BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
OPERATING SYSTEMS (CS C372, IS C362)
PROCESS MANAGEMENT CONCEPT
FIRST SEMESTER, 2015 -16

Process Creation and Execution – Part I

Objective:
Part-I of this tutorial describes how a program can create, terminate, and control child processes. Actually, there are three distinct operations involved: creating a new child process, causing the new process to execute a program, and coordinating the completion of the child process with the original program.

1. Process Creation Concepts

Processes are the primitive units for allocation of system resources. Each process has its own **address space** and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized **hierarchically**. Each process has a **parent process**, which explicitly arranged to create it. The processes created by a given parent are called its **child processes**. A child inherits many of its attributes from the parent process.

A **process ID number** names each process. A unique process ID is allocated to each process when it is created. The lifetime of a process ends when its termination is reported to its parent process; at that time, all of the process resources, including its process ID, are freed.

Processes are created with the **fork()** system call (so the operation of creating a new process is sometimes called **forking a process**). The child process created by fork is a copy of the original parent process, except that it has its own process ID.

After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling **wait()** or **waitpid()**. These functions give you limited information about why the child terminated--for example, its exit status code.
A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. *You can use the return value from fork to tell whether the program is running in the parent process or the child process*.

When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action. A process that is waiting for its parent to accept its return code is called a **zombie process**. If a parent dies before its child, the child (**orphan process**) is automatically adopted by the original "**init**" process whose PID is **1**.

Monitoring Processes: To monitor the state of your processes under Unix use the **ps** command.

ps [-option]

Used without options this produces a list of all the processes owned by you and associated with your terminal. The information displayed by the **ps** command varies according to which command option(s) you use and the type of UNIX that you are using.

Following are some of the column headings displayed by the different versions of this command.
PID SZ(size in Kb) TTY(controlling terminal) TIME(used by CPU) COMMAND

Examples:
1. To display information about your processes those are currently running:
$ps
2. To display information about all your processes
$ps -u avinash

3. To generate long list of all processes currently running:
$ps –ly

2. Process Identification:

The **pid_t** data type represents process IDs which is basically a signed integer type (**int**). You can get the process ID of a process by calling **getpid()**. The function **getppid()** returns the process ID of the parent of the current process (this is also known as the parent process ID). Your program should include the header files '**unistd.h**' and '**sys/types.h**' to use these functions.

Function: `pid_t getpid (void)`

Creating Multiple Processes
The fork function is the primitive for creating a process. It is declared in the header file "**unistd.h**". Function: pid_t fork (void)

The fork function creates a new process. If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of **0** in the child process and returns the **child's process ID** in the parent process. If process creation failed, fork returns a value of **-1** in the parent process and no child is created.

The specific attributes of the child process that differ from the parent process are:
- ✓ The child process has its own unique process ID.
- ✓ The parent process ID of the child process is the process ID of its parent process.
- ✓ The child process gets its own copy of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, both processes share the file position associated with each descriptor.
- ✓ The elapsed processor times for the child process are set to zero.
- ✓ The child doesn't inherit file locks set by the parent process.
- ✓ The child doesn't inherit alarms set by the parent process.
- ✓ The set of pending signals for the child process is cleared.

Ex1.c
```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
      printf("Hello World!\n");
      fork( );
      printf("I am after forking\n");
      printf("\tI am process %d.\n", getpid( ));
}
```

When this program is executed, it first prints Hello World! When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at the next statement.

Note the following:
- ✓ When a fork is executed, everything in the parent process is copied to the child process. This includes variable values, code, and file descriptors.
- ✓ Following the fork, the child and parent processes are **completely independent**.
- ✓ There is **no guarantee** which process will print I am a process first.
- ✓ The child process begins execution **at the statement immediately after the fork**, not at the beginning of the program.
- ✓ A parent process can be distinguished from the child process by examining the return value of the fork call. **Fork returns a zero to the child process and the process id of the child process to the parent**.
- ✓ A process can execute as many forks as desired. However, be wary of infinite loops of forks (there is a maximum number of processes allowed for a single user).

Ex2.c

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
      int pid;
      printf("Hello World!\n");
      printf("I am the parent process and pid is : %d .\n",getpid());
      printf("Here i am before use of forking\n");
      pid = fork();
      printf("Here I am just after forking\n");
      if (pid == 0)
            printf("I am the child process and pid is :%d.\n",getpid());
      else
            printf("I am the parent process and pid is: %d .\n",getpid());
}
```

Ex3.c (Multiple forks):

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
main(void)
{
      printf("Here I am just before first forking statement\n");
      fork();
      printf("Here I am just after first forking statement\n");
      fork();
      printf("Here I am just after second forking statement\n");
      printf("\t\tHello World from process %d!\n", getpid());
}
```

---

3. Process Completion

The functions described in this section are used to **wait** for a child process to terminate or stop, and determine its status. These functions are declared in the header file "**sys/wait.h**".

