Theoretical 2

2.1 Race conditions

Consider the two parallel threads $t1$ and $t2$ that share their data. Initially the valyes of y and $z = 0$:

```
int t1() {
  int x;
  // initialization code
  x = y + z
  //other code
}
```

```
int t2() {
  // initialization code
  y = 1;
  z = 2;
  // other code
}
```

a) Give all possible final values for x and the correspondig order of execution of instructions in $t1$ and $t2$

x can be 0, 1 and 3. In the code snippets below, comments with arrows indicate a task switch.

```
// Possibility 1
// t1
x = y + z; // x = 0
//t1-> t2
y = 1;
z = 2;

// possibility 2
// t2
y = 1;
// t2 -> t1
x = y + z; // x = 1
//t1-> t2
z = 2;

// possibility 3
// t2
y = 1;
z = 2;
// t2 -> t1
x = y + z; // x = 3
```

b) Is it possible to use semaphores so that the final value of x is 2? If so, give a solution using semaphores and wait/signal operations. If not, explain why.

No, it is not possible for x to be equal 2. This is because in whichever way we use semaphores, y would always get a value other than zero before z because is "higher" up in the code, meaning it will execute first.

2.2 Semaphores

Consider the two parallel threads $t1$ and t2.

```
int t1() {
  printf("w");
  printf("d");
}
```

```
int t2() {
  printf("o");
  printf("r");
  printf("l");
  printf("e");
}
```

a) Use semaphores and insert wait/signal calls into the two threads so that only "wordle" is printed

The threads share two common semaphores:

```
Semaphore lock1, lock2;
```

```
int t1() {
  printf("w");
  signal(&lock1); // new
  wait(&lock2);   // new
  printf("d");
  signal(&lock2); // new
}
```

```
int t2() {
  wait(&lock1);   // new
  printf("o");
  printf("r");
  signal(&lock2); // new
  wait(&lock1);   // new
  printf("l");
  printf("e");
  signal(&lock1); // new
}
```

b) Give the required initial values for the semaphores

```
Semaphore lock1 = 0, lock2 = 0;
```

2.3 Even more semaphores

Consider the parallel threads $t1$, $t2$ and t3 using the following common semaphores:

```
semaphore s_a = 0, s_b = 0, s_c = c;
```

```
int t1() {
  while(1) {
    printf("A");
    s_c.signal();
    s_a.wait();
  }
}
```

```
int t2() {
  while(1) {
    printf("B");
    s_c.signal();
    s_b.wait();
  }
}
```

```
int t3() {
  while(1) {
    s_c.wait();
    s_c.wait();
    printf("C");
    s_a.signal();
    s_b.wait();
  }
}
```

Which strings can be output when running the three threads in parallel?

For "C" to be printed, s_c needs to be unlocked twice. After either "A" or "B" is printed, $t1$ and $t2$ goes into a locked state, waiting for t3 to print "C", before s_a and s_b is unlocked again and a new cycle starts.
We therefore get the pattern:
$'((AB|BA)C)^{*}'$

2.4 Deadlocks

Consider the parallel threads $t1$ and $t2$ using the following common variables and semaphores:

```
1 int x = 0, y = 0, z = 0;
2 semaphore lock1 = 1, lock2 = 1;
```

```
1 int t1() {
2   z = z + 2;
3   lock1.wait();
4   x = x + 2;
5   lock2.wait();
6   lock1.signal();
7   y = y + 2;
8   lock2.signal();
9 }
```

```
1 int t2() {
2   lock2.wait();
3   y = y + 1;
4   lock1.wait();
5   x = x + 1;
6   lock1.signal();
7   lock2.signal();
8   z = z + 1;
9 }
```

a) Executing the threads in parallel could result in a deadlock. Why?

If $t1$ locks $lock1$ in line 3, then $t2$ locks $lock2$ in line 2, we would get a deadlock. In $t1$ line 5, the program would wait for lock2 to get unlocked, however in $t2$ line 4, the program waits for lock1 to be unlocked, resulting in that both threads are waiting for each other, hence a deadlock.

b) What are the possible values of x, y and z in the deadlock state?

We would get $x = 2$, $y = 1$ and $z = 2$.

c) What are the possible values of x, y and z if the program terminates successfully?

Line 2 in $t1$ and line 8 in $t2$ are not "protected" by a semaphore, and we could get a situation where z are accessed in memory at the same time in $t1$ and $t2$. Operations carried out at the same time could then be overwritten, meaning the possible values for z is 1, 2 and 3 (the program can also distribute processes such that line 2 in $t1$ and line 8 in $t2$ does not happen at the same time, yielding $z = 3$. x and y are "protected" by the semaphores and operations on them in $t1$ and $t2$ can not happen at the same time. This would yield $x = 3$ and $y = 3$.
To summarize:

$x = 3$, $y = 3$ and $z = 1, 2, 3$.