Process

Process is a program in execution.

Program by itself is not a process.

A program is a **passive entity**, such as the contents of the file stored on the disk, whereas a process is an **active entity**, with a program counter specifying the next instruction to execute and a set of associated resources.

Process is more than the program code, which is sometimes known as text section.

It also includes the current activity, as represented by the value of program counter and the contents of the processor's registers.

Additionally a process generally includes the process stack containing temporary data (like method parameters, return addresses, and local variables), and a data section containing global variables.

Process State

As a process executes, it changes state.

The state of process is defined in part by 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 an I/O completion or reception of a signal).

Ready: The process is waiting to be assigned to a processor.

Terminated: The process has finished execution.

Only one process can be running on any processor at any time instant, although many processes may be ready and waiting.

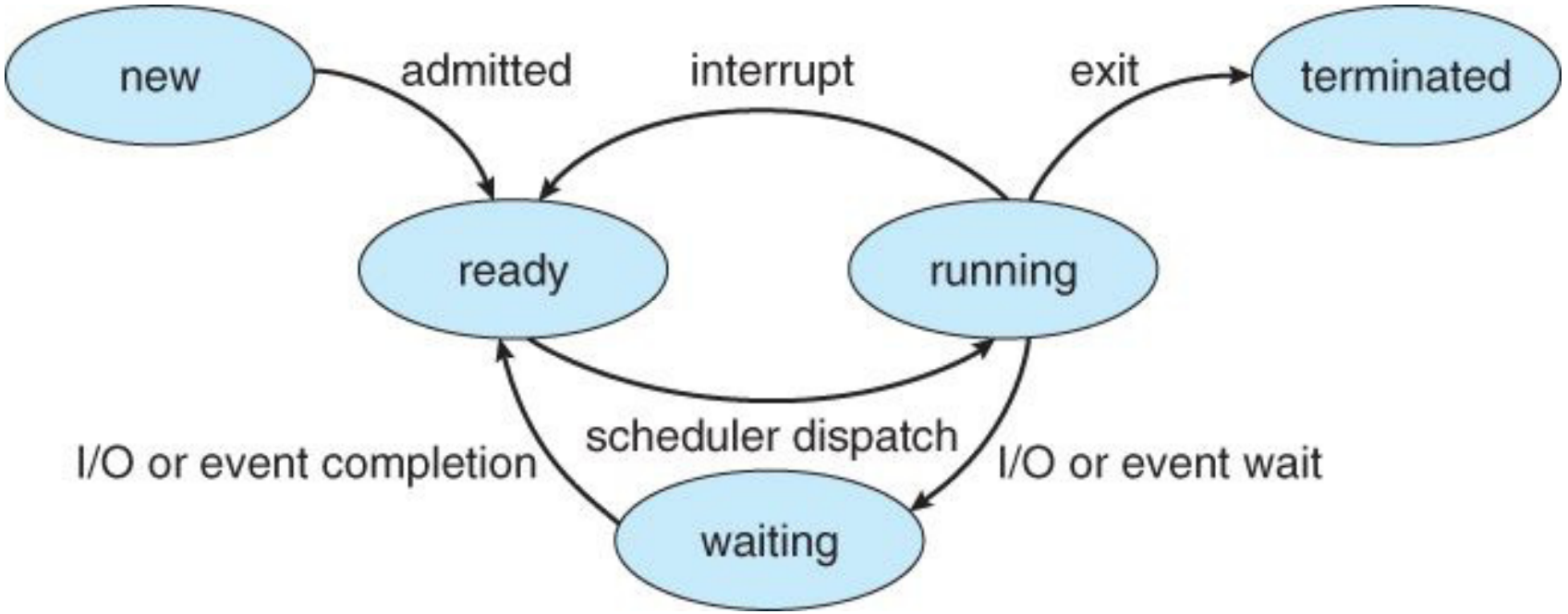

Fig: Diagram of process states.

Process Control Block (PCB)

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

PCB contains information associated with a specific process.

