EECS 678: Introduction to Operating Systems
IPC: Course Notes For Reference

Unix is a multitasking operating system which offers rich functionality in interprocess communication (IPC). Here we take a brief look at system calls which are used in process manipulation and interprocess communication. Process manipulation is important as it leads to multitasking and interprocess communication is important as it leads to network programming.

Process

A process in Unix terms is an instance of an executing program. Each process incorporates program code, the data values within program variables, and values held in hardware registers, program stack etc.

Associated with each process is a process control block (PCB) which stores all the information related to the process.
Unix provides a handful of system calls for creation and manipulation of processes.

- fork .... Creates a new process.
- exec .... Overlays the memory space of the process with a new program.
- wait .... synchronizes processes.
- exit .... Terminates a process.

The *fork* System Call.

`Usage`

```
int pid;
pid = fork();
```

The *fork* is the basic process creation system call in Unix, which transforms Unix into a multitasking operating system.

Any Unix process may create other processes. The calling process is called the parent process and the one being created is called the child process. Once created, both child and parent execute concurrently, resuming execution immediately after the call to *fork*, as shown in figure 1.

Figure 1 demonstrates the notion of process creation more clearly. The arrow labeled PC (Program Counter) shows the statement currently being executed. Before the call to *fork* one process exists and after the call to *fork* child process is created and now both processes run together (concurrently). From the figure 1 you can see that the child process is an exact replica of the parent process, (its PCB, code and data segments); immediately after the *fork* system call both the program counters point to the same instruction.

The call to *fork* returns an integer which is called the process id (pid). It is value of pid that distinguishes the child and the parent. In the parent pid is set to a non-zero, positive integer. In the child it is set to zero.

```
/* example: fork demonstration */

void main() {
  int pid;  /* holds the process id value in parent */

  printf(" So far one process \n");
  printf(" Calling fork \n");

  pid = fork();

  if (pid == 0)
    printf("I am the child\n");
  else if (pid > 0)
    printf("I am the parent\n");
  else
    printf("fork failed\n");
}
```

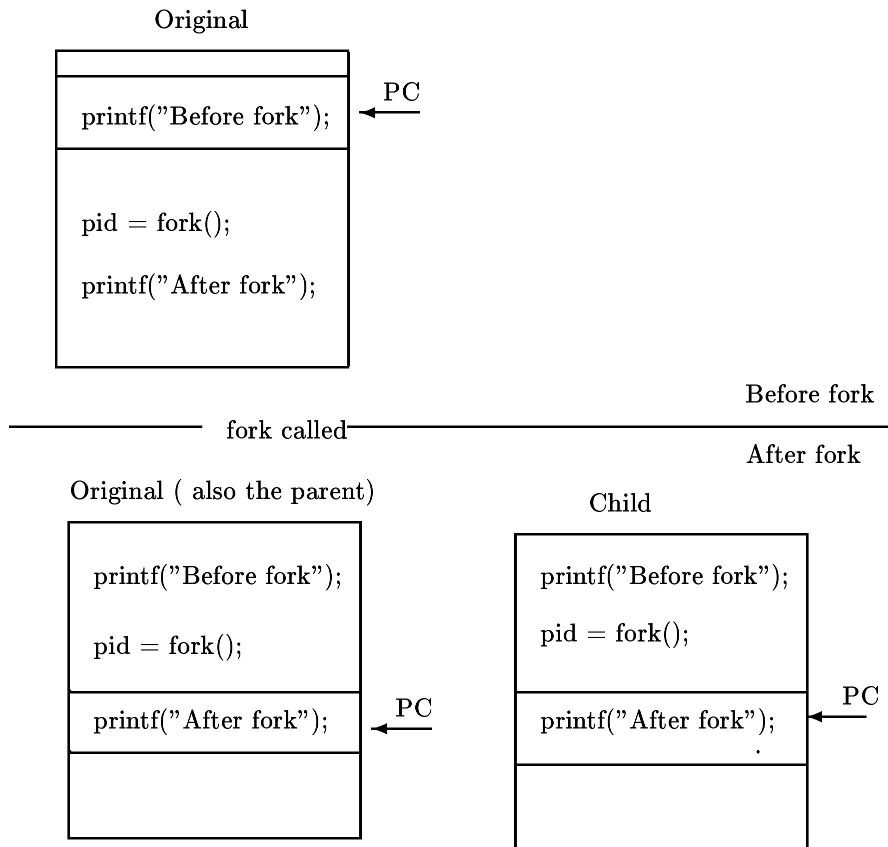

Figure 1 The fork system call

The *exec* System Call.

The only way in which a program is executed in Unix is for an existing program to issue an *exec* system call. The *exec* calls overlays the memory space of the calling process with a new program. When used together, *fork* and *exec* provides the means for the programmer to start another program, yielding multitasking.

