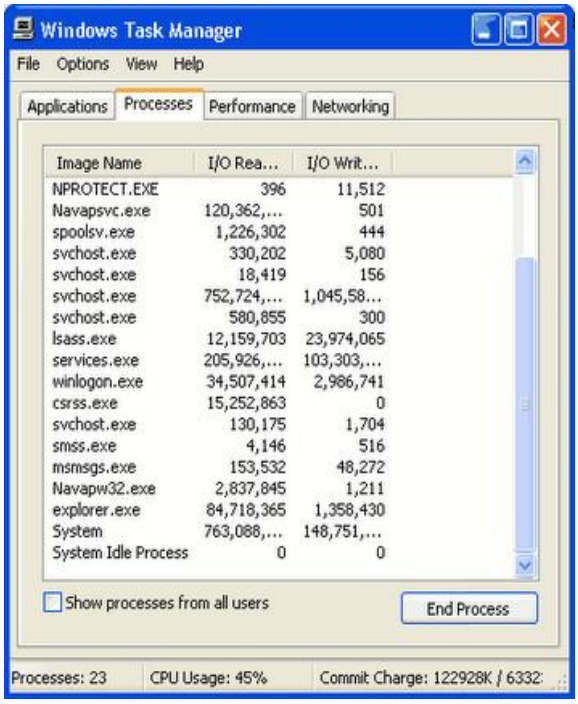
1. Concept of Process

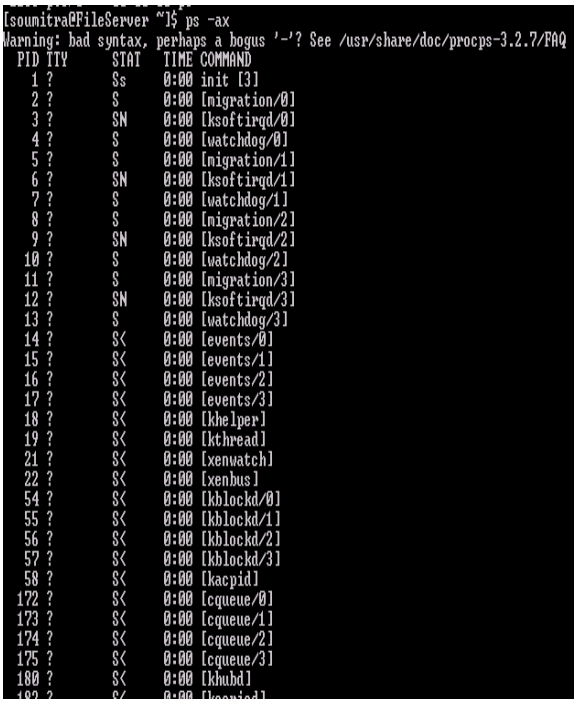
A process is a sequential program in execution. In simple words we can say that a process is a program that is running on our computer. Now this process may be running either in frontend or in backend of the computer. Each and every process is composed of one or more than one **thread**. It is possible that an **operating system** may contain more processes running than actual programs because some processes are running in the backend of the operating systems.

Viewing Processes

The *ps* command is used to list the currently running processes and their PIDs. At a bare minimum, two processes will be shown, the shell (usually *bash* on Linux) and ps, which itself is a process and which dies as soon as its output is displayed. Usually, there will be many more. The following will provide a full listing of the current processes:

ps -aux | less





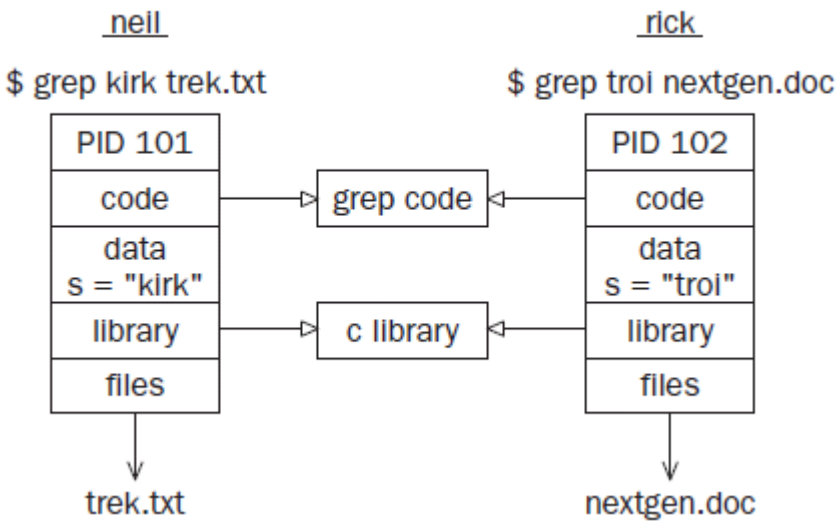
Process in windows system **Process in LINUX system**

