

## UNIT – IV: PROCESS AND SIGNALS

### 4.1 Process Structure

#### Introduction

- The UNIX standards, specifically IEEE Std 1003.1, 2004 Edition, defines a process as "an address space with one or more threads executing within that address space, and the required system resources for those threads."
- A multitasking operating system such as Linux lets many programs run at once. Each instance of a running program constitutes a process.
- As a multiuser system, Linux allows many users to access the system at the same time. Each user can run many programs, or even many instances of the same program, at the same time.
- process — that is running consists of program code, data, variables (occupying system memory), open files (file descriptors), and an environment. Typically, a Linux system will share code and system libraries among processes so that there's only one copy of the code in memory at any one time.

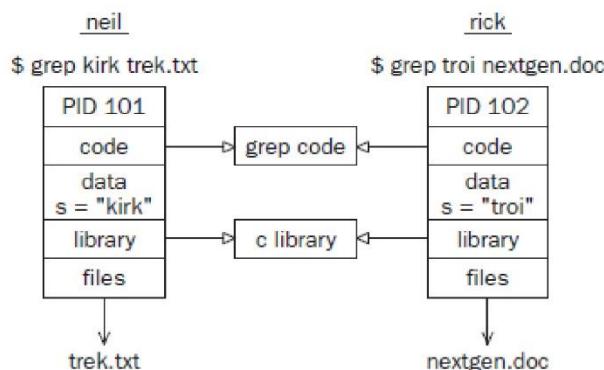


Fig 4.1 Different strings in different files

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  tty     2 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*. This is usually a positive integer between 2 and 32,768. When a process is started, the next unused number in sequence is chosen and the numbers restart at 2 so that they wrap around. The number 1 is typically reserved for the special init process, which manages other processes.
- The program code that will be executed by the grep command is stored in a disk file. Normally, a Linux process can't write to the memory area used to hold the program code, so the code is loaded into memory as read-only.
- The system libraries can also be shared. Thus, there need be only one copy of printf, for example, in memory, even if many running programs call it. This is a more sophisticated, but similar, scheme to the way dynamic link libraries (DLLs) work in Windows.

#### 4.1.1 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.

```
void (*signal(int sig, void (*func)(int)))(int);
```

This rather complex declaration says that signal is a function that takes two parameters, sig and func. The signal to be caught or ignored is given as argument sig. The function to be called when the specified signal is received is given as func. This function must be one that takes a single int argument (the signal received) and is of type void. The signal function itself returns a function of the same type, which is the previous value of the function set up to handle this signal, or one of these two special values:

*SIG\_IGN      Ignore the signal.  
SIG\_DFL      Restore default behavior.*

The function ouch reacts to the signal that is passed in the parameter sig. This function will be called when a signal occurs. It prints a message and then resets the signal handling for SIGINT (by default, generated by typing Ctrl+C) back to the default behavior.

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
```

The main function has to intercept the SIGINT signal generated when you type Ctrl+C. For the rest of the time, it just sits in an infinite loop, printing a message once a second.

```
int main()
{
    (void) signal(SIGINT, ouch);
    while(1) {
        printf("Hello World!\n");
        sleep(1);}}
```

raise to indicate the generation of a signal, and the term catch to indicate the receipt of a signal. Signals are raised by some error conditions, such as memory segment violations, floating-point processor errors, or illegal instructions.

They are generated by the shell and terminal handlers to cause interrupts and can also be explicitly sent from one process to another as a way of passing information or modifying behavior. In all these cases, the programming interface is the same. Signals can be raised, caught and acted upon, or (for some at least) ignored.

Signal names are defined by including the header file `signal.h`.

Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating-point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal quit
SIGSEGV	*Invalid memory segment access
SIGTERM	Termination
SIGUSR1	User-defined signal 1
SIGUSR2	User-defined signal 2

Fig 4.7 SIG header files

If a process receives one of these signals without first arranging to catch it, the process will be terminated immediately. Usually, a core dump file is created. This file, called `core` and placed in the current directory, is an image of the process that can be useful in debugging.

Signal Name	Description
SIGCHLD	Child process has stopped or exited.
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing. (Can't be caught or ignored.)
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process trying to read.
SIGTTOU	Background process trying to write.

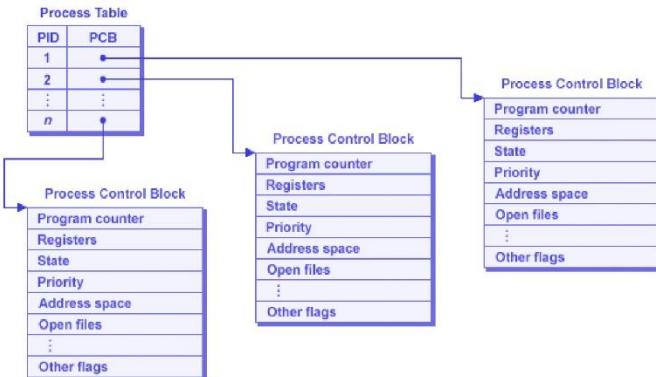
Fig 4.8 Additional signals

`SIGCHLD` can be useful for managing child processes. It's ignored by default. The remaining signals cause the process receiving them to stop, except for `SIGCONT`, which causes the process to resume. They are used by shell programs for job control and are rarely used by user programs.

Programs can handle signals using the signal library function.

