SRES's Sanjivani College of Engineering, Kopargaon
(*An Autonomous Institute*)
Department of Computer Engineering

SPOS Lab Manual

Assignment No. 12

AIM:
Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming)

PROBLEM DEFINITION:
Implement UNIX system calls like ps, fork, join, exec family, and wait for process management (use shell script/ Java/ C programming)

OBJECTIVES:
- To understand UNIX system call
- To understand Concept of process management
- Implementation of some system call of OS

THEORY:
SYSTEM CALL:
- When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

- When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

- Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

- To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

Examples of Windows and Unix System Calls -

	WINDOWS	UNIX
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile(), ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleeo()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Basics

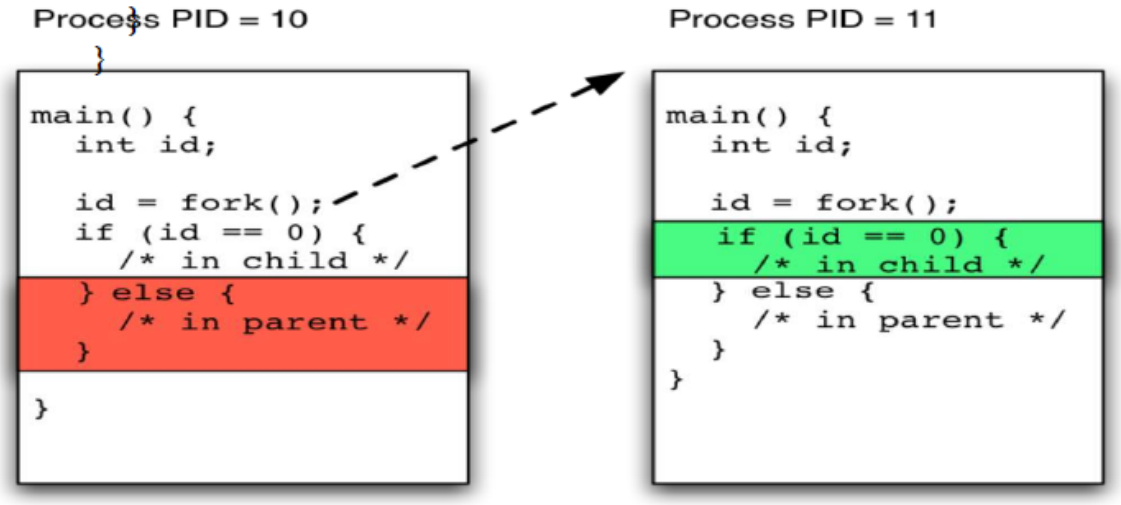
- Since system calls are functions, we need to include the proper header files
  - E.g., for getpid() we need
    - #include <sys/types.h>
    - #include <unistd.h>
- Most system calls have a meaningful return value
  - Usually, -1 or a negative value indicates an error
  - A specific error code is place in a global variable called
    - errno
  - To access errno you must declare it:
    - extern int errno;

UNIX Processes

- Recall a process is a program in execution
- Processes create other processes with the fork() system call
- fork() creates an identical copy of the parent process
- We say the parent has cloned itself to create a child
- We can tell the two process apart use the return value of fork()
    - In parent: fork() returns the PID of the new child
    - In child: fork() returns 0
- fork() may seem strange at first, that's because it is a bit strange!
- Draw picture

```
main() {
  int id;

  id = fork();
  if (id == 0) {
    /* in child */
  } else {
    /* in parent */
  }
}
```

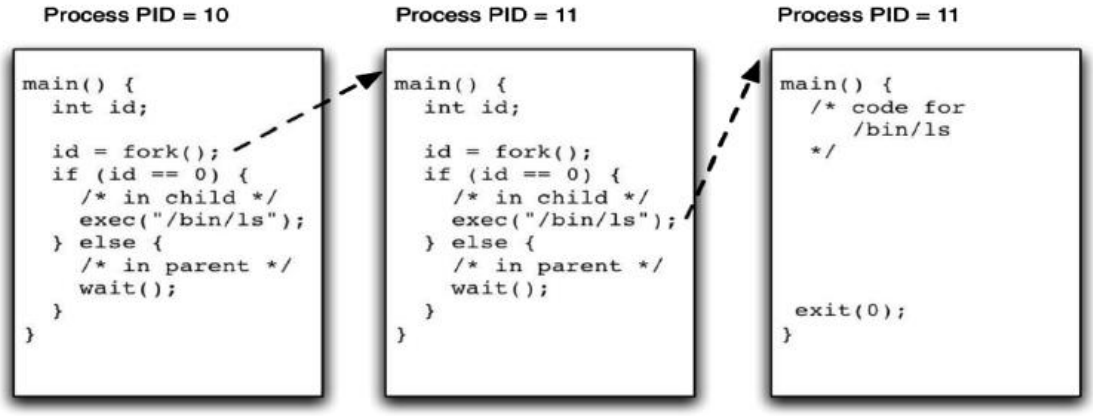


Starting New Programs

- fork() only allows us to create a new process that is a duplicate of the parent
- The exec() system call is used to start a new program
- exec() replaces the memory image of the calling processes with the image of the new program
- We use fork() and exec() together to start a new program

```
main() {
 int id;
id = fork();
```

```
if (id == 0) {
        /* in child */
        exec("/bin/ls");
} else {
        /* in parent */
        wait();
        }
}
```



Syscalls for Processes

- pid_t fork(void)
  - Create a new child process, which is a copy of the current process
  - Parent return value is the PID of the child proces
  - Child return value is 0

- int execl(char *name, char *arg0, ..., (char *) 0)
  - Change program image of current process
  - Reset stack and free memory
  - Start at main()
  - Also see other versions: execlp(), execv(), etc.

