execute and a set of associated resources.

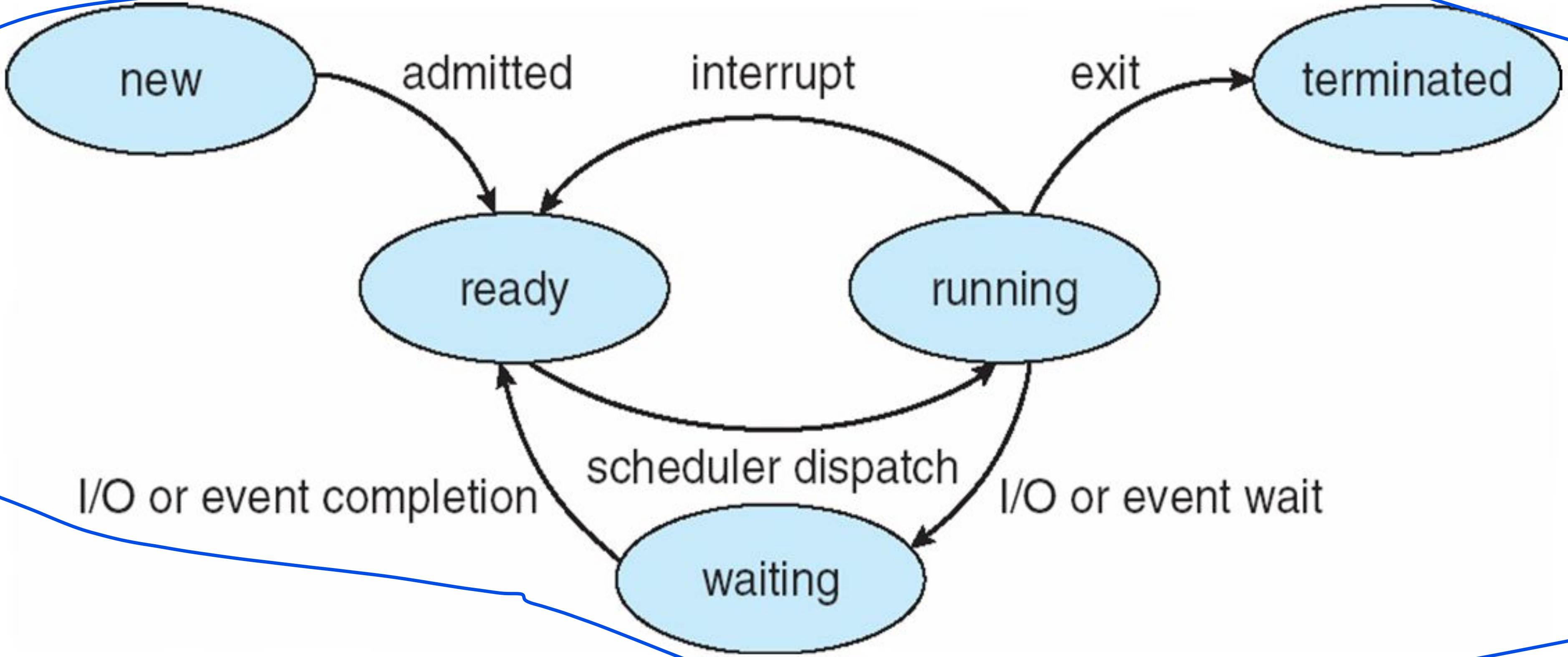
A program becomes a process when an executable file is loaded into memory and allocated with all necessary resources and its instructions are being executed.

Process State

- As a process executes, it changes state.
- The state of a process is defined as "the current activity of that process".
- Each process may be in one of the following states:
  - **new**: The process is being created.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur(such as I/O completion, reception of signal, etc.)
  - **ready**: The process is waiting to be assigned to a processor.
  - **terminated**: The process has finished execution.

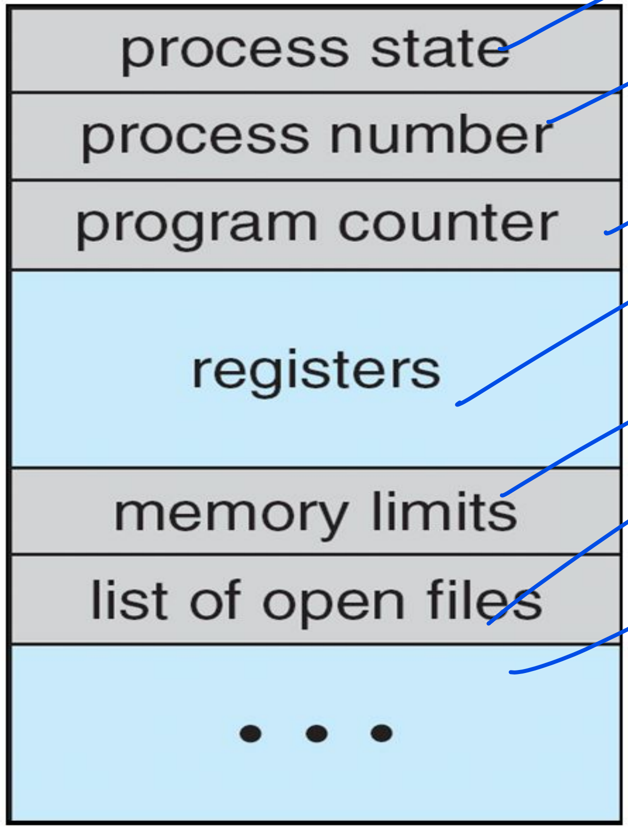
Diagram of Process State

PS



Process Control Block (PCB)

Each process is represented in the operating system by a **process control block (PCB)**-also called a *task control block*.