```
#include <signal.h>
```

- 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.



**Fig 4.2 Process Table**

- The process table major information is, Process ID, Process management . Registers, Program counter , Program status word , Stack pointer , Process state , 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 , Memory management, Pointer to text segment info, Pointer to data segment info , Pointer to stack segment info, File management , Root directory Working directory File descriptors User ID , Group IDProcess User ID, Process Priority, Process State, Process resource usage

#### 4.1.2 Viewing processes

The ps command shows the processes you're running, the process another user is running, or all the processes on the system. Here is more sample output:

\$ ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	433	425	0	18:12	tty1	00:00:00	[bash]
rick	445	426	0	18:12	tty2	00:00:00	-bash
rick	456	427	0	18:12	tty3	00:00:00	[bash]
root	467	433	0	18:12	tty1	00:00:00	sh /usr/X11R6/bin/startx
root	474	467	0	18:12	tty1	00:00:00	xinit /etc/X11/xinit/xinitrc
root	478	474	0	18:12	tty1	00:00:00	/usr/bin/gnome-session
root	487	1	0	18:12	tty1	00:00:00	gnome-smproxy--sm-client-id def
root	493	1	0	18:12	tty1	00:00:01	[enlightenment]
root	506	1	0	18:12	tty1	00:00:03	panel --sm-client-id default8
root	508	1	0	18:12	tty1	00:00:00	xscreensaver -no-splash -timeout
root	510	1	0	18:12	tty1	00:00:01	gmc --sm-client-id default10
root	512	1	0	18:12	tty1	00:00:01	gnome-help-browser --sm-client-i
root	649	445	0	18:24	tty2	00:00:00	su
root	653	649	0	18:24	tty2	00:00:00	bash
neil	655	428	0	18:24	tty4	00:00:00	-bash
root	713	1	2	18:27	tty1	00:00:00	gnome-terminal
root	715	713	0	18:28	tty1	00:00:00	gnome-pty-helper

```
root  717  716  13   18:28 pts/0 00:00:01 emacs
root  718  653  0    18:28 tty2 00:00:00 ps -ef
```

This shows information about many processes, including the processes involved with the Emacs editor under X on a Linux system. For example, the TTY column shows which terminal the process was started from, TIME gives the CPU time used so far, and the CMD column shows the command used to start the process. Let's take a closer look at some of these.

```
neil 655 428 0 18:24 tty4 00:00:00 -bash
```

The initial login was performed on virtual console number 4. This is just the console on this machine. The shell program that is running is the Linux default, bash.

```
root 467 433 0 18:12 tty1 00:00:00 sh /usr/X11R6/bin/startx
```

The X Window System was started by the command startx. This is a shell script that starts the X server and runs some initial X programs.

```
root 717 716 13 18:28 pts/0 00:00:01 emacs
```

This process represents a window in X running Emacs. It was started by the window manager in response to a request for a new window. A new pseudo terminal, pts/0, has been assigned for the shell to read from and write to.

