0

=> A process is more than the program code which is sometimes known as text section

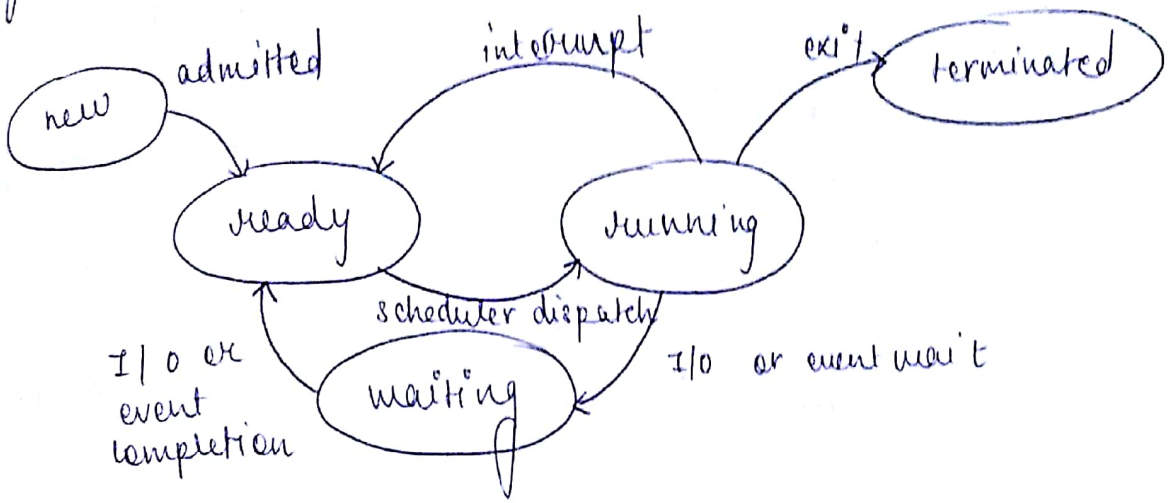
=> It also includes current activity as represented by value of program counter and contents of processor's registers

A process generally has
- process stack → it contains temporary data (like function parameters, return addresses, local variables)
- data section (contains global variables)
- heap (memory which is dynamically allocated during process run time)

⇒ process is an active entity

⇒ A program becomes a process when an executable file is loaded into memory

Diagram of process state




⇒ 2 processes may be associated with same program, they are considered 2 separate execution sequences.

For example → user may invoke many copies of Web Browser program

Each of these is a separate process; although the text sections are equivalent, the data, heap and stack sections vary.

⇒ As a process executes it changes state

(1) New : Process is created

(2) Running : Instructions are being executed

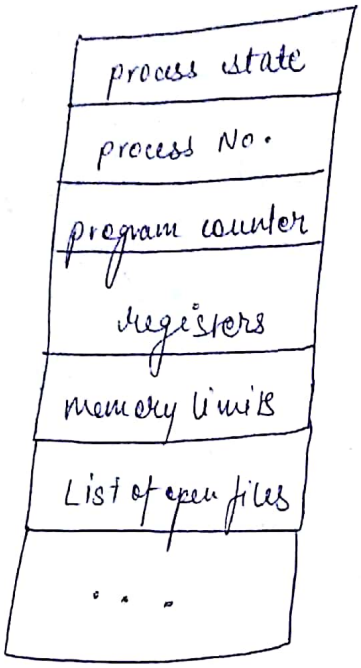
(3) Waiting : The process is waiting for some event to occur (such as I/O completion or reception of signal)

(4) Ready : The process is waiting to be assigned to a processor

(5) Terminated : The process has finished execution

⇒ Process Control Block (PCB)

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




(1) Process state : The state may be new, ready, running, waiting, halted and so on

(2) Program counter : The counter indicates the address of next instruction to be executed for this process.

(3) CPU registers : The registers vary in number and type, depending on computer architecture. They include accumulators, index registers, stack pointers, general-purpose registers plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

(4) CPU scheduling information : This information includes process priority, pointers to scheduling queues and other scheduling parameters.

