# The UNIX Process

- It is important to understand the difference between a **program** and a **process**.

- A program is a collection of instructions and data that is kept in an ordinary file on disk. In its i-node the file is marked executable, and the file's contents arranged according to rules established by the kernel.

- In order to run the program, the kernel is first asked to create a new process, which is an environment in which a program executes.

- A process consists of three segments: instruction segment, user data segment, and system data segment. After this initialization, there is no longer any fixed connection between a process and the program it is running.

- Several concurrently running processes can be initialized from the same program. There is no functional relationship, however, between these processes.

- A process in UNIX terms is simply an instance of an executing program, corresponding to the notion of a task in other environments.

- Each process incorporates program code, the data values within program variables, and more exotic items such as the values held in hardware registers, the program stack and so on.

- The shell creates a new process each time it starts up a program in response to a command. For example, the command line

  **$ cat file1 file2**

  results in the shell creating a process especially to run the cat command.

- The slightly more complex command line

  ***$ ls | wc -l***

  results in two processes being created to run the commands ***ls*** and ***wc*** concurrently. (In addition the output of the directory listing is piped into the word count program wc.)

- Because a process corresponds to an execution of a program, processes should not be confused with the programs they run; indeed several processes can concurrently run the same program, something that often happens with commonly used utilities.

- For example, several users may be running the same editor program simultaneously, each invocation of the one program counting as a separate process.

- Any UNIX process may in turn start other processes. This gives the UNIX process environments a hierarchical structure paralleling the directory tree of the file system.

- At the top of the process tree is a single controlling process, an execution of an extremely important program called init, which is ultimately the ancestor of all user processes.

- For the programmer, UNIX provides a handful of system calls for process creation and manipulation. The most important of these are:

  ***fork*** - Used to create a new process by duplicating the calling process. fork is the basic process creation primitive.

  ***exec*** - A family of system calls, each of which performs the same function: the

  transformation of a process by overlaying its memory space with a new program.

  The differences between exec calls lie mainly in the way their argument lists are constructed.

  ***wait*** - This call provides rudimentary process synchronization. It allows one process to wait until another related process finishes.

  ***exit*** - Used to terminate a process.

**Creating Processes**

- The **fork** system call

- The fundamental process creation primitive is the fork system call. It is the mechanism that transforms UNIX into a **multitasking** system.

- Usage

```
pid_t pid;

pid = fork ();
```

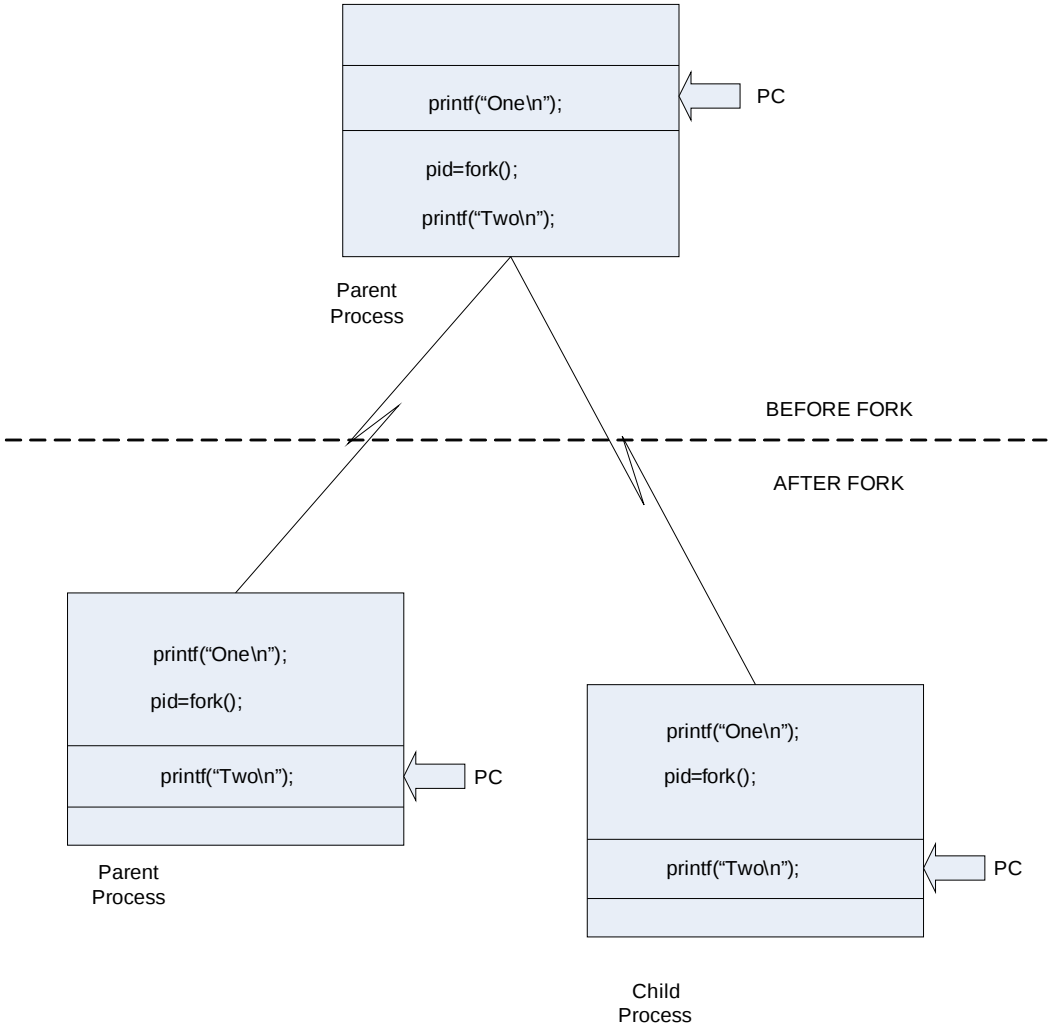
- A successful call to fork causes the kernel to create a new process which is a (more or less) exact duplicate of the calling process.