Under UNIX, many different users can be on the system at the same time. In other words, they have processes that are in memory all at the same time. The system needs to keep track of what user is running what process, which terminal the process is running on, and what other resources the process has (such as open files).

The components of a process are the following

- Program counter
- Stack
- Data section

Let's have a look at how a couple of processes might be arranged within the operating system. If two users, neil and rick, both run the grep program at the same time to look for different strings in different files, the processes being used might look like Figure.



If you could run the ps command as in the following code quickly enough and before the searches had finished, the output might contain something like this:

$ **ps -ef**

UID	PID	PPID	C	STIME	TTY	TIME	CMD
rick	101	96	0	18:24	tty2	00:00:00	grep troi nextgen.doc
neil	102	92	0	18:24	tty4	00:00:00	grep kirk trek.txt

Each process is allocated a unique number, called a *process identifier* or *PID*.

Program

Program is nothing but the set of all the instruction which requires carrying out some specific job. Before come into execution they must be convert in binary codes which should be understood by loader OS. Generally in Windows it like EXE file which is stored in Portable Executable (PE) Format on Secondary Memory Like hard disk.

Program differ from process

- Process is program under execution whereas Program is set of instructions
- Processes are dynamic whereas programs are static
- Process is the active state of an entity means the program on execution. Whereas program is the passive state of an entity means the set of logics that are applied to perform a task.
- Program are entitles as job whereas Processes are often referred to as tasks.
- Process can communicate to each other (Inter process communication) but program cannot.

1.2 Process state

As a process executes, it changes **state**. The state of a 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.