Process control block contains many pieces of information associated with a specific process, which is described as follows:

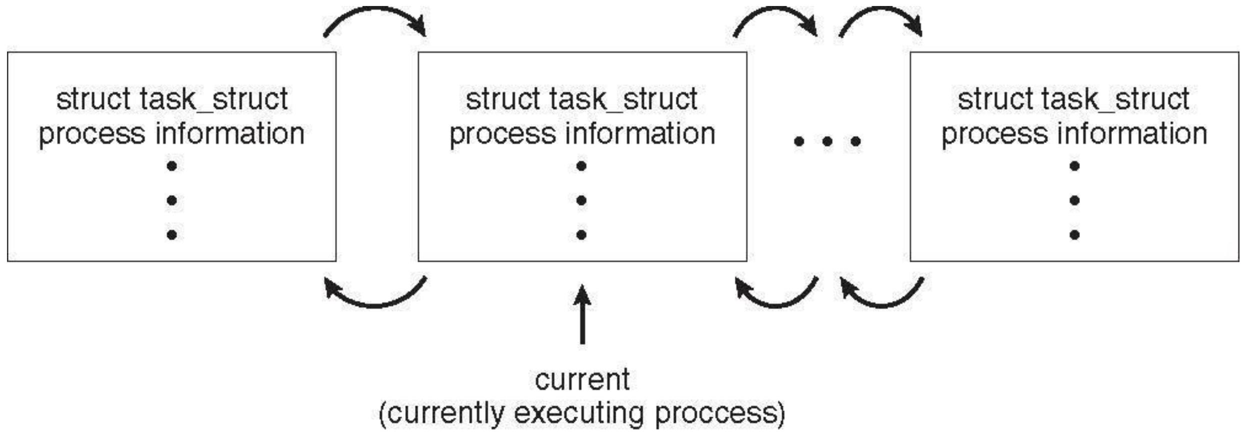
- **Process state** –The state may be new, ready, running, waiting, halted, etc.
- **Program counter** – indicates the address of the next instruction to be executed for this process.
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** Includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information** – includes value of the base and limit registers, the page tables, or the segment tables.
- **Accounting information** –includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, etc.
- **I/O status information** – I/O devices allocated to process, list of open files

Representation of Process control block in Linux

- The process control block in the Linux operating system is represented by the C structure task_struct.
- This structure contains all the necessary information for representing a process.
- Some of these fields include:

```
pid_t pid;     /* process identifier */
long state;      /* state of the process */
unsigned int  time_ slice    /* scheduling information*/
struct files_struct *files    /* list of open files */
struct mm_struct*mm;    /* address space of this process */
```

- With in the Linux kernel, all active processes are represented using a doubly linked list of task_struct, and the kernel maintains a pointer current to the process currently executing on the system.



