22/4

UNIT-2

Process Management

Process Concept

- A batch system executes jobs, whereas a time-shared system has user programs or tasks.
- Even on a single-user system, a user may be able to run several programs at one time: a word processor, a web browser and an e-mail package.
- The terms job and process are used almost interchangeably in this text.
- Although we personally prefer the term process, much of OS theory and terminology was developed during a time when the major activity of OS was job processing.

→ process has superseded job.

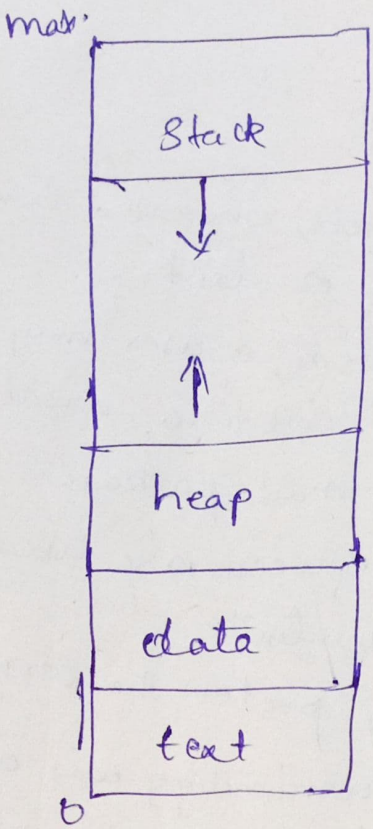
→ A process is a program in execution

→ A process is more than the program code, which is sometimes known as the text section.

→ It also includes the

→ A process generally also includes the process stack, which contain temporary data and 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.



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

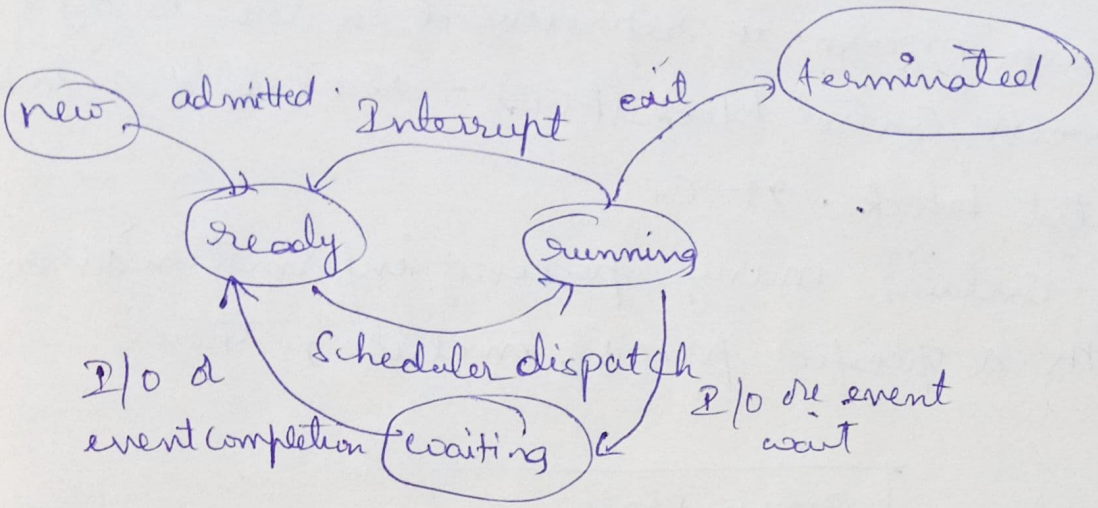
→ A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

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

→ Two common techniques for loading executable files are

* double clicking an icon representing the executable file
* entering the name of the executable file on the command line
(as in prog.exe or a.out)

process state diagram



→ As a process executes, it changes state.
→ The state of a process is defined in part by the current activity of that process.
→ A process may be in one of the following states.

