Norwegian University of Science and Technology

Operating Systems

Example solutions for Theoretical Exercise 2

Michael Engel

2.1 Race conditions

Consider the two parallel threads t1 and t2 that share their data (variables). Initially, the values of y and z = 0.

t1:

```
1 int t1() {
2    int x;
3    x = y + z;
4 }
```

t2:

```
1 int t2() {
2    y = 1;
3    z = 2;
4 }
```

a. Give all possible final values for x and the corresponding order of execution of instructions in t1 and t2 (indicate task switches ↯).

- t1 runs to the end first ↯ then t2 to the end: x = 0
- t2 to line 2 ↯ then t1 to the end ↯ then t2 to the end: x = 1
- t2 to the end ↯ then t1 to the end: x = 3

Are there other possibilities giving additional values?

2.1 Race conditions

t1:

```
1 int t1() {
2    int x;
3    x = y + z;
4 }
```

t2:

```
1 int t2() {
2    y = 1;
3    z = 2;
4 }
```

a. Give all possible final values for x and the corresponding order of execution of instructions in t1 and t2 (indicate task switches ↯).

Are there other possibilities giving additional values?

- This depends on the code generated by the compiler
- Additions (t1 l.3) often consist of multiple instructions in machine language, e.g.:
  - A. fetch operand y into register r1
  - B. fetch operand z into register r2
  - C. add r1 + r2, store result in r3
  - D. store r3 in memory location of x
- If a task switch to t2 occurs between machine instructions A and B and t2 runs to completion before switching back to t1, then:
  - y is read as 0 (t2 didn't set y yet)
  - z is read as 2 (t2 sets z before execution instruction B of add. in t1)
  - ***the sum is then x = 0 + 2***

2.1 Race conditions

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***t***. (typo in the exercise…)

- The addition x = y + z is a ***critical section***
  - We can protect it with a semaphore or mutex:

t1:

```
0 sem s = 1;
1 int t1() {
2    int x;
3    s.wait();
4    x = y + z;
5    s.signal();
6 }
```

t2:

```
1 int t2() {
2    s.wait();
3    y = 1;
4    z = 2;
5    s.signal();
6 }
```

  - But this can only guarantee that x can ***never*** have the value 2
  - The ***opposite*** would require splitting the addition into steps as shown on the previous slide

2.2 Semaphores

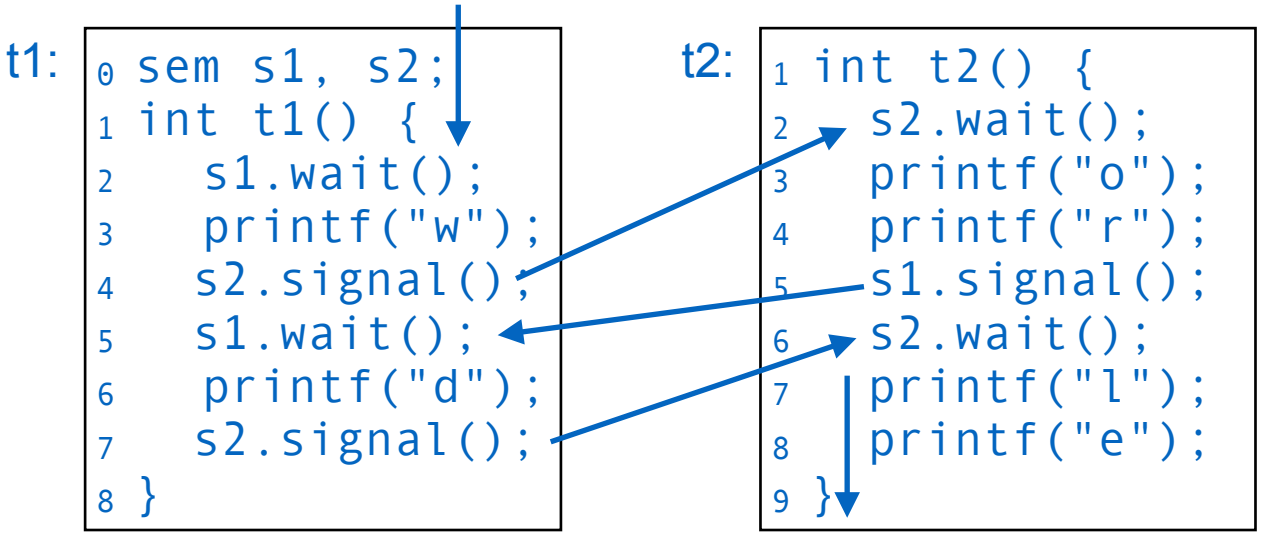
t1:

```
1 int t1() {
2   printf("w");
3   printf("d");
4 }
```

