Operating System Concepts Weekly

Week 4

Lucas van der Laan
s1047485

1

The code below does not have any deadlocks, because we safely handle all the shared resources and make sure that that cleaner always cleans up whenever the cleaner's semaphore gets released. The cleaner's thread gets created first, as he is on standby before any customer enters. Whenever a customer enters, we lock the semaphore that keeps track of the customers, we then check if there are more than 5 customers and if so the customer waits. The customers that are already at a table have a nice meal and when they leave, we again handle the customers variable with care but we also tell the cleaner to clean the table. Then when the cleaner cleans the table, the cleaner tells the customer that is waiting for a table that a table is ready. It does this in a loop, thus there will never be deadlocks.

```
#include <iostream>
#include <thread>
#include <semaphore.h>

using namespace std;

#define TOTAL_CUSTOMERS 50

sem_t cleanSempahore, customersSemaphore, tableSemaphore;

int customers = 0;

void clean() {
  sem_post(&tableSemaphore);
  cout << "Cleaned table" << endl;
}

void eat() {
  // This is an emtpy function.
  // The assignment states that the cleaner cleans when the customer leaves
  // not after they are done eating.
  // So this is just here to fulfill the requirement of having the function.
}
```

```
24
25  void leave() {
26    sem_wait(&customersSemaphore);
27    customers--;
28    sem_post(&customersSemaphore);
29    sem_post(&cleanSempahore);
30  }
31
32  void cleaner() {
33    while(true) {
34      sem_wait(&cleanSempahore);
35      clean();
36    }
37  }
38
39  void customer() {
40    sem_wait(&customersSemaphore);
41
42    if (customers >= 5) {
43      sem_wait(&tableSemaphore);
44    }
45
46    customers++;
47    sem_post(&customersSemaphore);
48
49    eat();
50    leave();
51  }
52
53  int main(int argc, char const *argv[]) {
54    sem_init(&cleanSempahore, 0, 1);
55    sem_init(&customersSemaphore, 0, 1);
56    sem_init(&tableSemaphore, 0, 1);
57
58    thread cleanerThread(cleaner);
59    sem_wait(&cleanSempahore);
60    for (size_t i = 0; i < TOTAL_CUSTOMERS; i++)
61    {
62      thread customerThread(customer);
63      customerThread.detach();
64    }
65
66
67    cleanerThread.join();
68    return 0;
69  }
```

2

The code below does not suffer from starvation, because we force the readers to give up the moment a writer asks for access. This results in the writer having preference over the reader.

Reader

```
readersAllowed.wait();
m_readers.wait();
readers++;
if (readers == 1)
    writersAllowed.wait();
m_readers.signal();
readersAllowed.signal();

/* Read from resource */

m_readers.wait();
readers--;
if (readers == 0)
    writersAllowed.write();
m_readers.signal();
```

Writer

```
m_writers.wait();
writers++;
if (writers == 1)
    readersAllowed.wait();
m_writers.signal();
writersAllowed.wait();

/* Write to resource */

writersAllowed.signal();

m_writers.wait();
writers--;
if (writers == 0)
    readersAllowed.signal();
m_writers.signal();
```

3

```cpp
#include <iostream>
#include <thread>
#include <semaphore.h>

sem_t nextSempahore;

const int n = 10000;
int next = 0;
bool primes[n];
std::thread threads[n];

bool isPrime (int n) {
  if (n == 0 || n == 1) {
    return false;
  }
  for (int i = 2; i < n; i++) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}

void T() {
  sem_wait(&nextSempahore);
  int index = next;
  next++;
  sem_post(&nextSempahore);

  primes[index] = isPrime(index);
}

int main(int argc, char const *argv[]) {
  sem_init(&nextSempahore, 0, 1);

  for (size_t i = 0; i < n; i++) {
    threads[i] = std::thread(T);
  }

  for (size_t i = 0; i < n; i++) {
    threads[i].join();
  }

  return 0;
}
```