CERTIFICATE

Name of the Lab : OPERATING SYSTEMS

Name of the Student : shaik sharmila

Student Regd. No. : 19BQ5A0528

CLASS : III B.TECH. I SEM CSE – D

GITHUB Link:<https://github.com/sharmila528/os-lab>

Or

https://github.com/212114/os-lab/upload/main

19BQ5A0528 Page 1

INDEX

S. No. Name of the Experiment Page No. 2 (a) Design a c program to implement the multiprogramming memory

3-10

management implementation of Fork() using System call.

2 (b) Design a c program to implement the multiprogramming memory

10-12

management implementation of exit() using System call.

2 (c) Design a c program to implement the multiprogramming memory

12-18

management implementation of exec() using System call.

2 (d) Design a c program to implement the multiprogramming memory

18-25

management implementation of wait() using System call.

19BQ5A0528Page 2

EXPERIMENT No. 2(a):

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

Memory Management

AIM: Design a c program to implement the multiprogramming memory management implementation of

Fork() exit(),exec(),wait(),by using System call.

fork() in C

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 is 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 are used 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 the process ID of the newly created child process.

PROGRAM:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main()

{

// make two process which run same

// program after this instruction

fork();

printf("Hello world!\n");

19BQ5A0528 Page 3

return 0;

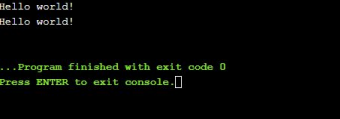
}

OUTPUT:

Hello world!

Hello world!

OUTPUT SCREENSHOT:

PROGRAM:

#include <stdio.h>

#include <sys/types.h>

int main() {

fork();

fork();

fork();

printf("hello\n");

return 0;

}

OUTPUT:

hello

hello

hello

19BQ5A0528 Page 4

hello

hello

hello

hello

hello

OUTPUT SCREENSHOT:



Number of times hello printed is equal to the number of processes created. Total Number of Processes = 2n where n is number of fork system calls. So here n = 3, 23 = 8

Let us put some label names for the three lines:

fork (); // Line 1

fork (); // Line 2

fork (); // Line 3

L1 // There will be 1 child process

/ \ // created by line 1.

L2 L2 // There will be 2 child processes

/ \ / \ // created by line 2

L3 L3 L3 L3 // There will be 4 child processes

19BQ5A0528 Page 5

// created by line 3

So there are a total eight processes (new child processes and one original process). PROGRAM:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

void forkexample() {

// child process because return value zero

if (fork() == 0) printf("Hello from Child!\n"); // parent process because return value non-zero.

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

}

int main() {

forkexample();

return 0;

}

OUTPUT:

1.

Hello from Child!

Hello from Parent!

(or)

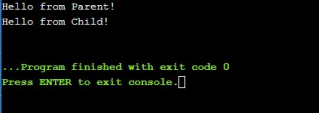
2.

Hello from Parent!

Hello from Child!

OUTPUT SCREENSHOT:

19BQ5A0528 Page 6



In the above code, a child process is created, fork() returns 0 in the child process and positive integer to the parent process. Here, two outputs are possible because the parent process and child process are running concurrently. So we don’t know if OS first gives control to which process a parent process or a child process.

note: Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and state for these two processes and also control the flow of these processes can be different.

1.

#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; }

2. Output:

3. Parent has x = 0

4. Child has x = 2

5. (or)

6. Child has x = 2

7. Parent has x = 0

8. Here, global variable change in one process does not affect other processes because data/state of these two processes are different. And also parent and child run simultaneously so two outputs are possible.

fork() vs exec()

The fork system call creates a new process. The new process created by fork() is a copy of the current process except the returned value. The exec system call replaces the current process with a new program.

