BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
OPERATING SYSTEMS (CS C372)
TUTORIAL - 4, PROCESS MANAGEMENT CONCEPT
FIRST SEMESTER, 2015 -16

1. OBJECTIVES

Upon the completion of this tutorial, students will be able to

- ✓ List various states in a Unix process lifecycle
- ✓ Use Unix system calls for process management: ***fork(), exec*(), wait*(), and exit()***

The asterisk (*) indicates that the system call has a number of variants.

2. Errors: errno in UNIX programs

Before starting to work with this labsheet you should have some idea of error handling and debugging C programs which involves UNIX system calls. You are encouraged to read this article (http://www.ibm.com/developerworks/aix/library/au-errnovariable/) carefully and get aquinted with error handling process.

3. INTRODUCTION TO PROCESSES

The notion of process is central to the understanding of operating systems. There is no universally agreed upon definition, but the definition "Program in Execution" is most frequently used. Process is not the same as program. A process is more than a program code. A process is an 'active' entity as oppose to program which consider to be a 'passive' entity.

Q. When does a program become a process?

The operating system reads the program into memory. The allocation of memory for the program image is not enough to make the program a process. The process must have an ID (the process ID) so that the operating system can distinguish among individual processes. The process state indicates the execution status of an individual process. The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system. The operating system also manages the memory occupied by the processes and the memory available for allocation.

When the operating system has added the appropriate information in the kernel data structures and has allocated the necessary resources to run the program code, the program has become a process. A process has an address space (memory it can access) and at least one flow of control called a thread (will be discussed later). The variables of a process can either remain in existence for the life of the process (static storage) or be automatically allocated when execution enters a block and deallocated when execution leaves the block (automatic storage).

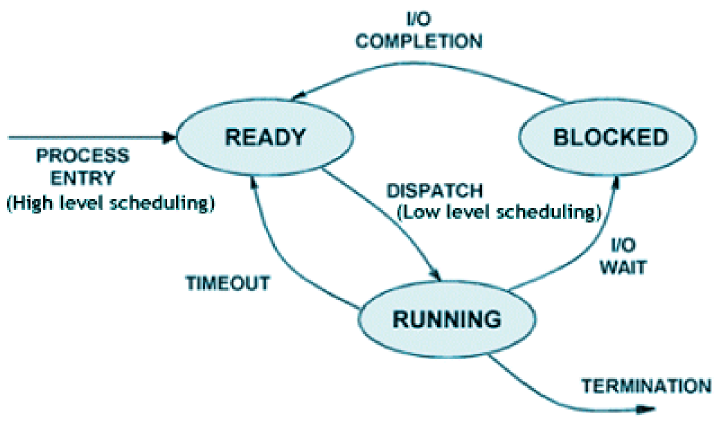
A process starts with a single flow of control that executes a sequence of instructions. The processor program counter keeps track of the next instruction to be executed by that processor (CPU). The CPU increments the program counter after fetching an instruction and may further modify it during the execution of the instruction, for example, when a branch occurs. Multiple processes may reside in memory and execute concurrently, almost independently of each other.

4. PROCESS STATES

The process state consist of everything necessary to resume the process execution if it is somehow put aside temporarily. The process state consists of at least following:

- ✓ Code for the program.
- ✓ Program's static data.
- ✓ Program's dynamic data.
- ✓ Program's procedure call stack.
- ✓ Contents of general purpose registers.
- ✓ Contents of program counter (PC)
- ✓ Contents of program status word (PSW).
- ✓ Operating Systems resource in use.

A process goes through a series of discrete process states.



- ✓ ***New State***: (PROCESS ENTRY) The process being created.
- ✓ ***Running State***: A process is said to be running if it has the CPU, that is, process actually using the CPU at that particular instant.
- ✓ ***Blocked (or waiting) State***: A process is said to be blocked if it is waiting for some event to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
- ✓ ***Ready State***: A process is said to be ready if it use a CPU if one were available. A ready state process is runable but temporarily stopped running to let another process run.
- ✓ ***Terminated state***: (TERMINATION) The process has finished execution.

5. PROCESS MANAGEMENT IN UNIX

5.1 *Process identification* –

UNIX identifies processes by a unique integral value called the *process ID*. Each process also has a *parent process ID*, which is initially the *process ID* of the process that creates it. If this parent process terminates, the process is adopted by a system process so that parent process ID always identifies a valid process.
Related System calls – The *getpid* and *getppid* functions return the process ID and the parent process ID, respectively. The *pid_t* is an unsigned integer type that represents a process ID. Neither the getpid nor the getppid functions can return an error.

