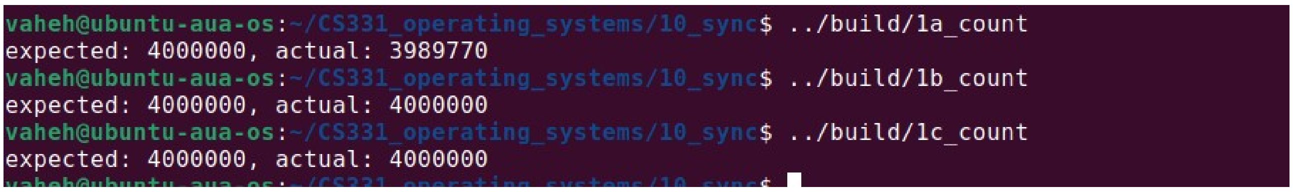
Synchronization report

P1

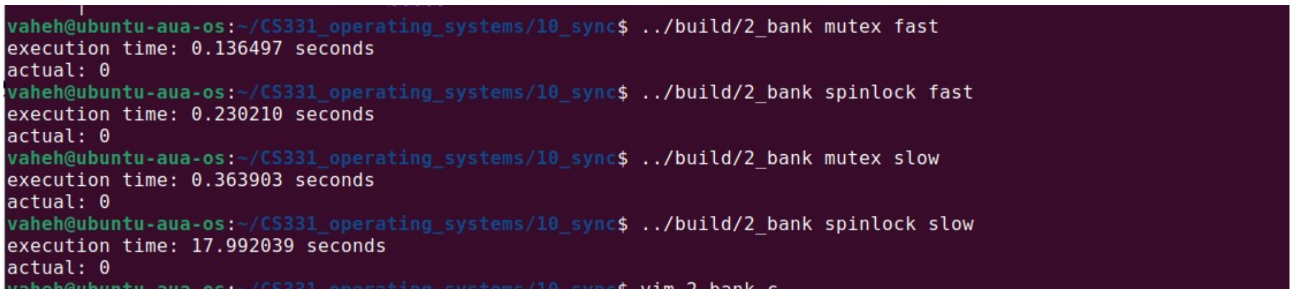
In the 1a case the value is almost never the same as the expected, because there's a race condition on the counter. So this is a critical section and needs to be protected.



```
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/1a_count
expected: 4000000, actual: 3989770
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/1b_count
expected: 4000000, actual: 4000000
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/1c_count
expected: 4000000, actual: 4000000
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$
```

P2

The experiments I did were using 20 threads. In all cases spinlock was slower (especially when using sleep). It seems that spinlock would be useful when thread count is small AND the critical section is small too. Otherwise too many threads take up processor's time unnecessarily. Mutex allows the running thread(s) to have more chance for running, because no unnecessary threads "pollute" the scheduler.



