ASSIGNMENT NO:

4 (B)

AIM: To implement Dining Philosopher's problem using 'C' in Linux

OBJECTIVE: Implement the deadlock-free solution to Dining Philosophers problem to illustrate the problem of deadlock and/or starvation that can occur when many synchronized threads are competing for limited resources..

THEORY:

What is Deadlock?

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause. Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no interrupts condition is needed to prevent an otherwise deadlocked process from being awake**.**

Conditions for Deadlock

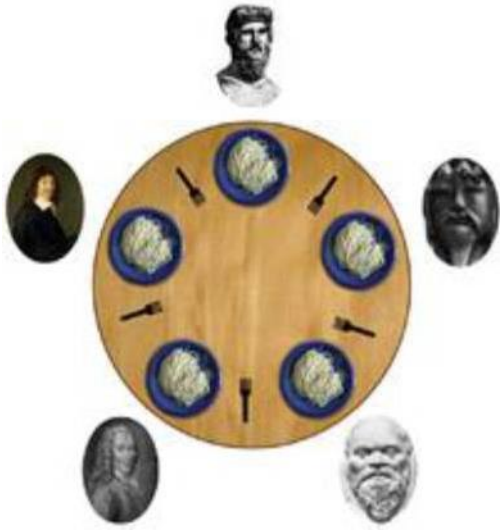
Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

In this problem, there are 5 philosophers present who spend their life in eating and thinking. Philosophers share a common circular table surrounded by 5 chairs, each belonging to 1 philosopher .In the center of the table. A bowl of slippery food(Noodles or rice) is present and across each philosopher a pair of chopstick is present. When a philosopher thinks, he does not interact with his colleague.

Whenever a philosopher gets hungry, he tries to pick up 2 chopsticks that are close to him. Philosopher may pick up one chopstick at a time. He cannot pick up a chopstick that is already in the hand of neighbor. When a hungry philosopher has both chopsticks at the same time, he start eating without releasing his chopstick and starts thinking again .The problem could be raised when all the philosopher try to keep the chopstick at the same time.

This may lead to deadlock situations. To synchronize all philosophers, semaphore chopsticks [5] are used as a variable where all the elements are first initialized to 1.The structure of philosophers is shown below;



SEMAPHORE CHOPSTICK [5]
Think: After eating

Eat: Hungry

Do

{

wait(chopstick[i]);

wait(chopstick[(i+1)%5]);

---

eat

signal(chopstick[i]);

signal(chopstick[(i+1)%5]);

---

think

}while(1);

1) The following three functions lock and unlock a mutex:

# include<pthread.h>
intpthread_mutex_lock(pthread_mutex_t * mptr);
intpthread_mutex_trylock(pthread_mutex_t * mptr);
intpthread_mutex_unlock(pthread_mutex_t * mptr);

2) The mutex or condition variable is initialized or destroyed with the following functions.

# include<pthread.h>
intpthread_mutex_init(pthread_mutex_t * mptr,constpthread_mutexattr_t * attr);
intpthread_mutex_destroy(pthread_mutex_t * mptr);

3) Functions used with their syntax :
Routines:

pthread_create (thread,attr,start_routine,arg) pthread_exit (status)

Creating Threads:

Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

pthread_create arguments:

a. thread: An opaque, unique identifier for the new thread returned by the subroutine.

b. attr: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.

c. start_routine: the C routine that the thread will execute once it is created.

d. arg: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.

When a program is started by exec, a single thread is created , called the initial thread or main thread . Additional threads are by pthread_create.

# include<pthread.h>
int pthread_create(pthread_t * tid,constpthread_attr_t * attr,void * (* func)void *),void* arg);

Thread Joining:

- "Joining" is one way to accomplish synchronization between threads.
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().

- A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.

- Two other synchronization methods, mutexes and condition variables, will be discussed later.

CONCLUSION:

Thus, we have implemented dining philosopher's problem using 'C' in Linux.

FAQ:

1. What is dead lock?
2. What are the necessary and sufficient conditions to occur deadlock?
3. What is deadlock avoidance and deadlock prevention techniques?