Fig: Process Control Block (PCB)

PCB includes following:

(i) Process State:

It includes present state of the process, i.e., new , ready, running, waiting, terminated, and so on.

(ii) Program Counter:

The program counter indicates the address of the next instruction to be executed for the process.

(iii) CPU Register:

The CPU register may vary in number and type, depending on the computer architecture.

They include accumulators, index-registers, stack pointers, general purpose registers, condition-code information.

Along with program counter, this state information must be saved when an interrupt occurs, so that the process to be continued correctly afterward.

(iv) CPU-Scheduling Information:

This information includes a process priority, pointer to scheduling queues, and any other scheduling parameters.

(v) Memory Management Information

This information may include value of the base and limit registers, page tables, segment tables depending on the memory segment used by the operating system.

(vi) Accounting Information:

This information includes the amount of CPU and real time used, time limits, process numbers and so on.

(vii) I/O Status Information

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

Process Scheduling

The objective of multiprogramming is to have some process running at all times to maximize the CPU utilization.

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

An uniprocessor system can have only one running process.

If more processes exist, then the rest must wait until the CPU is free and can be rescheduled.

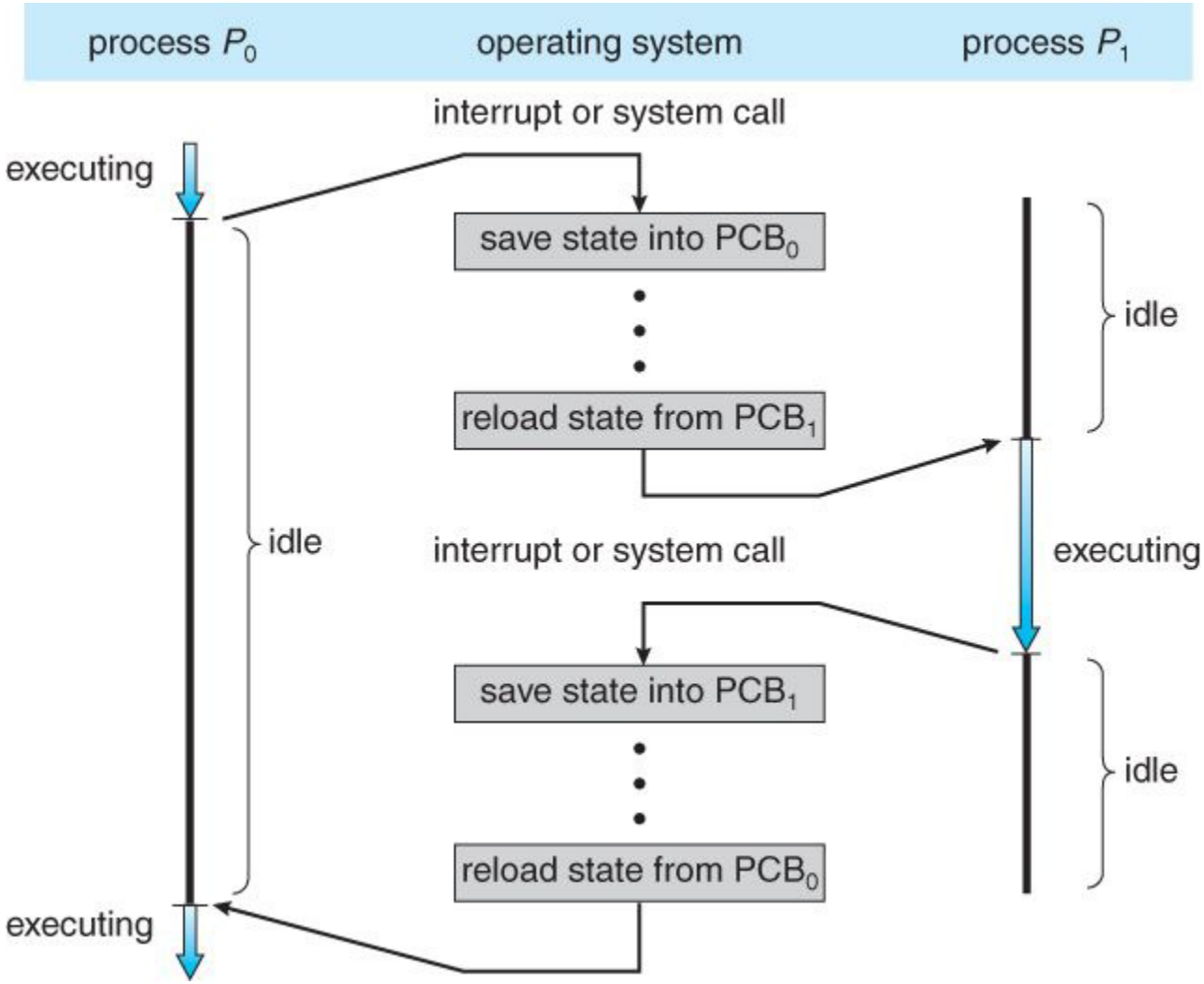

Fig: Diagram showing CPU switch from process to process.

Scheduling Queue

As a process enters the system, they are put into a **job queue**.

This queue consists of all process in the system.

The process that are residing in main memory and are ready and waiting to execute are kept on a list called **ready queue**.

This queue is generally stored as a linked list.

A ready queue header contains pointers to the first and final PCBs in the list.

When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.

The list of process waiting for a particular I/O device is called **device queue**. Each device has its own device queue.