```
root 512 1 0 18:12 tty1 00:00:01 gnome-help-browser --sm-client-i
```

This is the GNOME help browser started by the window manager.

By default, the ps program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. You can use ps to see all such processes using the -e option and to get "full" information with -f.

#### **4.1.3 System processes**

Here are some of the processes running on another Linux system. The output has been abbreviated for clarity. In the following examples you will see how to view the status of a process. The STAT output from

The getty processes wait for activity at the terminal, prompt the user with the familiar login prompt, and then pass control to the login program, which sets up the user environment and finally starts a shell. When the user shell exits, init starts another getty process.

You can see that the ability to start new processes and to wait for them to finish is fundamental to the system. You'll see later in this chapter how to perform the same tasks from within your own programs with the system calls fork, exec, and wait.

#### 4.1.4 Process Scheduling

- The Linux kernel uses a process scheduler to decide which process will receive the next time slice. It does this using the process priority. Processes with a high priority get to run more often, whereas others, such as low-priority background tasks, run less frequently.
- The operating system determines the priority of a process based on a "nice" value, which defaults to 0, and on the behavior of the program. Programs that run for long periods without pausing generally get lower priorities.
- Set the process nice value using **nice** and adjust it using **renice**. The nice command increases the nice value of a process by 10, giving it a **lower priority**. You can view the nice values of active processes using the **-l** or **-f** (for long output) option to **ps**.
- The value you are interested in is shown in the **NI (nice)** column.
- 

```
$ ps -l
  F S  UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
  000 S  500  1259  1254  0  75   0 -    710 wait4  pts/2    00:00:00 bash
  000 S  500  1262  1251  0  75   0 -    714 wait4  pts/1    00:00:00 bash
  000 S  500  1313  1262  0  75   0 -   2762 schedu pts/1    00:00:00 emacs
  000 S  500  1362  1262  2  80   0 -   789 schedu pts/1    00:00:00 oclock
  000 R  500  1363  1262  0  81   0 -   782 -      pts/1    00:00:00 ps
```

**Fig 4.4 NI (nice) column**

- Here you can see that the oclock program is running (as process 1362) with a default nice value. If it had been started with the command

```
$ nice oclock &
```

it would have been allocated a nice value of +10. If you adjust this value with the command

```
$ renice 10 1362 1362:
```

old priority 0, new priority 10

```
$ ps -l
  F S  UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
  000 S  500  1259  1254  0  75   0 -    710 wait4  pts/2    00:00:00 bash
  000 S  500  1262  1251  0  75   0 -    714 wait4  pts/1    00:00:00 bash
  000 S  500  1313  1262  0  75   0 -   2762 schedu pts/1    00:00:00 emacs
  000 S  500  1362  1262  0  90  10 -   789 schedu pts/1    00:00:00 oclock
  000 R  500  1365  1262  0  81   0 -   782 -      pts/1    00:00:00 ps
```

**Fig 4.5 modified nice value with ps**

The status column now also contains N to indicate that the nice value has changed from the default.

```
$ ps x
```

#### **4.2.4 Threads**

- Linux processes can cooperate, can send each other messages, and can interrupt one another. They can even arrange to share segments of memory between themselves, but they are essentially separate entities within the operating system. They do not readily share variables.
- There is a class of process known as a thread that is available in many UNIX and Linux systems. Though threads can be difficult to program, they can be of great value in some applications, such as multithreaded database servers. Programming threads on Linux (and UNIX generally) is not as common as using multiple processes, because Linux processes are quite lightweight, and programming multiple cooperation processes is much easier than programming threads
- Multiple strands of execution in a single program are called threads. A more precise definition is that a thread is a sequence of control within a process. All the programs you have seen so far have executed as a single process, although, like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution. All the processes that you have seen so far in this book have had just one thread of execution

#### **Following are some advantages of using threads:**

- Sometimes it is very useful to make a program appear to do two things at once. The classic example is to perform a real-time word count on a document while still editing the text. One thread can manage the user's input and perform editing.
- For a database server, this apparent multitasking is quite hard to do efficiently in different processes, because the requirements for locking and data consistency cause the different processes to be very tightly coupled. This can be done much more easily with multiple threads than with multiple processes.
- The performance of an application that mixes input, calculation, and output may be improved by running these as three separate threads. While the input or output thread is waiting for a connection, one of the other threads can continue with calculations. A server application processing multiple network connects may also be a natural fit for a multithreaded program.
- Now that multi-cored CPUs are common even in desktop and laptop machines, using multiple threads inside a process can, if the application is suitable, enable a single process to better utilize the hardware resources available.

#### **Threads also have drawbacks:**

- Writing multithreaded programs requires very careful design. The potential for introducing subtle timing faults, or faults caused by the unintentional sharing of variables in a multithreaded program is considerable. Alan Cox (the well respected Linux guru) has commented that threads are also known as "how to shoot yourself in both feet at once."
- Debugging a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control.
- A program that splits a large calculation into two and runs the two parts as different threads will not necessarily run more quickly on a single processor machine, unless the calculation truly allows multiple parts to be calculated simultaneously and the machine it is executing on has multiple processor cores to support true multiprocessing.

#### **4.3 Signals**

A signal is an event generated by the UNIX and Linux systems in response to some condition, upon receipt of which a process may in turn take some action. We use the term

a program that reads from its standard input and writes to its standard output, performing some useful transformation as it does so

Here's a very simple filter program, upper.c, that reads input and converts it to uppercase:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

When you run the program, it does what you expect:

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

You can, of course, use it to convert a file to uppercase by using the shell redirection

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ ./upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

This program, useupper.c, accepts a filename as an argument and will respond with an error if called incorrectly.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char *filename;
    if (argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
        exit(1);
    }
    filename = argv[1];
}
```