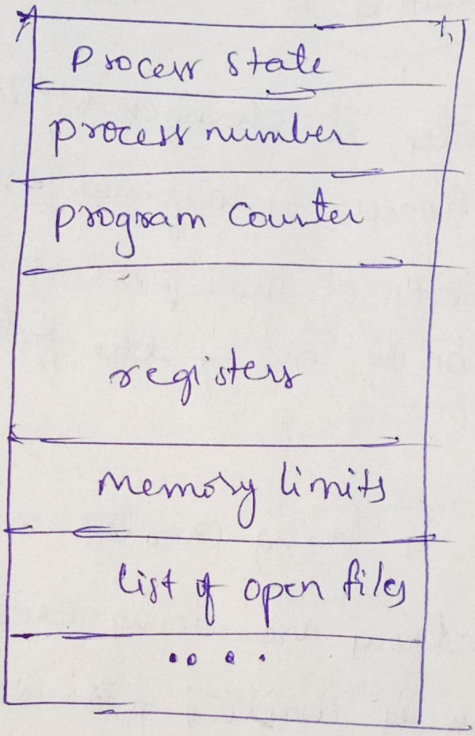
New: The process is being created.
Running: Instructions are being executed.
Waiting: The process is waiting for some event to occur.
Ready: The process is waiting to be assigned to a processor.
Terminated: The process has finished execution.

→ The states that they represent are found on all system.
→ It is important to realize that only one process can be running on any processor at any instant.
→ Many processes may be ready and waiting

Process Control block 23/4

Each process is represented in the OS by a process control block (PCB) - also called a task control block. It

It contains many pieces of information associated with a specific process, including these:



Process state : The state may be new, ready, running, waiting, halted and so on.

program Counter : the counter indicates the address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type depending on the computer architecture. They include accumulators, index registers, stack pointers and general purpose registers, plus any Condition-Code information

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

memory-management information: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the OS.

Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.

I/O status information: This information includes the list of the I/O devices allocated to the process, a list of open files and so on.

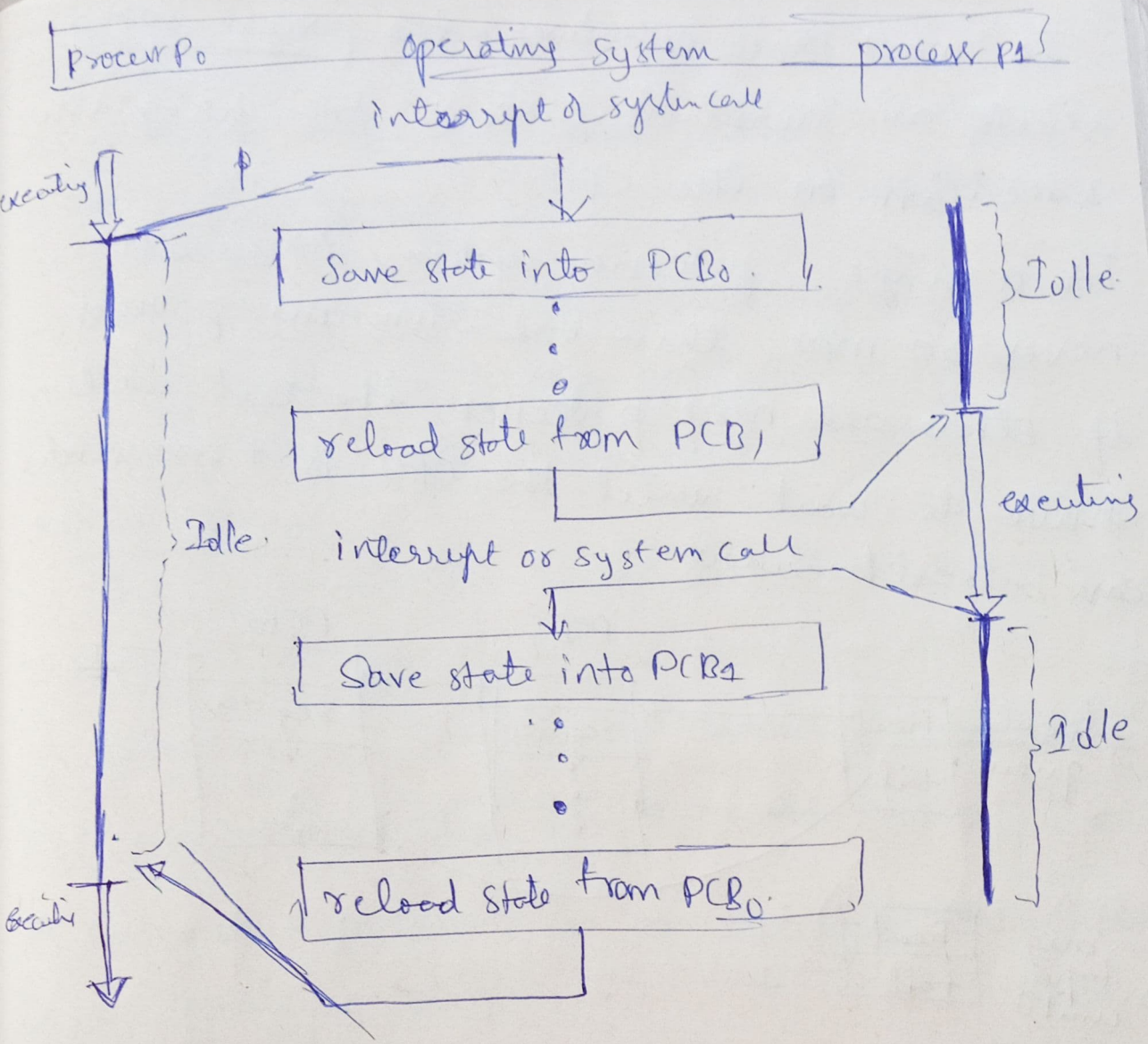
In brief, the PCB simply serves as the repository for any information that may vary from process to process.