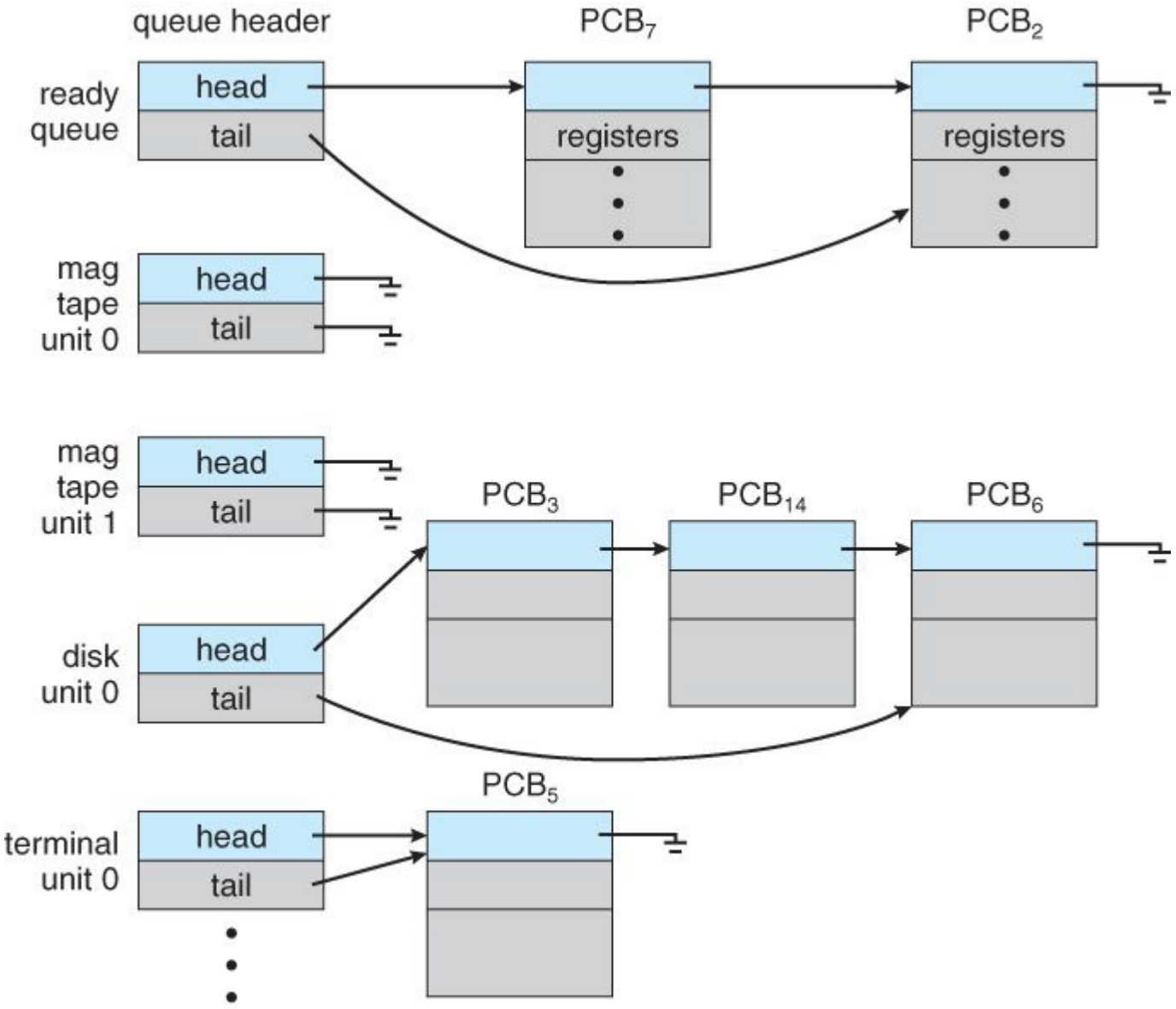

Fig: Ready queue and Device queue.

The process scheduling can be represented by queueing diagram.

Once a new process is initially put into the ready queue, it waits in the ready queue until it is selected for execution (or dispatched).

Once the process is assigned to the CPU and is executing, one of the several events could occur:

i) The process could issue an I/O request, and then be placed in an I/O queue.

ii) The process could create a new sub process and wait for its termination.

iii) The process could be removed forcibly from the CPU, as a result of an interrupt, and put back in the ready queue.

A process continues this cycle until it terminates.

After termination, the process is removed from all queues, and de-allocated its PCB and resources.