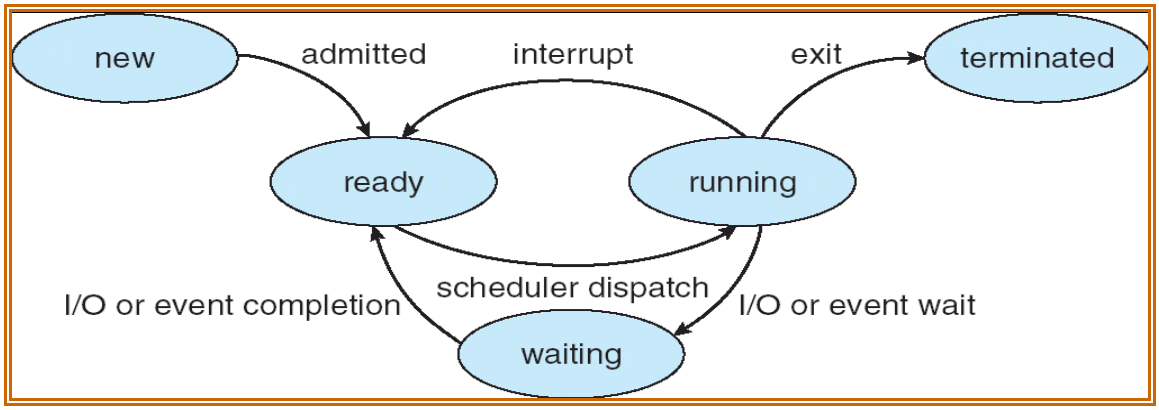
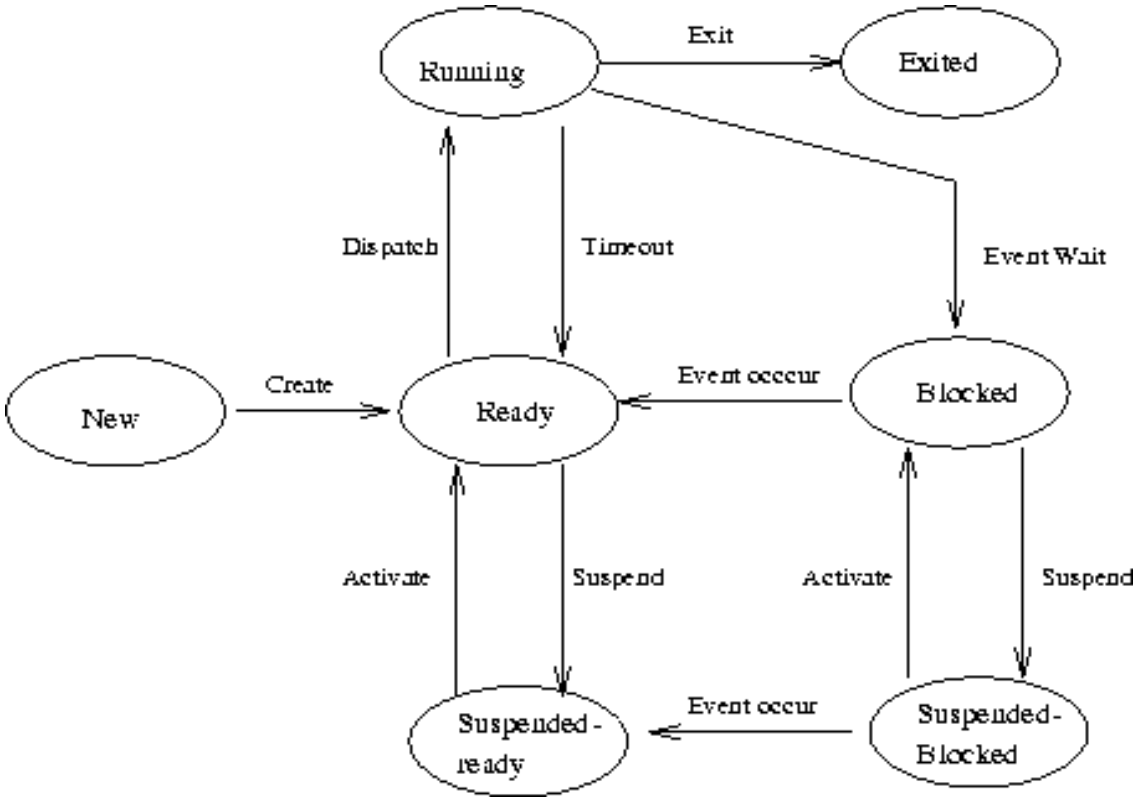


Diagram of Process State

Modern architectures rely on a hyerarchical organization of the available memory, only part of which exists in silicon on the machine board, while the rest is *paged* on a high speed storage peripheral like a hard disk. While the processor has the capability to address memory well above the amount of available RAM, it physically accessed the RAM only, and dedicated hardware is used to copy (or *page*) from the disk to RAM those pieces of memory which the processor refers to and are not available on RAM, and copy back areas of RAM onto disk when RAM space needs be made available. In particular, the act of copying entire processes back and forth from RAM to disk and vice versa is called *swapping*. As we'll see further on, swapping is often mandated by efficiency considerations even if paged memory is available, since the performance of a paged memory system can fall abruptly if too many active processes are mantained in main memory.

The OS could then perform a *suspend* transition on blocked processes, swapping them on disk and marking their state as *suspended* (after all, if they must wait for I/O, they might as well do it out of costly RAM), load into main memory a previously suspended process, *activating* into ready state and go on, However, swapping is an I/O operation in itself, and so at a first sight things might seem to get even worse doing this way. Again the solution is to carefully reconsider the reasons why processes are blocked and swapped, and recognize that if a process is blocked because it waits for I/O, and then suspended, the I/O event might occur while it sits swapped on the disk, We can thus classify suspended processes into two classes: *ready-suspended* for those suspended process whose restarting condition has occurred, and *blocked-suspended* for those who must still wait instead. This classification allows the OS to pick from the good pool of ready-suspended processes, when it wants to revive the queue in main memory. Provisions must be made for passing processes between the new states. In our model this means allowing for new transitions: *activate* and *suspend* between ready and ready-suspended, and between blocked-suspended and blocked as well (why also the activation?), end *event-occurred* transitions from blocked to ready, and from blocked-suspended to ready-suspended as well. The final scheme thus obtained is sketched in figure.