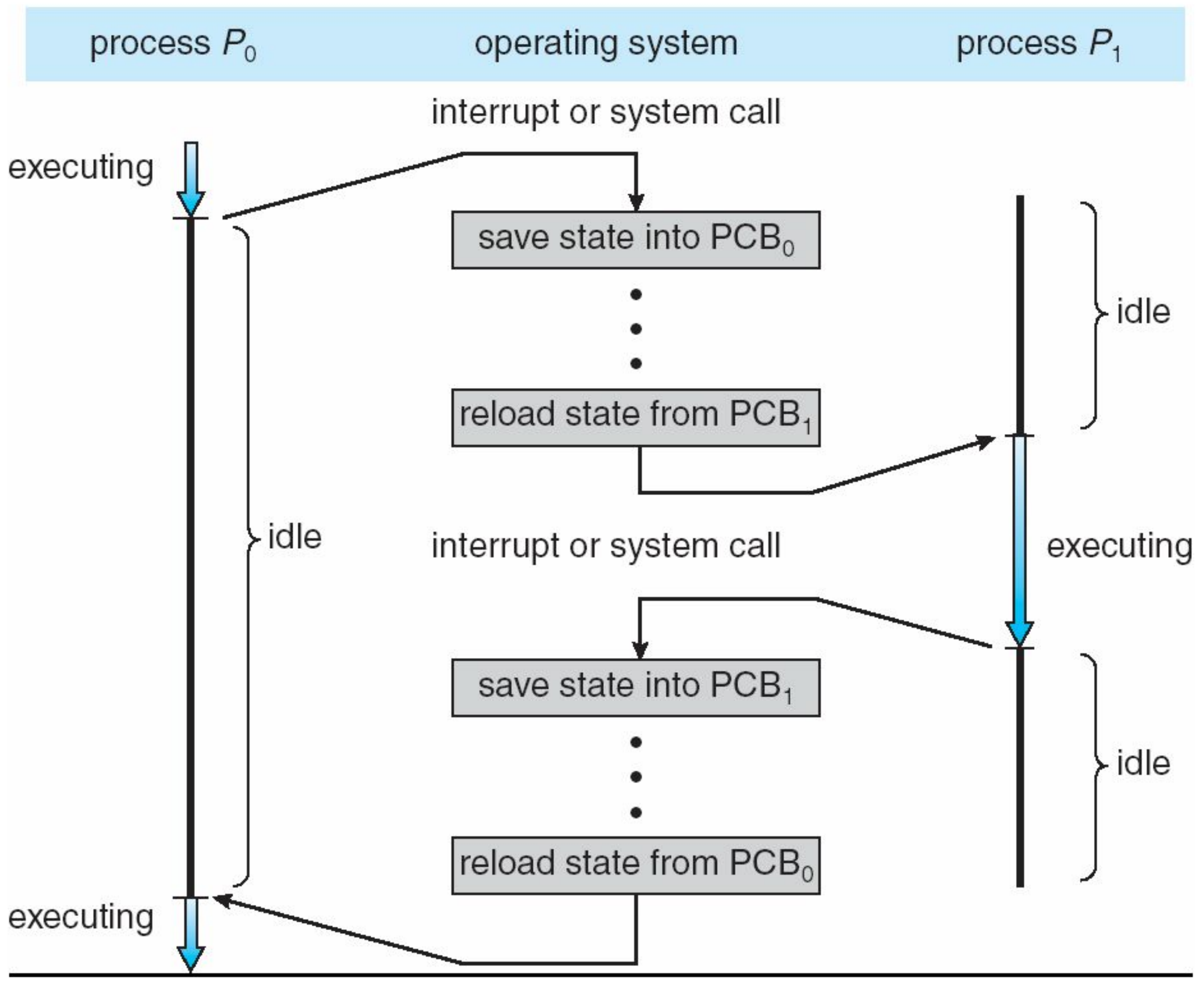
Context Switch

- Switching the CPU to another process requires performing a **state save** of the current process and a **state restore** of a different process. This task is known as a **context switch.**
- When an interrupt occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process;(it includes the value of the CPU registers, the process state and memory-management information.)
- We perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.

Context Switch contd..

- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.

CPU Switch From Process to Process



Process Scheduling

In case of multiprogramming and multitasking systems CPU switches among multiple processes to maximize the CPU Utilization.

To meet these objectives schedulers select an available process from a set of several available processes for execution.

OS maintains the following scheduling queues.

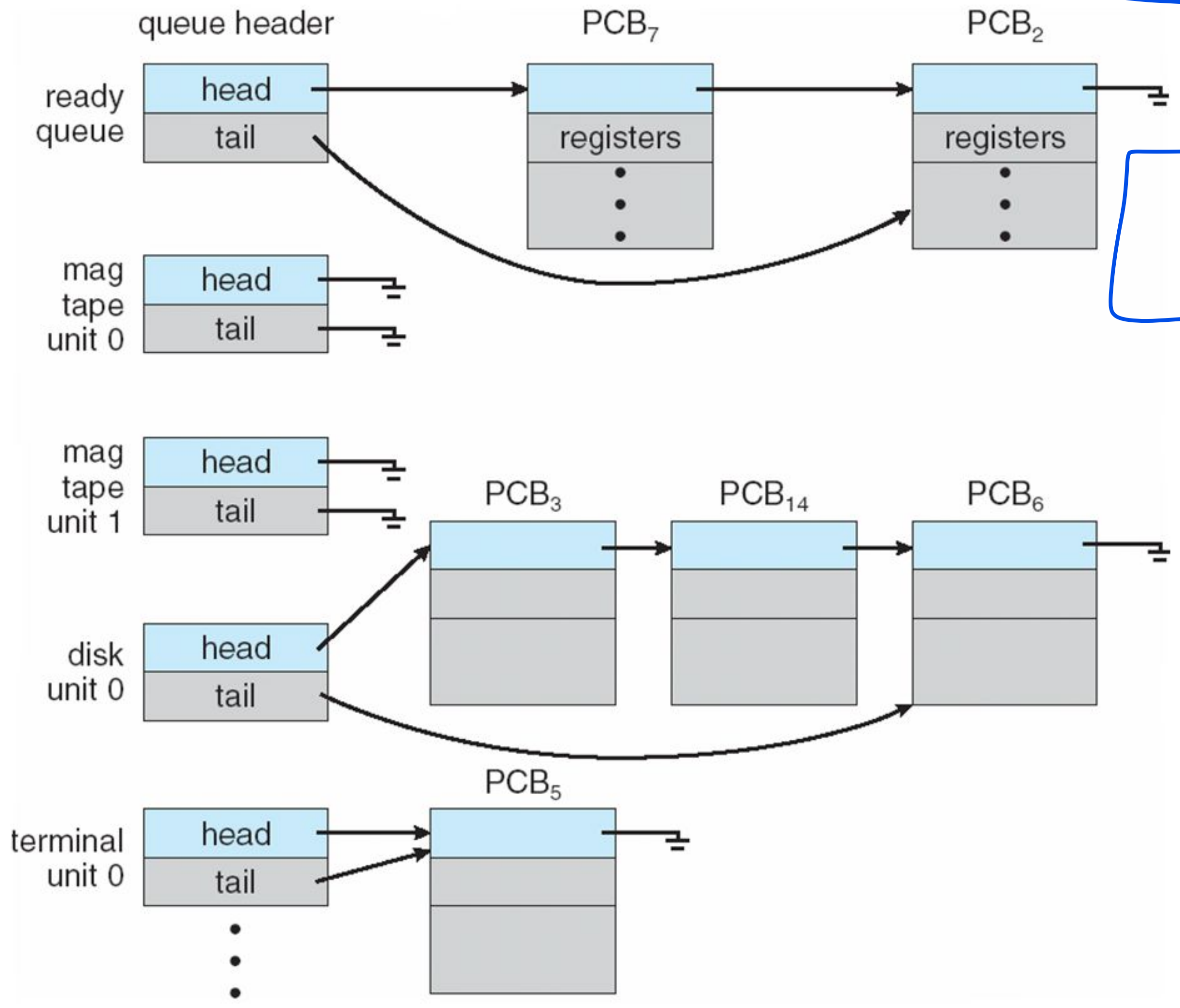
1. Job queue: Consists of all processes in the system

2. Ready queue: Consists of Processes that are residing in main memory, ready and waiting to be executed by processor.

3.Device queue: Consists of a set of processes waiting for an I/O device.