You reopen the standard input, again checking for any errors as you do so, and then use exec to call upper.

```
if(!freopen(filename, "r", stdin)) {
    fprintf(stderr, "could not redirect stdin from file %s\n", filename);
    exit(2);
}
```

```
exec("./upper", "upper", 0);
```

Don't forget that exec replaces the current process; if there is no error, the remaining lines are not executed.

```
perror("could not exec ./upper");
exit(3);
}
```

- If both parent and child write to the same descriptor, without any form of synchronization, their output will be intermixed, while this is possible, it's not the normal mode of operation.
  - There are two normal cases for handling the descriptors after a fork.
1. The parent waits for the child to complete. In this case, the parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or write to will have their file offsets updated accordingly.
  2. The parent and child each go their own way. Here, after the fork, the parent does the descriptors that it doesn't need and the child does the same thing. This way neither interfaces with other's open descriptors. This scenario is often the case with network servers.

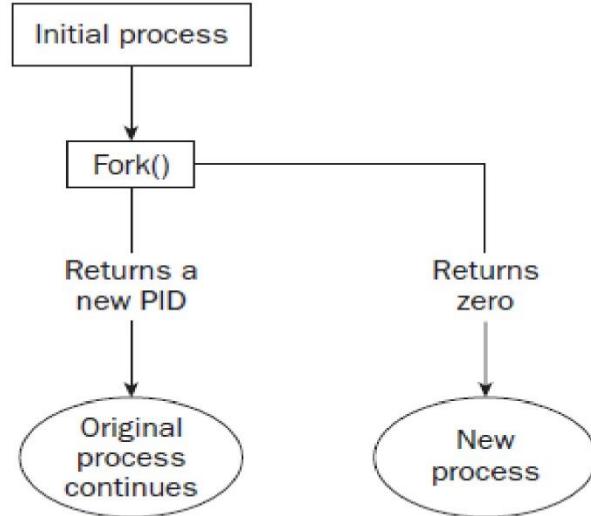
#### **vfork Function**

- The function vfork has the same calling sequence and share return values as fork. But the semantics of the two functions differ. vfork is intended to create a new process when the purpose of the new process is to exec a new program.
- vfork creates the new process, just like fork, without fully copying the address space of the parent into the child, since the child won't reference the address space – the child just calls exec right after the vfork.
- Instead, while the child is running, until it calls either exec or exit, the child runs in the address space of the parent. This optimization provides an efficiency gain on some paged virtual memory implementations of UNIX.
- Another difference between the two functions is that vfork guarantees that the child runs first, until the parent resumes.

#### **exit Function**

There are three ways for a process to terminate normally, and two forms of abnormal termination.

1. Normal termination:
  - a. Executing a return from the main function. This is equivalent to calling exit
  - b. Calling the exit function
  - c. Calling the \_exit function
2. Abnormal termination
  - a. Calling abort: It generates the SIGABRT signal
  - b. When the process receives certain signals. The signal can be generated by the process itself



**Fig:4.6 Duplicating a Process Image**

#### Sample program on fork

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
pid_t pid;
char *message;
int n;
printf("fork program starting\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
message = "This is the child";
n = 5;
break;
default:
message = "This is the parent";
n = 3;
break;
}
for(; n > 0; n--) {
puts(message);
sleep(1);
}
exit(0);
}
  
```

PID TTY STAT TIME COMMAND

1362 pts/1 SN 0:00 o'clock

- The PPID field of ps output indicates the parent process ID, the PID of either the process that caused this process to start or, if that process is no longer running, init (PID 1).
- The Linux scheduler decides which process it will allow to run on the basis of priority. Exact implementations vary, of course, but higher-priority processes run more often. In some cases, low-priority processes don't run at all if higher-priority processes are ready to run.

#### **4.2 Starting new processes**

- create a new process by using the system library function.