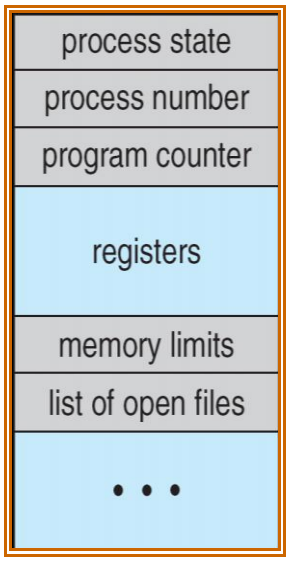


State process model

1.3 Process control Block(PCB)

The OS must know specific information about processes in order to manage, control them and also to implement the process model, the OS maintains a table (an array of structures), called the **process table**, with one entry per process. These entries are called **process control blocks** (PCB) - also called a task control block. This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from running to ready or blocked state so that it can be restarted later as if it had never been stopped. The role of the PCBs is central in process management: they are accessed and/or modified by most OS utilities, including those involved with scheduling, memory and I/O resource access and performance monitoring. It can be said that the set of the PCBs defines the current state of the operating system. Data structuring for processes is often done in terms of PCBs. For example, pointers to other PCBs inside a PCB allow the creation of those queues of processes in various scheduling states.

In modern sophisticated multitasking systems the PCB stores many different items of data, all needed for correct and efficient process management.



Process Control Block (PCB)

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Field of process table

- Such information is usually grouped into two categories: Process State Information and Process Control Information. 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 information as the value of the base and limit registers, 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 I/O devices allocated to the process, a list of open files, and so on.

1.4 **The Process Table**

The Linux process table is like a data structure describing all of the processes that are currently loaded with, for example, their PID, status, and command string, the sort of information output by ps. The operating system manages processes using their PIDs, and they are used as an index into the process table. The table is of limited size, so the number of processes a system will support is limited. Early UNIX systems were limited to 256 processes. More modern implementations have relaxed this restriction considerably and may be limited only by the memory available to construct a process table entry.

1.5 PROCESS STATE CODES

$ ps –ax will show you all process

```
Here are the different values that the s, stat and state output
specifiers
(header "STAT" or "S") will display to describe the state of a process.
```

```
D    Uninterruptible sleep (usually IO)
R    Running or runnable (on run queue)
S    Interruptible sleep (waiting for an event to complete)
T    Stopped, either by a job control signal or because it is being
traced.
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    Defunct ("zombie") process, terminated but not reaped by its
parent.

For BSD formats and when the stat keyword is used, additional
characters may
be displayed:
<    high-priority (not nice to other users)
N    low-priority (nice to other users)
L    has pages locked into memory (for real-time and custom IO)
s    is a session leader
l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+    is in the foreground process group
```

2. Process scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again. 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. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

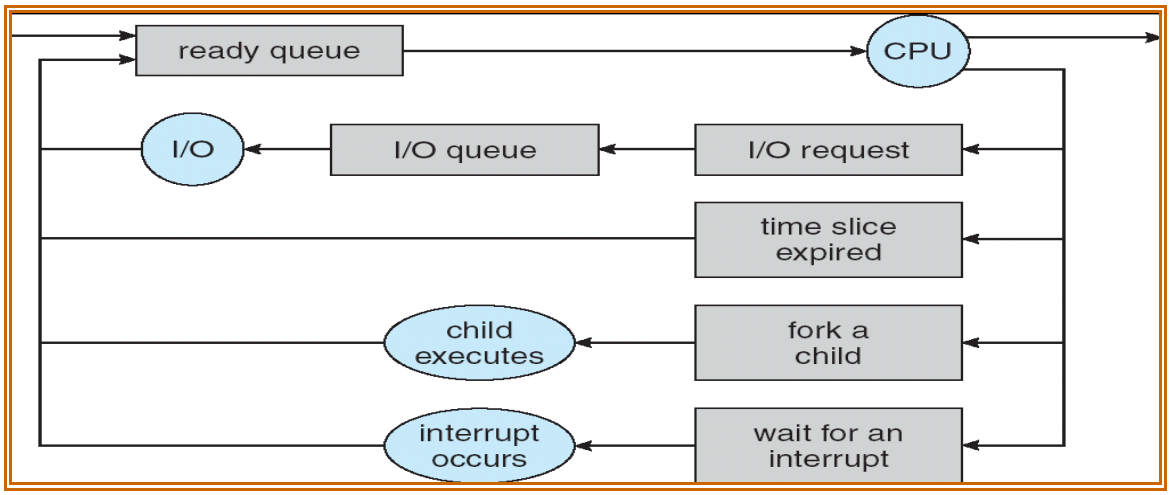
2.1 Scheduling Queues

- 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**.
- This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.
- The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue

A common representation for a discussion of process scheduling is a queuing diagram

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **dispatched**.
- 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.

