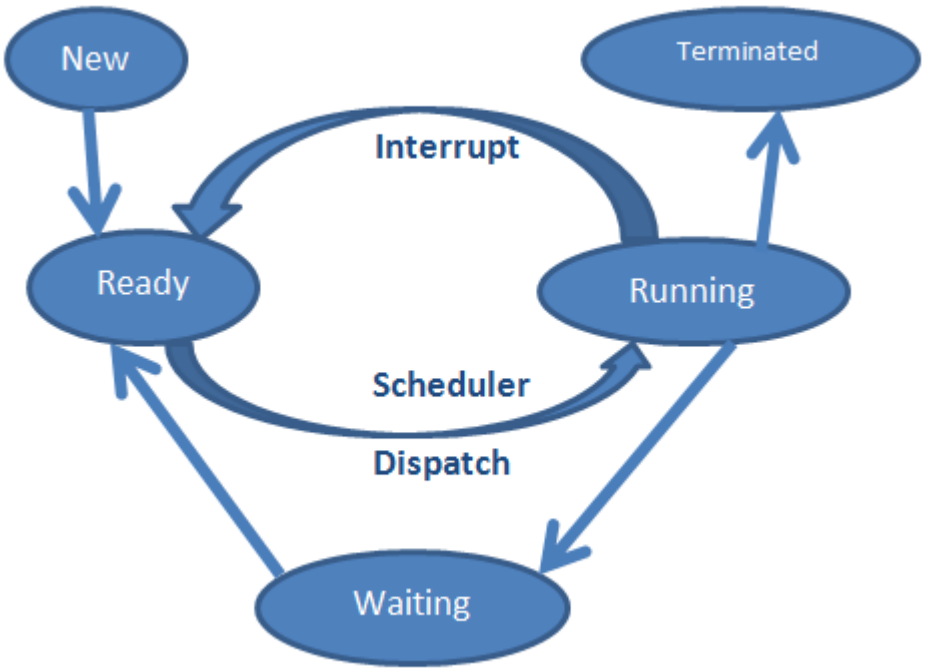
A process has the following attributes associated with it:

• PID (Process ID)

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Each process has a unique numeric process ID, better known as the PID. Each PID occurs only once on the system at any given moment, but if your system remains online for long enough, they are reused eventually. The PID is the primary way of identifying a particular process.

A process is also the unit for scheduling system resources for access. For instance, if 20 programs are running on a system with a single CPU, the Linux kernel alternates between each of them, giving them each a small amount of time to run, and then rapidly switching to the next. Thus, each process gets a small *time slice*, but because it gets these frequently, it seems as if the system is actually managing to run all 20 processes simultaneously. In systems with more than one CPU, the kernel decides which process should run on which CPU, and manages multitasking issues between them.



A process change of state

Terminated

Stopped by a signal

Of resumed by a signal

Process:

When you want to create a new process in Linux, the basic call to do this is fork(). **This is, incidentally, one of the few calls in Linux that are able to return twice;**

When you fork a process, the system creates another process running the same program as the current process. In fact, the newly created process, called the *child process*, has all the data, connections, and so on as the *parent process* and execution continues at the same place. The single difference between the two is the return value from the fork() system call, which returns the PID of the child to the parent and a value of 0 to the child. **Therefore, common practice is to examine the return value of the call in both processes, and do different things based on it.**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
pid_t pid;
pid = fork();
if (pid == 0) {
printf("Hello from the child process!\n");
} else if (pid != -1) {
printf("Hello from the parent. I've forked process %d.\n", pid);
} else {
printf("There was an error with forking.\n");
}
}
```

This code will fork. If the return value is 0, it means that the current process is the child from the fork. If the value is not -1, it means that the fork was successful and the return value indicates the PID of the new process. On the other hand, if the value is -1, then the fork failed.When you run this program, you will get two messages—one from the parent and one from the child. Because these are separate processes, these messages appear in essentially a random order. If you run the program several times, you'll get the messages in both orders.

The parent and child can communicate through a **pipe and signals**

Communicating through a pipe is much simpler and more reliable than using signals.

Parent Process
Child process

Zombie

when a process exits, its entry in the process table does not completely go away. This is because the operating system is waiting for a parent process to fetch some information about why the child process exited. This could include a return value, a signal, or something else along those lines. A process whose program terminated but still remains because its information was not yet collected is dubbed a zombie process.

Orphan

Every Child Process completion status has to be returned to the parent. It is the parent's responsibility to fetch the state of the child and handle it accordingly.

System call to establish proper synchronization between parent and child

Wait() system call:

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

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.

C program to demonstrate working of wait()

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
   pid_t cpid;
   if (fork()== 0)
      exit(0);          /* terminate child */
   else
   cpid = wait(NULL); /* reaping parent */ wait(NULL) will block the parent process until any of its children has finished.
   printf("Parent pid = %d\n", getpid());
   printf("Child pid = %d\n", cpid);

   return 0;
}
```

In the above program the parent process will be blocked until the child process returns an exit status to the operating system which is then returned to the parent process. If the child finishes before the parent reaches wait(NULL) then it will read the exit status, release the process entry in the process table and continue execution until it finishes as well.

