PES UNIVERSITY

(Established under Karnataka Act No.16 of 2013)
100-ft Ring Road, BSK III Stage, Bangalore – 560 085

Department of Computer Science & Engg

Session: Jan-May 2021

UE19CS254: Operating Systems

Unit 1 Notes

Read the sections mentioned

Chapter 1 Introduction	
Topics	**Page No**
1.1 What Operating Systems Do	**3-6**
1.2 Computer-System Organization	**7-12**
1.3 Computer-System Architecture	**13-18**
1.4 Operating-System Structure	**19-21**
1.5 Operating-System Operations	**21-24**
1.10 Kernel Data structures	**31-35**
1.11 Computing Environments	**35-43**
Chapter 2 Operating-System Structures	
2.1 Operating-System Services	**53-56**
2.6 Operating-System Design and Implementation	**73-75**
Chapter 3 Processes	
3.1 Process Concept	**103-107**
3.2 Process Scheduling	**108-112**
3.3 Operations on Processes	**113-119**
3.4 Interprocess Communication	**120-128**
3.6.3 pipes	**140-145**
Chapter 5 CPU Scheduling	
5.1 Basic Concepts	**201-205**
5.2 Scheduling Criteria	**205-206**
5.3 Scheduling Algorithms	**206-216**
5.5 Multiple-Processor Scheduling	**218-223**
5.6 Real-Time CPU Scheduling	**223-230**
5.7 Case Study: Linux/Windows Scheduling Policies	**230-237**

Text Book

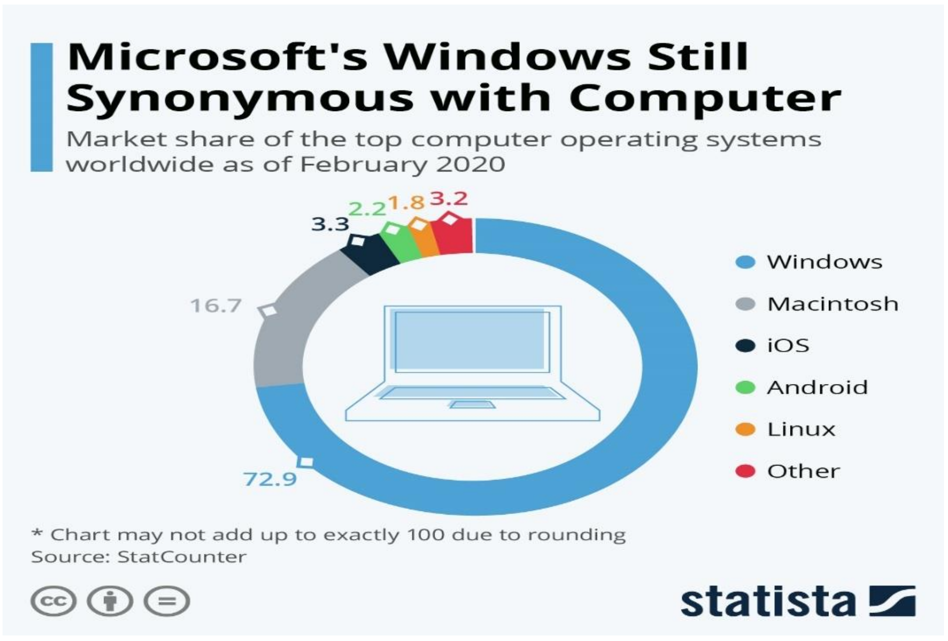
[1]. "Operating System Concepts", Abraham Silberschatz, Peter Baer Galvin, Greg Gagne 9th Edition, John Wiley & Sons, 2016, Indian Print.

Introduction to Operating System
The OS helps you to communicate with the computer without knowing how to speak the computer's language. It is not possible for the user to use any computer or mobile device without having an operating system.

