Experiment Number: 13

TITLE: Dining Philosophers Problem (Using Mutex)

OBJECTIVE:
1. To understand the use of POSIX threads and mutex in UNIX.
2. To study dining philosophers problem of operating system.

Theory:

The Dining Philosophers Problem

The dining philosophers problem was introduced by Dijkstra. Five philosophers live in a house, where a table is laid for them. The life of each philosopher consists principally of thinking and eating, and through years of thought, all of the philosophers had agreed that the only food that contributed to their thinking efforts was spaghetti. Due to a lack of manual skill, each philosopher requires two forks to eat spaghetti.

The eating arrangements are simple: a round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti.

The problem: devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock and starvation. This problem may not seem important or relevant in itself. However, it does illustrate basic problems in deadlock and starvation. Furthermore, attempts to develop solutions reveal many of the difficulties in concurrent programming. In addition, the dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources, which may occur when an application includes concurrent threads of execution. Accordingly, this problem is a standard test case for evaluating approaches to synchronization.
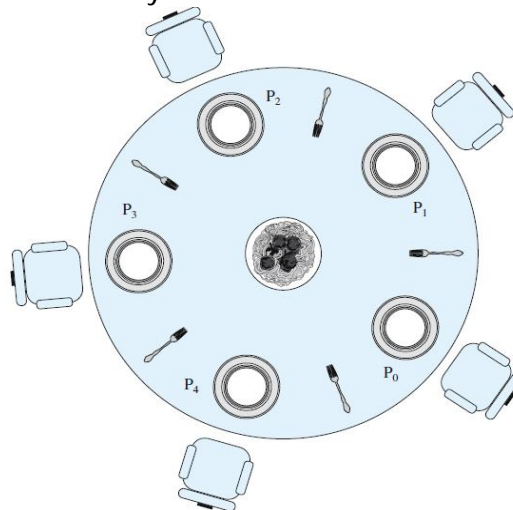
Dining Arrangement for Philosophers

Solution Using Semaphores

Each philosopher picks up first the fork on the left and then the fork on the right. After the philosopher is finished eating, the two forks are replaced on the table.

```
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true)
    {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin   (philosopher   (0),   philosopher   (1),
    philosopher (2), philosopher (3), philosopher (4));
}
```

Threads

Multiple strands of execution in a single program are called threads. A more precise definition is that a thread is a sequence of control within a process. Like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution.

POSIX Thread in Linux

Including the file pthread.h provides us with other definitions and prototypes that we will need in our code, much like stdio.h for standard input and output routines.

```
#include <pthread.h>
int  pthread_create(pthread_t  *thread,  pthread_attr_t  *attr,
void*(*start_routine)(void *), void *arg);
```

This function is used to create the thread. The first argument is a pointer to pthread_t. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread. The next argument sets the thread attributes. We do not usually need any special attributes, and we can simply pass NULL as this

argument. The final two arguments tell the thread the function that it is to start executing and the arguments that are to be passed to this function.

```
void *(*start_routine)(void *)
```

We must pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. Thus, we can pass any type of single argument and return a pointer to any type. Using fork causes execution to continue in the same location with a different return code, whereas using a new thread explicitly provides a pointer to a function where the new thread should start executing. The return value is 0 for success or an error number if anything goes wrong.

When a thread terminates, it calls the pthread_exit function, much as a process calls exit when it terminates. This function terminates the calling thread, returning a pointer to an object. Never use it to return a pointer to a local variable, because the variable will cease to exist when the thread does so, causing a serious bug. pthread_exit is declared as follows:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_join is the thread equivalent of wait that processes use to collect child processes. This function is declared as follows:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

The first parameter is the thread for which to wait, the identifier that pthread_create filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.

Unix Mutex Facilities:

The way of synchronizing access in multithreaded programs is with mutexes (short for mutual exclusions), which act by allowing the programmer to "lock" an object so that only one thread can access it. To control access to a critical section of code us lock a mutex before entering the code section and then unlock it when we have finished.

The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const
                       pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

As usual, 0 is returned for success, and on failure an error code is returned. As with semaphores, they all take a pointer to a previously declared object, in this case a pthread_mutex_t. The extra attribute parameter pthread_mutex_init allows us to provide attributes for the mutex, which control its behavior. The attribute type by default is "fast." This has the slight drawback that, if our program tries to call pthread_mutex_lock on a mutex that it has already locked, the program will block. Because the thread that holds the lock is the one that is now blocked, the mutex can never be unlocked and the program is deadlocked. It is possible to alter the attributes of the mutex so that it either checks for this and returns an error or acts recursively and allows multiple locks by the same thread if there are the same number of unlocks afterward.

References:

1. "Beginning Linux Programming" by Neil Mathew and Richard Stones, Wrox Publications.
2. "Operating System Internals and Design Implementation" by William Stallings, Pearson Education.

Tushar B Kute
(Subject Teacher)