FIT2100 Tutorial #5
Concurrency:
Mutual Exclusion and Synchronisation, Deadlock and Starvation
(Suggested Solutions)
Week 9 Semester 2 2017

Dr Jojo Wong
Lecturer, Faculty of IT.
Email: Jojo.Wong@monash.edu

September 13, 2017

Revision Status:

`$Id: FIT2100-Tutorial-05.tex, Version 1.0 2017/09/13 18:30 Jojo $`

Acknowledgement

The majority of the content presented in this tutorial was adapted from:

- William Stallings (2015). *Operating Systems: Internals and Design Principles (8th 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.

2 Pre-tutorial Reading

You should complete the following two sets of reading:

- Lecture Notes: Week 4(b), Week 5, and Week 7
- Stalling's textbook (7th/8th Edition): Chapter 5 and Chapter 6

3 Synchronisation and Mutual Exclusion

3.1 Review Questions

Question 1

Four design issues relevant to the concept of *concurrency*:

(a) Communicating among processes

(b) Sharing of and competing for resources

(c) Synchronisation of the activities of multiple processes

(d) Allocation of processor time to processes

Question 2

A race condition occurs when multiple processes or threads read and write data items, such that the final result depends on the order (relative timing) of the execution of instructions in the multiple processes.

Question 3

The ability to enforce *mutual exclusion*.

(a) Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.

(b) A process that halts in its non-critical section must do so without interfering with other processes.

(c) It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.

(d) When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.

(e) No assumptions are made about relative process speeds or number of processors.

(f) A process remains inside its critical section for a finite time only.

Question 4

A *binary* semaphore may only take on the values `0` and `1`. A *general* semaphore may take on any integer value.

Question 5

With *direct* addressing, the `send` primitive includes a specific identifier of the destination process. The `receive` primitive can be handled using two approaches. One possibility is to require that process explicitly designate a sending process. In other cases where it is impossible to specify the anticipated source process, an implicit approach is used, where the `source` parameter of the `receive` primitive possesses a value returned when the receive operation has been performed.

Indirect addressing is a more general approach, where messages are not sent directly from sender to receiver but rather are sent to a shared data structure consisting of *queues* that can temporarily hold messages. Such queues are generally referred to as *mailboxes*. Thus, one process sends a message to the appriopriate mailbox and the other process picks up the message from the mailbox.

3.2 Problem-Solving Tasks

3.2.1 Task 1

For "`x is 10`", the interleaving producing the required behavior is easy to find since it requires only an interleaving at the source language statement level.

The essential fact here is that the test for the value of `x` is interleaved with the increment of `x` by the other process. Thus, `x` was not equal to `10` when the test was performed, but was equal to `x` by the time the value of `x` was read from memory for printing.

	Memory (x)
`P1: x = x - 1;`	9
`P1: x = x + 1;`	10
`P2: x = x - 1;`	9
`P1: if (x != 10)`	9
`P2: x = x + 1;`	10
`P1: printf("x is %d", x);`	10

3.2.2 Task 2

(a) On casual inspection, it appears that `tally` will fall in the range $50 \leq$ `tally` ≤ 100 since from `0` to `50` increments could go unrecorded due to the lack of mutual exclusion. The basic argument contends that by running these two processes concurrently we should not be able to derive a result lower than the result produced by executing just one of these processes sequentially. But consider the following interleaved sequence of the load, increment, and store operations performed by these two processes when altering the value of the shared variable:

(1) Process A loads the value of `tally`, increments `tally`, but then loses the processor (it has incremented its register to `1`, but has not yet stored this value).

(2) Process B loads the value of `tally` (still zero) and performs forty-nine complete increment operations, losing the processor after it has stored the value `49` into the shared variable `tally`.

(3) Process A regains control long enough to perform its first store operation (replacing the previous tally value of `49` with `1`) but is then immediately forced to relinquish the processor.