History of OS

- Operating systems were first developed in the late 1950s to manage tape storage
- The General Motors Research Lab implemented the first OS in the early 1950s for their IBM 701
- In the mid-1960s, operating systems started to use disks
- In the late 1960s, the first version of the Unix OS was developed
- The first OS built by Microsoft was DOS. It was built in 1981 by purchasing the 86-DOS software from a Seattle company
- The present-day popular OS Windows first came to existence in 1985 when a GUI was created and paired with MS-DOS.

Different Operating systems with their market share



Services provided by the OS to the users

1. **Program Execution**: The Operating System is responsible for execution of all types of programs whether it be user programs or system programs. The Operating System utilises various resources available for the efficient running of all types of functionalities.

2. **Handling Input/Output Operations**: The Operating System is responsible for handling all sort of inputs, i.e, from keyboard, mouse, desktop, etc. The Operating System does all interfacing in the most appropriate manner regarding all kind of Inputs and Outputs.
   For example, there is difference in nature of all types of peripheral devices such as mouse or keyboard, then Operating System is responsible for handling data between them.
3. **Manipulation of File System**: The Operating System is responsible for making of decisions regarding the storage of all types of data or files, i.e, floppy disk/hard disk/pen drive, etc. The Operating System decides as how the data should be manipulated and stored.
4. **Error Detection and Handling**: The Operating System is responsible for detection of any types of error or bugs that can occur while any task. The well secured OS sometimes also acts as countermeasure for preventing any sort of breach to the Computer System from any external source and probably handling them.
5. **Resource Allocation:** The Operating System ensures the proper use of all the resources available by deciding which resource to be used by whom for how much time. All the decisions are taken by the Operating System.
6. **Accounting:** The Operating System tracks an account of all the functionalities taking place in the computer system at a time. All the details such as the types of errors occurred are recorded by the Operating System.
7. **Information and Resource Protection:** The Operating System is responsible for using all the information and resources available on the machine in the most protected way. The Operating System must foil an attempt from any external resource to hamper any sort of data or information.

Process Management

Process creation

Every process has a unique process ID, a non-negative integer. An existing process can create a new process by calling the `fork` function.

Prototype of fork:

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

Returns: 0 in child, process ID of child in parent, 1 on error

The new process created by fork is called the *child process*. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas 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 that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid() to obtain the process ID of its parent. Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent.The child gets a copy of the parent's data space, heap, and stack.

Program to demonstrate process creation

1. Program to print process ID’s

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid;
pid=fork();// return 3 values, 0--to the child process, >0 paren
t process, -1 error
if(pid<0)
{
printf("fork error\n");
exit(1);
}
else if(pid==0)
{
printf("child process\n");
printf("child id=%d, parent id=%d\n",getpid(),getppid());
}
else
{
sleep(2);
printf("parent process\n");
printf("parent id=%d, child id=%d\n",getpid(),pid);
}
return 0;
}
```

2. Program to demonstrate that copy of the data is given to the child process after forking.

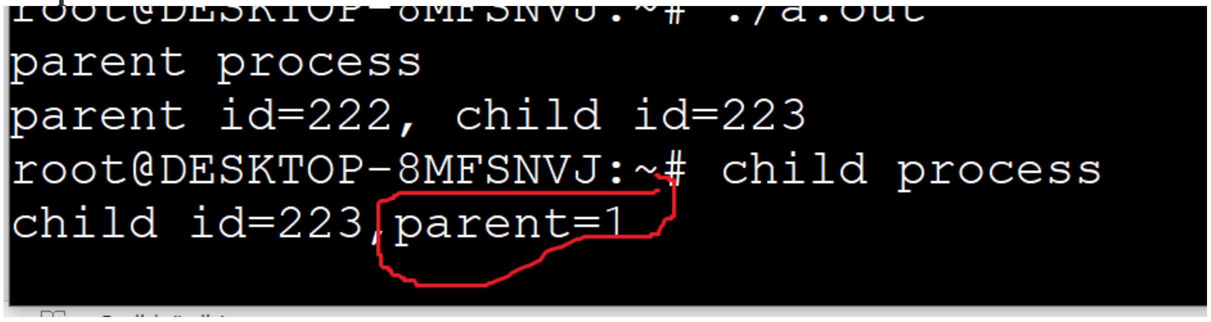
```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
```

