James Peacemaker

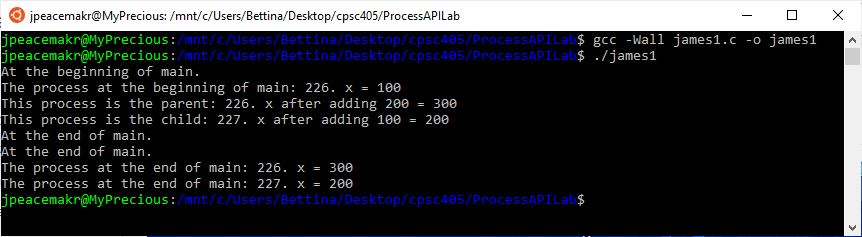
Feb. 14, 2021

CPSC 405

ProcessAPILab – Answer Document

Introduction

1. Write a program that calls fork. Before calling fork, have the main process access a variable (e.g., x) and set its value to something (e.g., 100). Copy/paste your run log here.



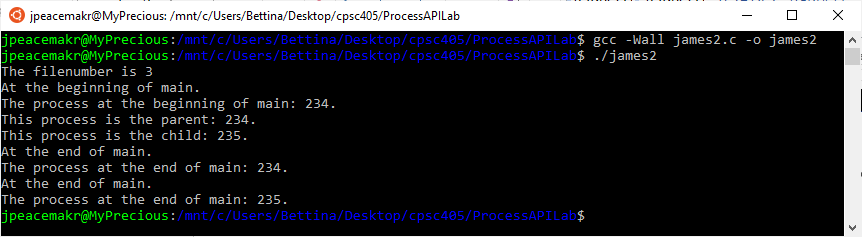
* 1. What value is the variable in the child process?

The value in the child process starts at what it was in main. It is just a copy. I wrote the program above that prints the initial value of x. It forks then modifies x and prints the value again with if statements for parent and child and then again after it returns to main.

* 1. What happens to the variable when both the child and parent change the value of x?

They are stored separately in memory so they do not add on top of each other.

1. Write a program that opens a file (with the open system call) and then calls fork to create a new process. Copy/paste your run log here.



This is the output to the file james2output.txt:

At the beginning of main.

The process at the beginning of main: 234

This process is the parent: 234

This process is the child: 235

The process at the end of main: = 234

The process at the end of main: = 235

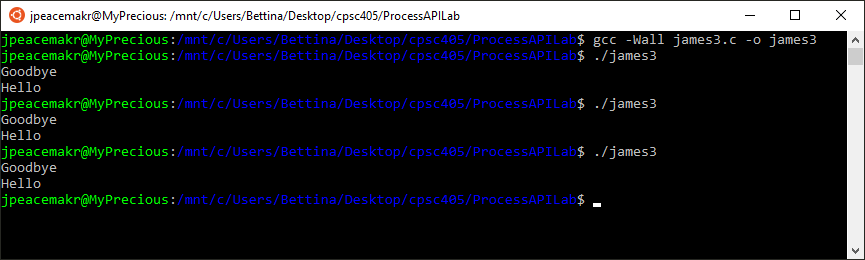
* 1. Can both the child and parent access the file descriptor returned by open?

Yes, they both wrote to the file as the program ran. The child just made a copy of the file descriptor variable.

* 1. What happens when they are writing to the file concurrently, i.e., at the same time?

I think the operating system handles the writing process for them so the operations don’t go at the same time. But there is no guarantee which one will go first.

1. Write another program using fork. The child process should print “hello”; the parent process should print “goodbye”. Copy/paste your run log here.

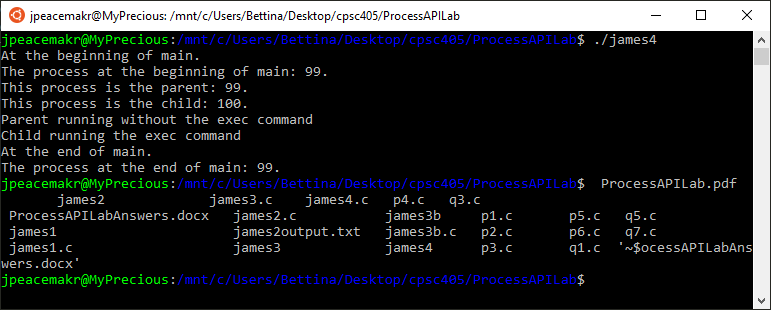


* 1. You should try to ensure that the child process always prints first; can you do this without calling wait in the parent?