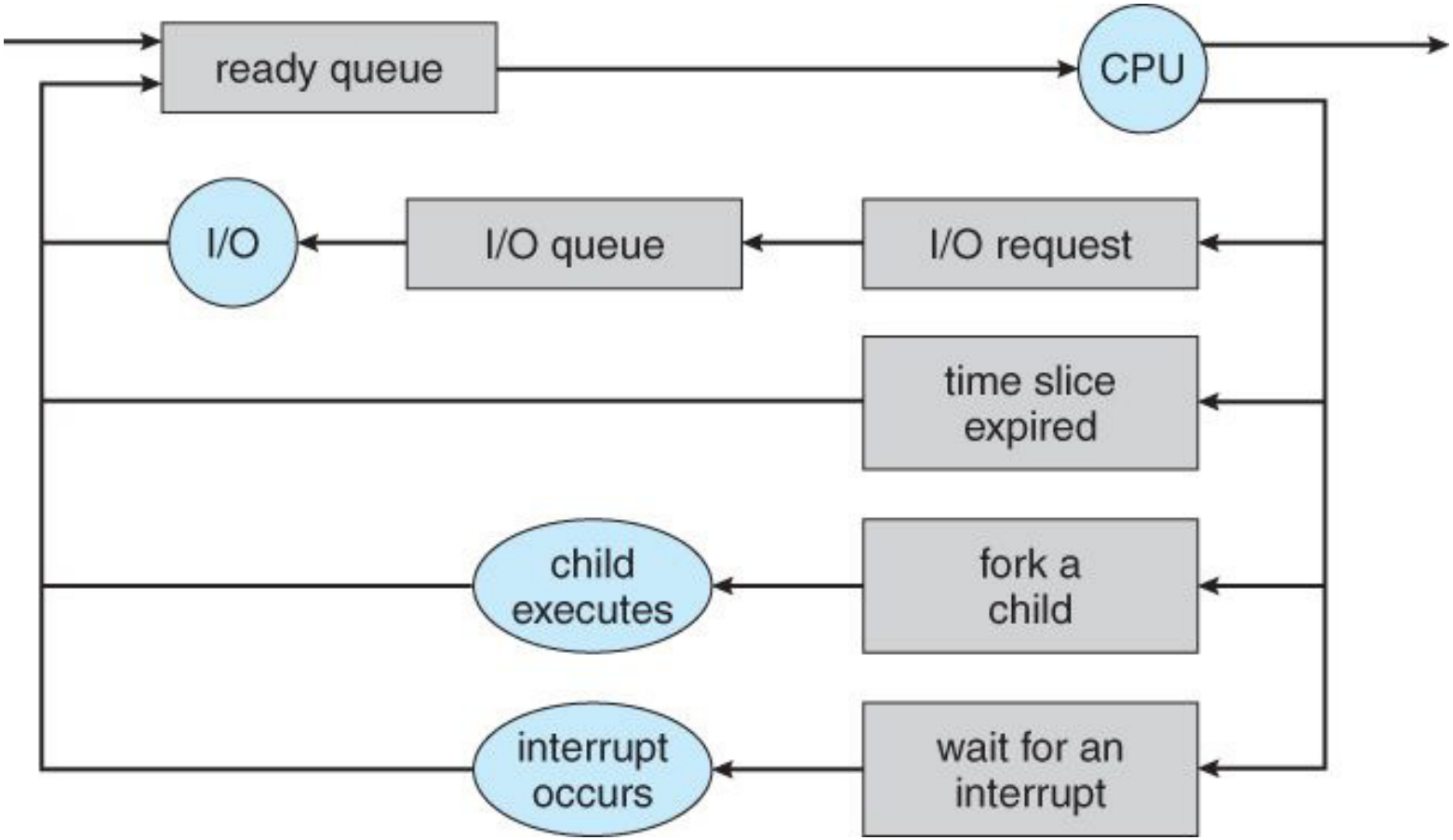

Fig: Queueing Diagram representation of process scheduling.

Schedulers

A process migrates between the various scheduling queues throughout its lifetime.

The selection process is carried out by the appropriate scheduler.

In a batch system, often more processes are submitted than can be executed immediately.

These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

The **long-term scheduler** (**Job scheduler**), select processes from this pool and loads them into memory for execution.

The **short-term scheduler** (**CPU scheduler**), selects from among the processes that are ready to execute, and allocates the CPU to one of them.

The primary distinction between these two schedulers is the frequency of their execution.

The short-term scheduler must select a new process for the CPU frequently.

On the other hand, the long-term scheduler executes much less frequently.

It controls the degree of multiprogramming – the number of process in memory.

The long-term scheduler must take a careful selection.

Generally, most processes can be described as either I/O bound or CPU bound.

An I/O bound process spends more of its time doing I/O than it spends doing computations.

Similarly, the CPU bound process generates I/O requests infrequently and using more of its time doing computation than an I/O bound process uses.

The long-term scheduler should select a good process mix of I/O bound and CPU bound process.

If all processes are I/O bound, then the ready queue will almost be empty, and short-term scheduler will have little to do.

Similarly, if all processes are CPU bound, then I/O waiting queue will almost always be empty, and long-term scheduler will have little to do, and the system will be unbalanced.

The system with best performance will have a combination of CPU-bound and I/O-bound process.

Some operating system, such as time-sharing systems, may introduce an additional intermediate level of scheduling called **medium-term scheduler**.

It removes processes from memory, and thus reduces the degree of multiprogramming.

At some later time, the process can be reintroduced into memory and its execution can be continued where it left off.

This scheme is called **swapping**.

The process is swapped out, and later it is swapped in, by the medium term scheduler.