19BQ5A0528 Page 7

Questions:

1. A process executes the following code.

for (i = 0; i < n; i++) fork();

The total number of child processes created is: (GATE CS 2008) (A) n

(B) 2^n – 1

(C) 2^n

(D) 2^(n+1) – 1;

Answer is :

(B) 2^n – 1

2. Consider the following code fragment:

if (fork() == 0) {

a = a + 5;

printf("%d, %d\n", a, &a);

}

else {

a = a –5;

printf("%d, %d\n", a, &a);

}

Let u, v be the values printed by the parent process, and x, y be the values printed by the child process. Which one of the following is TRUE? (GATE-CS-2005)

19BQ5A0528 Page 8

 (A) u = x + 10 and v = y

(B) u = x + 10 and v != y

(C) u + 10 = x and v = y

(D) u + 10 = x and v != y

Answer is:

(C) u + 10 = x and v = y

3. Predict output of below program.

#include <stdio.h>

#include <unistd.h>

int main() {

fork();

fork() && fork() || fork(); fork();

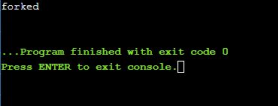
printf("forked\n");

return 0;

}

EXPECTED OUTPUT: forked

ACTUAL OUTPUT:



EXPERIMENT No. 2(b):

19BQ5A0528 Page 9



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

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).

PROGRAM:

#include <sys/types.h>

#include <sys/wait.h>

int main(void) {

pid\_t pid = fork();

if ( pid == 0 ) {

exit(9999); //passing value more than 255

}

int status;

waitpid(pid, &status, 0);

if ( WIFEXITED(status) ) {

int exit\_status = WEXITSTATUS(status);

19BQ5A0528 Page 10

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

}

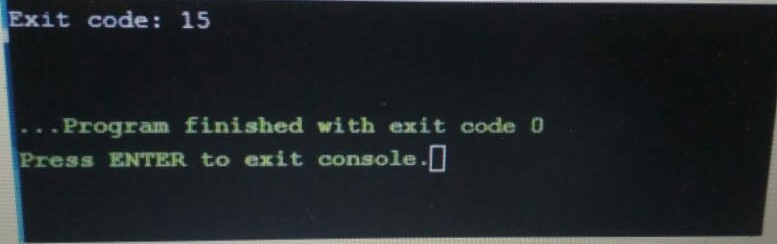
return 0;

}

OUTPUT:

Exit code: 15

OUTPUT SCREENSHOT:



Note that the above code may not work with an online compiler as fork() is disabled.

Explanation: It is the effect of 8-bit integer overflow. After 255 (all 8 bits set) comes 0. So the output is “exit code modulo 256”. The output above is actually the modulo of the value 9999 and 256 i.e. 15.

Note:

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush the stream buffer. Also, it does not call functions registered with atexit().

This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal.

EXPERIMENT No. 2(c):

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

exec family of functions in C

19BQ5A0528 Page 11



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 .

PROGRAM:

#include<stdio.h>

#include<unistd.h>

int main() {

int i;

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

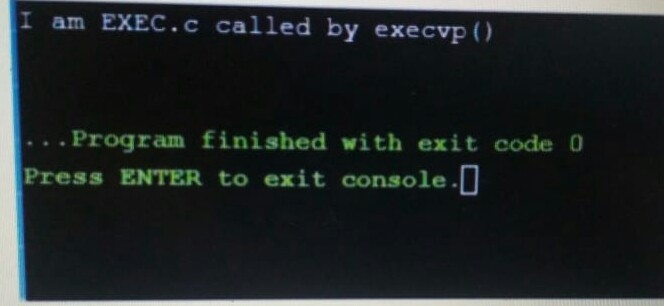
printf("\n");

return 0;

}

OUTPUT:

19BQ5A0528 Page 12



Now,create an executable file of EXEC.c using command

gcc EXEC.c -o EXEC

·

//execDemo.c

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main() {

//A null terminated array of character //

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

execvp(args[0],args); /\*All statements are ignored after execvp() call as this whole process(execDemo.c) is replaced by another process (EXEC.c) \*/