(5) Memory management information: This information may include such information as the value of base and limit registers, the page tables, or segment tables, depending on memory system used by operating system

(6) Accounting information: This inf includes amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on

(7) I/O status Information: This inf includes list of I/O devices allocated to the process, a list of open files and so on

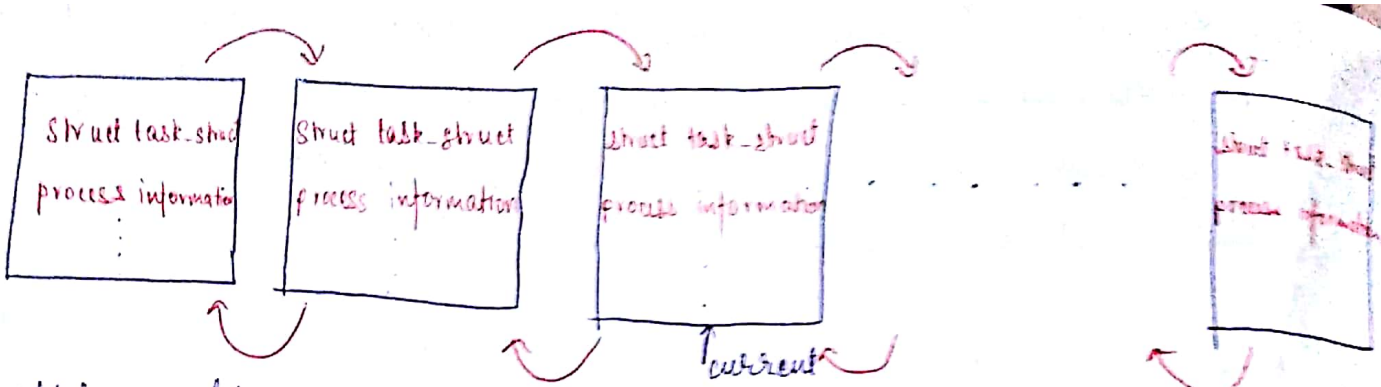
## Scheduling Queues

⇒ As the processes enter the system they are put in job queue which consists of all processes in the system

⇒ The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.

⇒ The process control block in the Linux operating system is represented by C structure task_struct.

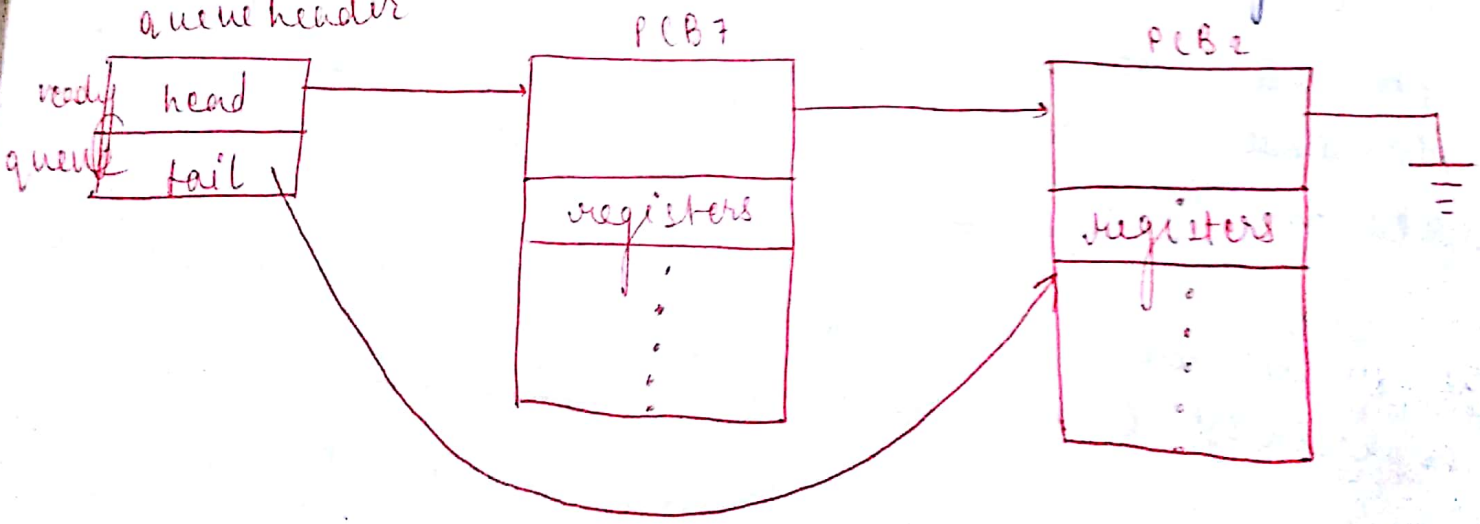
```
pid_t pid;                  /* process identifier */
long state;                 /* state of the process */
unsigned int time_slice;    /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children;  /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm;       /* address space of
                               this process */
```




Within a Linux kernel all the 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.