Threads

A process is a program that performs a single thread of execution

For example, when a process is running a word-processor program, a single thread of instructions is being executed.

→ This single thread of control allows the process to perform only one task at a time



Process Scheduling

-) The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. ~~The objective of time sharing~~

-) The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

Scheduling Queues

As processes enter the system, they are put into a **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 system also includes other queues.

→ The list of processes waiting for a particular I/O device is called **device queue**

→ Each device has its own device queue

A common representation of process ~~sc~~ scheduling is a queueing diagram

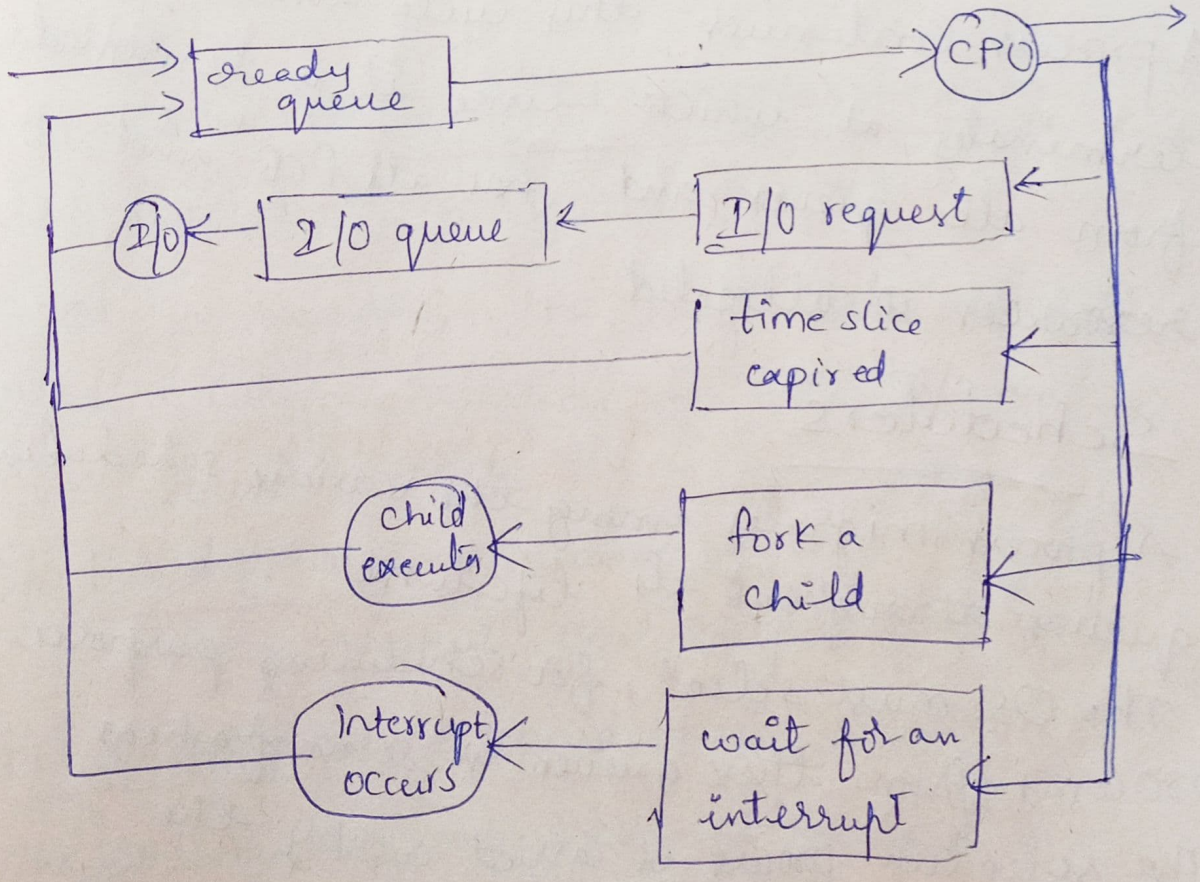


