Pablo Ruiz

pablo\_ruiz@student.uml.edu

Lab 1 report

Due 10/01/2021

Objectives:

The purpose of this lab was to demonstrate the creation of new “child” processes from an existing “parent” process using the fork() function. The first part of the assignment aims to use consecutive fork() calls to create multiple new processes which will then print their number and a message informing the user that they are running before exiting. In this case the parent process will use the waitpid() function to wait for all of the child processes to exit before exiting itself in order to prevent “zombie” processes from wasting resources. The second part of this lab aims to create [2\*n] new processes where n is an integer passed by the user as an argument. This will be accomplished using a for loop for their creation and another loop in which the parent waits for the processes to exit. Additionally, the children will print their process ID to the terminal before exiting.

Background:

Processes are a system tool that can be used to segment applications into separate operations that will be computed effectively in parallel rather than in series thus preventing bottlenecks and potentially using the available resources more effectively. New processes can be created from an existing process using the function fork() and are linked to the original process in several ways. The new processes are called “children” while the original process is known as the “parent”. Child processes inherit all the variables of the parent process that were declared up to that point and executes the rest of the program from that point independently. They are allocated their own stack and heap and share none of it with the parent process or any additional children. However, the parent process is still responsible for terminating its children processes once they have finished executing. If the parent process is terminated before it terminates its children, then those children will stay active indefinitely and become so called “zombie” processes which waste system resources. To prevent the parent process from terminating before its children the functions wait() or waitpid() can be used. The function wait() halts a parent process until there is a change of state in one of its children. This can be useful in simple applications however it lacks precision and can lead to unintended consequences when multiple children are active at once. The waitpid() function on the other hand takes the ID of a child process as its first argument and halts the parent process until the child changes to a state specified by the second argument. By default (if NULL is passed as the second argument), it waits until the child is terminated.

Each process running on a machine is given a process ID number when it is created using the fork() function. The return value for fork() is the process ID of the new child in the parent process and zero in the child process. This allows the user to perform different actions in the parent process and the child process by using conditional statements to differentiate between them. Additionally if the ID of a process wants to be known within a process the function getpid(), which returns the ID of the current process, can be used.

Algorithms:

To complete this lab two distinct algorithms were used, one for part 1 and another for part 2. In part 1 the fork() function was called three times consecutively and its return values were stored in the variables c1, c2 and c3. These calls generated 7 additional new processes for a total of 8 running processes including the parent. This happened because the number of processes grows exponentially since every existing child process also generates an additional new child with every subsequent fork() function call. This means that in this case the first child of the parent also has two children of its own. To control what code is executed by each process its necessary to use both c1, c2 and c3 as conditional variables. In the first child of the parent the variable c1 will be equal to zero. Subsequently in the first child of the first child of the parent both c1 and c2 will equal zero, and in its second child c1, c2, and c3 will be equal to zero. This allows the programmer to execute different code for each of the 8 processes. As an example, in this program each process will print its lineage and then use the execve() function to run a separate executable file called “b”. This executable will simply print the message “Program B is running” every time it is ran.

In part 2 of the lab the objective was to accept a user input n, and use a for loop to create [2\*n] new processes. Within the initial for loop a child is created with the function fork() and its process ID is stored in the variable “child”. Next, within the for loop a conditional check is performed on the variable child, if it is zero, meaning that the current process is the child, print the process ID of the child and of its parent. Otherwise, the current process can be assumed to be the parent and thus the child variable contains the child’s process ID which is then stored within an array so that the parent can later use the waitpid() function in the second for loop to wait for all of its children to exit iteratively before exiting itself. In this way the children processes are prevented from spawning their own children since they have exited before the next iteration of the for loop and the next call of the fork() function.

Results:

Part 1 Output:

Text

Description automatically generated

Part 2 Input:



Part 2 Output:

Text

Description automatically generated

Observations:

When using the function fork() multiple times in the same program it is important to be aware that previously created children processes will also spawn their own children processes unless they are prevented from executing the fork() function themselves, usually through the use of a conditional statement. Additionally, the output from part 1 shows that program “b” which was executed one time per child process using the function execve() was given a lower priority by the scheduler and consistently scheduled after all of the child processes had already ran. It is also worth noting that in part 2 the child processes all ran in order, almost like they were running sequentially. Its possible that the code within the if statement which the children processes run before exiting is so short that it is already done before the parent process is able to advance to the next iteration of the for loop and generate the next child process.