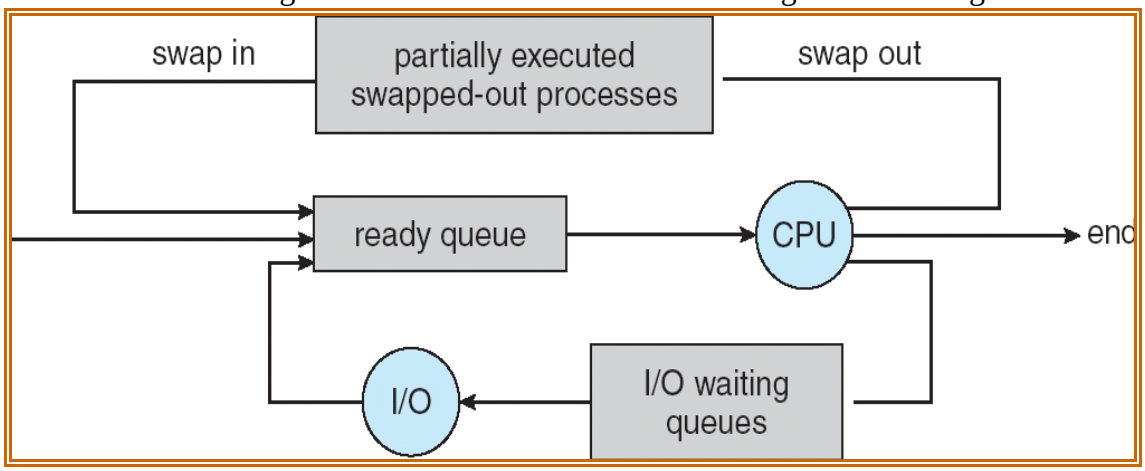
    - The process could create a new subprocess and wait for the subprocess's termination.
    - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.



2.1 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**.
- Often, in a batch system, 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**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.
- It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either I/O bound or CPU bound.
    - An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations.
    - A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes.

- On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.
- The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users.
- Some OSs, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure.

