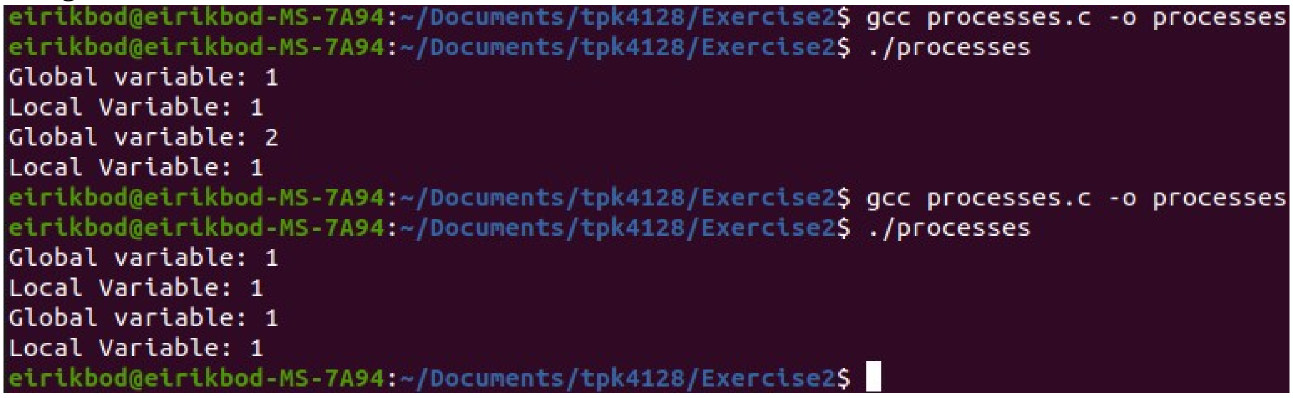
TPK4128 Exercise 2 – Concurrency

All code available at: https://github.com/bodsling/tpk4128

1.

Assignment A



```
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2$ gcc processes.c -o processes
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2$ ./processes
Global variable: 1
Local Variable: 1
Global variable: 2
Local Variable: 1
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2$ gcc processes.c -o processes
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2$ ./processes
Global variable: 1
Local Variable: 1
Global variable: 1
Local Variable: 1
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2$ 
```

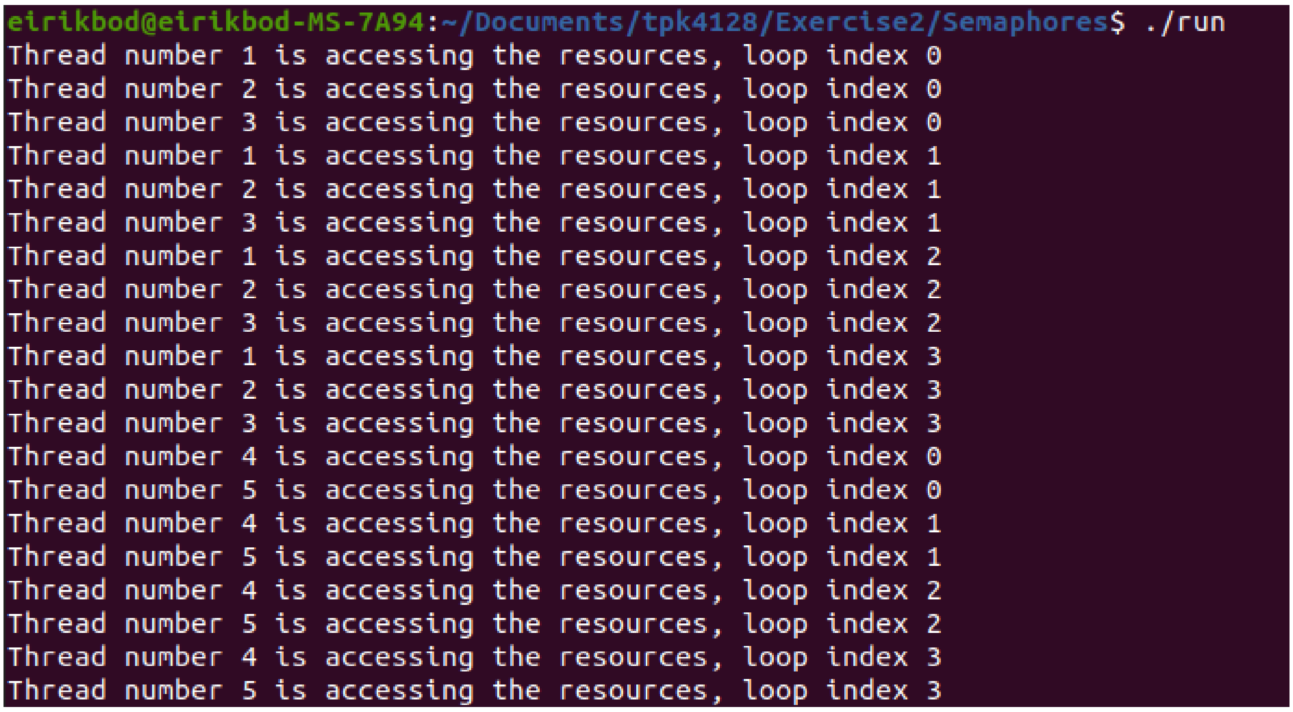
When run with vfork (first call above), the global variable is accessible from the child process, and thus is incremented in both functions. Vfork creates a child process and blocks the parent until the child process is exited, the child gets access to the memory space of the parent function, and can thus update the global variable, which is carried over to the parent process.