Swapping is necessary to improve process mix, as a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

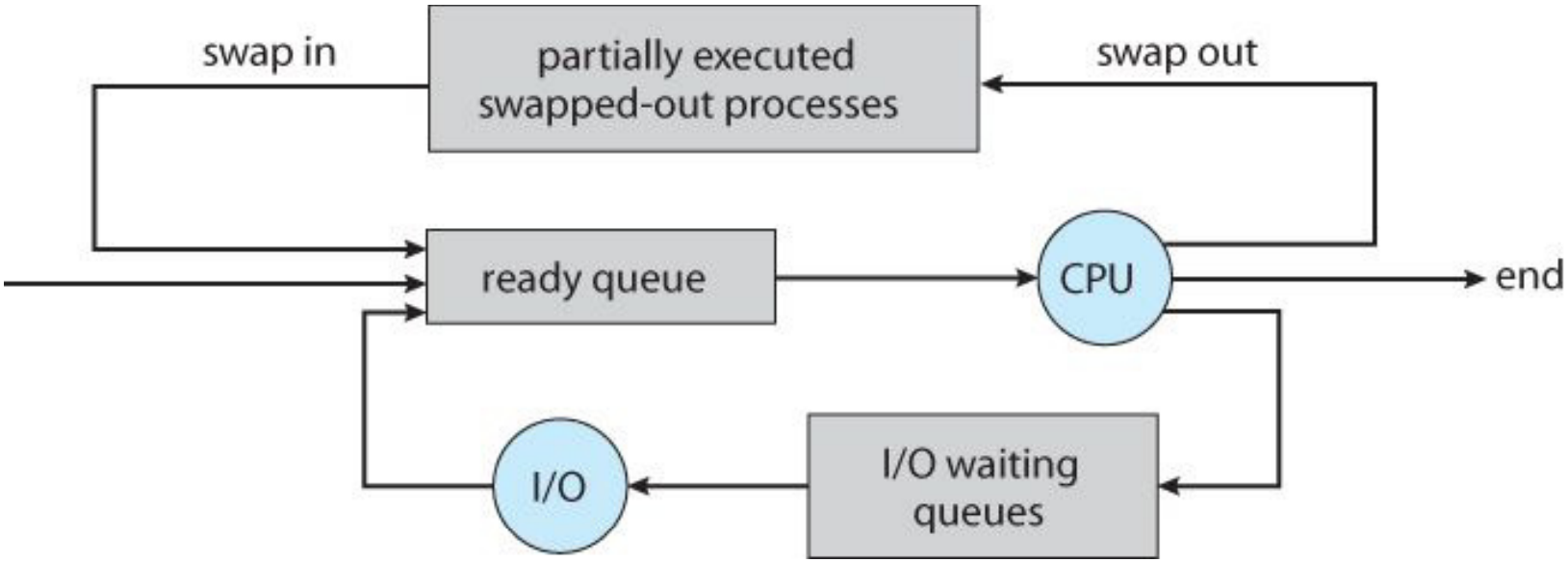

Fig: Addition of medium-term scheduling to the queueing diagram.

Context switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process.

This task is known as context switch.

The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, process state and memory management information.

When context switch occurs, the operating system 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, as the system does no useful work while switching.

Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, existence of special instructions.

The context switch times are highly dependent on hardware support.

Operation on processes

The process in the system can execute concurrently, and they must be created and deleted dynamically.

So, operating system provides a mechanism for process creation and termination.

Process Creation

A process may create several new processes during its execution.

The creating process is called parent process and the new processes are called children of that process.

Each of these new processes may in turn create other processes, forming a *tree* of processes.

Generally a process will need certain resources (like CPU time, memory, files, I/o devices) to complete its task.

When a process creates a sub-process, that sub-process may be able to obtain it resources directly from the operating system, or it may be constrained to a subset of resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (like memory or files) among several of its children.

When a process creates a new process two possibilities exist in terms of execution:

(i) The parent continues to execute concurrently with its children.

(ii) The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

(i) The child process is a duplicate of the parent process.

(ii) The child process has a program loaded into it.

In UNIX, each process is identified by its process identifier, which is a unique integer.

New process is created by the *fork* system call.

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

Therefore, the parent process can communicate easily with its child process.

Both parent and child processes continue execution at the instruction after the fork system call.

But there is one difference, i.e., the return code for the fork system call is zero for the child process, whereas the nonzero process identifier of the child is returned to the parent.