Fig:- Queueing-diagram representation of process scheduling

* Each rectangular box represents a queue.
* Circles represent the resources that serve the queues
* Arrows indicate the flow of processes in the syst.

Once the process is allocated the CPU and is executing, one of several events could occur.

* The process could issue an I/O request and then be placed in an I/O queue.
* The process could create a new child process and wait for the child's termination
* The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

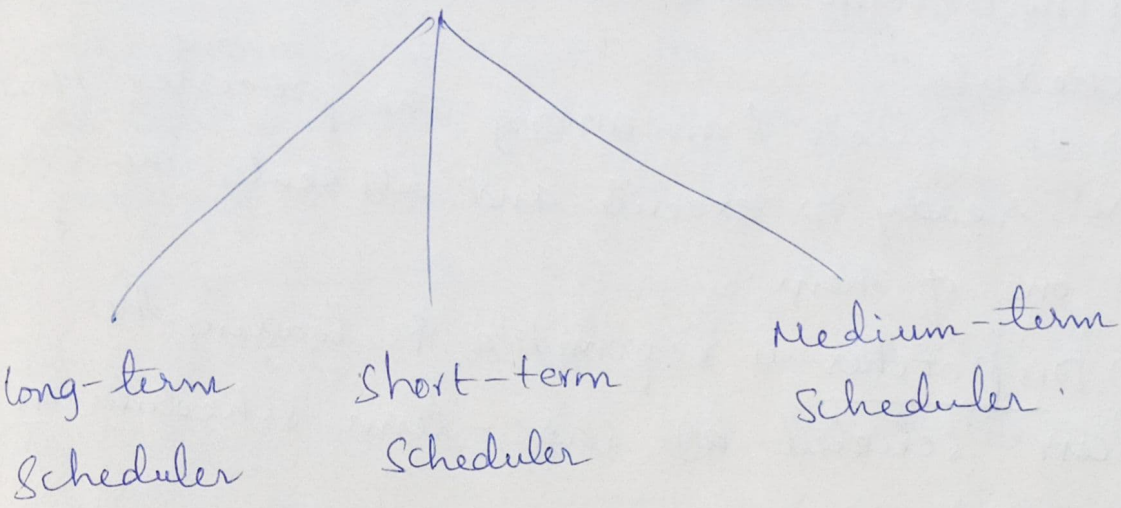
A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated

Schedulers

A process migrates among the various scheduling queues throughout its lifetime

The OS must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

Schedulers



Long-term scheduler

-) Long-term scheduler are also called as "job scheduler".