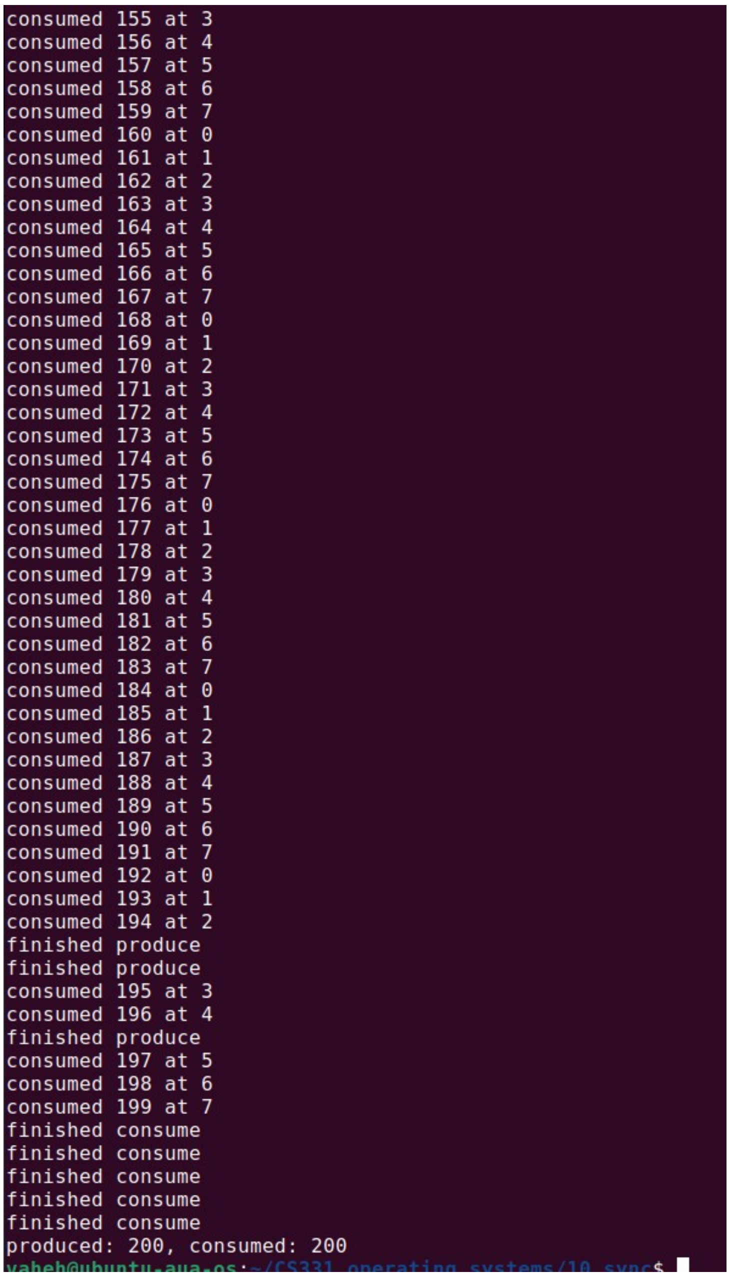
```
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/2_bank mutex fast
execution time: 0.136497 seconds
actual: 0
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/2_bank spinlock fast
execution time: 0.230210 seconds
actual: 0
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/2_bank mutex slow
execution time: 0.363903 seconds
actual: 0
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ ../build/2_bank spinlock slow
execution time: 17.992039 seconds
actual: 0
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$ vim 2_bank.c
```

P3

For this one I decided to make my life a bit harder and not share a mutex between the producer and consumer, to theoretically not block the one or the other, when there are slots available.

We still need mutexes here because the semaphor just guarantees that there are resources available, so when they're available multiple threads may start executing. And they'll access memory that is shared: critical sections. So we still need locking mechanisms to protect those.