- In other words the new process runs a copy of the same program as its creator, the variable within this having the same values as those within the calling process, with a single important exception which we shall discuss shortly.

- The newly created process is described as the **child process**, the one that called fork in the first place is called, not unnaturally, its **parent**.

- After the call the parent process and its newly created offspring execute **concurrently**, both processes resuming execution at the statement immediately after the call to fork.

- The diagram provided demonstrates the notion more clearly. It centers around three lines of code, consisting of a call to ***printf***, followed by a call to fork and then another call to ***printf***.




- The BEFORE section shows things prior to the invocation of ***fork***. There is a single process labeled A, with the PC pointing to the currently executing statement.

- The AFTER section shows the situation immediately following the call to ***fork***. There are now two processes A and B running together.

- Process A is the same as in the BEFORE part of the diagram. B is the new process spawned by the call to ***fork***.

- It is an exact copy of A and is executing the same program as A. A is the parent process while B is its child.

- The two PC arrows in this part of the diagram show that the next statement executed by both parent and child after the fork invocation is a call to ***printf***.

- In other words, both A and B pick up at the same point in the program code, even though B is new to the system. The result of this state is that the message "**Two"** is displayed twice.

## Process-Ids

- As you can see from the usage description, fork is called without arguments and returns an unsigned integer value (pid_t). Both processes (parent and child) receive the return.

- The return value is different, however, which is crucial, because this allows their subsequent actions to differ.

- It is the value of **pid** that distinguishes child and parent. In the **parent**, **pid** is set to a **non-zero, positive integer**. In the **child** it is set to **zero**.

- Because the return value in parent and child differs, the programmer is able to specify different actions for the two processes.

- The number returned to the parent is called the **process-id** of the child process. It is this number that identifies the new process to the system, rather like a user-id identifies a user.

- Since all processes are born through a call to fork, every UNIX process has its own **process-id**, which at any one time is unique to that process.

- A **process-id** can therefore best be thought of as the identifying signature of its process.

- The following program demonstrates the action of fork, and the use of process-id:

```
/*------ demonstrate fork ---------*/
int main (void)
{
        int pid;  /*process-d in parent */

        printf ("Just one process so far\n");
        printf ("Calling fork .....\n");

        pid = fork ();      /* create new process */

        if (pid == 0)
                printf ("I'm the child pid = %d\n", pid);
        else if (pid > 0)
             printf ("I'm the parent, child pid = %d\n", pid);
        else
                printf ("fork returned error code, no child\n");
}
```

- Also notice that, since the two processes generated by the example program will run concurrently and without synchronization, there is no guarantee that the output from parent and child will not become confusingly intermingled.