(4) Process B resumes long enough to load `1` (the current value of `tally`) into its register, but then it too is forced to give up the processor (note that this was B's final load).

(5) Process A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of `tally` to 50.

(6) Process B is reactivated with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.

Some thought will reveal that a value lower than 2 cannot occur. Thus, the proper range of final values is $2 \leq$ `tally` ≤ 100.

(b) For the generalised case of `N` processes, the range of final values is $2 \leq$ `tally` $\leq$ `(N x 50)`, since it is possible for all other processes to be initially scheduled and run to completion in Step (5) in part (a) before Process B would finally destroy their work by finishing last.

3.2.3 Task 3

The two sets of definitions are *equivalent*. For the definitions given in the lecture notes, when the value of the semaphore is negative, its value tells you how many processes are waiting. With the definitions given in this task, you do not have that information readily available. However, the two versions function the same.

4 Deadlock and Starvation

4.1 Review Questions

Question 1

The four conditions that create *deadlocks*:

(a) **Mutual exclusion:** Only one process may use a resource at a time.

(b) **Hold and wait:** A process may hold allocated resources while awaiting assignment of others.

(c) **No preemption:** No resource can be forcibly removed from a process holding it.

(d) **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

Question 2

Mutual exclusion should not be disallowed in order to prevent deadlocks. The reason is that if access to a resource requires mutual exclusion (such as to prevent race conditions), then mutual exclusion must be supported by the operating system.

Question 3

The *hold-and-wait* condition be prevented by requiring that a process requests all of its required resources at one time, and blocking the process until all requests can be granted simultaneously.

Question 4

First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.

Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

Question 5

(a) **Deadlock prevention:** Resource requests are constrained to prevent at least one of the four conditions of deadlock; this is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait.

(b) **Deadlock avoidance:** The three necessary conditions are allowed, but makes judicious choices to assure that the deadlock point is never reached.

(c) **Deadlock detection:** Requested resources are granted to processes whenever possible; periodically, the operating system performs an algorithm that allows it to detect the circular wait condition.

4.2 Problem-Solving Tasks

4.2.1 Task 1

Four conditions of deadlock apply to the given figure:

(a) **Mutual exclusion:** Only one car can occupy a given quadrant of the intersection at a time.

(b) **Hold and wait:** No car ever backs up; each car in the intersection waits until the quadrant in front of it is available.

(c) **No preemption:** No car is allowed to force another car out of its way.

(d) **Circular wait:** Each car is waiting for a quadrant of the intersection occupied by another car.

4.2.2 Task 2

(a) **Prevention:**

- Hold-and-wait approach: require that a car request both quadrants that it needs and blocking the car until both quadrants can be granted.
- No preemption approach: releasing an assigned quadrant is problematic, because this means backing up, which may not be possible if there is another car behind this car.
- Circular-wait approach: assign a linear ordering to the quadrants.

(b) **Avoidance:** The algorithms discussed in the lectures apply to this problem. Essentially, deadlock is avoided by not granting requests that might lead to deadlock.

(c) **Detection:** The problem here again is one to backup.

4.2.3 Task 3

(a) Creating the process would result in the safe state:

Process	Max	Hold	Claim	Free
1	70	45	25	25
2	60	40	20	
3	60	15	45	
4	60	25	35	

There is sufficient free memory to guarantee the termination of either **P1** or **P2**. After that, the remaining three jobs can be completed in any order.

(b) Creating the process would result in the trivially unsafe state:

Process	Max	Hold	Claim	Free
1	70	45	25	15
2	60	40	20	
3	60	15	45	
4	60	35	25	

4.2.4 Task 4

There is a deadlock if the scheduler goes — for example, P0→P1→P2→P0→P1→P2 (line by line). Each of the six resources will then be held by one process, so all three processes are now blocked at their third line inside the loop, waiting for a resource that another process holds.

This is illustrated by the circular wait (indicated by the arrows) in the resource allocation graph below: P0→C→P2→D→P1→B→P0.