Linux System Programming

Using C

Sarath M

Listings

Contents

1. Introduction

1.1 Topics

1. Process Management
2. File and file management
3. Memory and memory management
4. Signals and Signal handling
5. Thread and Thread management
6. Inter Process Communication
7. Process Synchronisation
8. Shell Scripitng

1.2 Defenition's

Operating System OS is a resource manager/allocater (rather than a mere interface between the user and hardware) which is responsible for managing the resources which is connected to the CPU

BIOS Basic Input Output Systems When we switch ON the system, BIOS program is executed. First job of BIOS is to check if basic input output are connected or not. After that the BIOS executes a program called Bootloader

Bootloader picks ups an OS from hard disk and loads it into RAM. Load time/Boot time is the time taken for this operation.

Grub loader Boot loader in Linux

NT loader Boot loader in Windows

1.3 OS in Embedded Systems

Why OS in embedded systems?

Without OS only one program can be run at a time

With OS multiple process can be run simultaneously. Thus performance of the product increases.

1.4 Components of an OS

1. Application
2. Services

Appication are optional services. Applications runs only when we intentionally run it.
Services are mandatory. Kernel starts executing when OS is loaded into RAM
Kernel All the services combined is called a kernel

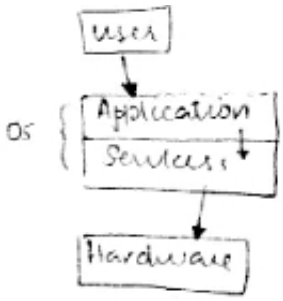

Figure 1.1: OS

2. Process Management

Process Thre program which is in execution is called as the process.To execute a file say a.out, a copy of a.out is loaded into RAM
Process Manager - Manages the different process.

ps -e Command to display the processes which are currently running

2.1 Process Manager

For every process, the process manager will provide a process ID, ie; the process manager identifies each process with a process ID.

- ./a.out & : If we want to run a command in background
- fg : command to bring the background process to foreground
- fg <jobId>: To move a specific process to foregraound
- ps -e | grep pts/0: If we want to list all programs running in terminal *pts/0*

Shell - Command Interpreter. When user wants to interact with the OS we use the shell. **ex: bash**

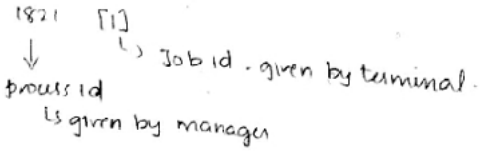

Figure 2.1: Output

kill: command used to send signals to a particular process
ex: kill -9 1769, where 1769 is process id

2.2 PID and PPID

Listing 2.1: programs/getpid.c

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

void main()
{
    printf("Hello!\n pid = %d, ppid = %d\n",getpid(),getppid()
       );

    printf("Waiting (while(1))\n");
    while(1);
}
```

- **get_pid**(): returns the process id
- **get_ppid**(): return the parent process id

The parent is nothing but the bash. Whenever a new terminal is opened, a bash process is creatted. This bash is the parent of the program being executed

2.3 System() function

Listing 2.2: programs/system.c

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

int main()
{
    printf("Hello\n");
```

```
    system("ls");
    printf("Hi\n");
}
```

Listing 2.2: Ouput

```
Hello
a.out getpid.c system.c system.txt
Hi
```

system() function executes a command specified by 'command' by calling `/bin/sh -c command` and returns after the command has been completed.

Example

Consider the following program

Listing 2.3: programs/example.c

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

int main()
{
    printf("Pid = %d, PPID = %d\n", getpid(), getppid());
    while(1);        // ./p1 will be stuck here
}
```

Suppose we created a new executable for this code name **p1.out**

Now we can run the above program in p2.c using system() as shown below:

Listing 2.4: programs/p2.c

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

int main()
{
    printf("Hello\n");
    system("./p1");
    printf("\nHi\n");        // Will be printed after ct+c
}
```

Listing 2.4: Ouput

```
Hello
Pid = 4040, PPID = 4039
^C
Hi
```

when *ctr+C* is pressed Hai will since only then control returns from *p1* because of *while(1)*

Example

Consider the following program, How many processes are created by exampl2.c

Listing 2.5: programs/example2.c

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

int main()
{
    printf("Hello\n");
    system("cal");
    system("ls");
    printf("\nHi\n");
}
```

Answer: 6

1. Bash
2. a.out
3. sh
4. cal
5. sh
6. ls

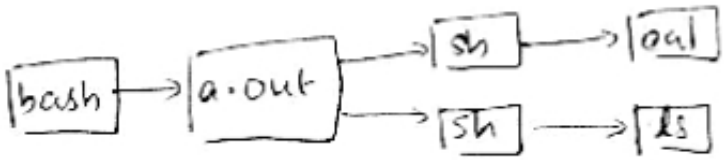

Figure 2.2: Output

2.4 Context Switch

Whenever a process is loaded into the RAM the process manager creates one table called **Process Control Block** which consists of information about the process.

Process manager stores the process information in linked list fashion.

When the process is executing the process information is loaded into CPU. For every process a particular time (time slice) is given. When the time slice expires the process manager loads the updated process information to the respective PCB and the next PCB is loaded into the CPU.

This loading and unloading of information is called ***Context Switch***

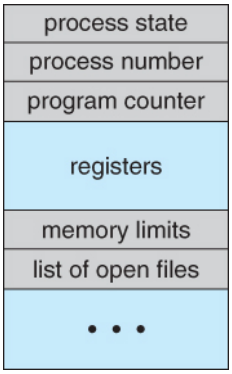

Figure 2.3: Process Control Block

When rhe process execution is complete the PCB is destoyed

2.5 fork() function