- ***fork*** becomes valuable when combined with other UNIX facilities. For example, it is possible to have a child and parent process perform different but related tasks, cooperating by using one of the UNIX interprocess communication mechanisms such as ***pipes*** (to be discussed).

- Another facility often used in combination with fork is the ***exec*** system call.

## The *exec* system call

- All varieties of ***exec*** perform the same function: they transform the calling process by loading a new program into its memory space.

- If the ***exec*** is successful the calling program is completely overlaid by the new program, which is then started from its beginning.

- The result can be regarded as a new process, but one that is given the same process-id as the calling process.

- It is important to stress that exec does not create a new subprocess to run concurrently with the calling process. Instead the old program is obliterated by the new. So, unlike fork there is no return from a successful call to exec.

- We will spotlight one of the exec calls, namely **execl**.

- Usage

  **char *arg0, *arg1, ..., *argn;**

  **ret = execl (path, arg0, arg1,..., argn, (char *)0);**

- All parameters for **execl** are **character pointers**. The first, path gives the name of the file containing the program to be executed; with ***execl*** this must be a valid pathname.

- The file itself must contain a true program; ***execl*** cannot be used to run a file of shell commands.

- The second parameter **arg0** is, by convention, the name of the program or command stripped of any preceding pathname element.

- This and the remaining variable number of arguments (**arg1** to **argn**) are available to the invoked program, corresponding to command line arguments within the shell.

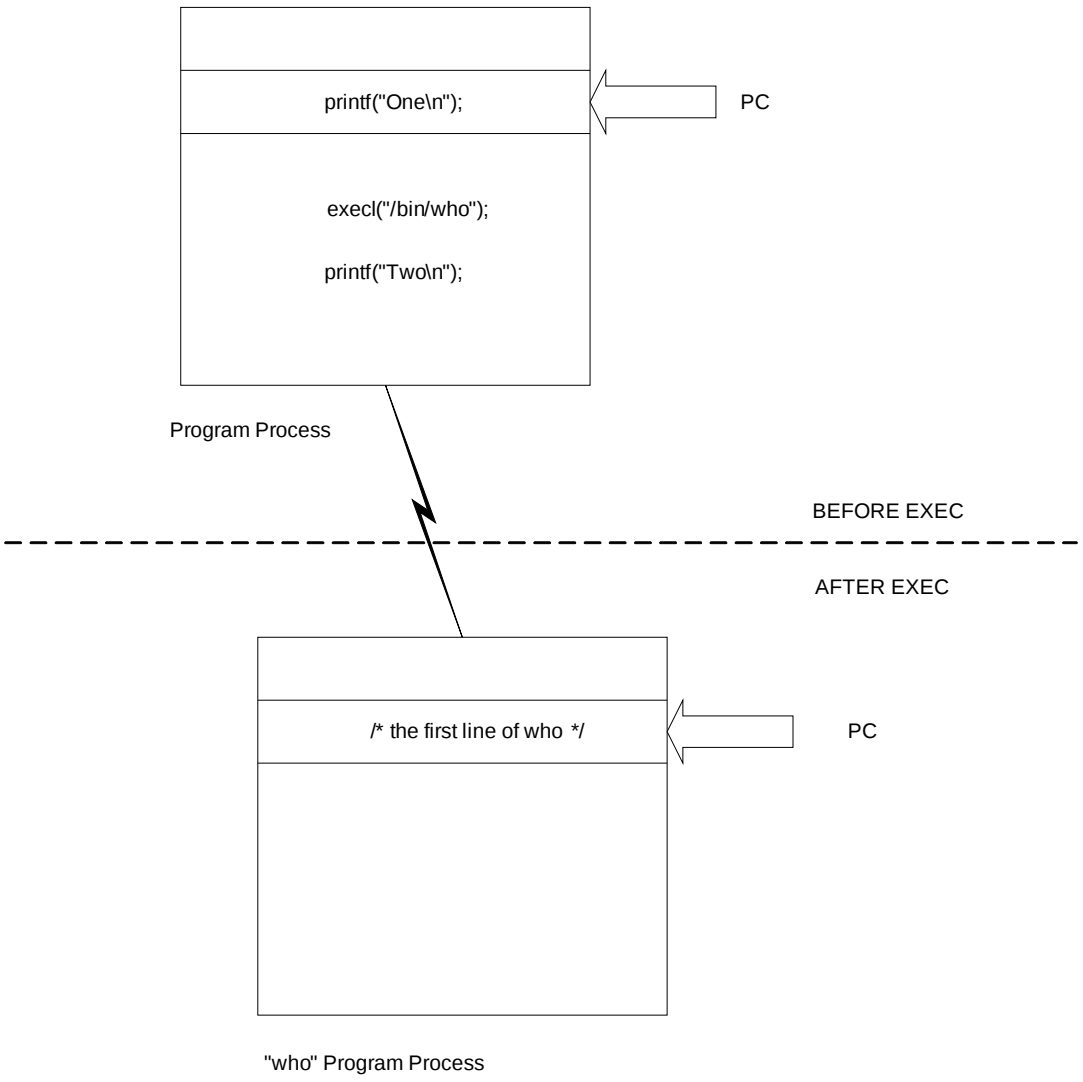
- Because the argument list is of arbitrary length, it must be terminated by a null pointer to mark the end of the list.