Instead of wait, I think you could use sleep () or have some substantial calculation or operation to stall the process before printing goodbye. It would have to be large though. I had it do a 1,000,000 count loop that added numbers and it still would print goodbye first occasionally. 10,000,000 made it print hello first consistently as far as I could tell but I wouldn’t bet the safety of an airline on it. Using sleep whenever you needed to do something like this would slow down the program and is still no guarantee if the other operation were large and the computer was bogged down with other programs running.

1. Write a program that calls fork and then calls some form of exec to run the program /bin/ls. See if you can try all of the variants of exec, including (on Linux) execl, execle, execlp, execv, execvp, and execvpe. Copy/paste your run log here.

I made it so the program ran the exec command on the child process after it forked.



* 1. Why do you think there are so many variants of the same basic call?

There are several different options.

execl() takes arguments as a list (an individual string separated by commas) (l is for list). This would be easier to use if the arguments are hard coded.

execv() takes arguments as an array (v for vector). This would be easier to use if you needed to pull in the arguments and assemble them or manipulate them. Manipulating one long string of arguments would be annoying.

Adding p makes it so it will check multiple paths looking for the program to execute.

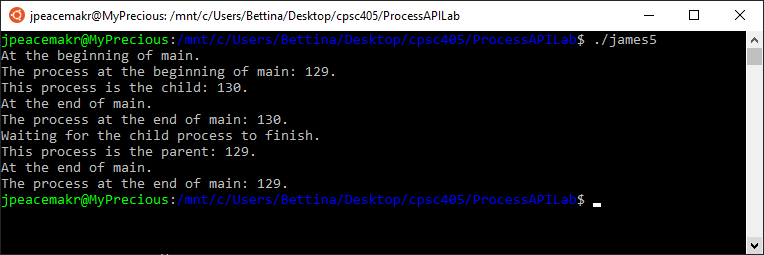
Adding e makes it run in a specified environment.

* 1. How do processes use file descriptors to redirect the standard output from a process

When the program runs, it starts with three file descriptors: [0] stdin (the screen input) [1] stdout (the screen output) [2] stderr (error messages to the screen).

When a new file is opened for output or input, it is assigned the next spot [3] to start, and so on.

1. Now write a program that uses wait to wait for the child process to finish in the parent. Copy/paste your run log here.



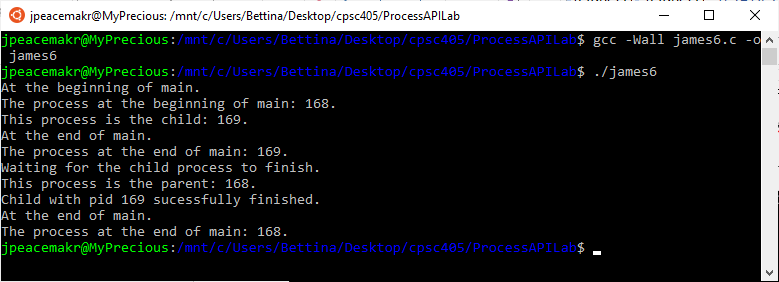
* 1. What does wait return?

It returns an int process id of a dead child.

* 1. What happens if you use wait in the child?

It would not have any effect in this program because the child does not have a child.

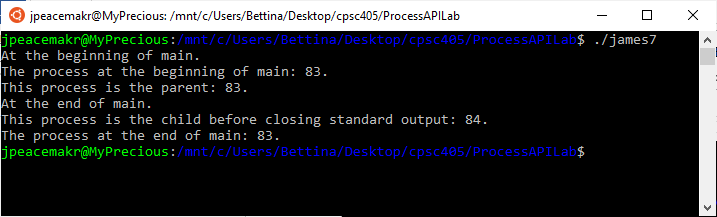
1. Write a slight modification of the previous program, this time using waitpid instead of wait. Copy/paste your run log here.



* 1. When would waitpid be useful?