Waitpid() system call:

```
#include <sys/wait.h>
      pid_t waitpid(pid_t pid, int *wstatus, int options);
```

wait(&wstatus) is

equivalent to:

```
   waitpid(-1, &wstatus, 0);

int main(void) {
   pid_t pid;
   pid = fork();
   if (pid == 0) {

     printf("Hello from the child process!\n");

     printf("The child is exiting now.\n");

   } else if (pid != -1) {

     printf("Hello from the parent, pid %d.\n", getpid());
     printf("The parent has forked process %d.\n", pid);
     waitpid (pid, NULL, 0);
     printf ("The child has stopped.  Sleeping for 60 seconds.\n");
     sleep(60);
     printf("The parent is exiting now.\n");

   } else {

     printf("There was an error with forking.\n");

   }
   return 0;
 }
```

Sleep() system call: Delay termination of a process

#include <unistd.h>

unsigned int sleep(unsigned int *seconds*);

Here's a quick example of this type of process:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
pid_t pid;
pid = fork();
if (pid == 0) {
printf("Hello from the child process!\n");
```

```
printf("The child is exiting now.\n");
} else if (pid != -1) {
printf("Hello from the parent, pid %d.\n", getpid());
printf("The parent has forked process %d.\n", pid);
sleep(60);
printf("The parent is exiting now.\n");
} else {
printf("There was an error with forking.\n");
}
}
```

There is now a sleep call in the parent to delay its exit for a minute

Pipes(unidirectional communication):

File descriptor basedcommunication. Pipes are provided for your use for setting up lines of communication between two processes on your local machine. Instead of using open() to create a pipe, you use pipe(). After that, however, you use standard system calls such as read() and write(), just as you would with a more "normal" file descriptor.

Pipes are intended solely for communication between two processes. When you create a pipe, you actually get two file descriptors—one for reading and one for writing. Any data that is written to the write side of the descriptor can later be read back from the read side. you cannot use a single pipe for more than two processes. Instead you might have to use a solution such as setting up a line of pipes (a pipeline) to shuttle data from one process to the next, which will certainly be slower. On the other hand, because pipes are used as standard file descriptors, they are the method of choice for communication between two processes that use file descriptor I/O already. Obtain pipe descriptors, and then use the fork() system call. Each end will close one of the descriptors. For instance, if the child process will do the writing and the parent the reading, the child process should close the reading end of the pipe and the parent should close the writing end.

Sample program to create a pipe and then communicate over it. **Notice how it must fork and then the two processes use the pipe file descriptors that were opened before the fork.**

Pipe():create a Pipe

int pipe(int *pipefd***[2]);**

Anything that is written on **fd[1]** may be read by **fd[0]**

read() and write:

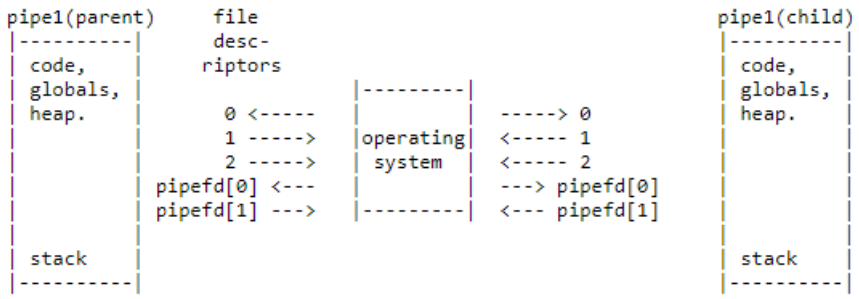
ssize_t read(int *fd***, void *****buf***, size_t** *count***);**

ssize_t write(int *fd***, const void *****buf***, size_t** *count***);**

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

When fork() is used:



we can view the running process as having 5 open file descriptors: standard input (0), standard output (1), standard error (2), the read end of the pipe (**pipefd[0]**), and the write end of the pipe (**pipefd[1]**).

When you say **write(1, buf, size)**, you are saying to print those bytes present in "buf" to standard output(instead of printf) ,
When you say read**(0, buf, size)**, you are saying to read those bytes from standard input in to "buf"(instead of scanf).

Sample program:

```
#include <stdio.h>
main()
{
  int pipefd[2];
  int i;
  char s[1000];
  char *s2;

  if (pipe(pipefd) < 0) {
    perror("pipe");
    exit(1);
  }

  s2 = "CSE E section";
  write(pipefd[1], s2, strlen(s2));
  i = read(pipefd[0], s, strlen(s2));
```

```
  s[i] = '\0';
  printf("Read %d bytes from the pipe: '%s'\n", i, s);
}
```

Using fork and pipe:

Assignment Program for practice:

Q1). Concatenate two strings

Parent

Get string as input “hi”
Pass it to child (“hi”)
Wait for the parent to process
Receive the concatenated string(“hi bye”) from child and display.

Child
Get a string as input (“bye”)
Receive the string from parent and concatenate (“hi bye”);
Send the result(“hi bye”) to parent

Q2. Pass array of numbers between process:

Process p1 get’s one integer input(A) passes it to the child process p2, child gets one integer input(b) passes(A and B) to its child process p3. P3 receives two values performs addition and displays the result.

Q3) Create a specific amount of child processes, lets say 3, and make the parent wait for each child to finish. Also we're supposed to have a pipe that all processes write to so that once the parent is done waiting, it would use the pipe's to output the sum of all the children's results.