**Operating Systems Design** :

Tutorial 06 - Semaphores

*By Tom-Eliott Herfray (Student n°2999664)*

***Question 1****. What are the differences between a process and a thread ?*

> A process is an active entity, which requires a set of resources, including a processor and special registers, to perform its function. We can also see this as a single instance of an executable program. Process has one or more thread, which is a smaller unit within a process, which can be scheduled and executed. All threads in a process share memory space and other resources.

***Question 2.*** *What is a critical section in relation to processes ?*

> The critical section is a part of a program that must complete execution before other processes can have access to the resources being used. This is a protected section, where the shared resource is accessed are protected and where it cannot be executed by more than one process at a time.

***Question 3.*** *What is a “busy waiting” and how is it solved ?*

> We have a “busy waiting” when a process checks repeatedly again and again for a condition. The process is waiting for the condition and busy checking for it. This usually uses CPU resources and one of the solutions to solve that is to use the WAIT & SIGNAL locking mechanism, which was designed from Test-and-set and where we have 2 new mutually exclusive operations : *WAIT* and *SIGNAL*. *WAIT* is activated when process encounters a busy condition code and *SIGNAL* when a process exits critical region and the condition code is to “free”.

***Question 4.*** *Name and explain one method in which synchronisation can be achieved between processes.*

> One of the methods in which synchronisation can be achieved between processes is using the Test-and-Set locking mechanism. Test-and-Set is an instruction executed in a single machine cycle to see if the key is available and, if it is, sets it to unavailable. It has the advantage of being easy to implement and it works well for a small number of processes. The key we are going to check for this method is a single bit in storage location that contain a 0 (which means it's free) or a 1 (which means it's busy). If a process tests the condition code and if there is no other process in this region, then the process is allowed to proceed and the condition code is changed from 0 to 1. When the process exits, the condition code is reset to 0 to allow other process to enter. We can have two problems with this method : Starvation (when many processes are waiting to enter a critical section) and busy waiting (see the answer to the previous question).

***Question 5.*** *Describe the operation of Semaphores.*

> Semaphore uses two operations to operate, where “s” is a semaphore variable : *P(s)* and *V(s).* They come from the Dutch verbs “proberen” and “verhogen” (which mean “test” and “increment”). *P(s)* holds the current process until a resource is available, which will be immediately allocated to the current process. *V(s)* is the inverse operation, it simply makes a resource available again after the process has finished using it. *P(s)* and *V(s)* operations enforce the concept of mutual exclusion.

*- P(s) : if s > 0, then s: = s – 1 (test, fetch, decrement and store sequence)*

*- V(s) : s: = s + 1 (fetch, increment and store sequence)*

***Question 6.*** *Explain clearly two suggested solutions to the Dining Philosophers problem.*

> There are several solutions to solve the Dining Philosophers problem.

One of these solutions is the hierarchy of resources (proposed by Dijkstra himself) : To solve this problem, we assign a partial order to the forks and assume that forks will be requested in order. The forks are numbered from 1 to the number of philosophers and a philosopher will always take first a fork whose number is the lowest (its first fork) then the one with the highest number (its 2nd fork). The order to put the fork is not important, but for example in the case where there are 5 philosophers at table, if 4 philosophers simultaneously take their first fork, only one philosopher will be able to feed and the last philosopher cannot have a set of cover. This solution is interesting to avoid deadlocks but it is not optimal when we do not know the entire list of resources in advance. Another interesting solution will be the use of semaphore by cover : We use a semaphore to represent a fork. A range can be picked up by performing a wait operation on the semaphore and released by executing a signal semaphore. When philosopher wants to eat, he will wait for the fork to his left and pick it up. Then he will wait until the right fork is available, then take it too. After eating, he puts both forks on the table. If all philosophers want to eat at the same time, we come across a deadlock because they wait indefinitely for a fork. We can manage this with an even number of philosophers or make sure that a philosopher can take his two forks only if they are fully available.