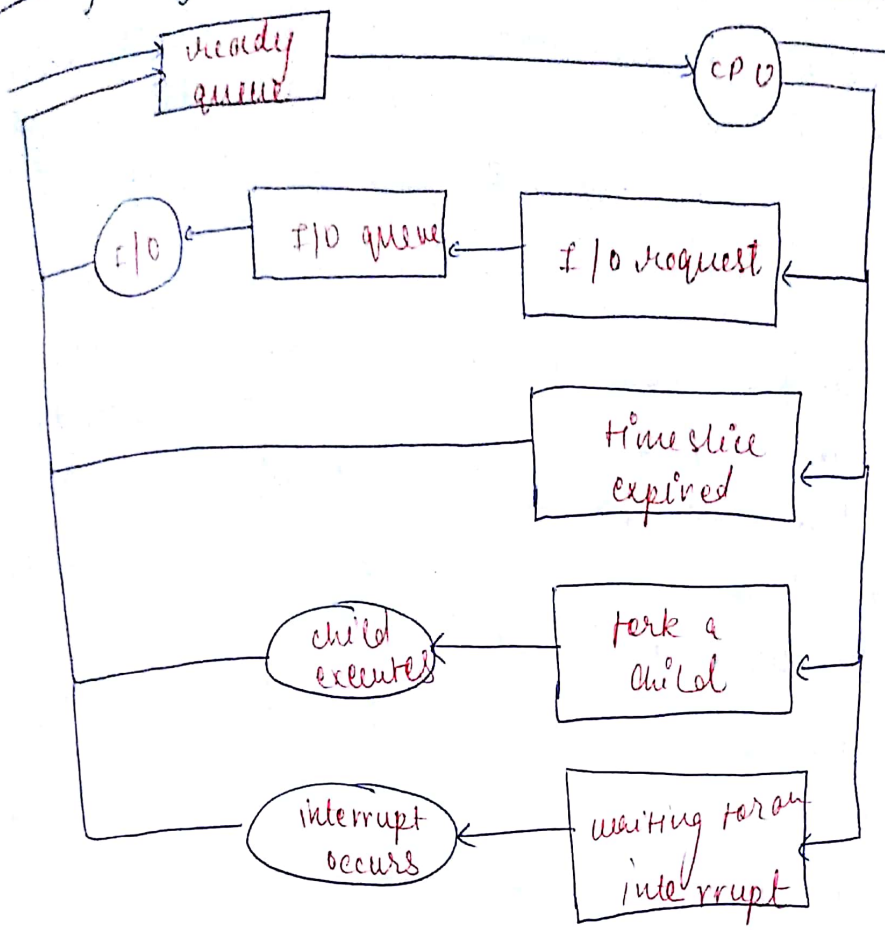
```
current -> state = new_state;
```

⇒ A ready queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to next PCB in ready queue.




⇒ The list of ~~devices~~ processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

Queuing Diagram representation of process scheduling




⇒ the selection process is carried out by appropriate scheduler.

⇒ A short term scheduler must select a new process for CPU frequently. It executes atleast once every 100 milliseconds.

⇒ Long term scheduler controls the degree of multiprogramming

⇒ Time sharing systems such as UNIX and Microsoft Windows system often have no long term scheduler.

⇒ Medium term scheduler : Some times it is advantageous to remove processes from memory and thus reduce the degree of multiprogramming. Later the process can be reintroduced in the memory and its execution can be continued where it left off. This is called swapping.

## Context Switch

⇒ Switching the CPU to another process requires performing a state save of current process and state restore of a different process. This task is known as context Switch.

⇒ When context switch occurs, the kernel saves the context of 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.

⇒ Its speed varies from machine to machine depending on —

1. memory speed
2. Number of registers that must be copied
3. Existence of special instructions.

## Process Creation

⇒ The creating process is called the parent process and the new processes are called child processes.

⇒ Most operating systems identify processes according to a unique **process identifier or (pid)** which is typically an integer number.

In Solaris the process at the top of the tree is **Sched** process with pid = 0.

The sched process creates various children processes like pageout and fsflush. These processes are necessary for managing memory and file systems.

⇒ The sched process also creates **init process** which serves as root parent process for all other processes.

2 children of init ⟶ inetd → responsible for networking services like telnet and ftp

⟶ dtlogin → process representing user login screen.

