CS314 - Operating Systems Homework 2 (Fall '18)

1 Administrivia

Homeworks should be done individually and turned in via Moodle. Problems requiring a programmatic solution should be placed in their own directory. Solutions for problems that can be answered without producing a program can all be placed into a single file.

Optionally, you may use LaTeX to typeset solutions (programs should still be placed in their own directories such that they may be compiled and executed separately). LaTeX version of this file is here: `/home/cs314/hw2.tex` To generate a pdf with your solutions copy the above file to someplace in your home directory and compile with `pdflatex hw2.tex`

Turn in one file, preferrably a tgz or zip, containing the answers to the questions below, along with all working code and/or scripts, where applicable.

2 Get On With It

1. (20 pts) See the man page for the `wc` utility. It will be present in any UNIX-like system distribution. Try the wc utility out on some input to familiarize yourself with its default output, then write a `wc` utility of your own that is invoked and produces similar output, counting the number of characters, words, and lines present in the input file. You do not have to implement support for options supported by the real `wc` utility, just the default behavior for one or more files specified on the command line. If no file is specified, input is taken from stdin until EOF appears (ctrl+d).

   Example:

```
$ wc wc.c
      34      85     455 wc.c
$ wc wc.c a.out
      34      85     455 wc.c
       6      40    8544 a.out
      40     125    8999 total
$ wc
this should work as well!
       1       5      26
```

2. (20 pts) See the man page for the `cat` utility. It reads files sequentially, writing out their contents to stdout. The files are read and written out to stdout in the order they appear in in the invocation (e.q. "cat file1 file2" prints contents of file1 then file2), printing an error message when it is unable to read a file or a file is missing and moving on to the next one in the list. Implement the basic functionality of cat (for no input files, where cat reads input from stdin, and for up to 5 files). Also add support for the "-b" flag, which numbers each file's lines starting at 1.

3. (20 pts) Consider the following C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int a = 0;
int main(){
  fork();
  printf("%d\n", ++a);
  fork();
  printf("%d\n", ++a);
  fork();
  printf("%d\n", ++a);
  exit(0);
}
```

   - (10 pts) Including the original process, how many processes end up calling exit()? Show your work or modified code and output showing the correct count.
   - (10 pts) What is one possible output of this program? Explain.

4. (20 pts) Included here is a trivial threaded program. The intended purpose is to construct a list of numbers from 1 to MAX-1, where each number occurs only once. Run the program on os.cs.s.siue.edu and report what you observe in the output (hint: use wc to count the number of lines in the output), answering the questions: Is the output correct? If it is incorrect, why is it incorrect?

   Now, if you've found that the output is incorrect, fix the program to produce the intended output. Describe what you've done, explaining why it works. Include your source as part of the answer.

   Also, In your answer, include the execution times you've observed for both the original program and for one you've fixed to produce correct output. (see: `man time`) Try to answer at least the following: How is the execution time different? Why is it different? How does the result change your expectations of improving performance by dividing work into threads?

   Hint 1: See http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html.
   Hint 2: Look at man pages for sem_init, sem_wait, sem_post.
   Hint 3: Develop and test your code on os.cs.siue.edu. It will save you a headache. Hint 4: Compile with the pthread library, e.g. gcc -lpthread yourprogram.c (See the program on the next page.)

```
#include <stdio.h>
#include <pthread.h>

#define MAX 100000
FILE* out;

int main()  {
  pthread_t f3_thread, f2_thread, f1_thread;
  void *f1();
  int i = 0;
  out = fopen("numbers", "w+");
  pthread_create(&f1_thread,NULL,f1,&i);
  pthread_create(&f2_thread,NULL,f1,&i);
  pthread_create(&f3_thread,NULL,f1,&i);
  pthread_join(f1_thread, NULL);
  pthread_join(f2_thread, NULL);
  pthread_join(f3_thread, NULL);
  fclose(out);
  return 0;
}

void *f1(int *x){
  while(*x < MAX){
    fprintf(out,"%d\n", *x);
    (*x)++;
  }
  pthread_exit(0);
}
```

5. (20 pts) Here is another trivial threaded program. Each thread simply prints the value of the passed argument (ignore compiler warnings; we are abusing the argument pointer to pass an integer). Enter this program and run it a bunch of times. You'll notice that the order of thread execution is random. Without changing main(), use semaphores to enforce that threads always get executed in order, so that what is printed are the values 1, 2, and 3, in that order.

```
#include <stdio.h>
#include <pthread.h>

int main()  {
  pthread_t thread1, thread2, thread3;
  void *f1();

  pthread_create(&thread1,NULL,f1,1);
  pthread_create(&thread2,NULL,f1,2);
  pthread_create(&thread3,NULL,f1,3);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  pthread_join(thread3, NULL);

  return 0;
}

void *f1(int x){

  printf("%d\n", x);

  pthread_exit(0);
}
```