Steve Patterson and Ethan Johnson

Comp 340

Program 3

4/30/113

**Technical Analysis:**

**Task1:**

1. Print memory before given process even runs
2. Print memory right after process begins
3. Fork a child
4. Print memory to see how much is automatically allocated
5. Child allocates and sets array to use up memory
6. Print memory to see how much is used
7. Child terminates
8. Fork a second child
9. Print memory to see how much is automatically allocated
10. Child allocates and sets array of greater size than last child to use up memory.
11. Print memory to see how much is used
12. Have second child sleep for 10 seconds
13. Print memory again to see if amount used has changed with time.

**Results:**

Process itself allocates around 140 kb each time.

Forking child 1 uses around 132 kb

After allocation, free memory is reduced by about 10 MB.

Forking child 2 uses around 132 kb (same amount as child 1)

Child 2 allocation uses nearly 100 MB

**Note:** Size of child doesn’t seem to affect memory allocated until the child actually allocates memory itself within its own code section.

After 10 seconds, about 100 kb less free memory. It is impossible to say whether or not that memory was given to the child though. More likely that background processes requested memory during the 10 seconds.

**Interesting sidebar:** If total free memory is **m** bytes, and a child is called that allocates **n** bytes, free memory is **m – n,** which makes sense. However, when the child terminates, free memory only returns to **m - 8**; i.e. 8 bytes that are used to in creation don’t seem to be freed. I am not sure if this is a coincidence and maybe other memory was being allocated by other processes but it seemed to be happening with 8 bytes far too much to be coincidence.

**Task2a:**

1. Print memory before process even starts
2. Print memory after process starts
3. Fork a child.
4. Print memory.
5. Once child terminates, fork another
6. Print memory.
7. Once child terminates, fork another
8. Print memory.

**Results:**

The original process takes about 100 kb this time, which makes sense because it is a smaller program.

Each child process takes up the exact same amount of memory when it is created as the other children.

Since the parent waits for each child to terminate before starting the next one, the free memory printed to the screen for the child is the same. If you want to see how much is used each time, just subtract the free amount printed after the given child process from the original free memory to get the amount used by each child.(Once again, this amount is the same).

**Task2b:**

1. Print memory before process is started
2. Print memory after process is started
3. Fork a child.
4. Use execv to start another process (The one made by compiling execvTest.c);
5. In new process, system call to print memory before any allocation is done.
6. New process allocates and sets an array
7. Print memory after allocation
8. Process terminates, so child terminates. Original process terminates.

**Results:**

Program itself takes about 32kb. It is the smallest of all

execv call uses up no extra memory until the called process actually allocates new memory.

This is different from forking a child process which automatically uses up the same amount of memory as the parent did that called it.

**What we learned:**

Using fork, children processes use up the same amount of memory that the parent process took.

Using execv, the process called does not use up the same amount of memory as the parent.

A problem we encountered early on is that malloc doesn’t actually allocate memory until you do something with it, such as using memset to set each element to 0, like we did.