Conclusions:

In conclusion the fork() function allows the programmer to effectively run multiple parts of his program in parallel rather than sequentially. Processes have their own memory and any variables that are created or modified after they have been created are exclusive to that process. Additionally, the order in which the child processes run is determined by the scheduler and in general can’t be anticipated so the program must be designed to accommodate that behavior. Furthermore, it is important that all the child processes spawned from a single parent exit before the parent, otherwise the child processes will never be terminated and become “zombies”, wasting system resources. To prevent this from happening the function waitpid() can be used to force the parent to wait for its children. Ultimately, processes are a powerful tool when used correctly and are vital to modern applications.

Source Code:

**Part 1:**

#include <iostream>

#include <string>

#include <vector>

#include <unistd.h>

#include <sys/wait.h>

using namespace std ;

int main( int argc, char \*argv[] ){

    int c1, c2, c3 ;

    string msg = "Child running: " ;

    char\* env[] = {NULL} ;

    // Create new processes with each fork call, the number of total processes created (including parent) will be 2^n,

    // where n is the number of consecutive fork() calls

    c1 = fork() ;

    c2 = fork() ;

    c3 = fork() ;

    if ( c1 == 0 && c2 != 0 && c3 != 0 ) { //first child of the parent

        // Print this child's lineage

        cout << "Child 1 of parent running" << endl ;

        // Run the executable b

        execve( "./b", argv , env) ;

        // wait for its children to end before exiting

        waitpid(c2, NULL, 0) ;

        waitpid(c3, NULL, 0) ;

        exit(0) ;

    }

    else if ( c2 == 0 && c1 != 0 && c3 != 0 ) {  //second child of parent

        cout << "Child 2 of parent running" << endl ;

        execve( "./b", argv , env) ;

        // wait for its child to end before exiting

        waitpid(c3, NULL, 0) ;

        exit(0) ;

    }

    else if ( c3 == 0 && c1 != 0 && c2 != 0 ) { //third child of parent

        cout << "Child 3 of parent running" << endl ;

        execve( "./b", argv , env) ;

        exit(0) ;

    }

    else if ( c1 == 0 && c2 == 0 && c3 != 0 ) { //first child of child 1

        cout << "Child 1 of Child 1 running" << endl ;

        execve( "./b", argv , env) ;

        // wait for its child to end before exiting

        waitpid(c3, NULL, 0) ;

        exit(0) ;

    }

    else if ( c1 == 0 && c2 != 0 && c3 == 0 ) { //second child of child 1

        cout << "Child 2 of Child 1 running" << endl ;

        execve( "./b", argv , env) ;

        exit(0) ;

    }

    else if ( c3 == 0 && c1 != 0 && c2 == 0 ) { //first child of child 2

        cout << "Child 1 of child 2 running" << endl ;

        execve( "./b", argv , env) ;

        exit(0) ;

    }

    else if ( c3 == 0 && c1 == 0 && c2 == 0 ) { //first child of child 1 of child 1

        cout << "Child 1 of child 1 of child 1 is running" << endl ;

        execve( "./b", argv , env) ;

        exit(0) ;

    }

    else if ( c1 != 0 && c2 != 0 && c3 != 0 ) { // Parent is running

        cout << "Parent Running" << endl ;

        // wait for each of the parent's child processes to end before exiting

        waitpid(c1, NULL, 0) ;

        waitpid(c2, NULL, 0) ;

        waitpid(c3, NULL, 0) ;

    }

    return 0 ;

}

**Part 2:**

#include <iostream>

#include <string>

#include <vector>

#include <unistd.h>

#include <sys/wait.h>

using namespace std ;

int main( int argc, char \*argv[] ){

    int i, child ;

    int \*childArr ;

    // Read the first argument, interpret as an int and multiply it times two

    int numProc = atoi(argv[1]) \* 2 ;

    // Print the number of processes that will be created (2\*n)

    cout << numProc << " processes will be created" << endl ;

    // Allocate space for the array of integers that will contain the process ID of all the children

    childArr = (int\*) malloc(numProc \* sizeof(int)) ;

    // Loop as 2\*n times and create a child each iteration

    for (i = 0 ; i < numProc ; i++){

        child = fork() ;

        // If its the children process it will enter this loop, print, and then exit so that it doesnt spawn additional children in the next loop

        if ( child == 0 ) {

            cout << "I am " << getpid() << " and my parent is " << getppid() << endl ;

            exit(0) ;

        }

        // The parent appends the ID of the new child to the array

        else{ childArr[i] = child ; }

    }

    // The parent loops 2\*n times, waiting for each of the child process IDs in the array.

    for (i = 0 ; i < numProc ; i++){

        waitpid(childArr[i], NULL, 0) ;

    }

    return 0 ;

}