FIT2100 Tutorial #6
Concurrency:
Mutual Exclusion and Synchronisation, Deadlock and Starvation
Week 10 Semester 2 2018

October 2, 2019

Revision Status:

`$Id: FIT2100-Tutorial-04.tex, Version 2.0 2018/08/28 15:30 Jojo $`

`Updated Oct 2019 by Daniel Kos`

Acknowledgement

The majority of the content presented in this tutorial was adapted from: William Stallings (2017). *Operating Systems: Internals and Design Principles (9th Edition)*, Pearson.

Contents

1 Background

This tutorial provides students with the opportunity to explore further on the various concepts of concurrency as discussed in the lectures.

You should complete the suggested reading in Section 2 before attending the tutorial. You should also prepare the solutions for the two sets of practice tasks given in Section 3 and Section 4 respectively.

Selected questions have marks attached. You should ensure you are marked within the tutorial session. These are used to guage your level of participation and to ensure you receive feedback in class. Other questions do not have marks attached but are equally important as discussion and problem-solving exercises.

2 Pre-tutorial Reading

You should complete the following two sets of reading:

- Lecture Notes: Week 8b, Week 9
- Stallings' textbook: Chapter 5 and Chapter 6

3 Synchronisation and Mutual Exclusion

3.1 Review Questions

Question 1

What is a *race condition*?

Question 2

What is mutual exclusion? Is mutual exclusion important for the execution of execution of concurrent processes?

Question 3

What is the difference between *binary* semaphores and *general* semaphores?

3.2 Problem-Solving Tasks

3.2.1 Task 1 (3 marks)

Consider the pseudocode of the following program:

```
P1:
{   shared int x;
    x = 10;
    while(1) {
        x = x - 1;
        x = x + 1;
        if (x != 10) {
            printf("x is %d", x);
        }
    }
}
```

```
P2:
{   shared int x;
    x = 10;
    while(1) {
        x = x - 1;
        x = x + 1;
        if (x != 10) {
            printf("x is %d", x);
        }
    }
}
```

Note that the scheduler in a uniprocessor system would implement pseudo-parallel execution of these two concurrent processes by *interleaving their instructions*, without restriction on the order of the interleaving.

Show a sequence (i.e. trace the sequence of inter-leavings of statements) such that the statement "`x is 10`" is printed.

3.2.2 Task 2

Consider the pseudocode of the following program:

```
const int n = 50;
int tally;

void total() {
    int count;
    for (count = 1; count <= n; count++)
        tally++;
}

void main() {
    tally = 0;
    parbegin (total(), total());
    printf ("%d\n", tally);
}
```

(a) Determine the proper lower- and upper-bound on the final value of the shared variable `tally` output by the concurrent program. Assume that the process can execute in any relative speed and that a value can be incremented after it has been loaded into a register by a separate machine instruction.

(b) Suppose that an arbitrary number of the above total processes are permitted to execute in parallel (say `N` such processes) under the same assumption of part (a). What effect this modification may have on the range of final values of `tally`?

3.2.3 Task 3

Consider the following definitions of *semaphores*:

```
1   void semWait(s)
2   {
3           if (s.count > 0) {
4                   s.count--;
5           }
6           else {
7                   /* place this process in s.queue */
8                   /* block this process */
9           }
10  }
11
12  void semSignal(s)
13  {
14          if (there is at least one process blocked on semaphore s) {
15                  /* remove a process P from s.queue */
16                  /* place process P on the ready list */
17          }
18          else {
19                  s.count++;
20          }
21  }
```

Compare this set of definitions with the other set of definitions presented in the lecture notes (Lecture 8, Slide 12). Note one difference: with the preceding definition, a semaphore can never take on a negative value. Is there any difference in the effect of the two sets of definitions when used in programs? Could you substitute one set for the other without altering the meaning of the program?

4 Deadlock and Starvation

4.1 Review Questions

Question 1

What are the four conditions that create *deadlocks*?

Question 2

Why should *mutual exclusion* not be disallowed in order to prevent deadlocks?

Question 3

How can the *hold-and-wait* condition be prevented?

Question 4

Suggest two ways in which the *no-preemption* condition can be prevented.

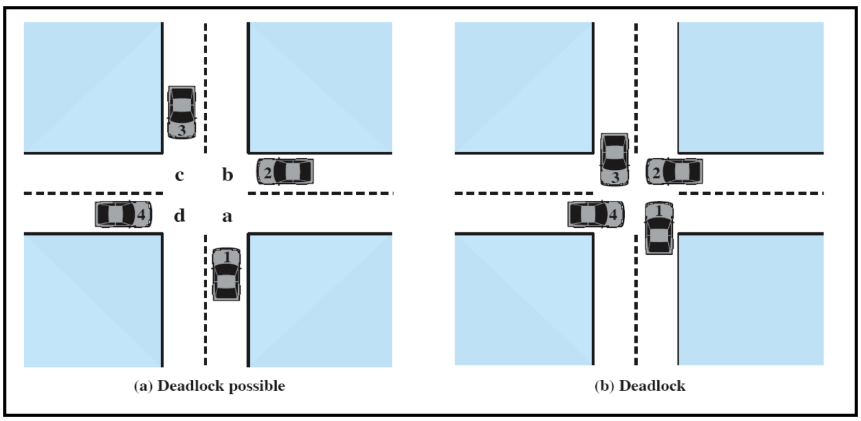
Question 5

Describe the three general approaches used for handling the problem of deadlocks.

4.2 Problem-Solving Tasks

4.2.1 Task 1

Demonstrate that the four conditions of deadlock apply to the following figure.



4.2.2 Task 2

Discuss how each of the approaches of *prevention*, *avoidance*, and *detection* can be applied to the deadlock situation presented in Task 1.

4.2.3 Task 3 (4 marks)

Consider a system with a total of 150 units of memory and are allocated to three processes as shown below:

Process	Max	Hold
1	70	45
2	60	40
3	60	15

Apply the **Banker's algorithm** to determine whether it would be safe to grant each of the following requests. If *yes*, indicate a sequence of terminations that could be guaranteed possible. If *no*, show the reduction of the resulting allocation table.

(a) A fourth process arrives, with a maximum memory need of 60 and an initial need of 25 units.

(b) A fourth process arrives, with a maximum memory need of 60 and an initial need of 35 units.

4.2.4 Task 4 (3 marks)

Consider the pseudocode of the following program, where three processes are competing for six resources labelled as **A** to **F**. By using a *resource allocation graph*, demonstrate the possibility of a deadlock in this implementation.

```
void P0()
{
   while (true) {
      get(A);
      get(B);
      get(C);
   // critical region:
   // use A, B, C
      release(A);
      release(B);
      release(C);
   }
}
```

```
void P1()
{
   while (true) {
      get(D);
      get(E);
      get(B);
   // critical region:
   // use D, E, B
      release(D);
      release(E);
      release(B);
   }
}
```

```
void P2()
{
   while (true) {
      get(C);
      get(F);
      get(D);
   // critical region:
   // use C, F, D
      release(C);
      release(F);
      release(D);
   }
}
```