(a) Function: `pid_t wait (int *status_ptr)`

wait() will force a parent process to wait for a child process to stop or terminate. **wait()** return the pid of the child or -1 for an error. The exit status of the child is returned to **status_ptr**.

(b) Function: `void exit (int status)`

exit() terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the status value.

By convention, a **status of 0** means *normal* termination. Any other value indicates an *error or unusual* occurrence. Many standard library calls have errors defined in the **sys/stat.h** header file. We can easily derive our own conventions.

If the child process must be guaranteed to execute before the parent continues, the **wait** system call is used. A call to this function causes the parent process to wait until one of its child processes exits. The **wait** call returns the **process id** of the child process, which gives the parent the ability to wait for a particular child process to finish.

(c) Sleep: A process may suspend for a period of time using the sleep command

Function: `unsigned int sleep (seconds)`

Ex4.c: Guarantees the child process will print its message before the parent process.

```
#include <stdio.h>
#include <sys/wait.h> /* contains prototype for wait */
int main(void)
{
      int pid;
      int status;
      printf("Hello World!\n");
      pid = fork( );
      if (pid == -1) /* check for error in fork */
      {
            perror("bad fork");
            exit(1);
      }
      if (pid == 0)
            printf("I am the child process.\n");
      else
      {
            wait(&status); /* parent waits for child to finish */
            printf("I am the parent process.\n");
      }
}
```

Ex5.c:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
main()
{
      int forkresult;
      printf("%d: I am the parent. Remember my number!\n", getpid());
      printf("%d: I am now going to fork ... \n", getpid());
      forkresult = fork();
      if (forkresult != 0)
      { /* the parent will execute this code */
            printf("%d: My child's pid is %d\n", getpid(), forkresult);
      }
      else /* forkresult == 0 */
      { /* the child will execute this code */
            printf("%d: Hi! I am the child.\n", getpid());
      }
      printf("%d: like father like son. \n", getpid());
}
```

4. Orphan processes

When a parent dies before its child, the child is automatically adopted by the original "init" process whose PID is 1. To, illustrate this insert a sleep statement into the child's code. This ensured that the parent process terminated before its child.

Ex6.c:

```
#include <stdio.h>
main()
{
      int pid ;
      printf("I'am the original process with PID %d and PPID %d.\n",
      getpid(), getppid()) ;
      pid = fork ( ) ; /* Duplicate. Child and parent continue from here */
```

```
        if ( pid != 0 ) /* pid is non-zero,so I must be the parent*/
        {
                printf("I'am the parent with PID %d and PPID %d.\n",
                getpid(), getppid()) ;
                printf("My child's PID is %d\n", pid ) ;
        }
        else /* pid is zero, so I must be the child */ {
                sleep(4); /* make sure that the parent terminates first */
                printf("I'm the child with PID %d and PPID %d.\n",
                getpid(), getppid()) ;
        }
        printf ("PID %d terminates.\n", getpid()) ;
}
```

The output is:
I'am the original process with PID 5100 and PPID 5011.
I'am the parent process with PID 5100 and PPID 5011.
My child's PID is 5101
PID 5100 terminates. /* Parent dies */
I'am the child process with PID 5101 and PPID 1.
/* Orphaned, whose parent process is "init" with pid 1 */
PID 5101 terminates.

5. Zombie processes

A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes. However, if a process's parent is alive but never executes a wait(), the process's return code will never be accepted and the process will remain a *zombie*.

The following program created a zombie process, which was indicated in the output from the ps utility. When the parent process is killed, the child was adopted by "init" and allowed to rest in peace.

Ex7.c:

```
#include <stdio.h>
main() {
        int pid ;
        pid = fork(); /* Duplicate. Child and parent continue from here */
        if ( pid != 0 ) /* pid is non-zero, so I must be the parent */ {
                while (1) /* Never terminate and never execute a wait ( ) */
                sleep (100) ; /* stop executing for 100 seconds */
        }
        else /* pid is zero, so I must be the child */ {
                exit (42) ; /* exit with any number */
        }
}
```

The output is:

```
$ ./a.out          & execute the program in the background
[1] 5186
$ ps               obtain process status
PID     TT      STAT    TIME    COMMAND
5187    p0       Z      0:00     <exiting> the zombie child process
5149    p0       S      0:01     -csh (csh) the shell
5186     p0      S      0:00     a.out the parent process
5188    p0       R      0:00     ps