-) It selects processes from this pool and loads them into memory execution.

-) It brings the new process to the 'Ready State'.

-) It Controls Degree of Multiprogramming (i.e., number of processes in memory)

-) It is important that long-term scheduler make a careful selection of both I/O and CPU bound process.

I/O bound process - Use much of their time in input and output operations

CPU bound process - spend their time on CPU.

-) The job scheduler increases efficiency by maintaining a balance b/w the two.

Short-term scheduler

→ Short-term scheduler also called as "CPU Scheduler"
→ It selects from among the processes that are ready to execute and allocates the CPU to one of them.
→ Dispatcher is responsible for loading the process selected by short-term scheduler on the CPU.
→ Context switching is done by dispatcher only.

Medium-term scheduler

→ Medium-term scheduler also called as "Swapper"
→ It is responsible for suspending and resuming the process.
→ It mainly does swapping (moving processes from main memory to disk and vice versa.
→ Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

Context switch 26/4

When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore the 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.

Generically, 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.

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 Context switch.

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.

→ Content - switch times are highly dependent on hardware support

→ A content switch here simply requires changing the pointer to the current register set.

Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory as before. Also, the more complex the operating system, the greater the amount of work that must be done during a content switch.

OPERATIONS ON PROCESSES

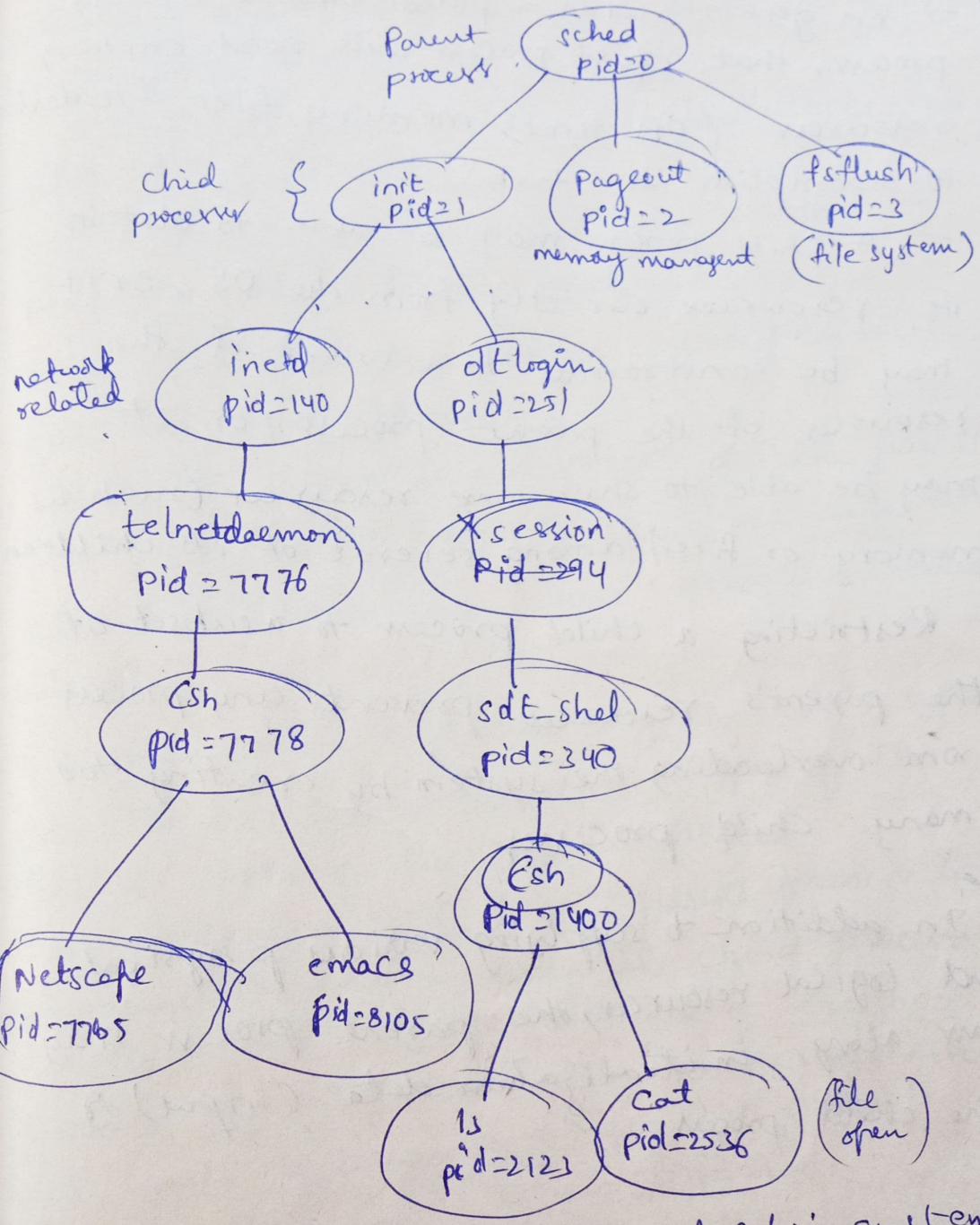
→ Process creation

→ Process Termination

Process Creation

A process may create several new processes via, a create-process system call, during the course of execution. The creating process is called a parent process and the new processes are called the children of that process.

Each of these new pro-



A tree of processes on a typical solaris system.

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

→ In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task