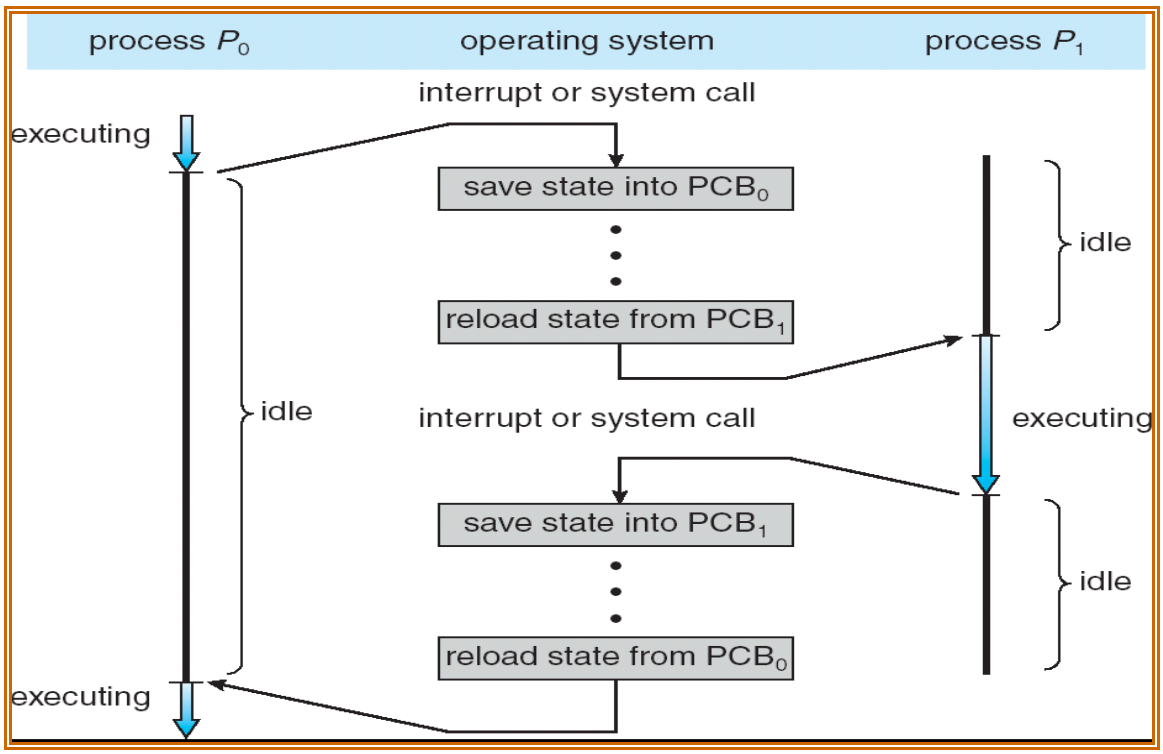


Medium-term scheduler

- The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**

2.2 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.
- 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 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.
    - process table keeps track of processes,
    - context information stored in PCB,
    - process suspended: register contents stored in PCB,
    - process resumed: PCB contents loaded into registers
- Context-switch time is pure overhead, because the system does no useful work while switching. Context switching can be critical to performance.
- Context switching time purely dependent on the hardware type.



CPU Switch from Process to Process

3. Operation on Process

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. Process goes through different operational state. During the operation of the process it utilizes the system resource like memory, CPU, I/O devices program counter etc. In this section we will discuss the how process may be created and hoe resource has been shared and duplicated among different running process. The approach will also include the UNIX and Solaris system to explain the process operation.

3.1 Process creation

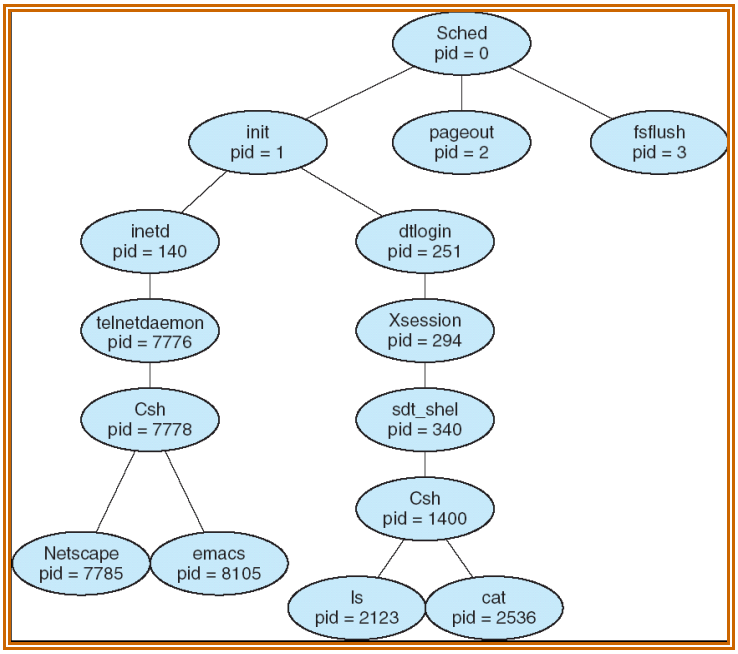
There are four principal events that cause processes to be created:

- During the system initialization, when an OS is booted, typically several processes are created.
- Execution of a process creation can also be done through system call by a running process. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes.
- A user request to create a new process. In interactive systems, users can start a program by typing a command or (double) clicking an icon.

- Initiation of a batch job. Here users can submit batch jobs to the system (possibly remotely). When the OS decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

In all these cases, a new process is created by having an existing process execute a fork () system call (in UNIX) for process creation.
The creating process is called a **parent process**, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most OSs (including UNIX and the Windows family of OSs) identify processes according to a unique process identifier (or pid), which is typically an integer number. The following figure shows process tree for Solaris Operating system with name of each process and it's pid. In Solaris, the process at the top of the tree is the shed process, with pid of 0. The shed process creates several children process including pageout and fsflush. These processes are responsible for managing memory and file system. The shed process also creates the init process, which serve as root parent process for all user process.