$ kill 5186        kill the parent process
[1] Terminated a.out

$ ps notice that the zombie is gone now
PID     TT      STAT    TIME    COMMAND
```

```
5149    p0      S       0:01    -csh (csh)
5189    p0      R       0:00    ps
```

Process Creation and Execution – Part II

Part-II of this tutorial describes how a program can replace its code with that of another executable file. Actually, there are three distinct operations involved:

- ✓ Creating a new child process,
- ✓ Causing the new process to execute a program, and
- ✓ Coordinating the completion of the child process with the original program.

1. Executing a file

A child process can execute another program using one of the **exec** functions. The program that the process is executing is called its **process image**. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

This section describes the exec family of functions, for executing a file as a process image. You can use these functions to make a child process execute a new program after it has been forked. The functions in this family differ in how you specify the arguments, but otherwise they all do the same thing. They are declared in the header file **"unistd.h".**

(a) Function: `int execv(const char *filename, char *const argv[])`

The ***execv()*** function executes the file named by filename as a new process image. The ***argv*** argument is ***an array of null-terminated strings*** that is used to provide a value for the ***argv*** argument to the main function of the program to be executed. The last element of this array must be a **null pointer**. By convention, the first element of this array is the file name of the program.

The **environment** for the new process image is taken from the **environ variable** of the current process image (description of the process environment is given in the end).

(b) Function: `int execl(const char *filename, const char *arg0, ...)`

This is similar to **execv**, but the **argv** strings are specified individually instead of as an array. A null pointer must be passed as the last such argument.

(c) Function: `int execvp(const char *filename, char *const argv[])`

The **execvp** function is similar to **execv**, except that it searches the directories listed in the **PATH** environment variable to find the full file name of a file from filename if filename does not contain a slash.

This function is useful for executing system utility programs, because it looks for them in the places that the user has chosen. Shells use it to run the commands that user's type.

(d) Function: `int execlp(const char *filename, const char *arg0, ...)`

This function is like **execl**, except that it performs the same file name searching as the **execvp** function.

These functions normally don't return, since execution of a new program causes the currently executing program to go away completely. A value of **-1** is returned in the event of a failure.

If execution of the new file succeeds, it updates the access time field of the file as if the file had been read.

Executing a new process image completely changes the contents of memory, copying only the argument and environment strings to new locations. But many other attributes of the process are unchanged:

- ✓ The process ID and the parent process ID.
- ✓ Session and process group membership.
- ✓ Real user ID and group ID, and supplementary group IDs.

- ✓ Current working directory and root directory. In the GNU system, the root directory is not copied when executing a setuid program; instead the system default root directory is used for the new program.
- ✓ File mode creation mask.
- ✓ Process signal mask.
- ✓ Pending signals.
- ✓ Elapsed processor time associated with the process; see section Processor Time.

The following programs execs the commands "ls -l -a" and "echo hello there" using the 4 most-used forms of exec. Enter each, compile, and run.

Ex8.c (Using execl)

```
#include <stdio.h>
#include <unistd.h>
main(){
      execl (     "/bin/ls", /* program to run - give full path */
                  "ls", /* name of program sent to argv[0] */
                  "-l", /* first parameter (argv[1])*/
                  "-a", /* second parameter (argv[2]) */
                  NULL); /* terminate arg list */

      printf ("EXEC Failed\n") ;
      /* This above line will be printed only on error and not otherwise */
}
```

Ex9.c (Using execlp)

```
#include <stdio.h>
#include <unistd.h>
main(){
      execlp (    "ls", /* program to run - PATH Searched */
                  "ls", /* name of program sent to argv[0] */
                  "-l", /* first parameter (argv[1])*/
                  "-a", /* second parameter (argv[2]) */
                  NULL); /* terminate arg list */

      printf ("EXEC Failed\n") ;
      /* This above line will be printed only on error and not otherwise */
}
```

Ex10.c (Using execv)

```
#include <stdio.h>
#include <unistd.h>
main (int argc, char *argv[]){
      execv("/bin/echo", /* program to load - full path only */ &argv[0]);
      printf ("EXEC Failed\n") ;
      /* This above line will be printed only on error and not otherwise */
}
```

Sample Output:

$ gcc Ex10.c -o myecho
$ myecho OS in first semester 2012-13
OS in first semester 2012-13

Ex11.c (Using execvp)

```
#include <stdio.h>
#include <unistd.h>
main(int argc, char *argv[]) {
execvp("echo", /* program to load - PATH searched */ &argv[0]) ;
printf ("EXEC Failed\n") ;
/* This above line will be printed only on error not otherwise */
```