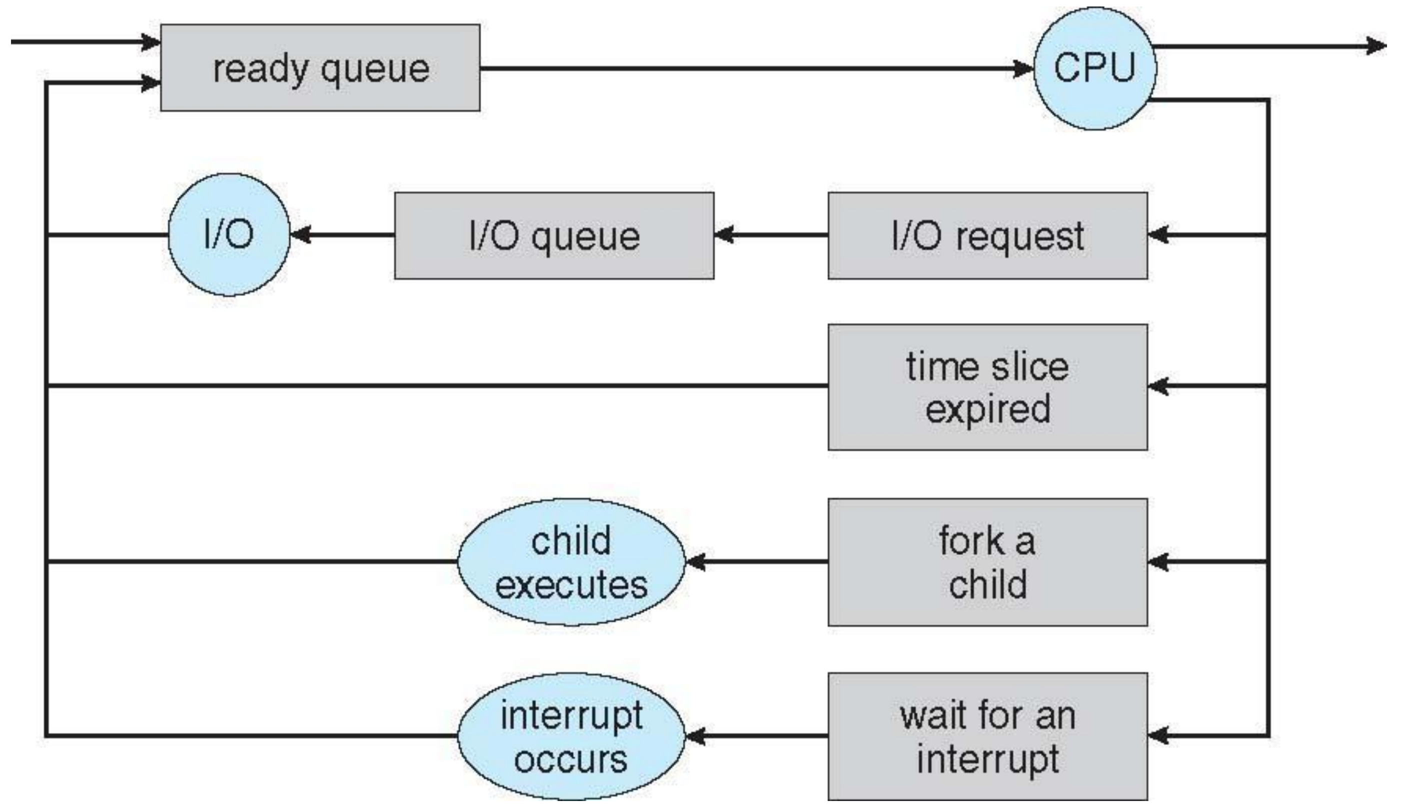
Processes are allowed to migrate among the various queues throughout their life time.

Ready Queue And Various I/O Device Queues



Queueing - diagram Representation of Process Scheduling

- Queueing diagram: **Rectangular boxes** represents queues, Circles represents the resources that serves the queue, and arrow represents flow

