Experiment Number: 13

TITLE: Dining Philosophers Problem (Using Semaphore)

OBJECTIVE:

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

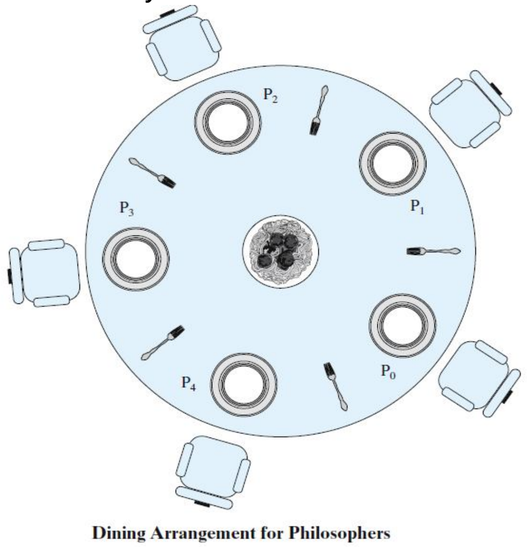
Theory:

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.

Linux Semaphore Facilities (Binary Semaphore)

A semaphore is created with the sem_init function, which is declared as follows:

```
    #include <semaphore.h>
    int sem_init(sem_t *sem, int pshared, unsigned int value);
```

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for *pshared* will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to sem_init.The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic “test and set” ability in a single function is what makes semaphores so valuable.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

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)