The second call uses fork, and this makes a child process and a parent process, which runs in different memory spaces, but the memory spaces has the same content at the time of fork. This means that the 2 processes cannot increment/update the same variable, ass seen from the printed values.

See git for code files.

- Difference between vfork and fork is that fork makes a child to the parent, which runs in different memory spaces (but initially equal), and cannot access each others variables. Vfork creates a child and blocks the parent, to child is ran first, and the child now has access to the same memory space as the parent. When the child exit, the parent can run.
- POSIX threads can modify variables, globally, or when passed to the thread function. If the variables are modified at the same time by two threads, missintention may happen, of this is not handled by i.e. Mutexes.

2.

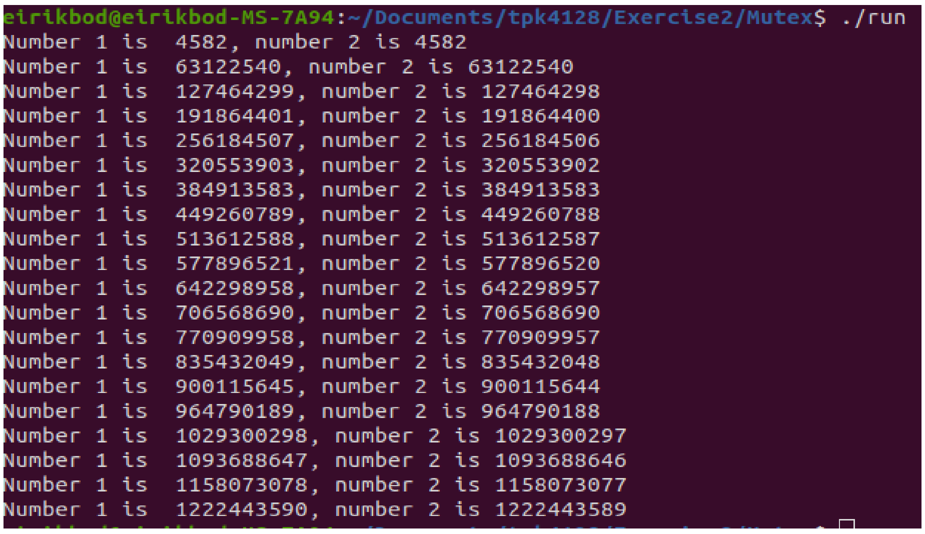


```
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2/Semaphores$ ./run
Thread number 1 is accessing the resources, loop index 0
Thread number 2 is accessing the resources, loop index 0
Thread number 3 is accessing the resources, loop index 0
Thread number 1 is accessing the resources, loop index 1
Thread number 2 is accessing the resources, loop index 1
Thread number 3 is accessing the resources, loop index 1
Thread number 1 is accessing the resources, loop index 2
Thread number 2 is accessing the resources, loop index 2
Thread number 3 is accessing the resources, loop index 2
Thread number 1 is accessing the resources, loop index 3
Thread number 2 is accessing the resources, loop index 3
Thread number 3 is accessing the resources, loop index 3
Thread number 4 is accessing the resources, loop index 0
Thread number 5 is accessing the resources, loop index 0
Thread number 4 is accessing the resources, loop index 1
Thread number 5 is accessing the resources, loop index 1
Thread number 4 is accessing the resources, loop index 2
Thread number 5 is accessing the resources, loop index 2
Thread number 4 is accessing the resources, loop index 3
Thread number 5 is accessing the resources, loop index 3
```