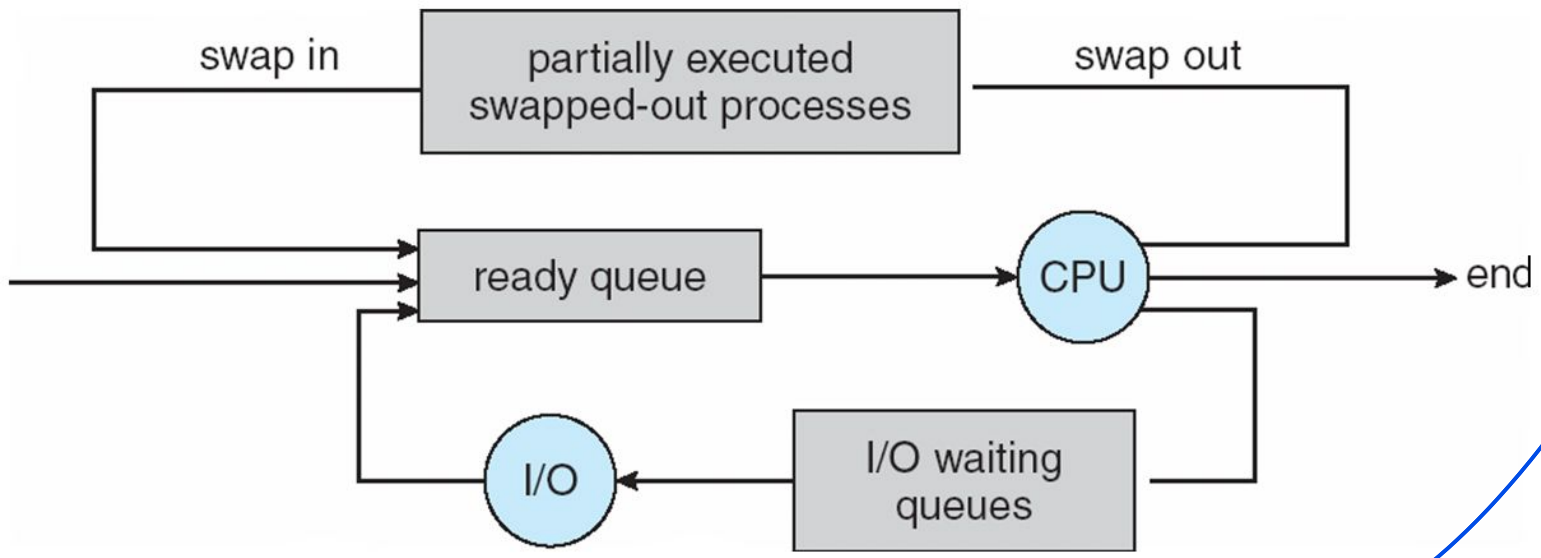


Schedulers

- **Long-term scheduler** (or **job scheduler**) – selects processes from job queue and loads them in to main memory for execution.(i.e. Processes that should be brought into the ready queue).
  - Long-term scheduler is invoked less frequently compared to CPU scheduler/short term scheduler.
  - The long-term scheduler controls the **degree of multiprogramming** **(the number of processes in memory).**
- Long-term scheduler strives for good ***process mix(proper combination of I/O and CPU bound processes) to ensure efficient resource utilization.***
- 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
- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU.
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**

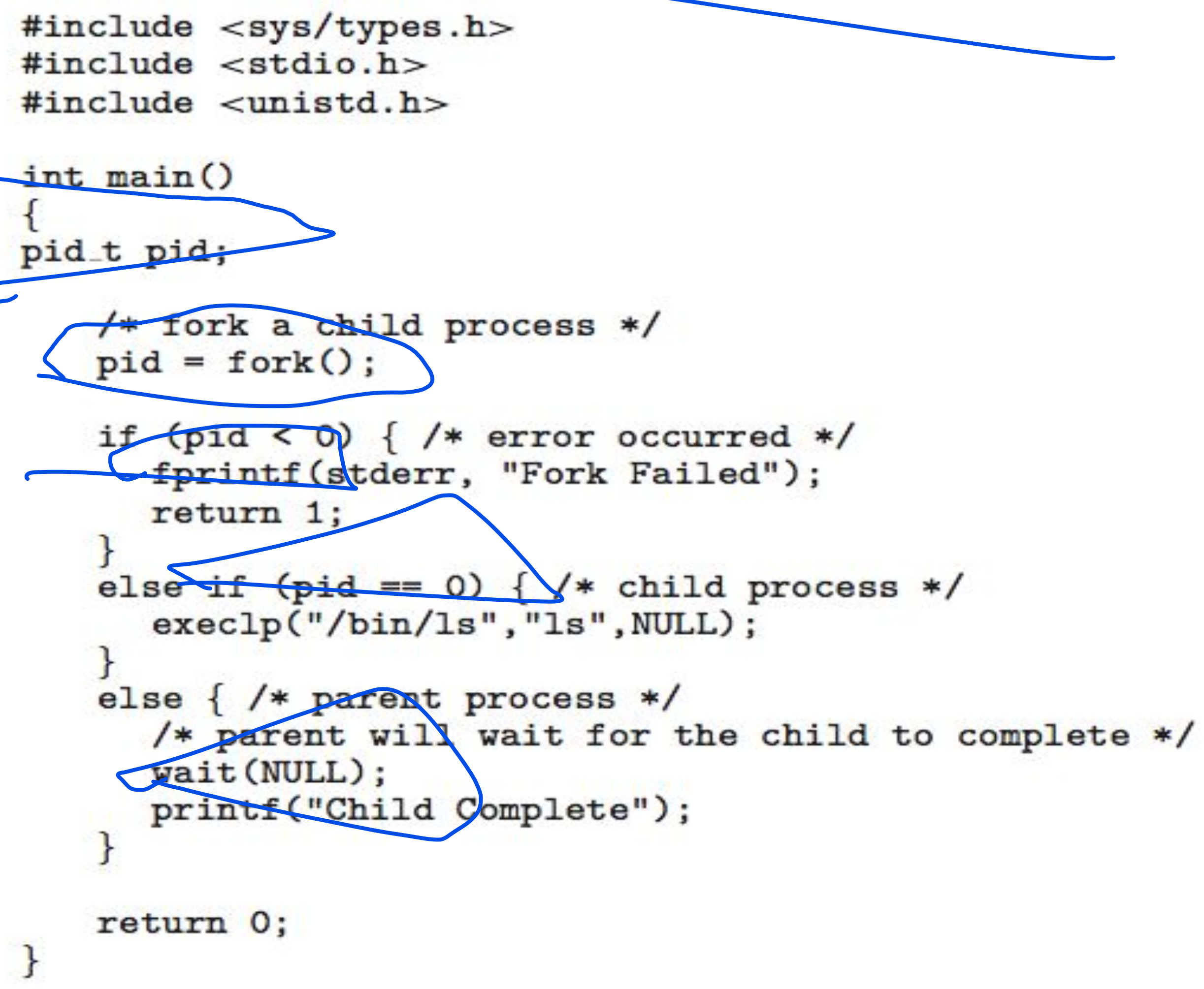


Operations on Processes

Two types

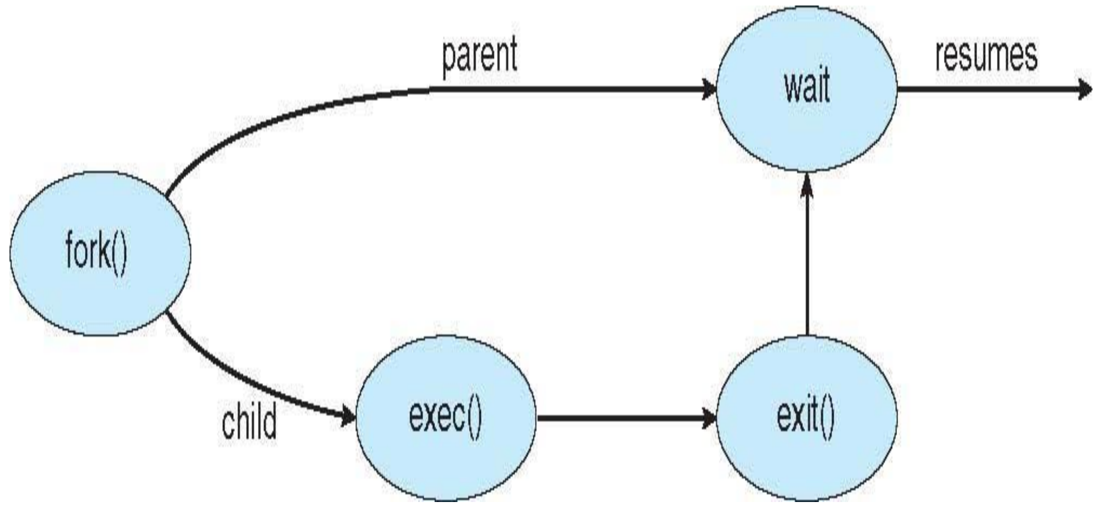
- process creation
- process termination