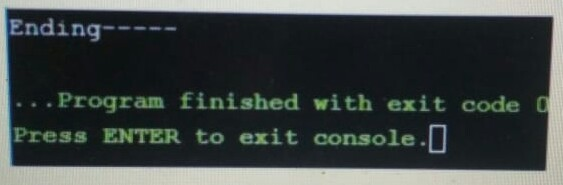
printf("Ending-----");

return 0;

}

OUTPUT:

19BQ5A0528 Page 13



· Now, create an executable file of execDemo.c using command · gcc execDemo.c -o execDemo

· After running the executable file of execDemo.cby using command ./excDemo, we get the following output:

· I AM EXEC.c called by execvp()

· When the file execDemo.c is compiled, as soon as the statement execvp(args[0],args) is executed, this very program is replaced by the program EXEC.c. “Ending—–” is not printed because as soon as the execvp() function is called, this program is replaced by the program EXEC.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 .

//EXEC.c

PROGRAM:

#include<stdio.h>

#include<unistd.h>

int main() {

int i;

19BQ5A0528 Page 14

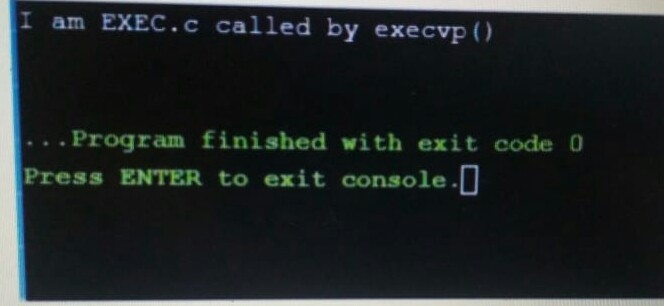
printf("I am EXEC.c called by execv() ");

printf("\n");

return 0;

}

OUTPUT:



Now,create an executable file of EXEC.c using command

gcc EXEC.c -o EXEC

·

//execDemo.c

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main() {

//A null terminated array of character //

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

execv(args[0],args); /\*All statements are ignored after execvp() call as this whole process(execDemo.c) is replaced by another process (EXEC.c) \*/

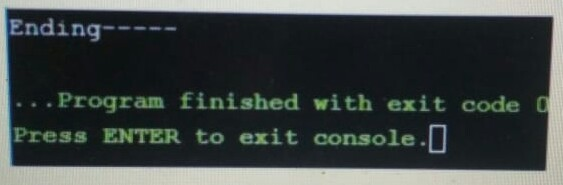
printf("Ending-----");

return 0;

}

19BQ5A0528 Page 15

OUTPUT: 



· Now, create an executable file of execDemo.c using command · gcc execDemo.c -o execDemo

· After running the executable file of execDemo.c by using command ./excDemo, we get the following output:

· I AM EXEC.c called by execv()

· execlp and execl : These two also serve the same purpose but the syntax of them are a bit different which is as shown below:Syntax:

· 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.

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 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. The synatxes of both of them are shown below : Syntax:

· int execvpe(const char \*file, char \*const argv[],char \*const envp[]); ·

· Syntax:

· int execle(const char \*path, const char \*arg, .../\*, (char \*) NULL, · char \* const envp[] \*/);

The syntaxes above shown has one different argument from all the above exec members, i.e. char \* const envp[]: allow the caller to specify the environment of

19BQ5A0528 Page 16

the executed program via the argument envp. 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 environment variable environ in the calling process.

EXPERIMENT No. 2(d):

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

Wait System Call in C Prerequisite : Fork System call

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent continues its execution after the 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 in c language:

#include

#include

// take one argument status and returns

// a process ID of dead children.

pid\_t wait(int \*stat\_loc);

If any process has more than one child processes, then after calling wait(), the 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 then wait() reap any arbitrary 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 is not NULL.