```
pid_t fork(void)
```

- Used to create a child process
- Creates a new process by duplicating the calling process
- The new process referred to as child process which is exact duplicate of calling process referred to as parent

Example

Listing 2.6: programs/fork1.c

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Hello\n");
    fork();
    printf("Hi PID = %d PPID = %d\n", getpid(), getppid());
    while(1);
}
```

Listing 2.6: Ouput

```
Hello
Hi PID = 5488 PPID = 3897
Hi PID = 5489 PPID = 5488
^C
```

```
 All the statements after the fork() are executed
twice!!
Once by the parent and then by the child
Parent executes in the foreground, whereas child
executes in the background !!
```

2.5.1 Race Condition

Normally the parent process gets the CPU time after creating the child

But when multiple processes are waiting for CPU time a race condition exists and the order of execution of these processes may vary

Example

Listing 2.7: programs/fork2.c

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

int main()
{
    printf("Hello\n");
    fork();
    fork();
    fork();
    printf("Hi, PID = %d, PPID = %d\n", getpid(), getppid());
    while(1);
}
```

Listing 2.7: Ouput

```
Hello
Hi, PID = 5684, PPID = 3897
Hi, PID = 5687, PPID = 5684
Hi, PID = 5685, PPID = 5684
Hi, PID = 5686, PPID = 5684
Hi, PID = 5688, PPID = 5686
Hi, PID = 5690, PPID = 5685
Hi, PID = 5689, PPID = 5685
Hi, PID = 5691, PPID = 5689
^C
```

Here we can't predict the order of the processes

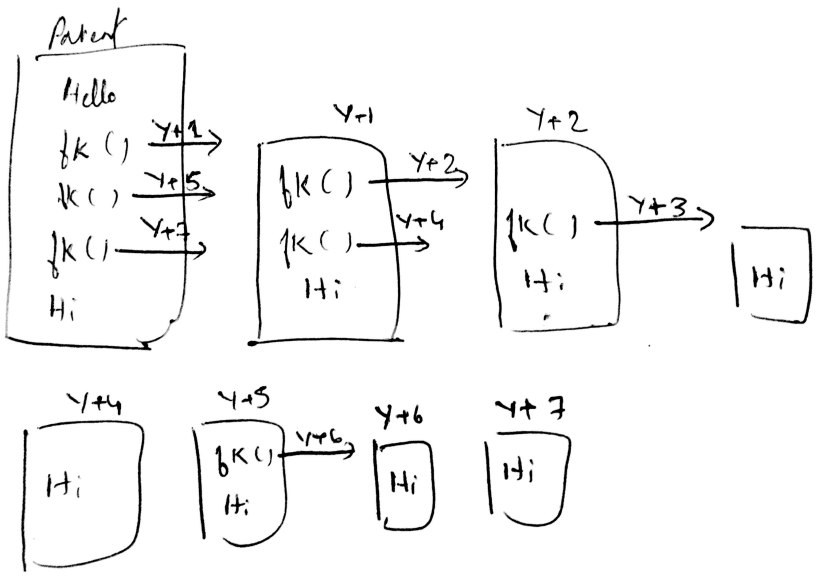

Figure 2.4: Explanation

Note

Upon successful creation of a child fork returns child process id in a parent and zero in a child. On failure fork() returns -1 and no child is created & errno is set appropriately.

Listing 2.8: programs/fork3.c

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int ret;
    printf("Hello\n");
    ret = fork();
    printf("ret = %d\n", ret);
    while(1);
}
```

Listing 2.8: Ouput

```
Hello
ret = 7254
ret = 0
^C
```

2.5.2 Seperating child and parent code using if-else

Listing 2.9: programs/fork4.c

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

int main()
{
    int ret;
    printf("Hello\n");
    ret = fork();

    if(ret == 0)
    {
        // Exclusive child code
        printf("In child, PID = %d, PPID = %d\n", getpid(),
            getppid());
    }
    else
    {
        // Exclusive parent code
        printf("In parent, PID = %d, PPID = %d\n", getpid(),
            getppid());
    }

    // Common code
    printf("Common code, pid = %d, ppid = %d\n", getpid(),
        getppid());
    while(1);
}
```

Listing 2.9: Ouput

```
Hello
In parent, PID = 7299, PPID = 3897
Common code, pid = 7299, ppid = 3897
In child, PID = 7300, PPID = 7299
Common code, pid = 7300, ppid = 7299
^C
```

Here child executes the if part and parent exuctes the else part. All code outside if, else is executed by both child and parent

Excercise

Write a program to create 3 child process from the same parent

Listing 2.10: programs/fork5.c

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

int main()
```

```
{
    printf("Hello\n");

    if(fork() == 0)
        printf("In Child 1, pid = %d, ppid = %d\n", getpid(),
           getppid());
    else
        if(fork() == 0)
            printf("In child2, PID = %d, PPID = %d\n", getpid
               (), getppid());
        else
            if(fork() == 0)
                printf("In child3, PID = %d, PPID = %d\n",
                   getpid(), getppid());
            else
                printf("In parent PID = %d, PPID = %d\n",
                   getpid(), getppid());
    while(1);
}
```

Listing 2.10: Ouput

```
Hello
In parent PID = 7462, PPID = 3897
In child3, PID = 7465, PPID = 7462
In child2, PID = 7464, PPID = 7462
In Child 1, pid = 7463, ppid = 7462
^C
```

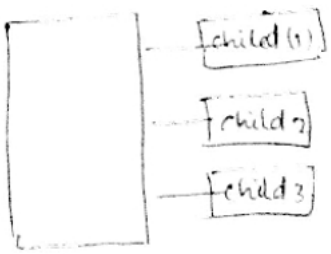

Figure 2.5: Structure

Bibliography

Books

Articles

Index