ID1206: Review Questions 2

Alexander Lundqvist

Fall 2022

1 Question 1

What is the meaning of the term busy waiting? Can busy waiting be avoided altogether? Explain your answer.

1.1 Answer

Busy waiting refers a technique where a process is continuously using the CPU for checking if a certain condition has been met like waiting for a certain keyboard input, a lock or some other shared resource to be available.

When it comes to avoiding busy waiting, the book describes one way which involves a process suspending itself and adding it to a list of waiting processes and then getting waked up when the lock/resource has been made available. This however is not a complete avoidance of busy waiting but rather moving it from the entry section to the critical section of the wait and signal commands, which leads to very short duration of busy waiting.

2 Question 2

Show that, if the **wait()** and **signal()** semaphore operations are not executed atomically, then mutual exclusion may be violated.

2.1 Answer

A semaphore is an integer variable shared between processes and used to solve critical section problems. It supports two operations that either increment(signal) or decrements(wait) the value of the semaphore. The value stands for the amount of available resources. The operations must be performed atomically, which means that when a process modifies the value of the semaphore, no other process can modify it at the same time.

Suppose we have two processes calling the wait operation at the same time and the semaphore has a value of 1. If it is not done atomically, then the first process will decrement the value and enter its critical state. At the same time, the other process will also see the value of 1 and not the decremented value (if the value is ≤ 0, then the process would wait), it will also decrement the value and enter its critical process and this would violate mutual exclusion.

3 Question 3

Consider the following snapshot of a system, and answer the following questions using the banker's algorithm:

	Allocation	Max	Available
	$ABCD$	$ABCD$	$ABCD$
T_0	0012	0012	1520
T_1	1000	1750	-
T_2	1354	2356	-
T_3	0632	0652	-
T_4	0014	0656	-

a. What is the content of the matrix **Need**?

b. Is the system in a safe state?

c. If a request from thread T_1 arrives for (0, 4, 2, 0), can the request be granted immediately?

3.1 Answer

a. The content of the matrix **Need**.

	Need
	$ABCD$
T_0	0000
T_1	0750
T_2	1002
T_3	0020
T_4	0642

b. We calculate this by applying the safety algorithm. Let $Work = \{1, 5, 2, 0\}$ and $Finish = \{0, 0, 0, 0, 0\}$. We compare the $Work$ array to the $Need$ matrix from the previous task to find those threads that fulfill the the criteria $Finish[i] == 0$ and $Need_i \leq Work$. We can see that either T_0 or T_3 follows this criteria. We start with T_3 increase the $Work$ array with the released resources by the formula $Work = Work + Allocation_3$ and set $Finish[3] = 1$. We then repeat the process for the rest of the threads. This will turn $Finish = \{1, 1, 1, 1, 1\}$ and thus, the system is in a safe state.

c. First, we have the new request for T_1 being $Request_1 = \{0, 4, 2, 0\}$. We start of by checking the $Need$ matrix if $Request_1 \leq Need_1 \rightarrow \{0, 4, 2, 0\} \leq \{0, 7, 5, 0\}$ which it is. We then need to check if $Request_1 \leq Available$ and we can see that $\{0, 4, 2, 0\} \leq \{1, 5, 2, 0\}$. We can now proceed to modify the tables to let the system pretend it has allocated the requested resources by the formula:

1. $Available = Available - Request_i$
2. $Allocation_i = Allocation_i + Request_i$
3. $Need_i = Need_i - Request_i$

The resulting table will look like this. Applying the safety algorithm

	Allocation	**Max**	**Available**	**Need**
	$ABCD$	$ABCD$	$ABCD$	$ABCD$
T_0	0012	0012	1100	0000
T_1	1420	1750	-	0330
T_2	1354	2356	-	1002
T_3	0632	0652	-	0020
T_4	0014	0656	-	0642

like in b showed that the system is safe and thus, we can grant the request.

4 Question 4

Consider the following resource-allocation policy.

- Requests for and releases of resources are allowed at any time.
- If a request for resources cannot be satisfied because the resources are not available, then we check any threads that are blocked waiting for resources.
- If a blocked thread has the desired resources, then these resources are taken away from it and are given to the requesting thread.
- The vector of resources for which the blocked thread is waiting is increased to include the resources that were taken away.

For example, a system has three resource types, and the vector *Available* is initialized to (4, 2, 2).

- If thread T_0 asks for (2, 2, 1), it gets them.
- If T_1 asks for (1, 0, 1), it gets them.
- Then, if T_0 asks for (0, 0, 1), it is blocked (resource not available).
- If T_2 now asks for (2, 0, 0), it gets the available one (1, 0, 0), as well as one that was allocated to T_0 (since T_0 is blocked).
- T_0's *Allocation* vector goes down to (1, 2, 1), and its *Need* vector goes up to (1, 0, 1).

a. Can deadlock occur? If you answer "yes", give an example. If you answer "no", specify which necessary condition cannot occur.

b. Can indefinite blocking occur? Explain your answer.

4.1 Answer

a. One of the required conditions that a system must have for a deadlock to arise is the lack of preemption. This means that a thread won't relinquish its allocated resources until it has completed its task. The third statement in the description states that if a thread is blocked and has resources that another thread is requesting, then the resources will be allocated to the requesting thread. Hence, there is preemption and deadlock cannot occur.

b. As the resource-allocation policy stated in the task always allows processes to acquire the resources of blocked processes we will have a predicament. If T_2 gets continuously preempted then there will always be processes that will be "starved" of required resources, hence there is a risk of indefinite blocking in the current system.