Jatin Kumar Sharma
Roll No: 2020563
Operating System
Assignment-5

Semaphore

Semaphores are used to ensure mutual exclusion of data and functions when multiple threads/processes try to access a resource simultaneously. For the given modifications of the philosopher's problem, I have defined my own counting semaphore using pthread_mutex_t and pthread_cond_t to solve the given variations of the dining philosopher's problem.

.

Implementation of counting Semaphore:

The semaphore struct my_semaphore consists of three members from the pthread library as follows:

1) **ssize_t count**:- this is the size of the semaphore to ensure that the semaphore object won't be accessible beyond what is supposed to exist.
2) **pthread_mutex_t mutex**: This allows us to lock/unlock the semaphore object to ensure that only one thread could access the struct at a single instant of time.
3) **pthread_cond_t cond**: Specifies the condition that may or may not be fulfilled, allowing threads to wait.

Semaphore functions:

1) **init()**: It takes the semaphore struct as an argument and assigns the Pthread_mutex_t and the pthread_cond_t so that my_semaphore struct is ready to be used.
2) **wait()**: This function works like the blocking version of the sem_wait(). It assigns the calling thread, the access to the object protected by the semaphore.
3) **signal()**: It is similar to sem_post(), it frees the semaphore from the calling thread and also notifies other threads waiting on the pthread_cond_t.

All three parts contain the main function and a function that is catering for the philosophers to have a deadlock-free dinner.

::

Part-1

As per the problem statement of part 1, there are 5 forks and 5 philosophers and they need to fork to eat. If all the philosopher takes a fork along with them then in this scenario none of the philosophers would be able to eat and there would be a situation of deadlock.

Avoiding deadlock:

To avoid a deadlock situation one can use ordering, for example here deadlock is arising if all of the 5 philosophers take up a fork along with them to avoid them we can ensure that if a philosopher is taking up a fork then he must have another fork as well that is done here by giving only even forks to the philosophers first and subsequently another fork. Which ensures that a deadlock won't arise.

Part-2

since we have 4 bowls and only a single fork is needed for this part . So there won't be any deadlock in this modification because each philosopher only needs one fork and 1 bowl to eat , 1 fork will always be available to each of the philosophers and the non acquired bowls can we take one by one and placed back on the table once a philosopher is done with eating. The logic of this is implemented in diner function. In the main function an array of threads of the philosophers and globally declare an array of bowls initialized with -1 and semaphore of bowl (counting) and forks of type semaphore (binary). In the main program we initialize all the semaphores using init function and the bowl semaphores have the value of count as 4.

Part-3

In the third program there will be a deadlock similar as of the first modification and hence it can be dealt by the similar way as we have done above for the forks and in the philosopher's only eats when he gets the unacquired bowl which is done in second modification. Therefore the modification is kind of a fusion of both the previous two modifications and hence the dinner function will be of the combination of the previous two.