OPERATING SYSTEMS

Lab-10: Dining Philosopher Problem and Readers Writers Problem

Name – Vinay Santosh Menon
Registration Number - 20BAI1103

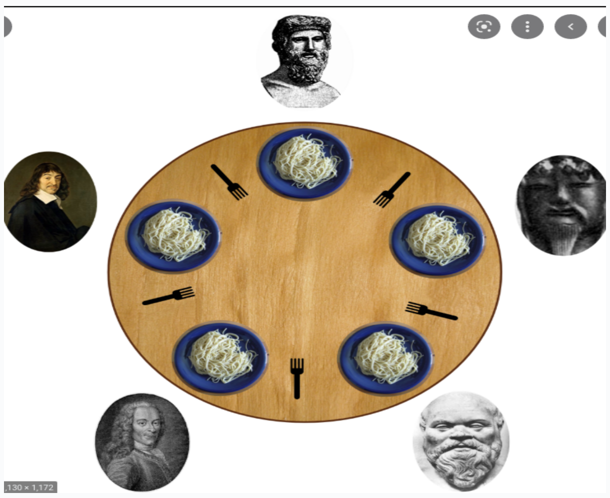
TASK:

To understand how the Dining Philosopher problem and Readers Writers problem works, and implementing it in C code.

Dining Philosopher problem:

According to the Dining Philosopher Problem, assume there are K philosophers seated around a circular table, each with one chopstick between them. This means that a philosopher can eat only if he/she can pick up both the chopsticks next to him/her. One of the adjacent followers may take up one of the chopsticks, but not both.

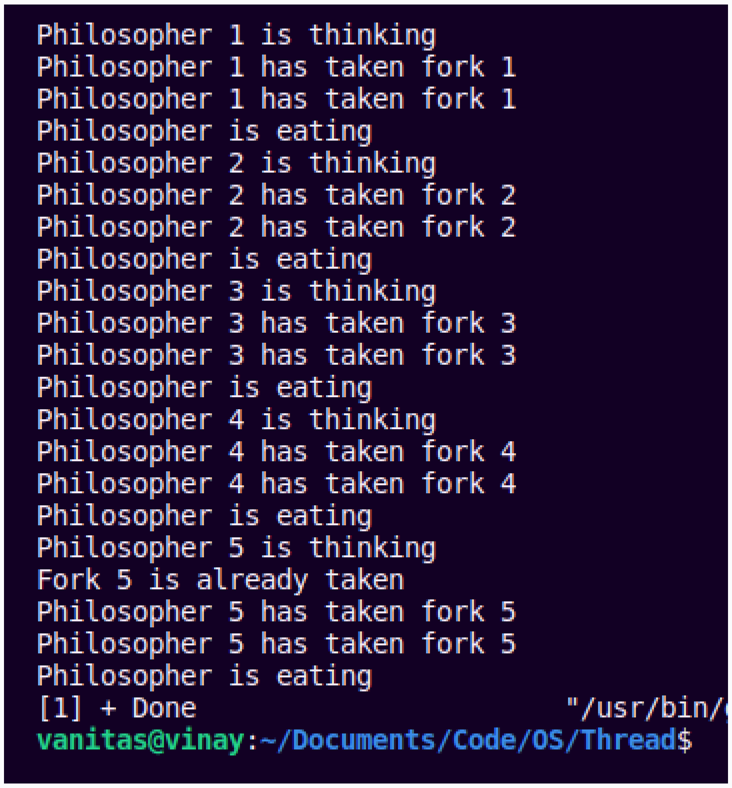
For example, let's consider P0, P1, P2, P3, and P4 as the philosophers or processes and C0, C1, C2, C3, and C4 as the 5 chopsticks or resources between each philosopher. Now if P0 wants to eat, both resources/chopstick C0 and C1 must be free, which would leave in P1 and P4 void of the resource and the process wouldn't be executed, which indicates there are limited resources(C0,C1..) for multiple processes(P0, P1..), and this problem is known as the Dining Philosopher Problem.



Code:

```c
#include <stdio.h>

#include <pthread.h>

int N = 5;

int fork_phil[] = {1, 1, 1, 1, 1};

void take_fork(int input)

{

   if(fork_phil[input] <= 0){

       printf("Fork %d is already taken\n", input);

   }

   else{

       fork_phil[input] = 0;

   }

}



void put_fork(int input)

{
   fork_phil[input] += 1;
}


void *philo(int *input) {
   printf("Philosopher %d is thinking\n", input);
   int i = (int *)input;
   take_fork(i);
   printf("Philosopher %d has taken fork_phil %d\n", input,
input);
   take_fork(((i + 1) % N));
   printf("Philosopher %d has taken fork_phil %d\n", input,
input);
   printf("Philosopher is eating\n");
   put_fork(i);
   put_fork(((i + 1)%N));
}


int main(){
   pthread_t p1, p2, p3, p4, p5;

   pthread_create (&p1, NULL, philo, 1);

   pthread_create (&p2, NULL, philo, 2);

   pthread_create (&p3, NULL, philo, 3);

   pthread_create (&p4, NULL, philo, 4);

   pthread_create (&p5, NULL, philo, 5);

   pthread_join (p1, NULL);

   pthread_join (p2, NULL);

   pthread_join (p3, NULL);

   pthread_join (p4, NULL);

   pthread_join (p5, NULL);

}
```

