Project 1b

Jonathan Poisson

poissonj@ufl.edu

Instruction

A make file is included. There are 5 commands

- `make increment` which compiles increment
- `make consecutive` which compiles consecutive
- `make safe_increment` which compiles safe_increment and creates a txt file with 10 lines labeled 1 through 10
- `make clean` to delete the compiled programs, foo.txt and config
- `make` that compiles everything.

The execution below follows running simply `make` and then `make clean` after the execution.

Execution

Increment

```
> cat foo.txt
> 1

> ./increment 10000 foo.txt & ./increment 10000 foo.txt
> ./consecutive < foo.txt
> 2
> 2
> 3
> 3
> 4
> 4
> 5
> 5
> ... for a long time
```

Safe Increment

```
> cat foo.txt
> 1
> ./safe_increment 10000 foo.txt config & ./safe_increment 10000 foo.txt config
> ./consecutive < foo.txt
>
```

Discussion

The program safe_increment provides a way to run two processes at the same time on a single text file while incrementing the last number in the text file for an inputted number of times. This is accomplished using Peterson's algorithm. The critical section for the program is the act of entering into the file times, reading the last value, incrementing by one and then writing the new value back into the file, all of this a total of N times. The flag and turn variables in Peterson's algorithm are created by using each process' shared_val as the turn variable, and then computing the XOR of the two shared_vals for the turn bit. While the turn bit is not necessarily shared by each program, the computation of the XOR between the different shared_vals allows for each process to have a copy of the turn bit that matches the other process' value. The program that does not use this type of mutual exclusion (increment) writes duplicates to the outputted text file as both processes are reading and writing into the file at the same time. A race condition appears almost instantly as the numbers written to the files are duplicates.

I chose for my critical section to be allowing one process into the critical section, allowing it to increment its full N times, and then having it leave the critical section and thus give control to the other process. This simpler method, however, still follows the Peterson's algorithm as each process is run at the same time and then the algorithm determines which process to run first and when the process is finished running. I was not sure what the critical section should have been (either process 0, then process 1, then process 0... incrementing or having process 0 increment followed by process 1 or vice versa depending on the execution) but the project specifications did not specify and the processes are still running at the same time and safely incrementing in a fair manor, just with a longer critical section. An argument could be made that the program is running serially instead of in a fair synchronized manor, but the algorithm for Peterson's solution is still in place, and the critical section is mutually exclusive while both processes are running.