

## 1 Administrivia

Homeworks should be done individually and turned in via Moodle.

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`

## 2 Get On With It

1. (20 pts) Consider the following C code:

```
void main(){
    fork();
    fork();
    fork();
    fork();
    exit();
}
```

Including the original process, how many processes end up calling `exit()`? Show your work or modified code and output showing the correct count.

### **Solution:**

Every process calling `fork` creates one more process, resulting in twice as many processes. The first `fork` splits one process into two, the second splits two processes into four, and so on. This results in  $2^n$  processes, where  $n$  is the number of times `fork` is called. Thus, the number of processes calling `exit()` is  $2^4 = 16$ .

The below code produces 16 `exit()` statements.

```
void main() {
    fork();
    fork();
    fork();
    fork();
    printf("exit()\n");
    exit(0);
}
```

2. (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
```

**Solution:**

Source code file is included, called `wc.c`.

3. (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.siu.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.siu.edu. It will save you a headache. Hint 4: Compile with the pthread library, e.g. `gcc -lpthread yourprogram.c`

```
#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);
}
```

### Solution:

Source code for solution is included, called `three.c`.

The original was incorrect because in the output text file "numbers" some numbers would show up more than once, out of order, or not at all. This is because of the fact that three threads were working with the same number `i`, but not atomically. To remedy this, I used a mutex to ensure that only one thread could read `i` and then print it to the file.

The runtimes were as follows:

- Original: \ real 0m0.039s \ user 0m0.026s \ sys 0m0.010s
- Solution: \ real 0m0.095s \ user 0m0.047s \ sys 0m0.083s

4. (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);
}
```

### Solution:

Source file is included, called `four.c`.

My solution involved changing `main()` slightly, though only to initialize some values. Besides that, all the changes occur in `f1()`.

```
void *f1(int x) {
    int waiting = 1;
    while(waiting) {
        sem_wait(&mutex);
        if(x == c) {
            printf("%d\n", x);
            c++;
            waiting = 0;
            sem_post(&mutex);
        }
    }
    pthread_exit(0);
}
```

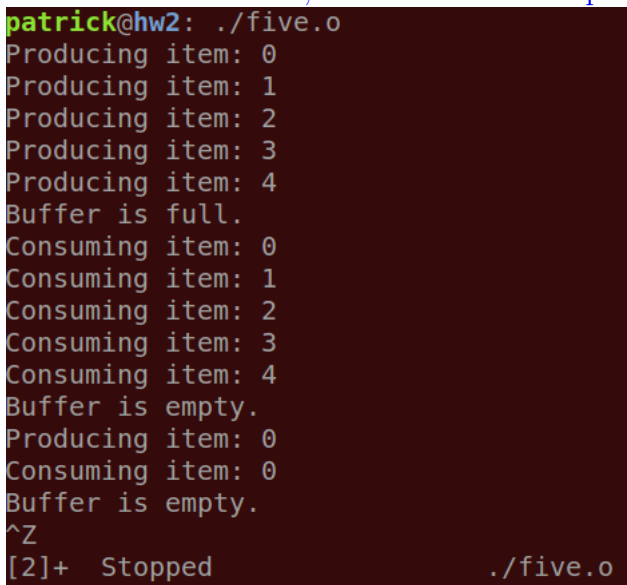
There is now an integer `c` which is atomic, only accessible from within a mutex. Its initial value is 1. If the number passed to the function `f1()` matches `c`, the function prints that argument. Otherwise, the thread continues to loop until it does get a chance to print. This

loop prevents a problem if "thread 2" tries to run before "thread 1", meaning that thread two and three will be caught waiting on the semaphore forever. The `c` prevents a later thread from printing before an earlier one.

5. (20 pts) Rewrite the example from the "Producer/Consumer with Semaphores" slide (Chapter 2 slides; also found in Figure 2-28 in Tanenbaum) to working C code. Write a `main()` that creates two threads, one for the consumer and one for the producer. You must use a bounded buffer having a size  $N > 1$ . Demonstrate that your implementation works (print at least when an item is added to the buffer, when an item is removed from the buffer, when the buffer is full, and when the buffer is empty). Turn in code and output.

### Solution:

Source file is included, called `five.c`. Output is included as screenshot below:



```
patrick@hw2: ./five.o
Producing item: 0
Producing item: 1
Producing item: 2
Producing item: 3
Producing item: 4
Buffer is full.
Consuming item: 0
Consuming item: 1
Consuming item: 2
Consuming item: 3
Consuming item: 4
Buffer is empty.
Producing item: 0
Consuming item: 0
Buffer is empty.
^Z
[2]+  Stopped                  ./five.o
```

I tested this with three consumers and three producers, as well as a buffer size of up to 32. I cut the buffer size down to 5 for simpler output, and i commented out the extra consumers and producers since the homework only asked to one of each. The `sleep()` statements are present just to give the producer thread enough time to caused a "**Buffer is full**" output before the consumer thread starts pulling stuff out. The consumer thread was changed to pull off one extra element, causing the program to hang, but also guaranteeing a "**Buffer is empty**" output.

I used `sem_trywait()` to see if the producers or consumers would wait on the `full` and `empty` semaphores respectively. If they would have hung, this triggers the "Buffer is full/empty" outputs. Then they wait for real on their respective semaphores.

### 3 What to Turn In

Turn in one file, preferably a tgz or zip, containing the answers to the questions above, along with all working code, where applicable.