The *execlp* system call is used after a fork system call by one of the two processes to replace the process memory space with a new program.

The execlp system call loads a binary file into memory and starts its execution.

In this manner, the two processes are able to communicate, and then to go their separate ways.

If the parent process has nothing else to do while the child runs, it can issue a *wait* system call to move itself off the ready queue until the termination of the child.

```
#include<stdio.h>
void main( int argc, char * argv[])
{ int pid;
  /* fork another process */
  pid=fork();
  if( pid < 0)        /* error occurred */
  { printf("\n Fork Failed");
    exit(-1);
   }
  else if (pid = = 0)          /* child process */
  { execlp("/bin/ls", "ls",NULL);
  }
else
{ /* parent process */
  wait(NULL);    /* parent will wait for the child to complete */
  printf("\n Child complete");
  exit(0);
 }
```

C program forking a separate process.

In this program parent process creates a child process using the *fork* system call.

Now two different processes running a copy of the same program.

The value of pid for the child process is zero; that for the parent is an integer value greater than zero.

The child process overlays its address space with the UNIX command /bin/ls (used to get directory listing) using the *execlp* system call.

The parent waits for the child process to complete with the *wait* system call.

When the child process completes, the parent process resumes from the call to wait where it completes using the exit system call.

Process termination

A process terminates when it finishes executing its final statement and asks the operating to delete it by using the *exit* system call.

At the point, the process may return data (output) to its parent process (via wait system call).

All the resources of the process are deallocated by the operating system.

A process can cause the termination of another process via an appropriate system call (like *abort*).

Only the parent of the process that is to be terminated can invoke such a system call.

Otherwise, users could arbitrarily kill each others job.

A parent may terminate the execution of one of its children for variety of reasons, such as:

i) The child has exceeded its usage of some of the resources that it has been allocated.

ii) The task assigned to the child is no longer required.

iii) The parent is terminated, and the operating system does not allow a child to continue.

If a process terminates (either normally or abnormally), then all its children also terminated by the operating system.

Such incident is known as **cascading termination**.

Cooperating processes

A process is called **independent process** if it cannot affect or be affected by the other processes executing in the system.

A process is called **cooperating process** if it can affect or be affected by the other processes executing in the system.

Process cooperation may be required for several reasons:

i) Information sharing:

Since several users may be interested for same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these type of resources.

ii) Computation speedup:

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

Such a speedup can be achieved only if the computer has multiple processing elements (such as CPU or I/O channels).

iii) Modularity:

We may construct the system in a modular fashion by dividing the system function into separate processes or threads.

iv) Convenience:

Even an individual user may have many tasks on which to work at one time.

For example, a user may be editing, printing and compiling in parallel.

Concurrent execution of cooperating processes requires mechanisms that allow processes to communicate with one another and to synchronize their actions.

The **producer-consumer problem** is a common example of cooperating process.

A producer process produces information that is consumed by a consumer process.

For example, a print program produces characters that are consumed by the printer driver.

To allow producer and consumer process to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

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.

In this situation, the consumer must wait until an item is produced.

The bounded buffer producer consumer problem 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.

The buffer may either be provided by the operating system through the use of an inter process communication (IPC) facility, or by explicitly coded by the application programmer with the use of shared memory.

Let consider the shared memory solution to the bounded buffer problem.

Producer and consumer processes share following variables:

```
#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)% BUFFER_SIZE)=*out*.

The code for producer process is as follows:

```
while(1)
{   while ((in+1)% BUFFER_SIZE) = =out);  /*do nothing*/

    buffer[in]=nextProduced;              /* produce an item in nextProduced  */
    in=(in+1)% BUFFER_SIZE;
}
```

The code for consumer process is as follows:

