Contents

1 Solution 1

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 Solution 2

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

3 Solution 3

Source code for solution is included, called `three.c`.
The original was incorrect because it 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

The run time for the solution is significantly greater because there is some time cost where threads are waiting for access to the *critical region* where they can read `i` and write to the file. When splitting work up across threads, there would usually be a speed increase because of concurrency, since more work is getting done at once. In cases like this, however, since all threads need to work in parallel with one atomic shared value, there will also be a speed cost when multithreading because of the waiting each thread will have to do in order to have a correct output.

4 Solution 4

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 `fi()`.

```
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 Solution 5

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