```
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";
int
main(void)
{
int var; /* automatic variable on the stack */
pid_t pid;
var = 88;
if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
printf("write error");
printf("before fork\n"); /* we don't flush stdout */
if ((pid = fork()) < 0) {
printf("fork error");
} else if (pid == 0) { /* child */
glob++; /* modify variables */
var++;
} else {
sleep(2); /* parent */
}
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
exit(0);
}
```

Program to demonstrate the creation of orphan process

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid;
pid=fork();// return 3 values, 0--to the child process, >0 pare>if(pid<0)
{
printf("fork error\n");
exit(1);
}
else if(pid==0)
{
sleep(5)
printf("child process\n");
printf("child id=%d,parent=%d\n",getpid(),getppid());

}
else
{

printf("parent process\n");
printf("parent id=%d, child id=%d\n",getpid(),pid);
exit(1);
}
return 0;
}
```

Output



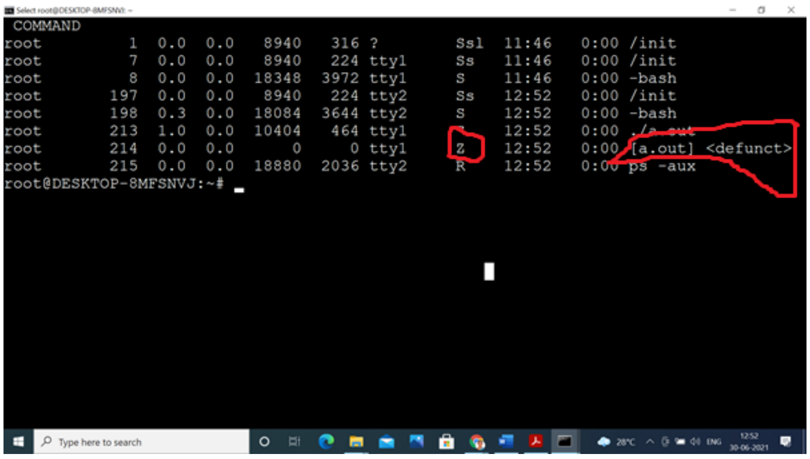
```
root@DESKTOP-8MFSNVJ:~# ./a.out
parent process
parent id=222, child id=223
root@DESKTOP-8MFSNVJ:~# child process
child id=223, parent=1
```

In this program the parent process terminates before the child process terminates. Thus, the child process will become orphan process. Any of the orphan process are taken care by the init process whose id is always 1. In the output you can observe that parent id for the child process is 1.

