Homework 2: Forking Processes

cs341
Spring 2002

Allen B. Downey
Computer Science Department

Due: Thursday 14 February

The purpose of this homework is to play around with the fork and wait system calls, and to run some experiments. For each of the sections labelled "Experiment", you should perform experiments to answer the questions and write a section in your lab report.

Before starting, please read "How to Write a Good Lab Report" (attached to this homework). Each group should turn in a single report.

Have fun!

Fork a process

1. Create a directory named `hw02` somewhere in your home directory and `cd` into it.

2. Download the fork program using `lynx`:

   ```
   lynx -dump http://rocky.wellesley.edu/cs341/code/hw02/fork.c > fork.c
   ```

   Examine it closely. Read the man pages for `fork` and `wait` (included with this homework).

3. To compile the program, type

   ```
   gcc fork.c
   ```

   The resulting executable is named `a.out`. Since the current directory is usually not on your path, to execute it you have to type

   ```
   ./a.out
   ```

   The dot means "this directory." You can specify an alternate name for the executable with the `-o` flag, and you can combine the two commands on a single line:

   ```
   gcc -o fork fork.c; ./fork
   ```

   Compile and run the program and make sure your output at least resembles the output in the comment.

Experiment 1

After the fork, both the parent and the child are runnable. The scheduler has to decide which to run first. When you run the program, which process prints its "Hello" message first? Is it the same one every time?

Experiment 2

In the parent thread, what is the elapsed time to create a child process and wait for it to complete? If you run the program repeatedly, how much does the elapsed time vary? You should express time values and the degree of variation in a way that is concise and easy to interpret.

Find out where the segments are

1. You can find out where the various segments of memory reside by creating variables in each segment and printing their addresses. To get the address of a variable, use the `&` operator. For example,

   ```
   int i;
   printf ("0x%x\n", &i);
   ```

   creates a local variable named `i` and prints its address in hexadecimal (the 0x prefix is the standard way to indicate that a number is hexadecimal).

Experiment 3

Find the location of each of the following: the program text, the static data segment, the heap and the stack. Draw a diagram of the address space. How big is the address space?

Do the stacks grow up or down? Does the heap grow up or down? Can you tell how big the text segment is? How does the size of the text segment compare to the size of the executable file (use `ls -l` to get the size of the file).

Experiment 4

As the `fork` man page explains, after the fork, the parent and the child are (almost) identical. Whatever was on the parent's stack is also on the child's stack. Whatever was in the parent's heap is also in the child's heap. What is not as clear is whether the OS has made a copy of these segments, or whether the parent and the child are sharing these segments.

Read the man pages for `fork` and `vfork` and see if you can figure out what they are talking about. Perform an experiment to see if you can distinguish between the behavior of `fork` and `vfork`.

Lots of processes

1. Make a copy of `fork.c` named `fork2.c`.

2. Right after the `#include` statements int `fork2.c`, add the line

   ```
   #define CHILDREN 2
   ```

   which defines the constant `CHILDREN` and sets it equal to 2. Note that in C, this type of definition does not end with a semi-colon.

3. Modify the program so that instead of creating a single child it creates as many children as the value of `CHILDREN`, such that all the children are running at the same time. The parent should do all the `fork`s and then print a message to indicate that it is done.

   To be sure that all the children are running at the same time, you should see the "done" message from the parent before the "hello" message from any of the children. You might have to adjust the duration of the `sleep` call in the children.

   Finally, the parent should wait for each child to complete and print a message acknowledging each completion.

   Each child should print a message like, "Hello from child 2." (Or whatever number the child happens to be). Each child should return its number as its exit status (the argument of `exit`).

Experiment 5

What do you notice about the order of the messages from the children? What can you say about the scheduling policy on this machine?

How many processes can you create before `fork` fails? Which error message do you get? Can you say anything about the amount of memory required for each process?

As the number of children increases, how does the elapsed time increase? Can you estimate how long it takes to create a process?