A tree of processes on a typical Solaris

In general, a process will need certain resources (CPU time, memory, files, I/0 devices) to accomplish its task. When a process creates a sub-process, that sub-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.

- The parent may have to partition its resources among its children,
- 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 subprocesses.

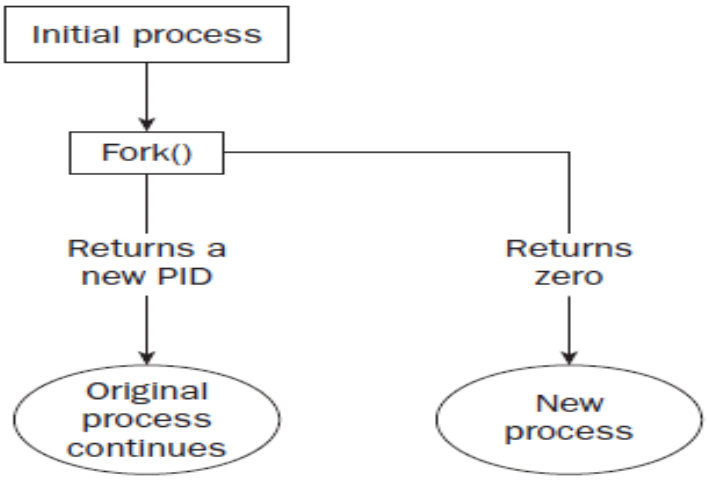
When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children, competing equally for the CPU.

2. The parent waits until some or all of its children have terminated (on UNIX, we can use {wait, waitpid, wait4, wait3}.).

Case: 1

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
pid_t fork(void);
int main()
{
pid_t p;
char *msg;
int n,i;
printf(" The fork program starting\n");
p=fork();
switch(p)
{
case -1:
        perror("fork failed");
        exit(1);
case 0:
        msg= "This is the child";
        n=5;
        printf("the child process id=%d\n",getpid());
        break;
default:
        msg= "This is the parent";
        n=3;
        printf("The parent process id=%d\n",getpid());
        break;
}
for(;n>0;n--)
{
        puts(msg);
        sleep(1);
}
printf("%d\n",getppid());
system("ps");
exit(0);
}
```



You can create a new process by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same

code but with its own data space, environment, and file descriptors. As you can see in Figure the call to fork in the parent returns the PID of the new child process. The new process continues to execute just like the original, with the exception that in the child process the call to fork returns 0. This allows both the parent and child to determine which is which. This program runs as two processes. A child is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

Case 2:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
pid_t wait(int *stat_loc);
int system(const char *string);
int main()
{
pid_t pid;
int n,exit_code;
char *msg;
printf("The for program is running\n");
pid=fork();
switch(pid)
{
case -1:
perror("The for system call fail\n");
break;
case 0:
msg="The child process is running";
exit_code=20;
n=5;
break;
default:
msg="The parent is running";
exit_code=0;
n=3;
break;
}
for(;n>0;n--)
{
puts(msg);
sleep(1);
}
if(pid!=0)
{
pid_t child_pid;
int stat_val;
child_pid=wait(&stat_val);
printf("The child has finished PID=%d\n",child_pid);
printf("The bash process PID=%d\n",getppid());
printf("The process id=%d\n",getpid());
if(WIFEXITED(stat_val))
printf("Child exited with code:%d\n",WEXITSTATUS(stat_val));
else
printf("Child terminated abnormally");
}
system("ps");
```

```
exit(exit_code);
}
```

In case 2 parent wait by calling wait () system call until child terminate.after termination of child process the parent proves terminate printing the child exit status.

If parent is exiting, some operating system do not allow child to continue if its parent terminates. In that situation, all children of that particular parent process terminated by the Operating system. This phenomenon is called *cascading termination.*