→ A child process may be able to obtain its resources directly from the OS, or it may be constrained to a subset of the resources of the parent process; or it may be able to share some resources (such as memory or files) among several of its children

- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes

- In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process

When a process creates a new process, two possibilities for execution exist.

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

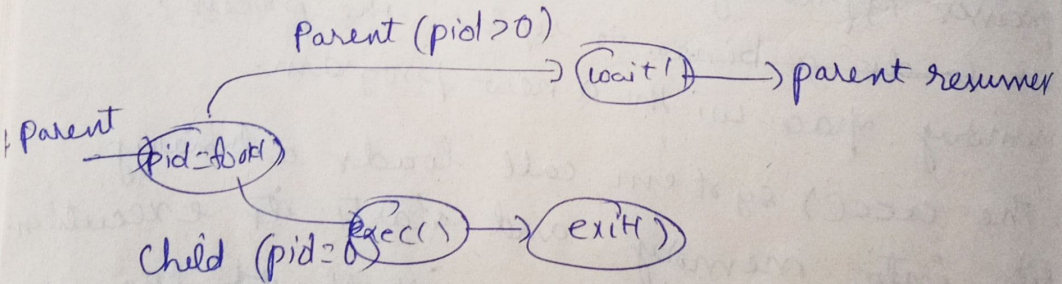
There are also two address-space possibilities for the new process.

1. The child process is a duplicate of the parent process (it has the same program and data as the parent.
2. The child process has a new program loaded into it

→ A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. After a fork() system call, one of the two processes typically uses the exec() system call ~~load a binary~~ to replace the process's memory space with a new program.

The exec() system call loads a binary file into memory and starts its execution. wait() system call to move itself off the ready queue until the termination of the child.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
 pid_t pid;
      pid = fork( )
      if (pid < 0){
        fprintf (stderr, "Fork failed");
        return 1;
      }
      else if (pid == 0)
      {
          execlp("/bin/ls", "ls", NULL);
      }
      else
      {
         wait (NULL);
         printf ("Child complete");
      }
      return 0;
}
```



process creation using the fork( ) system call

Process Termination

A process terminates when it finishes executing its final statement and asks the OS to delete it by using the exit() system call.
At that point, the process may return a status value (typically an integer) to its parent.

~~Termination can occur~~
A process can cause the termination of another process via an appropriate system call.
A parent may terminate the execution of one of its children for a variety of reasons, such as these:
→ the child has exceeded its useage 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 OS does not allow a child to continue if its parent terminates.

Note that a parent needs to know the identities of its children if it is to terminate them.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the

O.S

We can terminate a process by using the exit() system call, providing an exit status as a param