```
}
```

Sample Output:
$ gcc lab4.c -o myecho
$ myecho OS rocks
OS rocks

Exercise-1.c: Write a program where a child is created to execute a command.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
main(){
      int forkresult ;
      printf("%d: I am the parent. Remember my number!\n", getpid()) ;
      printf("%d: I am now going to fork ... \n", getpid()) ;
      forkresult = fork() ;
      if(forkresult != 0)
      {     /* the parent will execute this code */
            printf("%d: My child's pid is %d\n", getpid(), forkresult) ;
      }
      else /* forkresult == 0 */
      { /* the child will execute this code */
            printf("%d: Hi ! I am the child.\n", getpid()) ;
            printf("%d: I'm now going to exec ls!\n\n\n", getpid()) ;
            execlp("ls", "ls", NULL) ;
            printf("%d: AAAAH ! ! My EXEC failed ! ! ! !\n", getpid()) ;
            exit(1) ;
      }
      printf("%d: like father like son. \n", getpid()) ;
}
```

Sample Output:
$ gcc Exercise-1.c
$ ./ a.out
24639: I am the parent. Remember my number!
24639: I am now going to fork ...
24640: Hi ! I am the child.
24640: I'm now going to exec ls!
24639: My child's pid is 24640
24639: like father like son.

Run this program several times. You should be able to get different ordering of the output lines (sometimes the parent finished before the child, or vice versa). This means that after the fork, the two processes are no longer synchronized.

2. Process Completion Status

If the exit status value of the child process is zero, then the status value reported by wait is also zero. You can test for other kinds of information encoded in the returned status value using the following macros. These macros are defined in the header file "sys/wait.h".

(a) **Macro**: int WIFEXITED (int status) /*This macro returns a nonzero value if the child process terminated normally with exit() or _exit().*/
(b) **Macro**: int WEXITSTATUS (int status) /*If WIFEXITED is true of status, this macro returns the low-order 8 bits of the exit status value from the child process.*/
(c) **Macro**: int WIFSIGNALED (int status) /*This macro returns a nonzero value if the child process terminated because it received a signal that was not handled.*/

(d) **Macro**: int WTERMSIG (int status) /*If WIFSIGNALED is true of status, this macro returns the signal number of the signal that terminated the child process.*/

(e) **Macro**: int WCOREDUMP (int status) /*This macro returns a nonzero value if the child process terminated and produced a core dump. */

(f) **Macro**: int WIFSTOPPED (int status) /*This macro returns a nonzero value if the child process is stopped. */

(g) **Macro**: int WSTOPSIG (int status) /*If WIFSTOPPED is true of status, this macro returns the signal number of the signal that caused the child process to stop. */

Here's a program, which forks. The parent waits for the child. The child asks the user to type in a number from 0 to 255 then exits, returning that number as status.

Ex12.c (using program completion status)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
main()
{
      int number=0, statval ;
      printf ("%d: I'm the parent !\n", getpid());
      printf ("%d: number = %d\n", getpid(), number );
      printf ("%d: forking ! \n", getpid());
      if(fork() == 0) {
            printf("%d: I'm the child !\n", getpid());
            printf("%d: number = %d\n", getpid(), number);
            printf("%d: Enter a number : ", getpid());
            scanf("%d", &number);
            printf("%d: number = %d\n", getpid (), number);
            printf("%d: exiting with value %d\n", getpid(), number);
            exit(number) ;
      }
      printf("%d: number = %d\n", getpid(), number) ;
      printf("%d: waiting for my kid !\n", getpid()) ;
      wait(&statval) ;
      if(WIFEXITED(statval))
      {
            printf("%d: my kid exited with status %d\n",
                                    getpid(), WEXITSTATUS(statval)) ;
      }
      else
      {
            printf("%d: My kid was killed off ! ! \n", getpid()) ;
      }
}
```

Ex13.c

Here's an example call to **wait()**: a program spawns two children, then waits for their completion and behaves differently according to which one is finished. Try to compile and execute it.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
main(){
      pid_t whichone, first, second ;
      int howmany, status ;
      if((first = fork()) == 0) /* Parent spawns 1st child */ {
            printf("I am the first child, & my ID is %d\n", getpid());
            sleep(10);
            exit(0); }
```

```
        else if(first == -1) {
              perror("1st fork: something went wrong\n") ;
              exit(1);
        }
        else if((second = fork()) == 0) /* Parent spawns 2nd child */ {
              printf("I am the second child, & my ID is %d\n", getpid( ));
              sleep(15);
              exit(0);
        }
        else if (second == -1){
              perror ("2nd fork: something went wrong\n") ;
              exit(1);
        }

        printf("This is parent\n");
        howmany = 0;
        while(howmany < 2) /* Wait Twice */
        {
              whichone = wait(&status);
              howmany++;
              if(whichone == first)
                    printf("First child exited\n");
              else
                    printf("Second child exited\n");
              if((status & 0xffff) == 0)
                    printf("correctly\n");
              else
                    printf("Incorrectly\n");
        }
}
```