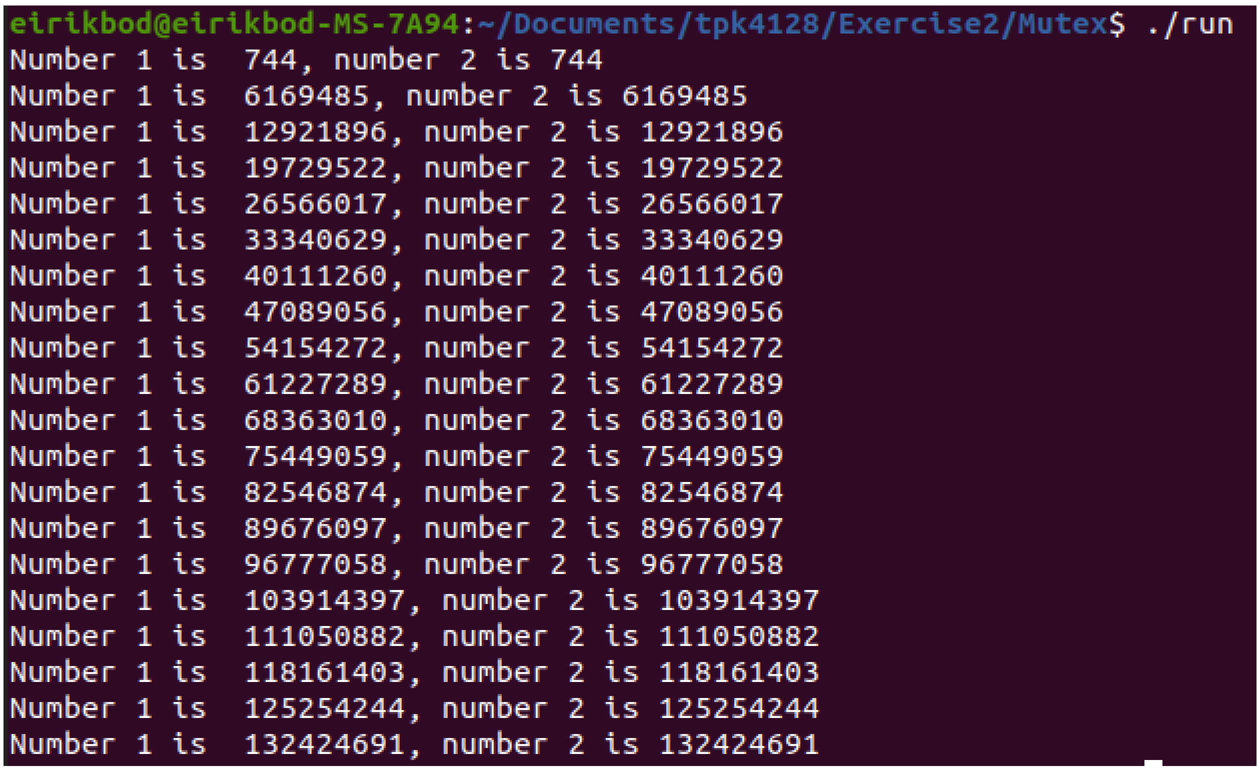
- The threads that does not get access to the resources straight away goes into a waiting state, where it uses no CPU time until it gets to increment the semaphore, and gets access to the resources.
- The 3 resources is allocated to the firsts three threads to be run, and is kept by the threads until they release their resource (in this example). By using semaphores, the semaphore may be incremented up by another process/thread.

3.

Running without mutex, the number 1 and 2 may be different, because the first thread change the variables between the time the second thread read the value of number 1 and number 2



```
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2/Mutex$ ./run
Number 1 is  4582, number 2 is 4582
Number 1 is  63122540, number 2 is 63122540
Number 1 is  127464299, number 2 is 127464298
Number 1 is  191864401, number 2 is 191864400
Number 1 is  256184507, number 2 is 256184506
Number 1 is  320553903, number 2 is 320553902
Number 1 is  384913583, number 2 is 384913583
Number 1 is  449260789, number 2 is 449260788
Number 1 is  513612588, number 2 is 513612587
Number 1 is  577896521, number 2 is 577896520
Number 1 is  642298958, number 2 is 642298957
Number 1 is  706568690, number 2 is 706568690
Number 1 is  770909958, number 2 is 770909957
Number 1 is  835432049, number 2 is 835432048
Number 1 is  900115645, number 2 is 900115644
Number 1 is  964790189, number 2 is 964790188
Number 1 is  1029300298, number 2 is 1029300297
Number 1 is  1093688647, number 2 is 1093688646
Number 1 is  1158073078, number 2 is 1158073077
Number 1 is  1222443590, number 2 is 1222443589
```