Example program to illustrate Process creation on Linux



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 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");
   }

   return 0;
}
```

Process creation using fork () system call



Following system calls are used in the example program:

1. fork()

Prototype:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

Return value: Returns a value 0 for child and process id of child for parent on success and -1 on error.

- An existing process invokes Fork function to create a new process.
- The new process created by fork is called child process and the process that invoked fork is called parent process.
- A newly created process by fork is a duplicate copy of parent process.
- By default Child gets a copy of parent's text, data , heap and stack segments. It can replace itself with a new program by invoking exec system call.
- Fork creates a duplicate copy of the file descriptors opened by parent.
- Parent and child execute concurrently upon fork or parent can wait until its child terminates.

2.exec()

- Exec functions are invoked by a newly created child process to replace itself with a completely new program.
- Exec replaces the calling process by a new program.
- The new program has same process ID as the calling process. i.e., upon invoking exec by a child process, No new process is created, instead, exec just replaces the current process by a new program.

UNIX supports six different exec functions and its prototypes are as follows:

```
#include <unistd.h>
• int execl ( const char *pathname, const char *arg0 ,… /*(char *) 0*/);
• int execv (const char *pathname, char * const argv[ ]);
• int execle (const char *pathname, const char *arg0 ,… /*(char *) 0,
                 char *const envp[ ] */);
• int execve ( const char *pathname, char *const argv[ ] , char *const envp [ ]);
• int execlp (const char *filename, const char *arg0 ,… /*(char *) 0*/);
• int execvp (const char *filename ,char *const argv[ ] );
```

l
v
le
ve
exec lp
vp

3. wait()

- Wait function can be invoked by parent process to fetch the termination status of the child process.
- When a child process is terminated the parent is notified by the kernel by sending a SIGCHLD signal.
- The termination of a child is an asynchronous event.(i.e., it can happen at any time).
- The parent can ignore or can provide a function that is called when the SIGCHLD signal occurs.

- The process that calls wait can:
  - Block itself.
  - Return immediately with termination status of the child upon child termination.
  - Return immediately with an error.

Prototype:

```
#include <sys/wait.h>
#include <sys/types.h>
pid_t wait (int *statloc);
```

- statloc is a pointer to integer.
- If statloc is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.
- The integer status returned by the two functions give information about exit status, signal number etc.

Memory layout of parent

Text segment

```
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); }
        }
```

data segment

No content because no global variables

Stack frame for main

Pid=20;

6

Heap

No content no dynamically allocated variables.

Memory layout of child(child gets a copy of text , data stack and heap segments on fork)

Text segment(shared by both parent and child upon fork)

```
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);
                        }
        }
```

data segment

No content because no global variables

Stack frame for main

~~Pid=0;~~

Heap

No content no dynamically allocated variables.

Note: As child process proceeds with its execution upon execution of execlp system call, entire text segment of child will be replaced with ls command executable. And as the execution of child continues data , stack and heap are also updated.

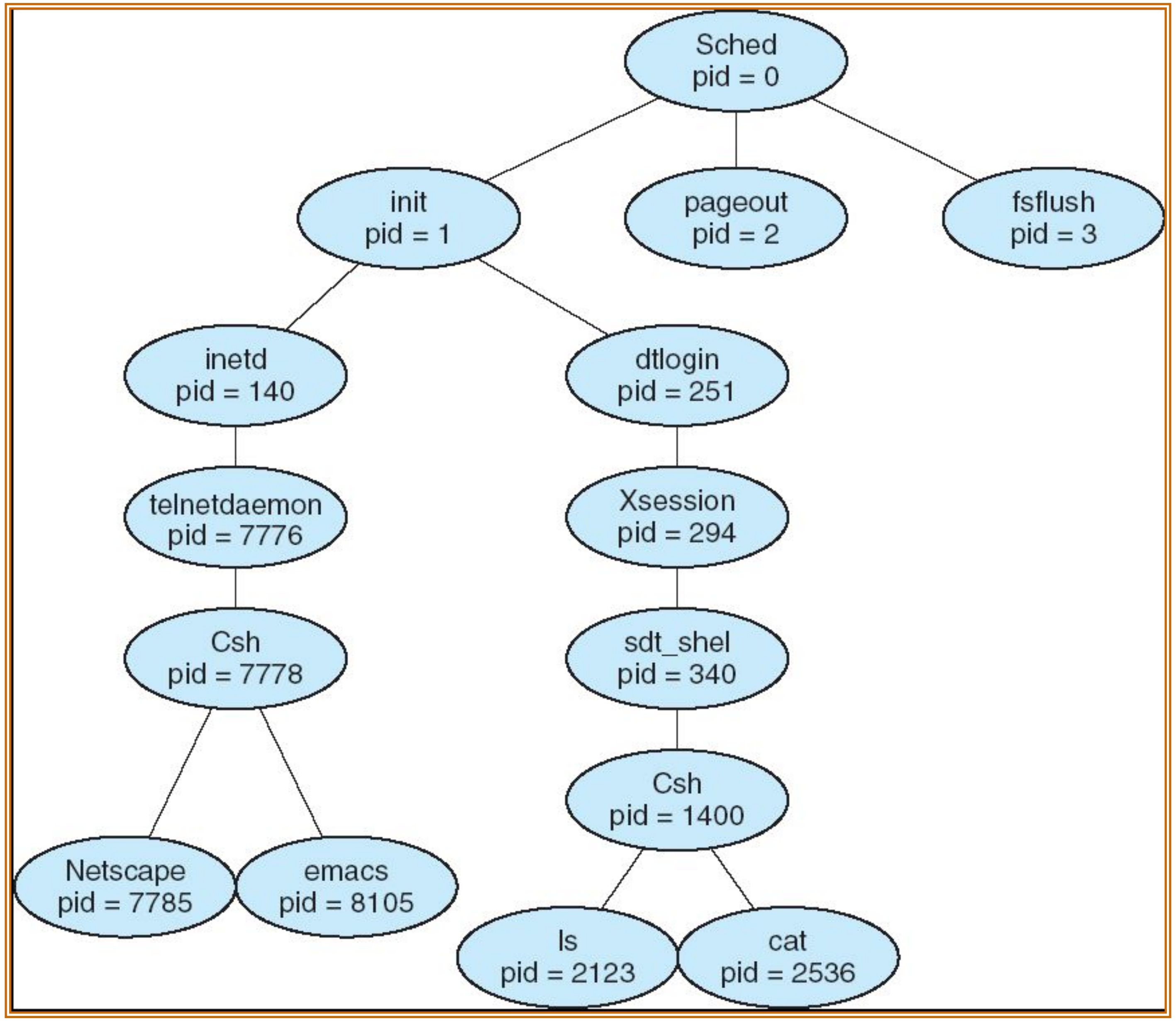


Fig: A tree of processes on a typical Solaris system

Example program to illustrate Process creation on windows