The first part of this example, up to the **howmany=0,** statement contains nothing new: just make sure you understand what the instruction flow is in the parent and in the children. The parent then enters a loop waiting for the children's completion. The **wait()** system call blocks the caller process until one of its immediate children (not children's children, or other siblings) terminates, and then returns the pid of the terminated process. The argument to **wait()** is the address on an integer variable or the NULL pointer. If it's not NULL, the system writes 16 bits of status information about the terminated child in the low-order 16 bits of that variable. Among these 16 bits, the ***higher*** 8 bits contain the ***lower*** 8 bits of the argument the child passed to **exit()**, while the ***lower*** 8 bits are all zero if the process exited correctly, and contain error information if not. Hence, if a child exits with 0 all those 16 bits are zero. To reveal if this is actually the case we test the bitwise AND expression (**status & 0xffff**), which evaluates as an integer whose lower 16 bits are those of status, and the others are zero. If it evaluates to zero, everything went fine, otherwise some trouble occurred. Try changing the argument passed to **exit()** in one of the children.

3. Process Groups

(a) The **setpgrp ( )** System Call: creates a new process group. The **setpgid()** system call adds a process to a process group.

The synopsis for **setpgrp()** follows:

```
#include <sys/types.h>
#include <unistd.h>
pid_t setpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

If the process calling **setpgrp()** is not already a session leader, the process becomes one by setting its GID to the value of its PID. **setpgid()** sets the process group ID of the process with PID **pid** to **pgid**. If **pgid** is equal to **pid** then the process becomes the group leader. If **pgid** is not equal to **pid** , the process becomes a member of an existing process group.

Compile and Run the following program to understand the process groups creation.

Ex14.c (process groups)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
main()
{
      pid_t parent_pid, child_pid, fork_pid, wait_pid;
      pid_t parent_grp, child_grp, grpid;
      int child_stat, exit_val;
      exit_val = 10;
      parent_pid = getpid();
      parent_grp = getpgrp();
      printf("\nParent process: process ID: %ld group ID: %ld\n",
                                    (long) parent_pid, (long) parent_grp) ;
      fork_pid = fork();
      switch(fork_pid)
      {
            case -1:
            perror("FORK FAILED\n");
            errno = 0 ;
            break ;

            case 0:
            child_pid = getpid();
            child_grp = getpgrp();
            printf ("Child process: process ID: %ld group ID: %ld " "parent
            process ID: %ld\n", (long) child_pid, (long) child_grp, (long)
            getppid());
            grpid = setpgrp(); /* Change the group of child */
            setpgid(child_pid, grpid);
            child_grp = getpgrp();

            printf ("Child process again: process ID: %ld group ID: %ld "
            "parent process ID: %ld\n", (long) child_pid, (long) child_grp,
            (long) getppid());

            printf ("Child process: terminate with
                                    \"exit\" - value: %d\n", exit_val);
            exit(exit_val);
            break;

            default:
            printf ("Parent process: child process with ID %ld created.\n",
                                                (long)fork_pid);
            wait_pid = wait (&child_stat);
            if(wait_pid == -1){
                  perror("wait");
                  errno = 0;
            }
            else {
                  printf ("Parent process: child process %ld
                                    has terminated.\n", (long) wait_pid);
            }
      }
}
```

Important NOTE:

1. The Shell acts as the parent process. All the processes started by the user are treated as the children of shell.
2. The status of a UNIX process is shown as the second column of the process table when viewed by the execution of the ps command. Some of the states are R: *running*, O: *orphan*, S: *sleeping*, Z: *zombie*.
3. The child process is given the time slice before the parent process. This is quite logical. For example, we do not want the process started by us to wait until its parent, which is the UNIX shell finishes. This will explain the order in which the print statement is executed by the parent and the children.
4. The call to the wait ( ) function results in a number of actions. A check is first made to see if the parent process has any children. If it does not, a -1 is returned by wait ( ). If the parent process has a child that has terminated (a zombie), that child's PID is returned and it is removed from the process table. However if the parent process has a child that is not terminated, it (the parent) is suspended till it receives a signal. The signal is received as soon as a child dies.