```
while(1)
{   while (in ==out);  /*do nothing*/

     nextConsumed=buffer[out];          /* consume an item in nextConsumed  */
    out=(out+1)% BUFFER_SIZE;
}
```

This scheme allows at most BUFFER_SIZE-1 items in the buffer at the same time.

Inter-processes communication

Cooperating process can communicate in a shared memory environment.

Another way by which cooperating processes can communicate with each other is inter process communication facility (IPC), provided by the operating system.

IPC provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space.

IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network.

For example, the chat program used on the world wide web.

IPC is provided by a message passing system which can be defined in many ways.

Message passing system

The function of a message passing system is to allow processes to communicate with one another without the need to resort to shared data.

An IPC facility provides at least the two operations: *send* (message) and *receive* (message).

Message sent by a process can be either fixed or variable size.

If the fixed sized messages can be sent, the system level implementation is straightforward.

This restriction makes the task of programming more difficult.

Similarly, the variable sized messages require a more complex system-level implementation, but the programming task become simpler.

If two processes *P* and *Q* want to communicate, they must send messages and receive messages from each other; a communication link must exist between them.

There are several methods for logically implementing a link and send/receive operations:

i) Direct or indirect communication

ii) Symmetric or asymmetric communication

iii) Send by copy or send by reference.

iv) Fixed-sized or variable-sized messages.

Naming

Processes that want to communicate must have a way to refer to each other.

They can use either direct or indirect communication.

Direct Communication

Each process that wants to communicate with direct communication, must explicitly name the recipient or sender of the communication.

In this scheme, the send and receive methods are defined as follows:

send (*P*, message): Send a message to process *P*.

receive (*Q*, message): Receive a message from process *Q*.

The communication for this scheme has the following properties:

i) A link is established between every pair of processes that want to communicate. The processes need each other's identity to communicate.

ii) A link is associated with exactly two processes.

iii) Exactly one link exists between each pair of processes.

The disadvantage of this scheme is that changing the name of process may necessitate examining all other process definitions.

All reference to the old name must be found, so that they can be modified to the new name.

Indirect Communication

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

Each mailbox has a unique identification.

In this scheme, a process can communicate with some other process via a number of different mailboxes.

Two processes can communicate if they share a mailbox.

send (*A*, message): Send a message to mailbox *A*.

receive (*A*, message): Receive a message from mailbox *A*.

Here the communication link has following properties:

i) A link is established between a pair of processes only if both members of the pair have a share mailbox.

ii) A link may be associated with more than two processes.

iii) A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Let there are three processes P_1, P_2 and P_3 share mailbox A.

Process P_1 sends a message to A; while P_2 and P_3 each execute a *receive* from A.

Now the process that should receive the message send by P_1 depends on the following schemes that we choose:

i) Allow a link to be associated with at most two processes.

ii) Allow at most one process at a time to execute a *receive* operation.

iii) Allow the system to select arbitrarily which process will receive the message (i.e., either P_2 and P_3, but not both, will receive the message). The system may identify the receiver to the sender.

A mailbox may be owned either by a process or by the operating system.

Synchronization

Communication between processes takes place by calls to send and receive methods.

There are different design options for implementing each primitive.

The message passing may be either blocking (synchronous) or non-blocking (asynchronous).

Blocking send:

The sending process is blocked until the message is received by the receiving process or mailbox.

Non-blocking send:

The sending process sends the message and resumes operation.

Blocking receive:

The receiver blocks until a message is available.

Non-blocking receive:

The receiver retrieves either a valid message or a null.

Buffering

Whether the communication is direct or indirect, message exchanged by communicating process reside in a temporary queue.

This type of queue can be implemented in the following three ways:

i) Zero capacity:

The queue has maximum length 0; thus the link cannot have any messages waiting in it.

In this case the sender must block until recipient receives the message.

ii) Bounded capacity:

The queue has finite length *n* and at most *n* messages can reside in it.

If the queue is not full when a new message is sent, the later is placed in the queue, and the sender can continue execution without waiting.

The link has a finite capacity.

If the link is full, the sender must block until space is available in the queue.

iii) Unbounded capacity:

The queue has potentially infinite length; thus any number of messages can wait in it.

The sender never blocks.

The zero capacity case is also known as message system with no buffering.

The other cases are referred to as automatic buffering.