Program to demonstrate the creation of zombie process

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid;
pid=fork();// return 3 values, 0--to the child process, >0 pare>if(pid<0)
{
printf("fork error\n");
exit(1);
}
else if(pid==0)
{
printf("child process\n");
printf("child id=%d,parent=%d\n",getpid(),getppid());
exit(1);
}
else
{
sleep(5);
printf("parent process\n");
printf("parent id=%d, child id=%d\n",getpid(),pid);
//exit(1);
}
return 0;
}
```

Output

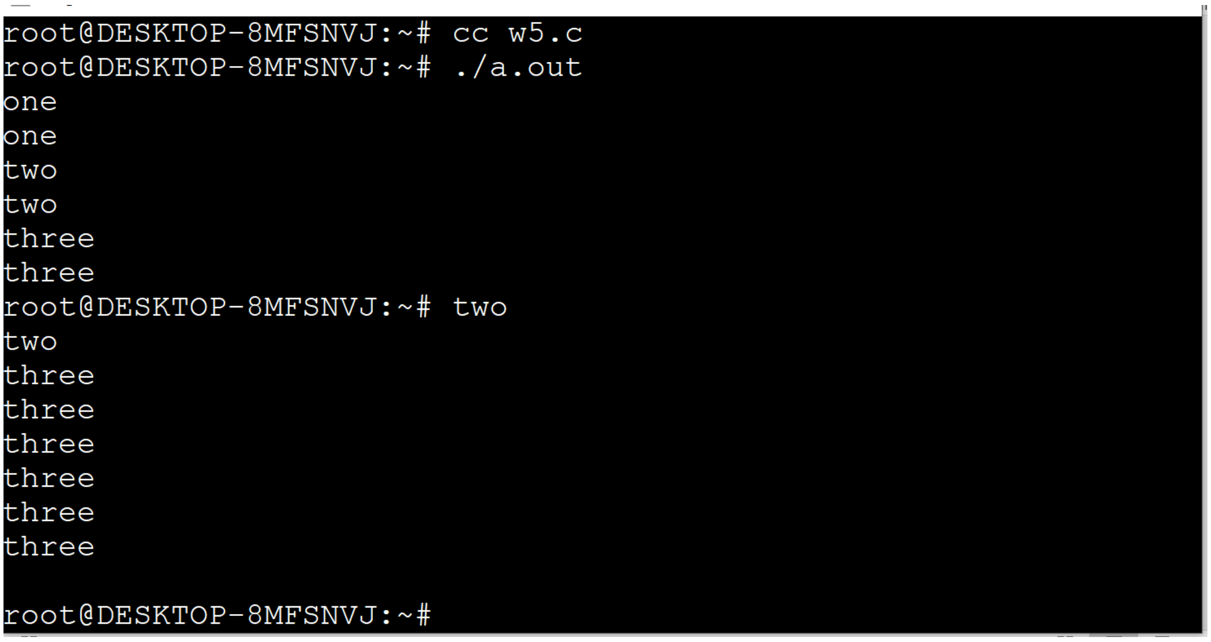


```
 COMMAND
root           1  0.0  0.0   8940   316 ?        Ssl  11:46   0:00 /init
root           7  0.0  0.0   8940   224 tty1     Ss   11:46   0:00 /init
root           8  0.0  0.0  18348  3972 tty1     S    11:46   0:00 -bash
root         197  0.0  0.0   8940   224 tty2     Ss   12:52   0:00 /init
root         198  0.3  0.0  18084  3644 tty2     S    12:52   0:00 -bash
root         213  1.0  0.0  10404   464 tty1     S    12:52   0:00 ./a.out
root         214  0.0  0.0      0     0 tty1     Z    12:52   0:00 [a.out] <defunct>
root         215  0.0  0.0  18880  2036 tty2     R    12:52   0:00 ps -aux
root@DESKTOP-8MFSNVJ:~# _
```

Execute the ps-aux command in the different terminal to see the status of the process. The “z” in the stat column indicates that the child process has become zombie. <defunct> marked in the out also shows that it is a zombie process. The child process has become zombie as the termination status of child is not taken by the parent. During this time the parent was sleeping.

Program to demonstrate that the code following the fork will be copied to both parent and child process.

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
fork();
printf("one\n");
fork();
printf("two\n");
fork();
printf("three\n");
}
```



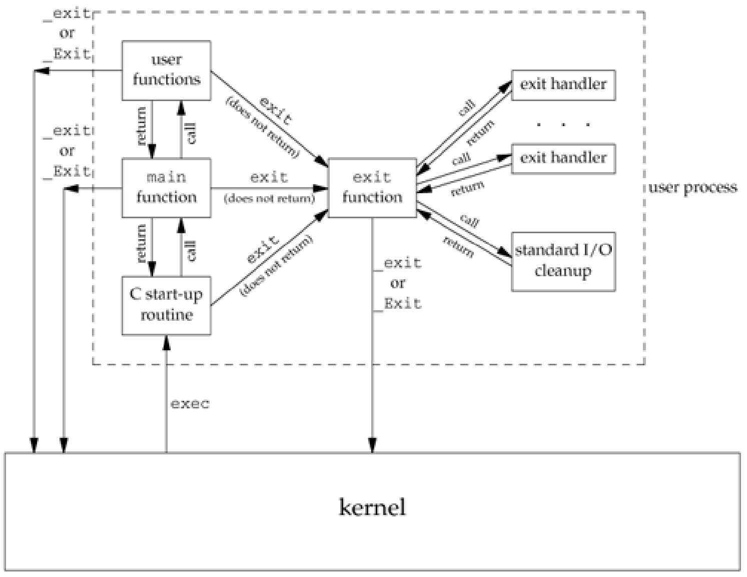
```
root@DESKTOP-8MFSNVJ:~# cc w5.c
root@DESKTOP-8MFSNVJ:~# ./a.out
one
one
two
two
three
three
root@DESKTOP-8MFSNVJ:~# two
two
three
three
three
three
three
three

root@DESKTOP-8MFSNVJ:~#
```