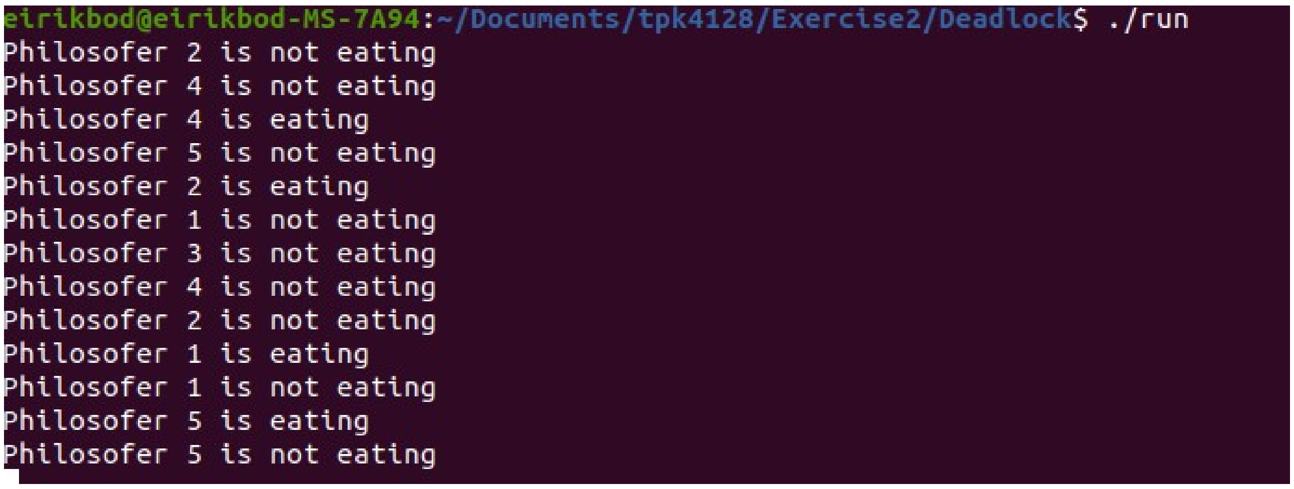


```
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2/Mutex$ ./run
Number 1 is  744, number 2 is 744
Number 1 is  6169485, number 2 is 6169485
Number 1 is  12921896, number 2 is 12921896
Number 1 is  19729522, number 2 is 19729522
Number 1 is  26566017, number 2 is 26566017
Number 1 is  33340629, number 2 is 33340629
Number 1 is  40111260, number 2 is 40111260
Number 1 is  47089056, number 2 is 47089056
Number 1 is  54154272, number 2 is 54154272
Number 1 is  61227289, number 2 is 61227289
Number 1 is  68363010, number 2 is 68363010
Number 1 is  75449059, number 2 is 75449059
Number 1 is  82546874, number 2 is 82546874
Number 1 is  89676097, number 2 is 89676097
Number 1 is  96777058, number 2 is 96777058
Number 1 is  103914397, number 2 is 103914397
Number 1 is  111050882, number 2 is 111050882
Number 1 is  118161403, number 2 is 118161403
Number 1 is  125254244, number 2 is 125254244
Number 1 is  132424691, number 2 is 132424691
```

Running with a mutex, we see that this problem is solves, and number 1 and 2 is read, while only thread 2 has access to the numbers. This makes number 1 and 2 always equal as intended by the code.

4.

- The deadlock happen when all philosophers has their left fork (or right fork) in their hand, but is waiting for the right fork to start eating. This way no philosopher will ever get to eat. As can be seen by the print, after a few iterations, no philosopher is eating.



```
eirikbod@eirikbod-MS-7A94:~/Documents/tpk4128/Exercise2/Deadlock$ ./run
Philosofer 2 is not eating
Philosofer 4 is not eating
Philosofer 4 is eating
Philosofer 5 is not eating
Philosofer 2 is eating
Philosofer 1 is not eating
Philosofer 3 is not eating
Philosofer 4 is not eating
Philosofer 2 is not eating
Philosofer 1 is eating
Philosofer 1 is not eating
Philosofer 5 is eating
Philosofer 5 is not eating
```

- I solved the deadlock by adding a new mutex, a waiter. The waiter needs to allow a philosopher to pick up forks, and only both at the same time. So if a philosopher is allowed by the waiter to pick up forks, and picks up one, every philosopher has to wait for him to pick up the second fork before they can get permission to pick up theirs.
- The resource hierarchy solution is proposed by wikipedia, where the forks are numbered 1 through 5, and only the highest numbered fork of the two of each philosopher is allowed to be picked up first (Not only left first or right first). This makes that one philosopher is not allowed to pick up their LH fork, if all other philosophers has their LH forks, and thus the deadlock will not happen.