19BQ5A0528 Page 17

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

#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;

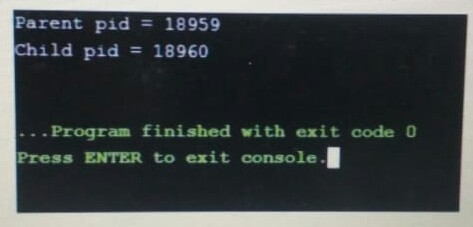
}

OUTPUT:

Parent pid = 12345678

Child pid = 89546848

OUTPUT SCREENSHOT:



19BQ5A0528 Page 18

PROGRAM:

#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: depend on environment

HC: hello from child

HP: hello from parent

CT: child has terminated

(or)

HP: hello from parent

HC: hello from child

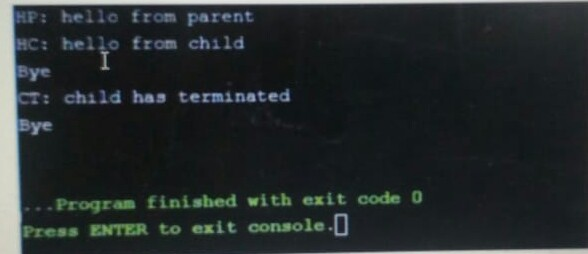
CT: child has terminated // this sentence does

// not print before HC

// because of the wait.

19BQ5A0528 Page 19

OUTPUT SCREENSHOT:



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);

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

#include<sys/wait.h>

#include<unistd.h>

void waitexample() {

19BQ5A0528 Page 20

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();

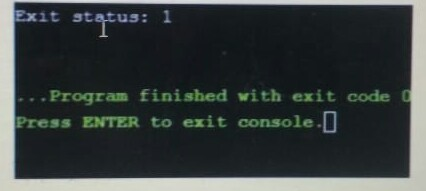
return 0;

}

OUTPUT:

Exit status: 1

OUTPUT SCREENSHOT:



We know if more than one child processes are terminated, then wait() reaps any arbitrarily child process but if we want to reap any specific child process, we use waitpid() function.

Syntax in c language: pid\_t waitpid (child\_pid, &status, options); 19BQ5A0528 Page 21

Options Parameter

· If 0 means no option, the parent has to wait for the child.

· If WNOHANG means parent does not wait if child does not terminate just check and return waitpid().(not block parent process)

· If child\_pid is -1 then it means any arbitrarily child, here waitpid() work is the same as wait() work.

Return value of waitpid()

· pid of child, if child has exited

· 0, if using WNOHANG and child hasn’t exited

PROGRAM:

#include<stdio.h>

#include<stdlib.h>

#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))

19BQ5A0528 Page 22

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

} // Driver code

int main() {

waitexample();

return 0;

}

OUTPUT:

Child 50 terminated with status: 100

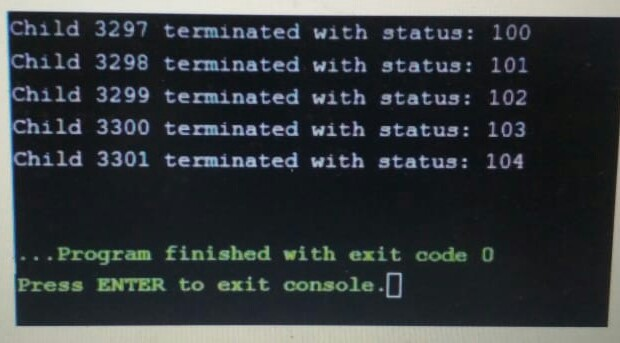
Child 51 terminated with status: 101

Child 52 terminated with status: 102

Child 53 terminated with status: 103

Child 54 terminated with status: 104

OUTPUT SCREENSHOT:



Here, Children pids depend on the system but in order print all child information. 19BQ5A0528 Page 23

19BQ5A0528 Page 24