```
/* exit with status 1 */
exit(1);
```

The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child

```
pid_t pid;
int status;
pid = wait(&status);
```

A process that has terminated, but whose parent has not yet called wait(), is known as zombie process.

The init process periodically invokes wait() thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table ~~entry~~ entry.

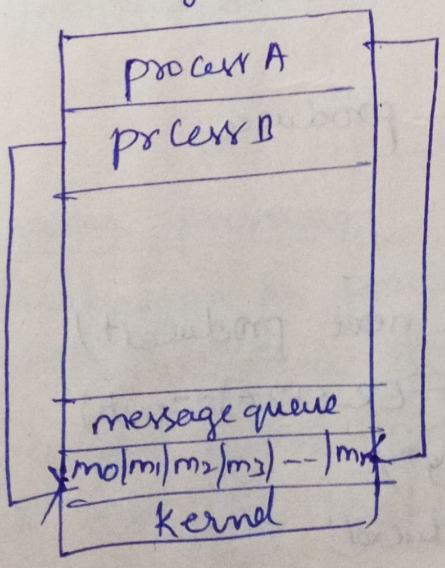
27/4

Interprocess Communication

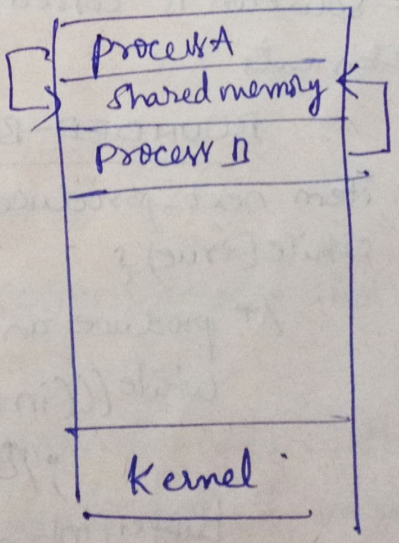
→ Processes within a system may be independent or cooperating.

- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating process:
  - Information sharing.
  - Computation speedup.
  - Modularity
  - Convenience
- Cooperating process need interprocess communication (IPC)
- Two models of IPC
  - shared memory
  - message passing.

(a) Message passing



(b) Shared memory



Producer – consumer problem

- Paradigm for cooperating processes, producer process produce 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

```
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 */
}
```

Inter process Communication shared memory.

- An area of memory shared among the processes that wish to communication.
- The communication is under the control of the users processes not the operating system
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory
- Synchronization is discussed

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)
  - receive (message)

- The message size is either fixed & variable.

- If processor P and Q wish to communicate, they need to:
  * Establish a communication link b/w them
  * Exchange messages via send / receive.

* Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be b/w every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

* Implementation of communication link
  * Physical:
    - → Shared memory
    - → Hardware bus
    - → Network.
  * Logical:
    - → Direct or indirect
    - → Synchronous or asynchronous
    - → Automatic or explicit buffering.

Direct Communication

- process 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 & bidirectional

* Operations
  - Create a new mail box (port)
  - send and receive messages through mail box
  - destroy a mail box.

* Primitives are defined as
  send (A, message) - send a message to mailbox A
  receive (A, message) - receive a message from mail box A

* Mail box sharing
  - ~~P1, P2~~, P_1, P_2 and P_3 share mail box A.
    - P_2 sends; P_2 and P_3 receive
    - who gets the message?

* Solutions
  - Allows 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 arbitarily 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 - the sender is blocked until the message is received.

Blocking receive – the receiver is blocked until a message is available.

- Non blocking is considered asynchronous.
  - Non blocking send -- the sender sends the message and continue
  - Non-blocking receive - the receiver receives:
    - A valid message, or
    - Null message.
- Different combination possible.
  - If both send and receive are blocking we have a rendezvous.

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways:

1. Zero capacity - no messages are queued on a link

   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