Rupanshoo Saxena | 2019096 | CSE

Operating Systems - Assignment 4
Date: 4th December 2020

Approach behind implementing the counting semaphore using mutex:

The **my_semaphore** structure consists of an integer **counter** to store the value of semaphore and **2 mutex** - one to lock threads and one to lock the counter of the semaphore. All these together make synchronisation possible without any deadlocks.

mutexForThreadBlock ensures that at a time only one thread can access the semaphore's counter and it either blocks or makes the other threads wait in case no more resources are available.
mutexCounter keeps check of the resources i.e. the counter and stops any other thread from changing it when it is locked.

The following **functions** have been implemented for my_semaphore:

1. **void initialise(my_semaphore *, int)** → this function initializes the semaphore structure with the integer provided as a parameter. In case initialization fails, it returns an error. Here both mutex are initialized and the value of the semaphore is set.
2. **void wait_blocking(my_semaphore *)** → this function blocks the thread calling wait in case the value of the semaphore cntr <= 0 i.e. no more resources are available. The thread remains blocked till the time resources become available and the thread can access it.
3. **void wait_non_blocking(my_semaphore *)** → this function doesn't block the thread calling wait in case the semaphore cntr <= 0. The thread can try again and again till the resource becomes available but it is not blocked. pthread_mutex_trylock has been used here.
4. **void signal(my_semaphore *)** → it unblocks the thread which is blocked by wait_blocking and also restores the resources i.e. the semaphore counter.
5. **void printValue(my_semaphore *)** → it prints the current value of the semaphore given as parameter to the function. It has been used only for debugging and checking.
6. **void kill_Sem(my_semaphore *)** → this function is used to destroy the mutex of the semaphore and thus the semaphore itself.

Approach for the Advanced Dining Philosophers Problem:

The **approach** that I have followed is that till the time a philosopher doesn't have all the equipments he requires to eat, he won't eat and if he isn't able to acquire a certain resource needed to eat, then all the previous resources that he had acquired will be released, i.e. for example, if Philosopher 1 acquires chopsticks 1 and 2 but isn't able to acquire the first bowl, then he will have to give up the chopsticks too and chopstick 1 and 2 will become available for some other philosopher to use.

Input : the number of philosophers and chopsticks (should be >1 else an error message appears and program exits.)

Chopsticks are represented by **binary semaphores** which are implemented using the my_semaphore structure and **bowls** are represented by **counting semaphores** which is also implemented using the my_semaphore structure. The **philosophers** are taken as **threads** and have IDs assigned to them.

malloc is used to assign memory to the chopsticks and bowls semaphores and to the philosophers threads and their integer IDs.

void* philosopher(void * phi_ID) function is the one that implements the complete cycle of thinking-->waiting-->eating-->thinking of the philosophers that keeps going on in an infinite loop.
The philosopher acquires the left chopstick by the blocking wait function and gets the rest of the components by the non-blocking wait function. In case any of the resources aren't available, the earlier resources acquired by the philosopher are returned and thus **deadlock** doesn't happen. After the philosopher is done eating, all the resources acquired by him are released.

Considering a situation where we have 3 philosophers and 3 chopsticks (let's ignore the 2 bowls for the time being). If all philosophers pick up the chopstick on their left by blocking wait, now if philosopher 1 tries to acquire the chopstick on his right, he won't be able to because philosopher 2 has already acquired it and so will also release his left chopstick which can be later acquired by philosopher 3. In this way, such a **deadlock will never happen.**

When the program terminates, all the allocated space is freed and the semaphores are killed.