(*exec* is a family of system calls; see the *man* pages for more information.)

The sample code provided below shows how *fork* and *exec* can be used together to achieve multitasking. Here the parent process creates a child and then uses *execl* system call to overlay the *ls* program on top of the child's address space causing it to run the *ls* program. The *wait* system call (discussed next) is used to suspend the parent process till the child terminates.

Figure 2 illustrates this idea. There you see once the child process executes the *execl* system call, the address space of the child is replaced by the *ls* program.

```
/* example fork exec together */
void main() {
  int pid;

  pid = fork();

  /* child executing ls program */
  if (pid == 0) {
    execl("/bin/ls", "ls", "-l", (char *)0);
  }

  /* parent waits for child to finish */
  if (pid > 0)
    wait((int *)0);
}
```

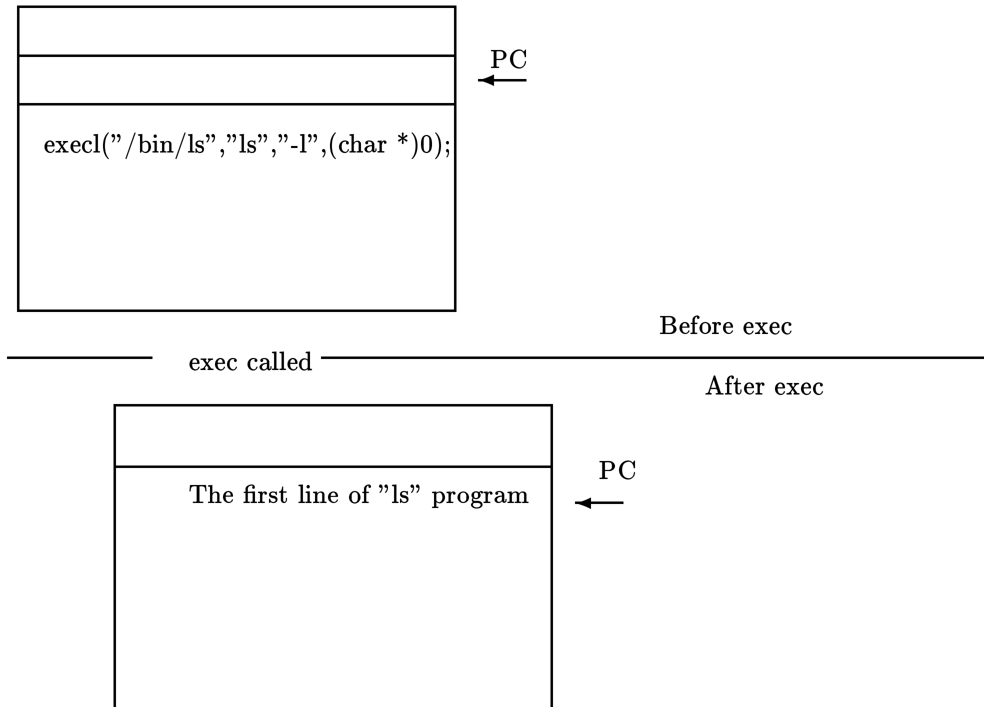

Figure 2 The exec system call

The `wait` System Call.

Usage

```
int status, retval;
retval = wait(&status);

or

retval = wait((int *)0);
```

The *wait* is a basic process synchronization call in Unix. It temporarily suspends the execution of the parent process. In certain applications, the parent process is suspended using *wait* system call while the child process is running; the waiting parent process restarts once the child finishes execution.

The *exit* System Call.

Usage

```
int status;
exit(status);
```

The *exit* system call is used to terminate a process. A process also stops when it reaches the end of the main function or when a *return* statement is executed in the main function.

Interprocess Communication (IPC)

For processes to cooperate in performing a task, they need to share data. The fundamentals of network programming lies in processes' ability to share data among themselves.

Unix allows concurrent processes to communicate by using a variety of IPC methods, in the form of pipes, signals, message queues, shared memory, semaphores and sockets. Here we discuss only the basic ideas of pipes and sockets.

Pipes

`Usage`

```
int p[2];
int retval;
retval = pipe(p);
```

A call to *pipe* creates a pair of file descriptors, pointed to a pipe inode. Pipes are normally used to couple the output of one program to the input of of another program without having to store data in an intermediate file. They are usually used as unidirectional communication channels, which operate in First In First Out (FIFO) basis.

Note: Do not confuse this with the FIFO IPC mechanism.

Usually, pipe is called with a two integer array, which will hold the file descriptors associated with the pipe. The file descriptor denoted by the first element in the array (p[0]) opens the pipe for reading while the other file descriptor (p[1]) opens the pipe for writing.

Once created we can use pipes to communicate with the process itself or with any child process.

