Semaphores

Name: Samiksha Modi
Roll No: 2019331

I implemented my own semaphore using the pthread library. I initialise the semaphore with the required pthread library functions. I also use the blocking and the non-blocking functions available in the Pthreads library to implement the primitives wait, signal and signal(printValue). For the shared resource that is the k forks and 2 sauce bowls, I have used a semaphore initialized with count 1. I implemented two programs - blocking (blocking_2019331.c) and non-blocking (nonblocking_2019331.c) variants.

Blocking

Inside an infinite loop in the philosopher thread function, the philosopher first tries to pick up the forks. To avoid deadlock in this scenario, the philosopher at an even position should first pick the right chopstick and then the left chopstick while the philosopher at an odd position should first pick the left chopstick and then the right chopstick.

After the philosopher has acquired both the forks, he tries to pick up both bowls. To avoid deadlock, the philosopher who has already acquired both the forks, must first acquire bowl 0 and only after acquiring bowl 0, he must acquire bowl 1.

After the philosopher has acquired both the forks and both the bowls, he eats and puts down both the forks and bowls making them available to other philosopher threads.

The semaphore is a struct with 3 fields-count, pthread_mutex_t, and pthread_cond_t. To implement wait pthread_mutex_lock() was used.We need to make the wait function an atomic operation so we acquire the mutex lock. We need to decrement the count but also check that the count is greater than 0. So we wait while the count is 0 using pthread_cond_wait(). After decrementing the count we release the lock.

To implement signal pthread_mutex_lock was used. We need to make the signal function an atomic operation so for that we acquire the mutex lock. We then need to increment the count which I do. Then we can either always notify or notify when the count becomes 1 from 0, because only then there is a likelihood of someone waiting. I chose to notify in the latter case. After that we release the mutex lock.

Non-blocking

Inside an infinite loop in the philosopher thread function, the philosopher tries to pick up the resources that are the 2 forks he can access and the 2 bowls. To avoid deadlock in this scenario, the philosopher at an even position should first pick the right chopstick and then the left chopstick while the philosopher at an odd position should first pick the left chopstick and then the right chopstick. If philosopher gets the first chopstick, he tries to pick the second chopstick. If he gets the second chopstick he tries to pick the sauce bowls. If he does not get the second chopstick, he releases the first chopstick.

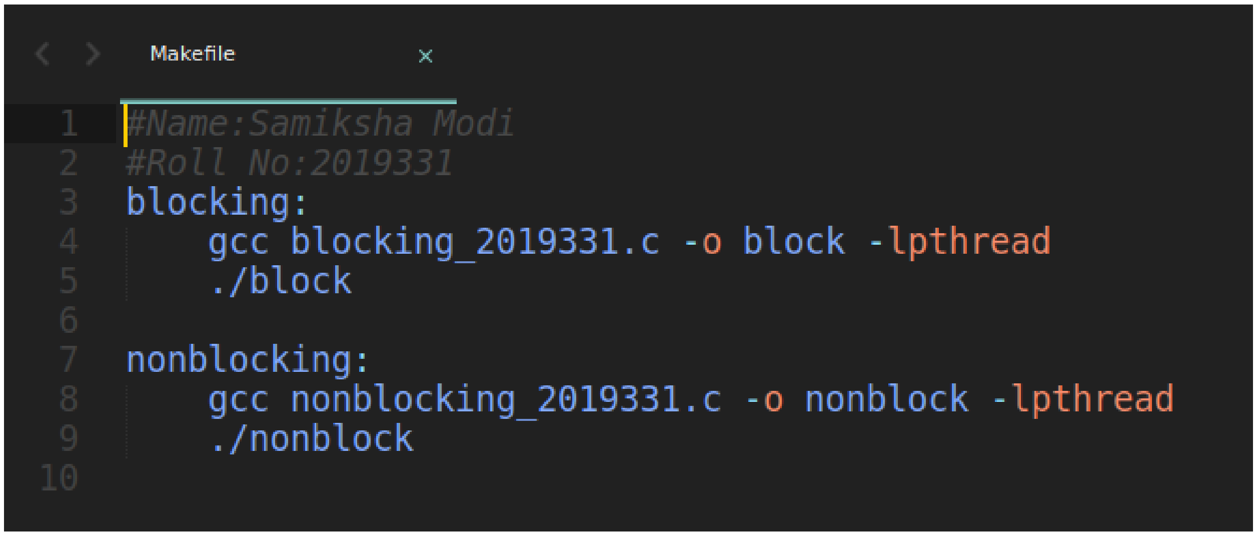
Now if the philosopher gets both the first and the second chopstick, then he tries to get the two sauce bowls. If the philosopher gets the first bowl, he then tries to get the second bowl. If he gets the second bowl also then he eats and releases both the bowls and both the forks. But if the philosopher can't get the second bowl then he releases the first bowl, thus avoiding any deadlock.