- As always, a short example is worth a thousand words, and the following program uses ***execl*** to run the directory listing program ***ls***:

```
/*runls -- use "execl" to run ls */

int main (void)
{
        printf ("executing ls\n");

        execl ("/bin/ls", "ls", "-l", (char *)0);

        /* if execl returns, the call has failed, so ...*/
        perror ("execl failed to run ls");
        exit (1);
}
```

- Notice that in the example, the call to ***execl*** is followed by an unconditional call to the library routine ***perror***. If the calling program does survive and ***execl*** returns, then an error must have occurred.

- As a corollary of this, when ***execl*** and its relatives do return, they will always return -1.


printf("One\n");
PC
execl("/bin/who");
printf("Two\n");
Program Process
BEFORE EXEC
AFTER EXEC
/* the first line of who */
PC
"who" Program Process

## The wait System Call

- ***wait*** temporarily suspends the execution of a process while a child process is running. Once the child has finished, the waiting parent is restarted.

- If more than one child is running then wait returns the first time one of the parent's offspring exits.

- ***wait*** is often called by a parent process just after a call to **fork**. For example:

```
pid = fork();       /* create new process */

if (pid == 0)
{
        /* child  */
        /* do something */
}
else
{
        /* parent, so wait for child */
        wait ((int *)0);
}
```

- This combination of ***fork*** and ***wait*** is used when the child process is intended to run a completely different program through ***exec***.

- The return value from wait is normally the process-id of the exiting child. If wait returns -1, it can mean that no child exists.

- ***wait*** takes one argument. This can either be a pointer to an integer or a null pointer. If a null pointer is used, it simply ignored.

## Using *exec* and *fork* and *wait* together

- ***fork*** and ***exec*** combined offer the programmer a powerful tool. By forking, then using exec within the child, a program can run another program within a subprocess and without obliterating itself.

- The following example shows how:

```
int main (void)
{
        pid_t pid;

        pid = fork();

        /* if parent, use wait to suspend execution
          until child finishes */
        if (pid > 0)
        {
                wait (int *)0);
                printf ("ls completed\n");
                exit(0);
        }

        /* if child then exec ls */
        if (pid == 0)
        {
                execl ("/bin/ls", "ls", (char *)0);
                fatal ("execl failed\n");
        }

        /* getting here means pid is negative,
          so error has occurred */

        fatal ("fork failed\n");
}

/* print error message and die! */
fatal (s)
char *s;
{
        perror (s);
        exit(1);
}
```

- In this example wait is invoked immediately after it has created a child through a call to fork.

- The system will respond to this by putting the parent to sleep until the child terminates.

- ***wait*** therefore allows the programmer to synchronize process execution in a fairly rudimentary manner.