```c
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

- Processes are created on windows using createProcess() function.
- fork () has the child process inheriting the address space of its parent, where as CreateProcess () requires loading a specified program into the address space of the child process at process creation.
- In the example program CreateProcess() function creates a child process that loads the application mspaint.exe .
- Two parameters passed to CreateProcess() are instances of the STARTUPINFO and PROCESS_INFORMATION structures.
- STARTUPINFO specifies many properties of the new process, such as window size and appearance and handles to standard input and output files.
- The PROCESS_INFORMATION structure contains a handle and the identifiers to the newly created process and its thread.

- We invoke the ZeroMemory () function to allocate memory for each of these structures before proceeding with CreateProcess().
- The first two parameters passed to CreateProcess () are the application name and command line parameters.
- If the application name is NULL (which in this case it is), the command line parameter specifies the application to load.

Process termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call(Normal termination).
- Upon termination all the resources of the process-including physical and virtual memory, open files, and I/O buffers are deallocated by the operating system.
- A process can cause the termination of another process via an appropriate system call such as abort(),kill(),etc.(Abnormal termination).
- Only parent process is allowed to terminate its child processes. This restriction is enforced to avoid user processes killing each other.

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
  - The child has exceeded its usage of some of the resources that it has been allocated.
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates(cascading termination).
- On some operating systems, If parent process terminates and child continues to execute "init" process is assigned as parent of such processes.

Inter process communication

Processes executing concurrently on operating systems are broadly classified in to two types:

1.Independent Processes:

process cannot affect or be affected by the execution of another process.

2.Cooperating Processes:

process can affect or be affected by the execution of another process.

Advantages of process cooperation

- Information sharing

  Several users may be interested in same piece of information. Hence we must provide concurrent access to such information.

- Computation speed-up

  Parallel execution of task to speedup the overall execution.

- Modularity

  System might be constructed in modular fashion.

- Convenience

  Single user working on multiple tasks.

Inter Process communication

Two fundamental models of Interprocess communication:

- **Shared memory:**

In this model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to shared region.

- **Message Passing system:**

In this model, communication takes place by means of messages exchanged between the cooperating processes.