The semaphore is a struct with 2 fields-count, pthread_mutex_t.

To implement non-blocking wait pthread_mutex_trylock() was used. If a thread is not able to acquire the lock, wait returns 1. Else if it is able to acquire the lock, it checks the count. If count is less than 0, wait again returns 1. When 1 is returned it simply means that the philosopher can’t acquire that resource right now. If it can acquire the resource, the count is decremented and we release the lock

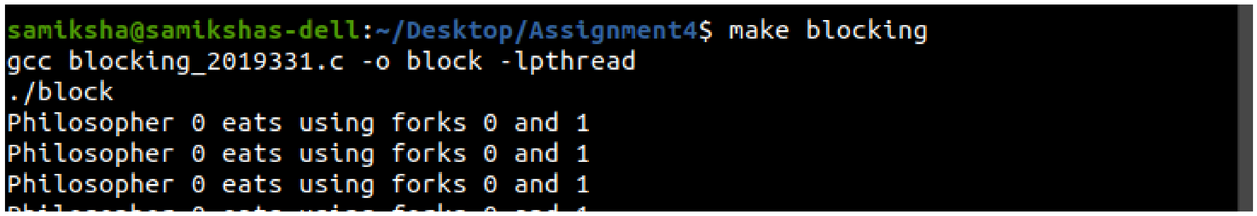
To implement signal pthread_mutex_lock was used. We need to make the signal function an atomic operation so for that we acquire the mutex lock. We then need to increment the count which I do. After that we release the mutex lock.

Makefile



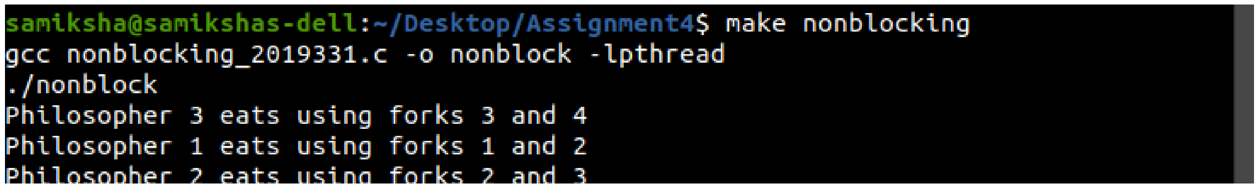
```
#Name:Samiksha Modi
#Roll No:2019331
blocking:
	gcc blocking_2019331.c -o block -lpthread
	./block

nonblocking:
	gcc nonblocking_2019331.c -o nonblock -lpthread
	./nonblock
```

To run the blocking version go to the directory where the files are stored and write *make blocking* on the terminal.



```
samiksha@samikshas-dell:~/Desktop/Assignment4$ make blocking
gcc blocking_2019331.c -o block -lpthread
./block
Philosopher 0 eats using forks 0 and 1
Philosopher 0 eats using forks 0 and 1
Philosopher 0 eats using forks 0 and 1
```

To run the nonblocking version go to the directory where the files are stored and write *make nonblocking* on the terminal.



```
samiksha@samikshas-dell:~/Desktop/Assignment4$ make nonblocking
gcc nonblocking_2019331.c -o nonblock -lpthread
./nonblock
Philosopher 3 eats using forks 3 and 4
Philosopher 1 eats using forks 1 and 2
Philosopher 2 eats using forks 2 and 3
```

References

- https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/
  Used to learn the correct syntax of locking and unlocking the pthread_mutex_t mutex
- Also referred to manpages for the syntax for trylock and other pthread library functions used
- https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
  Used to learn the implementation of a counting semaphore and wait and signal operations
- NPTEL Lecture 25: Multiple producer-multiple consumer queue, Semaphore by Prof Sorav Bansal http://www.cse.iitd.ernet.in/os-lectures Used to learn semaphores and how to implement my own counting semaphore using lock and cv. Then implemented them using the pthread library
- Initializing the corresponding fields https://linux.die.net/man/3/pthread_cond_init
  https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutex_init.html
  Referred to the above links for how to initialise