SYNOPSIS –
```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Exercise-1: The following program outputs its *process ID* and its *parent process ID*. Notice that the return values are cast to *long* for printing since there is no guarantee that a *pid_t* will fit in an *int*.

outputPID.c
```
#include <stdio.h>
#include <unistd.h>
int main(void) {
      printf("I am process %ld\n", (long)getpid());
      printf("My parent is %ld\n", (long)getppid());
      return 0;
}
```

5.2 The PS utility –

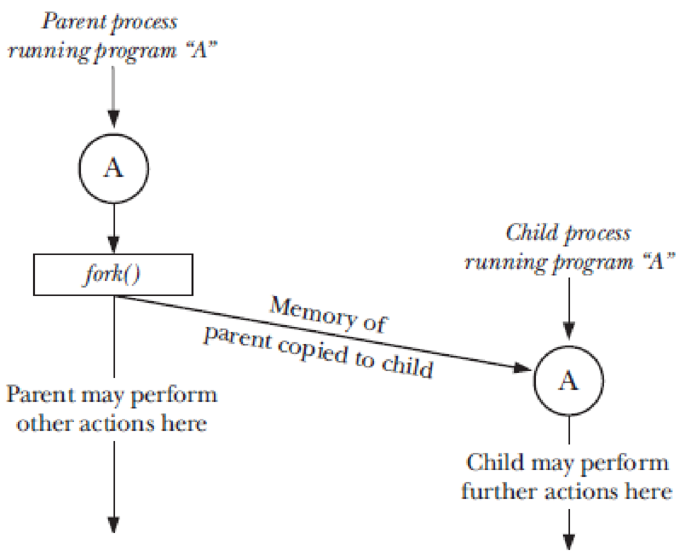
The ps utility displays information about processes. By default, ps displays information about processes associated with the user. The -a option displays information for processes associated with terminals. The -A option displays information for all processes.

The following is sample output from the ps -a command.

```
$ ps -a
  PID TTY          TIME CMD
20825 pts/11   0:00 pine
20205 pts/11   0:01 bash
20258 pts/16   0:01 telnet
20829 pts/2    0:00 ps
20728 pts/4    0:00 pine
19086 pts/12   0:00 vi
```

5.3 UNIX Process Creation and *fork* –

A process can create a new process by calling *fork*. The calling process becomes the *parent*, and the created process is called the *child*. The *fork* function copies the parent's memory image so that the new process receives a copy of the address space of the *parent*. Both processes continue at the instruction after the *fork* statement (executing in their respective memory images).



Creation of two completely identical processes would not be very useful. The *fork* function return value is critical characteristic that allows the *parent* and the *child* to distinguish themselves and to execute different code. The *fork* function *returns 0* to the *child* and returns the *childs process ID* to the *parent*. When a *fork* fails, it *returns -1* and sets the *errno*. If the system does not have the necessary resources to create the *child* or if limits on the number of processes would be exceeded, *fork* sets *errno* to *EAGAIN*. In case of a failure the fork doesn't create a child.

SYNOPSIS –

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

Exercise-2: In the following program, both parent and child execute the x = 1 assignment statement after returning from *fork*.

simplefork.c

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
      int x;
      x = 0;
      fork();
      x = 1;
      printf("I am process %ld & my x is %d\n", (long)getpid(), x);
      return 0;
}
```

- In this program, before the *fork*, one process executes with a single x variable. After the fork, two independent processes execute, each with its own copy of the x variable. Since the *parent* and the *child* process execute independently, they don't execute the code in lock step or modify the same memory locations.

In Exercise-2, the *parent* and the *child* processes execute the same instructions because the code didn't test the return value of fork. The program given below in Exercise-3 demostrates how to test the return value of fork.

Exercise-3: Describe the working of the above program. What should be the output?

twoProcess.c

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
      pid_t childpid;

      childpid = fork();
      if(-1 == childpid){
            perror("Failed to fork");
            return 1;
      }
      if(0 == childpid){
            printf("I am child %ld\n", (long)getpid());
      else
            printf("I am parent %ld\n", (long)getpid());

      return 0;
}
```

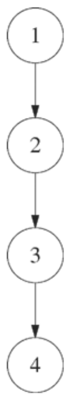
Exercise-4: Identify the problem with the following code and suggest the solution.

badprocessID.c

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
      pid_t childpid;
      pid_t mypid;

      mypid = getpid();
      childpid = fork();
      if(-1 == childpid){
            perror("Failed to fork");
            return 1;
      }
      if(0 == childpid){
            printf("I am child %ld, ID = %ld\n",
                                 (long)getpid(),(long)mypid());
      else
            printf("I am parent %ld, ID = %ld\n",
                                 (long)getpid(),(long)mypid());
      return 0;
}
```

Exercise-5 (Creating process chains with fork): we can create chain of "n" processes by calling *fork* in a loop. Have a look at the following program:



