CERTIFICATE

Name of the Lab : OPERATING SYSTEMS

Name of the Student : JASWANTH YENDURI

Student Regd. No. : 18BQ1A05M8

CLASS : III B.TECH. I SEM CSE – D

GIT HUB LINK: <https://github.com/Jaswanth-yenduri/OS-Lab>

INDEX

|  |  |  |
| --- | --- | --- |
| S. No. | Name of the Experiment | Page No. |
| 2 (a) | Design a c program to implement the multiprogramming memory management implementation of Fork() using System call. | 3-6 |
| 2 (b) | Design a c program to implement the multiprogramming memory management implementation of exit() using System call. | 6-7 |
| 2 (c) | Design a c program to implement the multiprogramming memory management implementation of exec() using System call. | 7-10 |
| 2 (d) | Design a c program to implement the multiprogramming memory management implementation of wait() using System call. | 10-15 |

**EXPERIMENT NO: 2(a)**

**AIM:**

 Design a c program to implement the multiprogramming memory management implementation of Fork() using System call.

**DESCRIPTION:**

Fork system call use for creates a new process, which is called child process, which runs concurrently with process (which process called system call fork) and this process is called parent process. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

**PROGRAMMING LANGUAGE USE:**  C

**LIBRARIES USED**: stdio.h, sys/types.h, unistd.h

**SYNTAX:** fork()

**Program1:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

fork();

printf("Hello world!\n");

return 0;

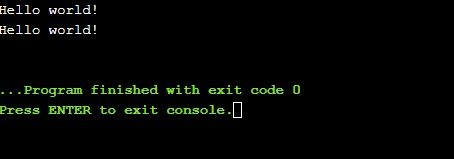
}

**Output:**

Hello world!

Hello world!

**Screenshot:**



**Program2:**

#include <stdio.h>

#include <sys/types.h>

int main()

{

fork();

fork();

fork();

printf("hello\n");

return 0;

}

**Output:**

Hello

Hello

Hello

Hello

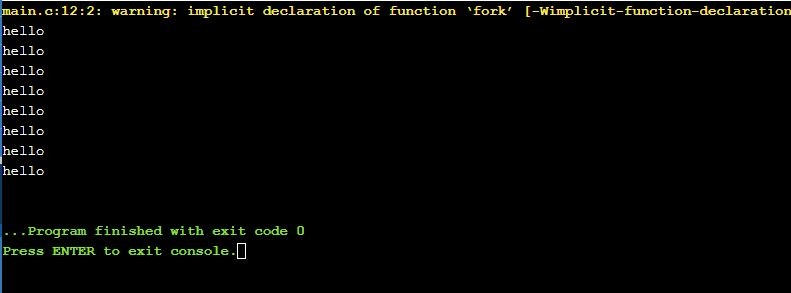
Hello

Hello

Hello

Hello

**Screenshot:**



**Program3:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

void forkexample()

{

if (fork() == 0)

printf("Hello from Child!\n");

else

printf("Hello from Parent!\n");

}

int main()

{

forkexample();

return 0;

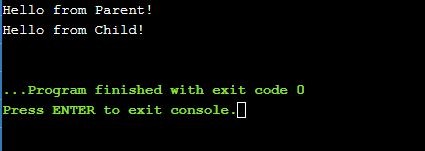
}

**Output:**

Hello from parent!

Hello from child!

**Screen shot:**



**Program4:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

void forkexample()

{

int x = 1;

if (fork() == 0)

printf("Child has x = %d\n", ++x);

else

printf("Parent has x = %d\n", --x);

}

int main()

{

forkexample();

return 0;

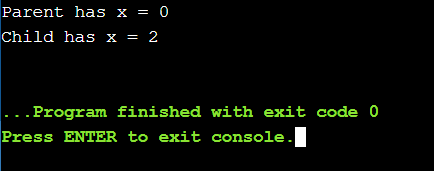
}

**Output:**

Parent has x=0

Child has x=2

**Screenshot:**



**EXPERIMENT NO:2(b)**

**Aim:**

Design a c program to implement the multiprogramming memory management implementation of exit() using System call.

**Description:**

exit()

void exit ( int status );

exit() terminates the process normally. status: Status value returned to the parent process. Generally, a status value of 0 or EXIT\_SUCCESS indicates success, and any other value or the constant EXIT\_FAILURE is used to indicate an error. exit() performs following operations. \* Flushes unwritten buffered data. \* Closes all open files. \* Removes temporary files. \* Returns an integer exit status to the operating system.

When exit() is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.

The mystery behind exit() is that it takes only integer args in the range 0 – 255 . Out of range exit values can result in unexpected exit codes. An exit value greater than 255 returns an exit code modulo 256. For example, exit 9999 gives an exit code of 15 i.e. (9999 % 256 = 15).