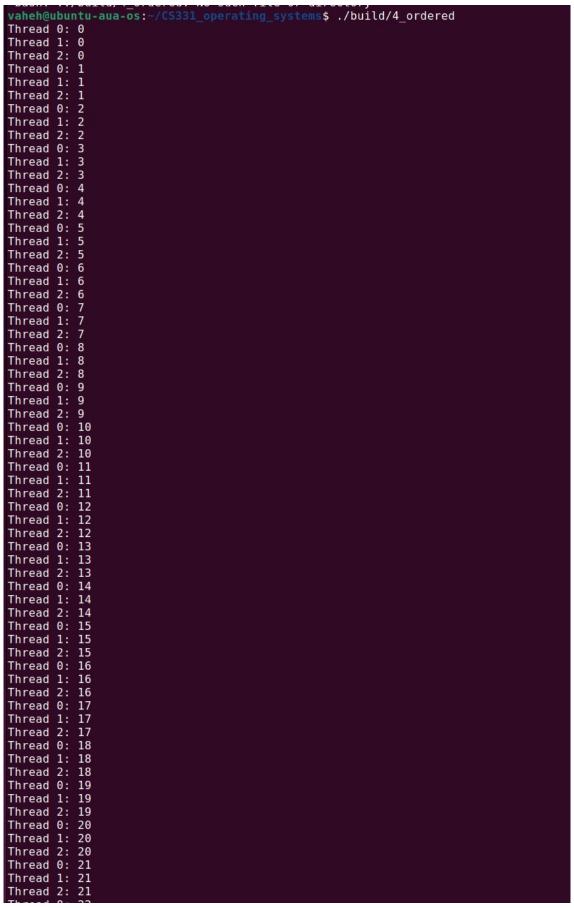
Also note: I didn't include all the error checking here because it was making hard to read and debug.



```
consumed 155 at 3
consumed 156 at 4
consumed 157 at 5
consumed 158 at 6
consumed 159 at 7
consumed 160 at 0
consumed 161 at 1
consumed 162 at 2
consumed 163 at 3
consumed 164 at 4
consumed 165 at 5
consumed 166 at 6
consumed 167 at 7
consumed 168 at 0
consumed 169 at 1
consumed 170 at 2
consumed 171 at 3
consumed 172 at 4
consumed 173 at 5
consumed 174 at 6
consumed 175 at 7
consumed 176 at 0
consumed 177 at 1
consumed 178 at 2
consumed 179 at 3
consumed 180 at 4
consumed 181 at 5
consumed 182 at 6
consumed 183 at 7
consumed 184 at 0
consumed 185 at 1
consumed 186 at 2
consumed 187 at 3
consumed 188 at 4
consumed 189 at 5
consumed 190 at 6
consumed 191 at 7
consumed 192 at 0
consumed 193 at 1
consumed 194 at 2
finished produce
finished produce
consumed 195 at 3
consumed 196 at 4
finished produce
consumed 197 at 5
consumed 198 at 6
consumed 199 at 7
finished consume
finished consume
finished consume
finished consume
finished consume
produced: 200, consumed: 200
vaheh@ubuntu-aua-os:~/CS331_operating_systems/10_sync$
```

P4

Sleep is not reliable in this case because we have no guarantees about the order of execution. Semaphore solves this because we're kind of using it as an inter-thread communication here and one thread unlocks the other when it's done.