Output:



```
Philosopher 1 is thinking
Philosopher 1 has taken fork 1
Philosopher 1 has taken fork 1
Philosopher is eating
Philosopher 2 is thinking
Philosopher 2 has taken fork 2
Philosopher 2 has taken fork 2
Philosopher is eating
Philosopher 3 is thinking
Philosopher 3 has taken fork 3
Philosopher 3 has taken fork 3
Philosopher is eating
Philosopher 4 is thinking
Philosopher 4 has taken fork 4
Philosopher 4 has taken fork 4
Philosopher is eating
Philosopher 5 is thinking
Fork 5 is already taken
Philosopher 5 has taken fork 5
Philosopher 5 has taken fork 5
Philosopher is eating
[1] + Done                       "/usr/bin/
vanitas@vinay:~/Documents/Code/OS/Thread$
```

Reader Writer Problem:

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers-writers problem.

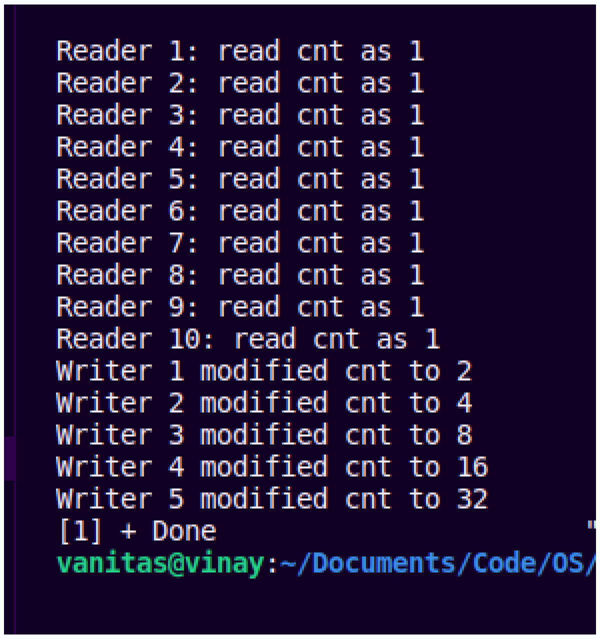
The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

Code:

```
#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>
```

```
sem_t wrt;

pthread_mutex_t mutex;

int cnt = 1;

int numreader = 0;



void *writer(void *wno)

{

   sem_wait(&wrt);

   cnt = cnt*2;

   printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);

   sem_post(&wrt);



}

void *reader(void *rno)

{

   pthread_mutex_lock(&mutex);

   numreader++;
   if(numreader == 1) {

       sem_wait(&wrt);

   }

   pthread_mutex_unlock(&mutex);

   printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

   pthread_mutex_lock(&mutex);

   numreader--;

   if(numreader == 0)

   {

       sem_post(&wrt);

   }

   pthread_mutex_unlock(&mutex);

}



int main()

{
   pthread_t read[10],write[5];

   pthread_mutex_init(&mutex, NULL);

   sem_init(&wrt,0,1);



   int a[10] = {1,2,3,4,5,6,7,8,9,10};



   for(int i = 0; i < 10; i++)

   {

       pthread_create(&read[i], NULL, (void *)reader, (void
*)&a[i]);

   }

   for(int i = 0; i < 5; i++)

   {

       pthread_create(&write[i], NULL, (void *)writer, (void
*)&a[i]);

   }



   for(int i = 0; i < 10; i++)
   {

       pthread_join(read[i], NULL);

   }

   for(int i = 0; i < 5; i++)

   {

       pthread_join(write[i], NULL);

   }

   pthread_mutex_destroy(&mutex);

   sem_destroy(&wrt);



   return 0;



}
```

Output:



```
Reader 1: read cnt as 1
Reader 2: read cnt as 1
Reader 3: read cnt as 1
Reader 4: read cnt as 1
Reader 5: read cnt as 1
Reader 6: read cnt as 1
Reader 7: read cnt as 1
Reader 8: read cnt as 1
Reader 9: read cnt as 1
Reader 10: read cnt as 1
Writer 1 modified cnt to 2
Writer 2 modified cnt to 4
Writer 3 modified cnt to 8
Writer 4 modified cnt to 16
Writer 5 modified cnt to 32
[1] + Done                         "
vanitas@vinay:~/Documents/Code/OS/
```