**PROGRAMMING LANGUAGE USED:**  C

**LIBRARIES USED:** sys/types.h, sys/wait.h

**SYNTAX:** void exit ( int status );

**Program1:**

#include <sys/types.h>

#include<stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include<unistd.h>

int main(void)

{

pid\_t pid = fork();

if ( pid == 0 )

{

exit(9999);

}

int status;

waitpid(pid, &status, 0);

       if ( WIFEXITED(status) )

       {

int exit\_status = WEXITSTATUS(status);

       printf("Exit code: %d\n", exit\_status);

       }

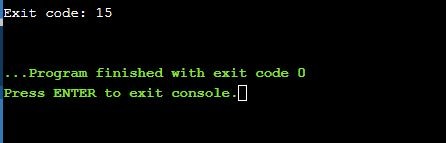
return 0;

}

**Output:**

Exit code: 15

**Screenshot:**



**EXPERIMENT NO: 2(c)**

**Aim:**

Design a c program to implement the multiprogramming memory management implementation of exec() using System call.

**Description:**

exec family of functions in C

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file unistd.h. There are many members in the exec family which are shown below with examples.

· execvp :

 Using this command, the created child process does not have to run the same program as the parent process does. The exec type system calls allow a process to run any program files, which include a binary executable or a shell script . Syntax:

int execvp (const char \*file, char \*const argv[]);

file: points to the file name associated with the file being executed.

 argv: is a null terminated array of character pointers.

Let us see a small example to show how to use execvp() function in C. We will have two .C files , EXEC.c and execDemo.c and we will replace the execDemo.c with EXEC.c by calling execvp() function in execDemo.c .

execv :

This is very similar to execvp() function in terms of syntax as well. The syntax of execv() is as shown below:Syntax:

int execv(const char \*path, char \*const argv[]);

path: should point to the path of the file being executed. argv[]: is a null terminated array of character pointers.

Let us see a small example to show how to use execv() function in C. This example is similar to the example shown above for execvp() . We will have two .C files , EXEC.c and execDemo.c and we will replace the execDemo.c with EXEC.c by calling execv() function in execDemo.c

·       execlp and execl : These two also serve the same purpose

The same C programs shown above can be executed with execlp() or execl() functions and they will perform the same task i.e. replacing the current process the with a new process.

·       execvpe and execle : These two also serve the same purpose but the syntax of them are a bit different from all the above members of exec family.

·       envp:This argument is an array of pointers to null-terminated strings and must be terminated by a null pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

**PROGRAMMING LANGUAGE USED:**  C

**LIBRARIES USED:** stdio.h, sys/types.h, unistd.h

**SYNTAX:**

int execvp (const char \*file, char \*const argv[]);

file: points to the file name associated with the file being executed.

argv:  is a null terminated array of character pointers.

int execv(const char \*path, char \*const argv[]);

path: should point to the path of the file being executed.

argv[]: is a null terminated array of character pointers.

int execlp(const char file, const char \*arg,.../ (char  \*) NULL \*/);

int execl(const char path, const char \*arg,.../ (char  \*) NULL \*/);

file:  file name associated with the file being executed

const char \*arg and ellipses : describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program

**program1:**

//EXEC.c

#include<stdio.h>

 #include<unistd.h>

int main()

{

int i;

printf("I am EXEC.c called by execvp() ");

printf("\n");

return 0;

}

//execdemo.c

#include<stdio.h>

 #include<stdlib.h>

#include<unistd.h>

 int main()

 {

char \*args[]={"EXEC",NULL};

execvp(args[0],args);

/\*All statements are ignored after execvp() call as this wholeprocess(execDemo.c)

is replaced by another process (EXEC.c) \*/

printf("Ending-----");

return 0;

 }

**Output:**

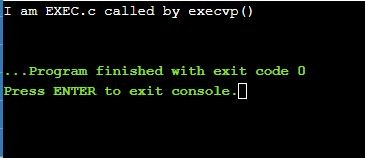
gcc EXEC.c -o exec

gcc execdemo.c -o execdemo

./execdemo

Iam EXEC.c called by execvp()

**Screenshot:**



**Program 2:**

//EXEC.c

#include<stdio.h>

 #include<unistd.h>

int main()

{

int i;

printf("I am EXEC.c called by execv() ");

printf("\n");

return 0;

}

//execdemo.c

#include<stdio.h>

 #include<stdlib.h>

#include<unistd.h>

int main()

{

char \*args[]={"EXEC",NULL};

execv(args[0],args);

/\*All statements are ignored after execv() call as this wholeprocess(execDemo.c)

is replaced by another process (EXEC.c) \*/

printf("Ending-----");

return 0;

}

**Output:**

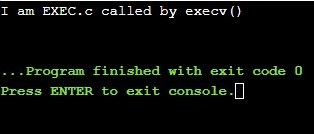
gcc EXEC.c -o exec

gcc execdemo.c -o execdemo

./execdemo

Iam EXEC.c called by execv()

**Screenshot:**



**EXPERIMENT NO: 2(d)**

**Aim:**

Design a c program to implement the multiprogramming memory management implementation of wait() using System call.