- pid_t wait(int *status)
  - Wait for a child process (any child) to complete
  - Also see waitpid() to wait for a specific process

- void exit(int status)
  - Terminate the calling process
  - Can also achieve with a return from main()

- int kill(pid_t pid, int sig)
  - Send a signal to a process
  - Send SIGKILL to force termination

- **UNIX SYSTEM CALLS :-**

A] Ps command :

The *ps* (i.e., *process status*) command is used to provide information about the currently running *processes*, including their *process identification numbers* (PIDs).

A process, also referred to as a *task*, is an *executing* (i.e., running) instance of a program. Every process is assigned a unique PID by the system.

The basic syntax of ps is

ps [options]

When ps is used without any options, it sends to *standard output*, which is the display monitor by default, four items of information for at least two processes currently on the system: the *shell* and ps. A shell is a program that provides the traditional, text-only user interface in Unix-like operating systems for issuing commands and interacting with the system, and it is *bash* by default on Linux. ps itself is a process and it *dies* (i.e., is terminated) as soon as its output is displayed.

The four items are labeled PID, TTY, TIME and CMD. TIME is the amount of CPU (central processing unit) time in minutes and seconds that the process has been running. CMD is the name of the command that launched the process.

B] Fork()

- The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.
- The process which called the fork() call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.
- Both processes start execution from the next line of code i.e., the line after the fork() call. Let's look at an example:

```
//example.c
#include <stdio.h>
void main() {
int val;
val = fork(); // line A
printf("%d",val); // line B
}
```

- When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the fork() call.
- The difference is that, in the parent process, fork() returns a value which represents the **process ID** of the child process. But in the child process, fork() returns the value 0.

- This means that according to the above program, the output of parent process will be the **process ID**of the child process and the output of the child process will be 0.

C] Join Command :

The join command in UNIX is a command line utility for joining lines of two files on a common field. It can be used to join two files by selecting fields within the line and joining the files on them. The result is written to standard output.

Join syntax :

Join [option]….. file1 file2

How to join two files

To join two files using the join command files must have identical join fields. The default join field is the first field delimited by blanks. For the following example there are two files college.txt and city.txt.s

cat college.txt

1 pvg
2 met
3 mit

cat city.txt

1 nashik
2 nashik
3 pune

These files share a join field as the first field and can be joined.

join college city.txt

1 pvg nashik
2 met nashik
3 mit pune

D] Exec()

- The exec() system call is also used to create processes. But there is one big difference between fork() and exec() calls. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.
- It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist
- There are many flavors of exec() in UNIX, one being exec1() which is shown below as an example

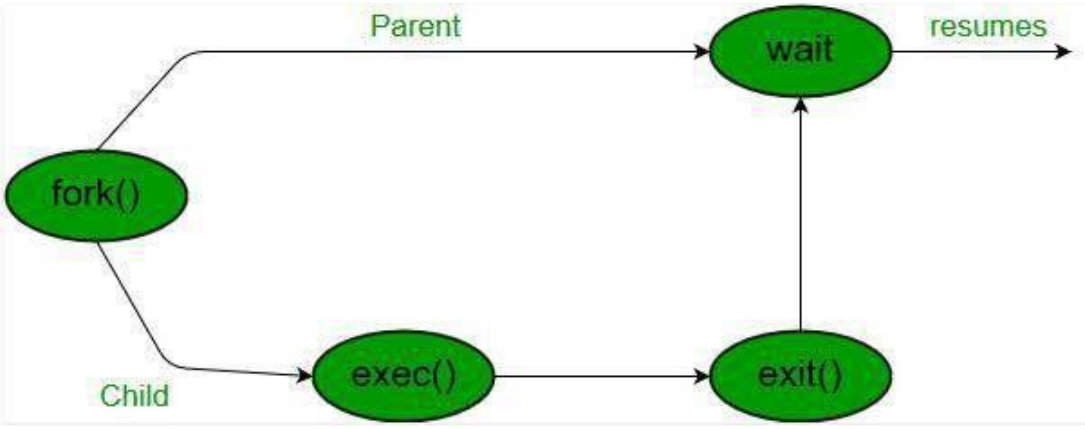
```
//example2.c
#include
Void main() {
execl("/bin/ls", "ls", 0); //line A
printf("This text won't be printed unless an error occurs in exec().");
}
```

- As shown above, the first parameter to the execl() function is the address of the program which needs to be executed, in this case, the address of the **ls** utility in UNIX. Then it is followed by the name of the program which is **ls** in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).
- When the above example is executed, at line A, the **ls** program is called and executed and the current process is halted. Hence the printf() function is never called since the process has already been halted. The only exception to this is that, if the **execl()** function causes an error, then the printf()function is executed.

E] Wait ()

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. Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.




```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t * infop , int options );
```

System Calls vs Library Functions

- A system call is executed in the kernel
    - p = getpid();
- A library function is executed in user space
    - n = strlen(s);
- Some library calls are implemented with system calls
    - printf() really calls the write() system call
- Programs use both system calls and library functions

ALGORITHM:
Write algorithm as per your program

CONCLUSION:

In this assignment we learn and implemented ps, fork, join, exec and wait system calls.

References:
https://en.wikipedia.org/wiki/System_call
https://en.wikipedia.org/wiki/Process_management_(computing)
https://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture25.pdf
https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/14/lec.html
https://www.thegeekstuff.com/2012/03/c-process-control-functions/

Prepared by
Prof.N.G.Pardeshi
Subject Teacher

Approved by
Dr. D.B.Kshirsagar
HOD