t2:

```
1 int t2() {
2   printf("o");
3   printf("r");
4   printf("l");
5   printf("e");
6 }
```

a. Use semaphores and insert wait/signal calls into the two threads so that only “wordle” is printed.



2.2 Semaphores

b. Give the required initial values for the semaphores.

t1:

```
0 sem s1, s2;
1 int t1() {
2   s1.wait();
3   printf("w");
4   s2.signal();
5   s1.wait();
6   printf("d");
7   s2.signal();
8 }
```

t2:

```
1 int t2() {
2   s2.wait();
3   printf("o");
4   printf("r");
5   s1.signal();
6   s2.wait();
7   printf("l");
8   printf("e");
9   s2.signal();
10}
```

- t1 has to run first to print "w", so `s1` has to be set to 1 initially.
- t2 has to wait until the "w" has been printed.
  - It is signalled by t1, so the initial value of s2 has to be 0.

2.3 Even more semaphores

```
1 int t1() {
2   while(1) {
3     printf("A");
4     s_c.signal();
5     s_a.wait();
6   }
7 }
```

```
1 int t2() {
2   while(1) {
3     printf("B");
4     s_c.signal();
5     s_b.wait();
6   }
7 }
```

```
1 int t3() {
2   while(1) {
3     s_c.wait();
4     s_c.wait();
5     printf("C");
6     s_a.signal();
7     s_b.signal();
8   }
9 }
```

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

Which strings can be output when running the 3 threads in parallel?

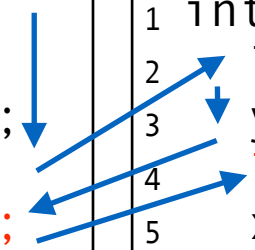
- Either t1 or t2 could start first, so the first letter can be A or B
- Then both t1 and t2 signal s_c, only after both have signalled s_c, t3 can start and print C
- This, t3 signals s_a and s_b, which start in arbitrary order again
- Accordingly, the output is `([AB|BA]C)+`
  - so print A or B in arbitrary order, then print C, then the process starts again
  - Here, we have used a ***regular expression*** to indicate the structure of a text pattern. Regular expressions (short: regexps) are a common tool in Unix

2.4 Deadlocks

```
int x=0, y=0, z=0;
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?

- t1 runs first until line 4 (so lock1=0, lock2=1) ↯ switch to t2
- t2 starts and runs until line 3 (so lock1=0, lock2=0) ↯ back to t1
- t1 waits for lock2 in line 5 ↯ switch to t2, waits for lock1 in line 4
- This results in a ***mutual waiting condition*** which is not resolved

Note that this deadlock does not occur in all execution/task switch orders!

2.4 Deadlocks

```
int x=0, y=0, z=0;
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?

Are there other execution orders leading to a deadlock?

- t2 runs first until line 2 (so lock2=0, lock1=1) ↯ switch to t1
- t1 starts and runs until line 3 (so lock1=0, lock2=0) ↯ back to t2
- t2 waits for lock2 in line 4 ↯ switch to t1, waits for lock1 in line 5

2.4 Deadlocks

```
int x=0, y=0, z=0;
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 }
```

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

- x = 2, y = 1, z = 2

c. What are the possible values of x, y and z if the program terminates successfully (i.e., without a deadlock)?
Hint: Remember that an assignment z = z + 1 consists of multiple atomic operations on x.

- t1 runs first to the end, then t2 (or vice versa): x=3, y=3, z=3
- But a thread switch could e.g. also occur in the "middle" of line 8 of t2, e.g. before z is written back ↯ switch to t1 (z=2), run t1 to the end ↯ switch to t2, write back its value of z ➙ z=1!

Can you find other possible orders that run to completion?