When a process has more than one child. It can wait for a specific child or all. And there are options to waitpid to handle them depending on what you want to happen.

1. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). Copy/paste your run log here.



* 1. What happens if the child calls printf to print some output after closing the descriptor?

The printf in the child process no longer prints to the screen. The parent process can still print to the screen though.

1. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the pipe system call. One child is the reader and the other child is the writer. The program reads a string argument from the command line. The writer child process writes to the pipe (a) the string argument (b) followed by some other string (c) followed by closing the pipe. The reader child process reads from the pipe until it is closed. The reader child process must print to standard output the string received on the pipe followed by the reverse of the string argument. The following demonstrates the running of this program.

./q8 gusty

Parent of two children connected via pipe. (pid:8587)

parentpid:8587 of child1pid:8588 and child2pid:8589

child 1: pipe reader: (pid:8588)

child 2: pipe writer: (pid:8589)

gustyCooperytsug

childpid: 8589 finished

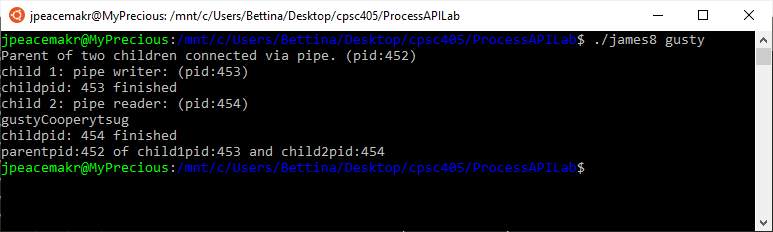
childpid: 8588 finished

The following demonstrates running the program without the string argument.

$ ./q8

usage: q8 <string>

Copy/paste your run log here. I couldn’t get them to print in the same order as above.



1. Write a program that connects a child process and a parent process with two pipes.

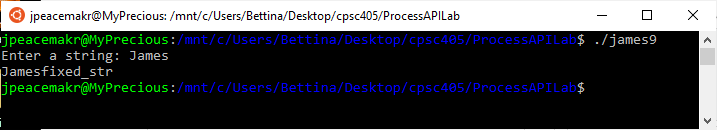
Parent process

1. reads string from terminal prior to fork
2. writes string read from terminal to first pipe
3. reads from second pipe
4. writes what is read from second pipe to terminal

Child process

* 1. reads from first pipe
  2. concatenates fixed\_str to what was read
  3. writes concatenated string to second pipe
  4. closes pipes

Copy/paste your run log here.



1. What is a pipe? What is a pipe good for? This question reminds me of a song from the 1970’s. <https://www.youtube.com/watch?v=bX7V6FAoTLc> .  
   Note that Linux shells provide a | operator that allows you to connect commands. For example, I can pipe the output of the ls command to grep by doing the following.  
   $ ls | grep p1  
   When doing this, the shell creates two child processes that are connected via a pipe. The left-hand child process runs the program ls. The left-hand’s stdout is connected to the output end of the pipe. The right-hand’s stdin is connected to the input end of the pipe.

A pipe uses a buffer that passes text from one process to another. You can send the output from one process to the input of another. It allows you to combine multiple operations from different processes.

1. Explain how processes use file descriptors to redirect the standard input/output to accomplish the following. This is seen from a shell as   
   $ ./prog < input.txt > output.txt
   1. When a process reads from standard input, it read from a file
   2. When a process writes to standard output, it writes to a file

The arrow pointing toward ./prog means that it is reading input from input.txt instead of standard input (input from the keyboard). The arrow pointing away from ./prog means that it is sending output to output.txt instead of to standard output (the screen). The default file descriptors are standard in, standard out and standard error (like out but an error). The arrows reassign one of the file descriptors to a file.

1. Study the code in p1.c, p2.c, p3.c, p4.c, p5.c, and p6.c. The code gets more interesting as the number increases. The code in p6.c is a hardcoded version of the following shell command.  
   $ ls | grep p1  
   There is a slight variation to the hardcoded in that you can pass the p1 as an argument. For example, if you execute p6 as follows  
   $ ./p6 p3  
   It performs the following shell command.  
   $ ls | grep p3

Write a new program, p7.c, that mimics p6.c to be the hardcoded version of the following shell command.  
$ cat p7.c | grep close