Sample code below demonstrates how a pipe is created using *pipe* system call and the how the returned file descriptors can be used to writing down the pipe and reading from it.

```
#define MSGSIZE 16
char *msg1 = "Hello world #1";
char *msg2 = "Hello world #2";

void main() {
  char buf[MSGSIZE];
  int p[2], j;

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

  /* writing down the pipe */
  write(p[1], msg1, MSGSIZE);
  write(p[1], msg2, MSGSIZE);
```

```
  /* read from the pipe */
  for (j=0; j<2; j++) {
    read(p[0], buf, MSGSIZE);
    printf("%s\n", buf);
  }
}
```

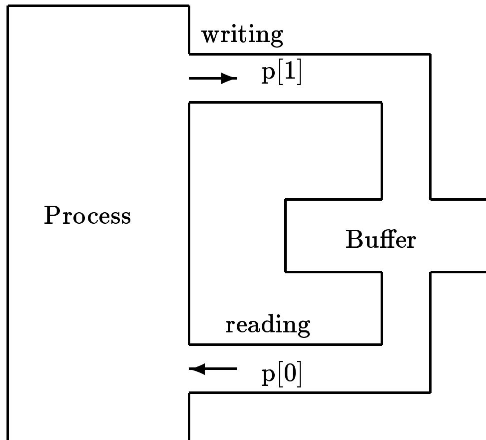

Figure 2 Process sending data to itself through a pipe.

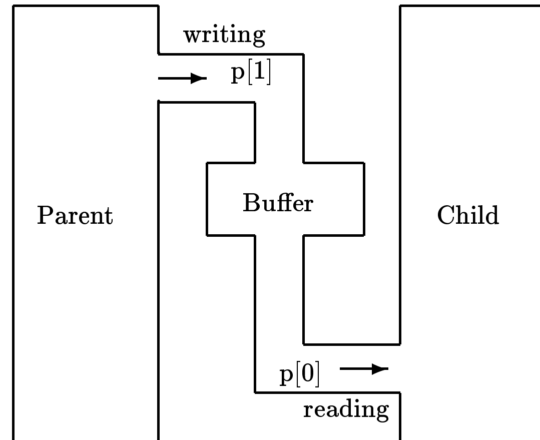

Figure 3 Parent sending data to the child through a pipe.

Note: A buffer is a system maintained data structure accessible to both ends of the pipe. The writer writes on to the buffer while the reader reads from it.

Sockets

A socket is an end point for communication which assumes client-server relationship between the processes.

The server normally waits for requests at a fixed address. The address is a combination of the machine *Internet* address and the port address at which the service is offered. Clients get the service by sending requests to the fixed address of the server.

Associated with sockets are a handful of system calls. Sockets are created by the *socket* system call which returns a socket descriptor on success. Both the client and the server call *socket* initially. The server then calls *bind* to establish its address. The *listen* call specifies the number of requests that can wait on the queue. Servers handle these requests in different ways depending on the application, which we are not going to discuss in detail here. Interested readers may read *Internetworking with TCP/IP - vol 3 by Douglas B. Comer.*

The client, after calling *socket*, may call bind to establish its address though it is not necessary. The client then calls *connect* to request connection to the server.

At this point the client and the server are in communication. They can use the *read* and *write* system calls for communication. Once the communication is done, *close* is called to close the connection.

Standard Input, Standard Output, Standard Error

A Unix shell automatically opens three files for any executing program.

- standard input (stdin) - keyboard by default.
- standard output (stdout) - screen by default.
- standard error (stderr) - screen by default.

Within the program they are always identified by file descriptors 0, 1, 2 respectively.

Filters

By now you should be familiar with commands like this at the shell prompt.

```
$ ls | wc
```

Did you ever wonder how this works ? Here the output of the *ls* program is sent to the input of the program *wc* through a pipe.

To achieve this, before invoking the *wc* program, the shell sends standard output of *ls* to the write end of the pipe and standard input of *wc* to the read end of the pipe. This is achieved with the *dup2* or *dup* system calls.

The *dup2* system call

`Usage`

```
dup2(int oldfd, int newfd);
```

The system call *dup2* makes new file descriptor (newfd) be the copy of old file descriptor (oldfd), closing new file descriptor first if necessary. Hence coupling of oldfd to newfd is achieved.

```
dup2(fd, 1);  /* now fd coupled to stdout */
              /* anything written to fd now be sent to stdout */
```

The *dup* system call

The *dup* is called with some file descriptor as an argument, and returns a new file descriptor that refers to the same file. The new file descriptor value will be the lowest number available in the system. By closing any of stdin, stdout or stderr we can couple any file descriptor to any of the standard file descriptors as shown below.

```
close(1);    /* stdout is closed */
dup(fd);     /* now fd coupled to stdout */
```

Read the *man* pages for more information.