**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS**

**Information and Computer Science Department**

**ICS 431 Operating Systems**

**Lab # 4**

**Process Creation and Execution**

**Objectives:**

* To learn how to create child processes
* To know how to coordinate the completion of the child process with its parent
* To know how to execute a new program in a process

# 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.

These 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**

1. To display information about all your processes

**% ps -u mohammed**

1. To generate long list of all processes currently running:

## **% ps -ly**

**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)

The **getpid()** function returns the process ID of the current process.

### Function: pid\_t getppid (void)

The **getppid()** function returns the process ID of the parent of the current process.

# 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 copies 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.

* 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 execute 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).

# 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**".

## 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**.

## 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.

**Example#1:**

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main()

{

    pid\_t pid;

    pid = fork();

    if(pid == -1)

        printf("fork error");

    else if(pid == 0)

        printf("Child process\n");

    else

    {

        wait(NULL);

        printf("Parent process\n");

    }

    return 0;

}

# Sleep

A process may suspend for a period of time using the sleep command

## Function: unsigned int sleep (seconds)

# 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 ensures that the parent process terminated before its child.

**Example#2:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <sys/wait.h>

int main()

{

    pid\_t pid;

    pid = fork();

    if(pid == -1)

        printf("fork error");

    else if(pid == 0)

    {

        sleep(4);

        printf("Child PID %d and PPID %d.\n", getpid(), getppid());

    }

    else

    {

        //wait(NULL);

        printf("Parent PID %d and PPID %d.\n", getpid(), getppid());

        printf("Child PID is %d\n", pid );

    }

    printf("PID %d terminates.\n", getpid());

    return 0;

}

# 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 creates 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 finally terminate.

**Example#3:**

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main()

{

    pid\_t pid;

    pid = fork();

    if(pid == -1)

        printf("fork error");

    else if(pid == 0)

        printf("Child PID %d Done\n", getpid());

    else {

        printf("Parent PID %d \n", getpid());

        while(1);

    }

    return 0;

}

# 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".**

Macro: int WIFEXITED (int status)

This macro returns a nonzero value if the child process terminated normally with exit.

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.

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.

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.

Macro: int WCOREDUMP (int status)

This macro returns a nonzero value if the child process terminated and produced a core dump.

Macro: int WIFSTOPPED (int status)

This macro returns a nonzero value if the child process is stopped.

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.

**Important NOTE:**

1. The Shell acts as the parent process. All the processes started by the user are treated as the children of shell.

1. 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*.

1. 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.

1. 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.

**Example#4:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main() {

    int number, status;

    pid\_t pid;

    pid = fork();

    if(pid == -1)

        printf("fork error");

    else if(pid == 0)

    {

        number = 255;

        printf("Child %d\n", getpid());

        return number;

    }

    else

    {

        printf("Parent %d: waiting for child %d\n", getpid(), pid);

        wait(&status);

        printf("Child returned %08X\n", status);

        if(WIFEXITED(status))

            printf("Normal termination, exit code %d\n", WEXITSTATUS(status));

        else

            printf("Abnormal termination\n");

    }

    return 0;

}

# 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".**

## 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 sans directory names.

The environment for the new process image is taken from the environ variable of the current process image.

## 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.

## 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.

## 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.

**Example#5:**

#include <stdio.h>

#include <unistd.h>

int main() {

    char \*arg[] = {"ls", "-l", "-a", NULL};

    printf("EXEC test\n");

    execvp(arg[0], arg);

    printf("EXEC Failed\n");

    return 0;

}

Exercise: Build a simple shell program. You are going to read commands from the user after printing a suitable prompt showing your ID. If the command is exit, your program will exit. Otherwise it should create a new process using fork() system call. The new process will execute any given command using execvp(). The parent process should wait until the new process terminates and should print a report on the exit status of the child process like:

2xxxxxxx> ls

Terminated : Normally

Exit status : 0

Then your shell should loop back to read the next command.

Note: Replace 2xxxxxxx with your ID number.

Cfopasfvmsapgvsmdao[gbmnearso[gmnsdao[gnaro[gnero[ngerao[gn