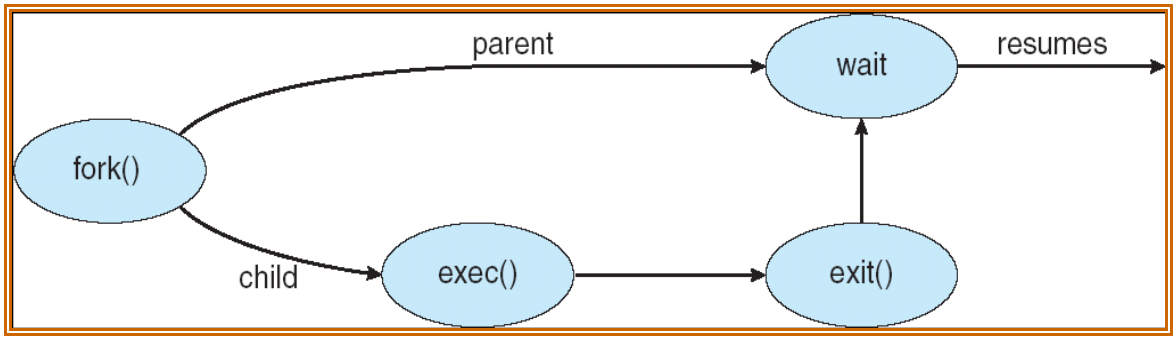
3.2 Zombie process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a *zombie process*.

You can see a zombie process being created if you change the number of messages in the fork example program. If the child prints fewer messages than the parent, it will finish first and will exist as a zombie until the parent has finished.

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<unistd.h>
pid_t fork(void);
int main()
{
pid_t p;
char *msg;
int n,i;
printf(" The fork program starting\n");
p=fork();
switch(p)
{
case -1:
        perror("fork failed");
        exit(1);
case 0:
        msg= "This is the child";
        n=1;
        printf("the child process id=%d\n",getpid());
        break;
default:
        msg= "This is the parent";
        n=5;
        printf("The parent process id=%d\n",getpid());
        break;
}
for(;n>0;n--)
{
        puts(msg);
        sleep(1);
}
printf("%d\n",getppid());
system("ps -ax");
exit(0);
}
```

If you run the preceding program and then call the ps program after the child has finished but before the parent has finished, you'll see a line such as this. (Some systems may say <zombie> rather than <defunct>.)
If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (init) as parent. The child process is now a zombie that is no longer running but has been inherited by init because of the abnormal termination of the parent process.This kind of process is called **Orphan process**. The zombie will remain in the process table until collected by the init process. The bigger the table, the slower this procedure. You need to avoid zombie processes, because they consume resources until init cleans them up.



State diagram of process creation

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

1. The child process is a duplicate of the parent process (it has the same program and data as the parent, an exact clone). The two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files.
2. The child process has a new program loaded into it.

Case 1: where the main() function call the "ps-ax" through a function system after starting the execution of system() function, the main() function became the parent and system() function which call "ps-ax" command to be executed became child and they run independently. The two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. The status of the process "ps-ax" will show R+ ie as fore ground process.

```
#include<stdio.h>
#include<stdlib.h>
int system(const char *string);
int main()
{
printf("Running the ps with system\n");
system("ps -ax");
printf("%d\n",(int) getppid());
printf("Done\n");
exit(0);
}
```

If we call the function system ("ps-ax &"), it is more understandable when the called function (child process) is running in background and parent process completed its task before the execution of child and child process "ps-ax" process status will only R ie as back ground process.

Case2:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main()
{
printf("Running ps with execlp\n");
printf("%d\n",(int) getpid());
printf("%d\n",(int) getppid());
execlp("ps","ps","-ax", 0);
printf("%d\n",(int) getpid());
printf("%d\n",(int) getppid());
printf("Done\n");
exit(0);
}
```

The program prints its first message and then calls execlp, which searches the directories given by the PATH environment variable for a program called ps. It then executes this program in place of the main program, starting it as if you had given the shell command. When ps finishes, you get a new shell prompt. You don't return to main, so the second message doesn't get printed. The PID of the new process is the same as the original, as are the parent PID and nice value i.e the child process overlay the its address space with the UNIX command ps-ax using the system call execlp().

3.4 Process termination

Normal 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 process wait ( ) system call. All the resources of the process -including physical and virtual memory, open files, and I/O buffers- are deallocated by the OS.

Abnormal termination: programming errors, run time errors, I/O, user intervention.

- **Error exit (voluntary):** An error caused by the process, often due to a program bug (executing an illegal instruction, referencing non-existent memory, or dividing by zero). In some systems (e.g. UNIX), a process can tell the OS that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.
- **Fatal error (involuntary):** i.e.; no such file exists during the compilation.
- **Killed by another process (involuntary):** A process can cause the termination of another process via an appropriate system call (for example,Terminate process( ) in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

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. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- 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 (cascading termination).