**Description:**

Wait System Call in C

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction. Child process may terminate due to any of these:

· It calls exit();

· It returns (an int) from main

· It receives a signal (from the OS or another process) whose default action is to terminate.

**Syntax :**

pid\_t wait(int \*stat\_loc);

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.

If only one child process is terminated, then return a wait() returns process ID of the terminated child process.

If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process. When wait() returns they also define exit status (which tells our, a process why terminated) via pointer, If status are not NULL.

If any process has no child process then wait() returns immediately “-1”.

Child status information: Status information about the child reported by wait is more than just the exit status of the child, it also includes

· normal/abnormal termination

· termination cause

· exit status

For find information about status, we use WIF….macros

1. WIFEXITED(status): child exited normally • WEXITSTATUS(status): return code when child exits

2. WIFSIGNALED(status): child exited because a signal was not caught • WTERMSIG(status): gives the number of the terminating signal

3. WIFSTOPPED(status): child is stopped • WSTOPSIG(status): gives the number of the stop signal

/\*if we want to prints information about a signal \*/

void psignal(unsigned sig, const char \*s);

PROGRAMMING LANGUAGE USED:  C

LIBRARIES USED:  stdio.h ,stdlib.h ,sys/types.h, sys/wait.h

SYNTAX:

pid\_t wait(int \*stat\_loc);

void psignal(unsigned sig, const char \*s);

pid\_t waitpid (child\_pid, &status, options);

**Program1:**

#include<stdio.h>

 #include<stdlib.h>

 #include<sys/wait.h>

 #include<unistd.h>

 int main()

 {

pid\_t cpid;

if (fork()== 0)

exit(0);  /\* terminate child \*/

else

cpid = wait(NULL);   /\* reaping parent \*/

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

printf("Child pid = %d\n", cpid);

return 0;

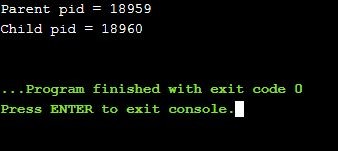
}

**Output:**

Parent pid = 18959

Child pid = 18960

**Screenshot:**



**Program2:**

#include<stdio.h>

#include<sys/wait.h>

#include<unistd.h>

int main()

{

if (fork()== 0)

printf("HC: hello from child\n");

else

{

printf("HP: hello from parent\n");

wait(NULL);

printf("CT: child has terminated\n");

}

printf("Bye\n");

return 0;

}

**Output:**

HP: hello from parent

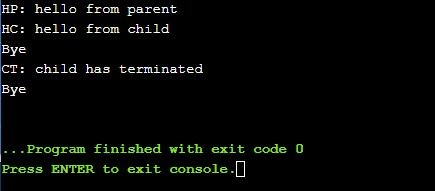
HC: hello from child

Bye

CT: child has terminated

Bye

**Screenshot:**



**Program 3:**

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>

void waitexample()

{

    int stat;

    // This status 1 is reported by WEXITSTATUS

    if (fork() == 0)

        exit(1);

    else

        wait(&stat);

    if (WIFEXITED(stat))

        printf("Exit status: %d\n", WEXITSTATUS(stat));

    else if (WIFSIGNALED(stat))

        psignal(WTERMSIG(stat), "Exit signal");

}

// Driver code

int main()

{

    waitexample();

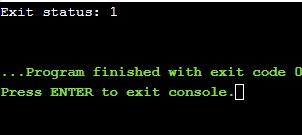
    return 0;

}

**Output:**

Exit status: 1

**Screenshot:**



**Program4:**

#include<sys/wait.h>

#include<unistd.h>

void waitexample()

{

    int i, stat;

    pid\_t pid[5];

    for (i=0; i<5; i++)

    {

        if ((pid[i] = fork()) == 0)

        {

            sleep(1);

            exit(100 + i);

        }

    }

    // Using waitpid() and printing exit status

    // of children.

    for (i=0; i<5; i++)

    {

        pid\_t cpid = waitpid(pid[i], &stat, 0);

        if (WIFEXITED(stat))

            printf("Child %d terminated with status: %d\n", cpid, WEXITSTATUS(stat));

    }

}

// Driver code

int main()

{

    waitexample();

    return 0;

}

**Output:**

Child 32381 terminated with status: 100

Child 32382 terminated with status: 101

Child 32383 terminated with status: 102

Child 32384 terminated with status: 103

Child 32385 terminated with status: 104

**Screenshot:**