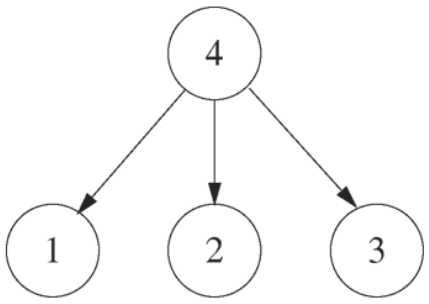
simpleChain.c

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
      pid_t childpid = 0;
      int i, n;
      /*check for valid number of command line arguments*/
      if(argc != 2){
            fprintf(stderr, "Usage %s processes\n", argv[0]);
            return 1;
      }

      n = atoi(argv[1]);
      for(i=1;i<n;i++)
            if(childpid = fork())
                  break;
      fprintf(stderr, "i: %d process ID: %ld parent ID: %ld
                                child ID: %ld\n", i, (long)getpid(),
                                (long) getppid(), (long)childpid);
      return 0;
}
```

On each iteration of the loop, the *parent* process has a non-zero *childpid* and hence breaks out of the loop. The *child* process has a zero value of *childpid* and becomes a *parent* in the next loop iteration. In case of error, *fork* returns -1 and the calling *process* breaks out of the loop.

Exercise-6 (Creating process fans with fork): create a fan of n processes by calling *fork* in a loop. On each iteration, the newly created process breaks from the loop while the original process continues. In contrast, the process that calls *fork* in program **simpleChain.c** breaks from the loop while the newly created process continues for the next iteration.



simpleFan.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
      pid_t childpid = 0;
      int i, n;
      /*check for valid number of command line arguments*/
      if(argc != 2){
            fprintf(stderr, "Usage %s processes\n", argv[0]);
            return 1;
      }

      n = atoi(argv[1]);
      for(i=1;i<n;i++)
            if((childpid = fork()) <= 0) ──► TEST
                  break;
      fprintf(stderr, "i: %d process ID: %ld parent ID: %ld
                        child ID: %ld\n", i, (long)getpid(),
                        (long) getppid(), (long)childpid);
      return 0;
}
```

Exercise-7: Explain what happens when you replace the TEST with:

```
   if((childpid = fork()) == -1)
```

More on fork –

a. The *fork* function creates a new process by making a copy of the parent's image in memory. The child *inherits* parent attributes such as environment and priviledges. The child also inherits some of the parent's resources such as open files and devices.
b. Not every parent attribute or resource is inherited by the child. For instance, the child has a new process ID and of course a different parent ID. The child's times for CPU usage are reset to 0. The child child doesn't get locks that the parent holds. If the parent has set an alarm, the child is not notified when the parent's alarm expires. The child starts with no pending signals, even if the parent had signals pending at the time of the *fork*.
c. A child inherits its parent's process priority and scheduling attributes, it competes for processor time with other processes as a separate entity.

5.4 The wait function

When a process creates a child, both parent and child proceed with execution from the point of the fork. The parent can execute *wait* or *waitpid* to block until the child finishes.

The *wait* function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal (signals will be discussed later). A process status most commonly becomes available after termination, but it can also be available after the proess has been stopped.

The *waitpid* function allows a parent to wait for a particular child. This function also allows a parent to check whether a child has terminated without blocking.

The *waitpid* function takes three parameters: a *pid*, a pointer to a location for returning the status and a flag specifying options. If *pid* is -1 *waitpid* waits for any child. If *pid* is greater than 0, *waitpid* waits for the specific child whose process ID is *pid*. Two other possibilities are allowed for the *pid* parameter. If *pid* is 0, *waitpid* waits for any child in the same process group as the caller. Finally, if *pid* is less than -1, *waitpid* waits for any child in the process group specified by the absolute value of *pid* (***process group will be discussed later***).

Two options are of particular importance:

(a) *WNOHANG* option causes waitpid to return even if the status of child is not immediately available.

(b) *WUNTRACED* option causes waitpid to report the status of unreported child processes that have been stopped.

SYNOPSIS –

```
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

If *wait* or *waitpid* returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return -1 and set *errno*. If you have not used *errno.h* and global variable *errno* before for the debugging purposes then read this article - *Working with the standard error mechanism* available on this link.

If called with *WNOHANG* option, *waitpid* return 0 to report that there are possible unwaited-for children but that their status is not available. Below is the list of mandatory errors for *wait* and *waitpid*.

errno	**cause**
ECHILD	Caller has no unwaited-for children (wait), or process specified by pid doesn't exist (waitpid)
EINTR	Function was interrupted by a signal
EINVAL	*options* parameter of waitpid was invalid

Exercise-6: The following code segment waits for a child

```
#include<sys/wait.h>
#include<stdio.h>
 int main(void){
      pid_t childpid;
      childpid = wait(NULL);
      if(childpid != -1)
           printf("Waited for child with pid %ld\n", childpid);
      else
           printf("No child to wait for");
}
```

Take a look at the following function carefully (**r_wait.c**) , it restarts the wait function if it is interrupted by a signal. We will use this function on different occasions.

```
#include<errno.h>
#include<sys/wait.h>
pid_t r_wait(int *stat_loc){
      int retval;
      while(((retval = wait(stat_loc)) == -1) && (errno == EINTR));
      return retval;
}
```