## The *exit* System call

- **Usage**

  **int status;**

  **void exit (status);**

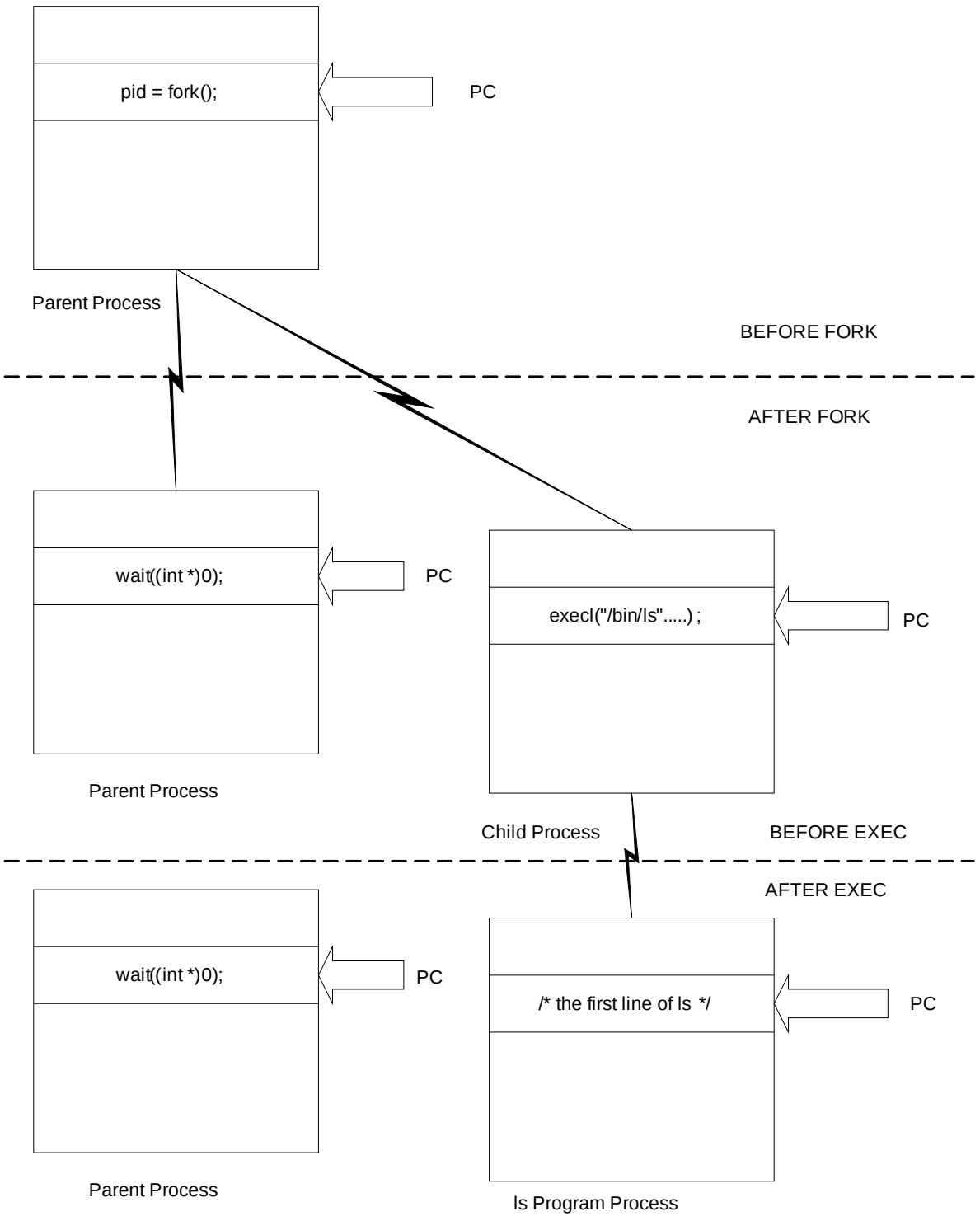
- It is used to terminate a process, although a process will also stop when it runs out of program by reaching the end of the main function, or when main executes a return statement.

- As well as stopping the calling process, exit has a number of other consequences: most importantly, all open file descriptors are closed.

- If the parent process has executed a wait call, as in the last example, it will be restarted.

- The single, integer argument to exit is called the process's exit status. This is used to indicate the success or failure of the task performed by the process.

- By convention a process returns zero on normal termination, and a nonzero (usually 1) value if an error occurred.


pid = fork();
PC
Parent Process
BEFORE FORK
AFTER FORK
wait((int *)0);
PC
execl("/bin/ls".....) ;
PC
Parent Process
Child Process
BEFORE EXEC
AFTER EXEC
wait((int *)0);
PC
/* the first line of ls */
PC
Parent Process
ls Program Process