Process Termination

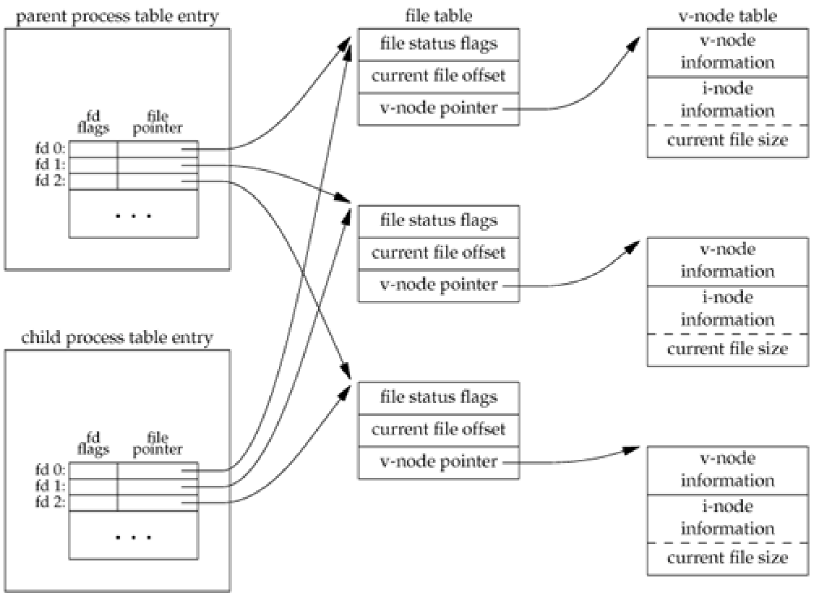
1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine.
5. Calling pthread_exit from the last thread
Abnormal termination occurs in three ways:
6. Calling abort
7. Receipt of a signal
8. Response of the last thread to a cancellation request

Three functions that terminate a program normally are _exit and _Exit, which return to the kernel immediately, and exit, which performs certain clean up processing and then returns to the kernel. The exit function has always performed a clean shutdown of the standard I/O library. All three exit functions expect a single integer argument, which we call the exit status.

How a C program is started and how it terminates



Sharing of the files between parent and child after fork



It is important that the parent and the child share the same file offset. Consider a process that forks a child, then waits for the child to complete. Assume that both processes write to standard output as part of their normal processing. If both parent and child write to the same descriptor, without any form of synchronization, such as having the parent wait for the child, their output will be intermixed.

wait Function

The **wait**() **system call** suspends execution of the current process until one of its children terminates.

Prototype

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

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates. If only one child process is terminated, then return a wait() returns process ID of the terminated child process.

If more than one child processes are terminated than wait() reap any ***arbitrarily child*** and return a process ID of that child process.

When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**.

Program to demonstrate working of wait

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

int main()
{
	if (fork()== 0)
		printf("child process\n");
	else
	{

		wait(NULL);
		printf("Parent process\n");
	}


	return 0;
}
```

Reference Books:

1. W. Richard Stevens, Stephen A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, **Addison Wesley Professional, 2005**