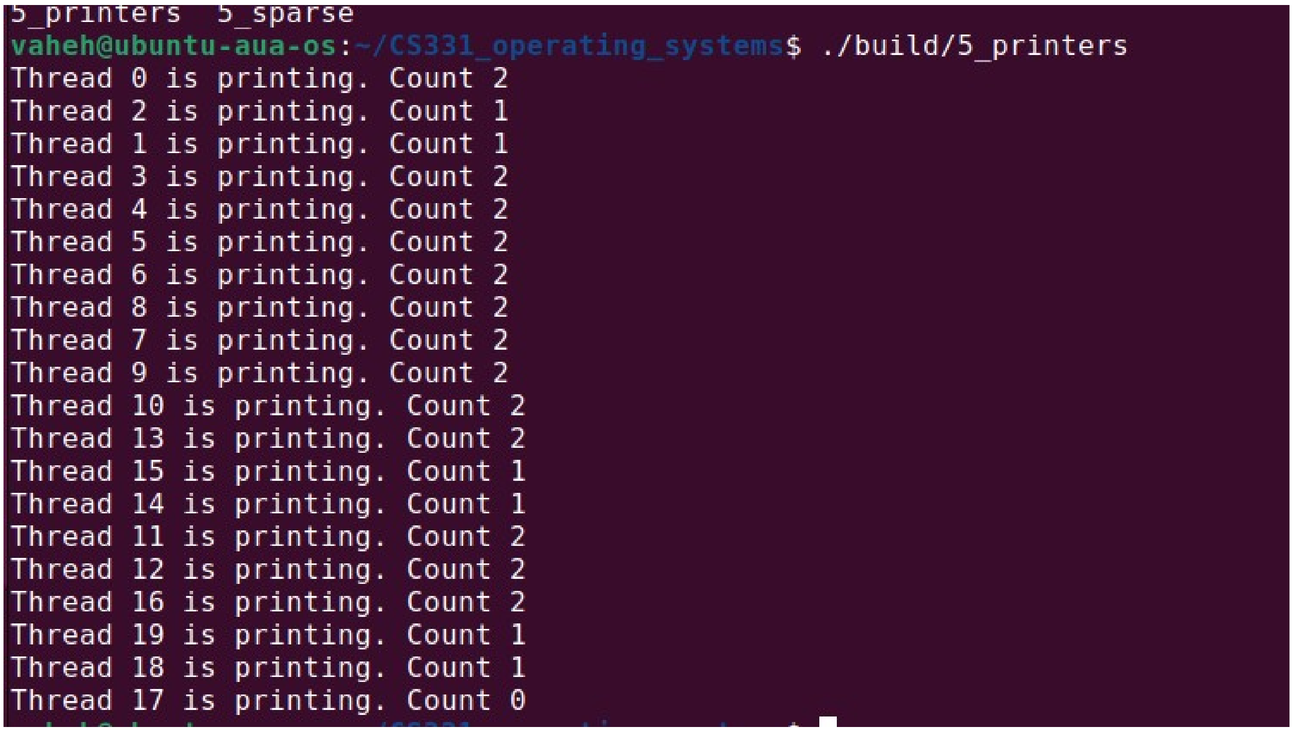


```
vaheh@ubuntu-aua-os:~/CS331_operating_systems$ ./build/4_ordered
Thread 0: 0
Thread 1: 0
Thread 2: 0
Thread 0: 1
Thread 1: 1
Thread 2: 1
Thread 0: 2
Thread 1: 2
Thread 2: 2
Thread 0: 3
Thread 1: 3
Thread 2: 3
Thread 0: 4
Thread 1: 4
Thread 2: 4
Thread 0: 5
Thread 1: 5
Thread 2: 5
Thread 0: 6
Thread 1: 6
Thread 2: 6
Thread 0: 7
Thread 1: 7
Thread 2: 7
Thread 0: 8
Thread 1: 8
Thread 2: 8
Thread 0: 9
Thread 1: 9
Thread 2: 9
Thread 0: 10
Thread 1: 10
Thread 2: 10
Thread 0: 11
Thread 1: 11
Thread 2: 11
Thread 0: 12
Thread 1: 12
Thread 2: 12
Thread 0: 13
Thread 1: 13
Thread 2: 13
Thread 0: 14
Thread 1: 14
Thread 2: 14
Thread 0: 15
Thread 1: 15
Thread 2: 15
Thread 0: 16
Thread 1: 16
Thread 2: 16
Thread 0: 17
Thread 1: 17
Thread 2: 17
Thread 0: 18
Thread 1: 18
Thread 2: 18
Thread 0: 19
Thread 1: 19
Thread 2: 19
Thread 0: 20
Thread 1: 20
Thread 2: 20
Thread 0: 21
Thread 1: 21
Thread 2: 21
```

P5

If we initialize the semaphore to 1, we'll effectively have a mutex-like lock on the resource. Only one thread will print at a time. If we have too large of a value (e.g. >= thread count), we'll just remove all restrictions because any number of threads can print at a time.

Here I'm using getvalue function after sem_wait. Initial value was 3 so maximum that can be printed is 2 because at that point at least current thread will have decremented 1 from the 3.



```
5_printers  5_sparse
vaheh@ubuntu-aua-os:~/CS331_operating_systems$ ./build/5_printers
Thread 0 is printing. Count 2
Thread 2 is printing. Count 1
Thread 1 is printing. Count 1
Thread 3 is printing. Count 2
Thread 4 is printing. Count 2
Thread 5 is printing. Count 2
Thread 6 is printing. Count 2
Thread 8 is printing. Count 2
Thread 7 is printing. Count 2
Thread 9 is printing. Count 2
Thread 10 is printing. Count 2
Thread 13 is printing. Count 2
Thread 15 is printing. Count 1
Thread 14 is printing. Count 1
Thread 11 is printing. Count 2
Thread 12 is printing. Count 2
Thread 16 is printing. Count 2
Thread 19 is printing. Count 1
Thread 18 is printing. Count 1
Thread 17 is printing. Count 0
```

Github repo: https://github.com/VaheHayrapetyan24/CS331_operating_systems