```
#include <stdlib.h>
```

```
int system (const char *string);
```

- The system function runs the command passed to it as a string and waits for it to complete. The command is executed as if the command:

```
$ sh -c string
```

has been given to a shell. system returns 127 if a shell can't be started to run the command and -1 if another error occurs. Otherwise, system returns the exit code of the command.

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("Running ps with system\n");
```

```
    system("ps ax");
```

```
    printf("Done.\n");
```

```
    exit(0);
```

```
}
```

**Compile and Run:** \$./system1

#### **fork Function**

- The only way a new process is created by the UNIX kernel is when an existing process calls the fork function.

```
#include<sys/types.h>
```

```
#include<unistd.h>
```

```
pid_t fork(void);
```

Return: 0 is child, process ID of child in parent, -1 on error

- The new process created by fork is called child process. This is called once, but return twice that is the return value in the child is 0, while the return value in the parent is the process ID of the new child.
- The reason the child's process ID is returned to the parent is because a process can have more than one child, so there is no function that allows a process to obtain the process IDs of its children.
- The reason fork return 0 to the child is because a process can have only a single parent, so that child can always call getppid to obtain the process ID of its parent.



STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, “runnable,” that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or “zombie” process.
N	Low priority task, “nice.”
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

**Fig 4.4 Common codes**

\$ ps ax

PID	TTY	STAT	TIME COMMAND
1 ?	Ss	0:03	init [5]
2 ?	S	0:00	[migration/0]
3 ?	SN	0:00	[ksoftirqd/0]
4 ?	S<	0:05	[events/0]
5 ?		S<	0:00 [khelper]
6 ?	S<	0:00	[kthread]
840 ?	S<	2:52	[kjournald]
888 ?	S<s	0:03	/sbin/udevd --daemon
3069 ?	Ss	0:00	/sbin/acpid
3098 ?	Ss	0:11	/usr/sbin/hald --daemon=yes
3099 ?	S	0:00	hald-runner
8357 ?	Ss	0:03	/sbin/syslog-ng
8677 ?	Ss	0:00	/opt/kde3/bin/kdm
9119 ?	S	0:11	konsole [kdeinit]
9120 pts/2	Ss	0:00	/bin/bash
9151 ?	Ss	0:00	/usr/sbin/cupsd
9457 ?	Ss	0:00	/usr/sbin/cron
9479 ?	Ss	0:00	/usr/sbin/sshd -o PidFile=/var/run/sshd.init.pid

In general, each process is started by another process known as its parent process. A process so started is known as a child process. When Linux starts, it runs a single program, the prime ancestor and process number 1, init. This is, if you like, the operating system process manager and the grandparent of all processes. Other system processes you'll meet soon are started by init or by other processes started by init.

One such example is the login procedure. init starts the getty program once for each serial terminal or dial-in modem that you can use to log in. These are shown in the ps output like this:

9619 tty2 Ss+ 0:00 /sbin/mingetty tty2

#### 4.2.2 Zombie process

- The process that has terminated, but whose parent has not yet waited for it, is called zombie process.
- The 'ps' command prints the status of the **zombie process as Z**. If a long running program that forks many child processes, unless it waits for the processes to fetch their termination status, they become zombie.
- **init** calls one of the wait functions to fetch the termination status.
- 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.

fork2.c is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code

```
switch(pid)
{
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        message = "This is the child";
        n = 3;
        break;
    default:
        message = "This is the parent";
        n = 5;
        break;
}
```

#### wait and waitpid Functions

- When a process terminates, either normally or abnormally, the parent is notified by the kernel sending the parent **SIGCHLD** signal.
- Since the termination of a child is an asynchronous event, this signal is the asynchronous notification from the kernel to the parent.
- The default action for this signal is to be ignored. A parent may want for one of its children to terminate and then accept its child's termination code by executing wait.
- A process that calls wait and waitpid can:
  1. block (if all of its children are still running).
  2. return immediately with termination status of a child ( if a child has terminated and is waiting for its termination status to be fetched) or
  3. return immediately with an error (if it doesn't have any child process).
- If a child has already terminated and is a **zombie**, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates: if the caller blocks and has multiple children, wait returns when one terminates, we can know this process by PID return by the function.
- For both functions, the argument **statloc** is pointer to an integer. If this argument is not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument.

#### 4.2.3 Input and output redirection

Processes to alter the behavior of programs by exploiting the fact that open file descriptors are preserved across calls to fork and exec. The next example involves a filter program —