Communication Models

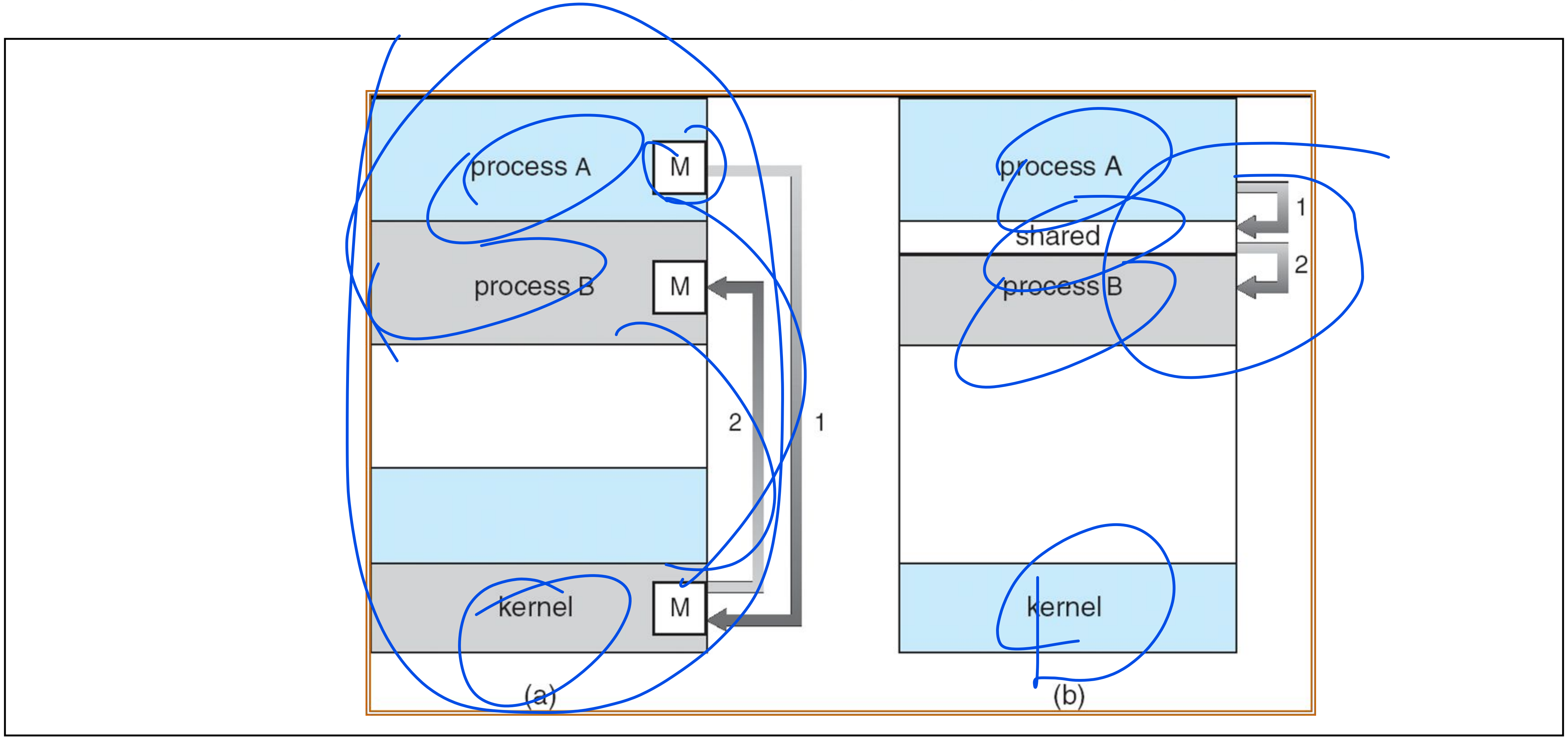


Fig: communication models (a)Message Passing (b)Shared Memory

Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
- Shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- Normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction.

- Processes can exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Example for cooperative processes

- In order to understand the concept of cooperative processes let's consider the producer-consumer problem, which is a common paradigm for cooperating processes.
- Producer process – is a process that produces information.
- consumer process- Process that consumes the information produced by the producer.
  - Example: A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
  - Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting there source.

Solution to producer consumer problem using shared memory

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

- Two types of buffers can be used:
  - The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  - The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Bounded-Buffer – Shared-Memory Solution

- Let us explore how the bounded buffer can be used to enable processes to share memory.
- The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

  - The shared buffer is implemented as a circular array with two logical pointers: in and out.
- The variable in points to the next free position in the buffer;
- out points to the first full position in the buffer.
- The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFERSIZE) == out.

The code for the producer and consumer processes is shown below

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

- The producer process has a local variable nextProduced in which the new item to be produced is stored.
- The consumer process has a local variable nextConsumed in which the item to be consumed is stored.
- This scheme allows at most BUFFERSIZE - 1 items in the buffer at the same time.

Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- Message passing facility(IPC facility)provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message sent by processes can be* either fixed size or variable size.

Message passing contd..

- If processes *P* and Q want to communicate, they must send messages to and receive messages from each other; a **communication link** must exist between them.
- Implementation of communication link can be either
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical: (we focus only on logical implementation of the link)
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Naming

- Processes that want to communicate must have a way to refer to each other.
- They can use either direct or indirect communication.
- Under **Direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- send()and receive() primitives under direct communication.
  - send(P, message) -Send a message to process P.
  - receive (Q, message) -Receive a message from process Q.

A communication link in direct communication has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.

The addressing in direct communication can be symmetric or asymmetric

- Symmetric addressing:

In this approach, both the sender process and the receiver process must name the other to communicate.

- Asymmetric addressing:

In this approach only the sender names the recipient; the recipient is not required to name the sender.

In Asymmetric addressing, the send()and receive() primitives are defined as follows:

- send(P, message) - Send a message to process P.
- receive (id, message) - Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

Drawback of symmetric and asymmetric addressing(Direct communication)

- The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

Indirect Communication

In case of indirect communication, the messages are sent to and received from mailboxes, or ports.

- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
- Each mailbox has a unique identification.
- Two processes can communicate only if the processes have a shared mailbox.
- The send() and receive() primitives are defined as follows:
  - send (A, message) -Send a message to mailbox A.
  - receive (A, message) -Receive a message from mailbox A.

The communication link in case of indirect communication has following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

- Now suppose that processes P1, P2, and *P3* all share mailbox *A.* Process P1 sends a message to *A,* while both P2 and *P3* execute a receive() from *A.* Which process will receive the message sent by *P1?* The answer depends on which of the following methods we choose:
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time to execute a receive () operation.(Allow the system to select arbitrarily which process will receive the message (that is either P2 or *P3* but not both will receive the message). The system also may define an algorithm for selecting which process will receive the message (that is, *round robin* where processes take turns receiving messages). The system may identify the receiver to the sender.)

- A mailbox may be owned either by a process or by the operating system.
- If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox).
- When a process that owns a mailbox terminates the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.
- A mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:
  - Create a new mailbox.
  - Send and receive messages through the mailbox.
  - Delete a mailbox.
- The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls

Synchronization

- Communication between processes takes place through calls to send() and receive () primitives.
- There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.
- Blocking send-The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Nonblocking send- The sending process sends the message and resumes operation.
- Blocking receive- The receiver blocks until a message is available.
- Nonblocking receive-The receiver retrieves either a valid message or a

null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity-The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- Bounded capacity- The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- Unbounded capacity- The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks,

Thank you