⇒ When a user logs in dtlogin creates an X windows session which in turn creates sdt-shel process

⇒ Below sdt-shel a user's command line shell the c shell or ash is created

⇒ In this command line interface the user can invoke various child processes such as ls and cat commands

---

⇒ In Unix the command ps -el will list complete information for all processes currently active in the system

⇒ When a process creates a new process, 2 possibilities exist in terms of execution

1. The parent continues to execute concurrently with its children
2. The parent waits until some or all of its children have terminated

⇒ There are 2 possibilities in terms of address space of new process

1. The child process is a duplicate of parent process
2. The child process has a new program loaded into it

# UNIX operating system

⇒ Each process is identified by a unique process identifier.

⇒ A new process is created by fork() system call.

The new process consists of a copy of address space of original process

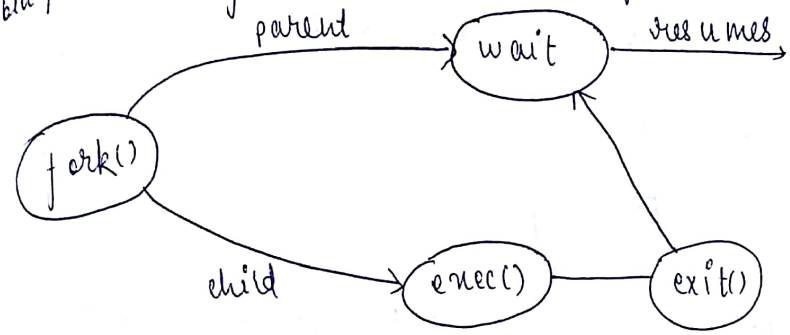
⇒ The return code for fork() is zero for new (child) process whereas (nonzero) pid of child is returned to parent

⇒ The exec() system call is used after fork() system call by one of the 2 processes to replace the process's memory space with a new program. The exec() system call loads a binary file in memory and starts its execution

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{  pid_t pid;
     pid = fork();  // fork a child process
     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;
}
```

→ the child process overlays its address space with UNIX command /bin/ls using the execlp() system call.




(Process creation using fork system call)

## Process Termination

→ A process terminates when it finishes executing its final statement and asks the OS to delete it by exit() system call.

→ The process may return a status value (an integer) to the parent process

Parent may terminate execution of one of its children due to:

① child has exceeded its usage of some of the resources.
② Task assigned to child is no longer required
③ The parent is exiting.

→ {Cascading Termination}

If parent terminates, all its children have assigned as their new parent the init process. Thus children still have a parent to collect their status and execution statistics.

# Interprocess Communication

→ Processes executing concurrently in the operating system are

{independent} {cooperating}

→ (Process cooperation) its benefits:

1. Information sharing
2. Computation speed up
3. Modularity
4. Convenience

→ Two models for Interprocess communication (IPC)

1. Shared memory
2. Message passing

## Shared Memory

A region of memory is shared by cooperating processes

Processes can exchange information by reading and writing data to shared region

→ Faster than message passing

→ Reason: Message passing is typically implemented using system calls requiring kernel intervention.

In shared memory systems, system calls are required only to establish shared memory regions. Once shared memory region is established — all accesses are treated as routine memory accesses and no assistance from kernel required!

## Message passing

Communication takes place by means of messages exchanged

→ Useful for small information

→ Easier to implement for intercomputer communication

→ 2 Types of Buffers Bounded and Unbounded Buffer.

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.

Bounded Buffer → Assumes a fixed Buffer size. Consumer must wait if buffer is empty and the producer must wait if the buffer is full.

Pg 114

⇒ If P and Q want to send and receive messages, a communication link must exist b/w them

Several methods for logically implementing a link:

1. Direct or Indirect Communication
2. Synchronous or asynchronous communication
3. Automatic or explicit buffering.

Direct communication

- send (P, message) → Send a message to process P
- receive (Q, message) → Receive a message from process Q

This scheme exhibits symmetry in addressing

Asymmetry:

- send (P, message)
- receive (id, message) → Receives a message from any process. id → {with which communication has taken place earlier}

Disadvantage → Limited modularity of resulting process definition

Indirect Communication

⇒ Messages are sent to and received from mail boxes or ports.

⇒ A mail box can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

⇒ A process can communicate with some other process via no. of different mailboxes.
2 processes can communicate only if the processes have a shared mailbox.

⇒ A mailbox is owned either by a process or the operating system

## Synchronization

Message passing may either be blocking or non-blocking ~~also~~ also known as synchronous and asynchronous.

① Blocking send : The sending process is blocked until the message is received by receiving process or by the mailbox.

② Non blocking send : The sending process sends the message and resumes operation.

③ Blocking receive : The receiver blocks until a message is available.

④ Non blocking receive : The receiver ~~blocks until a message is available~~ retrieves either a valid message or a NULL.