OS Lab-8 Assignment

Anirudh Chimpidi SE20UCSE019

Dhanush Bommavaram SE20UCSE039

Naga Tharun Makkena SE20UCSE105

Sri Harsha Vandanapu SE20UCSE184

Q1. Dining Philosophers Problem

Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

// defining the number of philosophers to 5
#define N 5
// variables used for knowing the state of the philosopher
#define THINKING 2
#define HUNGRY 1
#define EATING 0
// used for checking the availablity of forks on table by
checking stae of adjacent philosophers

#define LEFT (num_of_philosopher + 4) % N
#define RIGHT (num_of_philosopher + 1) % N

// stores the state of philosopher
int state[N];
// ids of the philosophers
int phil[N] = {0, 1, 2, 3, 4};

// binary semaphore for preventing collision for multiple
```

```
operations
sem_t mutex;
// binary semaphore used for each philosopher
sem_t S[N];

void test(int num_of_philosopher){
      // if philospher state is hungry and the left and
right forks are available by checking state of adjacent
philosophers
      if(state[num_of_philosopher]==HUNGRY &&
state[LEFT]!=EATING && state[RIGHT]!=EATING){
            // update state to eating
            state[num_of_philosopher] = EATING;
            sleep(2);
            // printing the state of philosopher
            printf("Philosopher %d takes fork %d and %d\n",
num_of_philosopher+1, LEFT+1, num_of_philosopher+1);
            printf("Philosopher %d is Eating\n",
num_of_philosopher + 1);

            /* sem_post(&S[num_of_philosopher]) has no
effect
            during takefork
            used to wake up hungry philosophers
            during putfork */
            sem_post(&S[num_of_philosopher]);
      }
}

void take_fork(int num_of_philosopher){
      // mutex is reduced to prevent processes interfering
with each others shared data
      sem_wait(&mutex);

      // change the philosopher state from thinking to
hungry
      state[num_of_philosopher] = HUNGRY;
      printf("Philosopher %d is Hungry\n",
num_of_philosopher + 1);

      // if the neighour is eating then wait till he changes
his state
      test(num_of_philosopher);
      // releases the mutex after completion of the process
      sem_post(&mutex);
```

```
        // if unable to eat then wait till s changes
        sem_wait(&S[num_of_philosopher]);
        sleep(1);
}

void put_fork(int num_of_philosopher){
        // mutex is reduced to prevent processes interfering
with each others shared data
        sem_wait(&mutex);

        // change the state of philsopher to thinking
        state[num_of_philosopher] = THINKING;
        printf("Philosopher %d putting fork %d and %d
down\n",num_of_philosopher + 1, LEFT + 1, num_of_philosopher
+ 1);
        printf("Philosopher %d is thinking\n",
num_of_philosopher + 1);

        // allows the adjacent philosophers to check if they
have the forks or resources to change their state
        test(LEFT);
        test(RIGHT);
        // releases the mutex after completion of the process
        sem_post(&mutex);
}

void* philosopher(void* num){
        while(1){
                // i is the philosopher id
                int* i = num;
                sleep(1);
                // philosopher takes the  fork if available else
wait
                take_fork(*i);
                sleep(0);
                // philospher puts the fork after eating
                put_fork(*i);
        }
}

int main(){
        int i;
        pthread_t thread_id[N];

        // initialize the semaphores
        sem_init(&mutex, 0, 1);
```

```
	for(i=0; i<N; i++)
	{
		sem_init(&S[i], 0, 0);
	}
	// creating philosopher processes
	for(i=0; i<N; i++){
		pthread_create(&thread_id[i], NULL,philosopher,
&phil[i]);
		printf("Philosopher %d is thinking\n", i + 1);
	}

	for(i=0; i<N; i++){
		pthread_join(thread_id[i], NULL);
	}
}
```

The problem: The Dining Philosopher Problem states that some philosophers are seated around a circular table with one fork/chopstick between every two philosophers. A philosopher may eat if he can pick up the two forks/chopsticks adjacent to him. One fork/chopstick may be picked up by any one of its adjacent followers but not by both of them at the same time.

Solution using semaphores:

We take n semaphores, n is the number of forks/chopsticks on the table which will be equal to the number of the Philosophers.

A philosopher will try to grab a fork/chopstick by executing a wait operation on the semaphore associated with that particular fork/chopstick.

A philosopher will try to place the fork/chopstick he is holding back on the table executing a signal operation on the semaphore associated with that particular fork/chopstick.

We have 3 states for a philosopher (thinking, hungry, eating). Depending on these states we decide whether a philosopher should wait before picking up a fork/chopstick or releasing a fork/chopstick.

We use a mutex semaphore to prevent processes interfering with each other’s shared data (critical section).

Wait function (take_fork):

We first check the state of the current philosopher (whether he is hungry).

If the philosopher is hungry, we check whether the 2 philosophers adjacent to him are eating (hold fork/chopstick), if not them we assign the adjacent forks/chopsticks to the philosopher.

Signal function (put_fork):

We wait until the current philosopher is no longer eating (in Eating state).

After that we release the semaphores of the two forks/chopsticks adjacent to him/her.

```
tharun@DESKTOP-HD8TAHP:/mnt/c/Users/nagat/OneDrive/Desktop/Semester 5/Operating Systems/Lab/Lab_8$ gcc -pthread temp_philosopher.c
tharun@DESKTOP-HD8TAHP:/mnt/c/Users/nagat/OneDrive/Desktop/Semester 5/Operating Systems/Lab/Lab_8$ ./a.out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 3 is Hungry
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 1 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 4 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
^C
tharun@DESKTOP-HD8TAHP:/mnt/c/Users/nagat/OneDrive/Desktop/Semester 5/Operating Systems/Lab/Lab_8$
```

Q2. Reader Writer Problem

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

// defining READERS = 4 and WRITERS = 2 according to problem
#define READERS 4
#define WRITERS 2

// binary semaphore used by writers and readers
sem_t db;
// binary semaphore used by readers
pthread_mutex_t mutex;

// sample data of the database used by readers and writers
int count = 1;
// count of the readers reading the database
int reader_count = 0;

void *writer(void *writer_no){
    // if db is 1 then it becomes 0 and writer can access
the database else writer waits
    sem_wait(&db);
    // writer modifying the database
    count = count*2;
    printf("Writer %d modified count to %d\n",(*((int
*)writer_no)),count);
    // making db back to 1 after the writer left
    sem_post(&db);
}

void *reader(void *reader_no){
    // reducing mutex before modifying reader_count
    pthread_mutex_lock(&mutex);
    reader_count++;
    // if a reader exist in database then block the writers
from accessing database
    if(reader_count == 1){
        sem_wait(&db);
    }
    // releasing the mutex
```

```
    pthread_mutex_unlock(&mutex);

    // reading section
    printf("Reader %d: read count as %d\n",*((int
*)reader_no),count);

    // reducing mutex for modifying reader_count when reader
job is done
    pthread_mutex_lock(&mutex);
    reader_count--;
    // if there are no readers then writer can be allowed
    if(reader_count == 0){
        sem_post(&db);
    }
    // relasing the mutex after reader_count updation
    pthread_mutex_unlock(&mutex);
}

int main(){

    pthread_t read[READERS],write[WRITERS];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&db,0,1);

    // used for numbering writers and readers
    int a[4] = {1,2,3,4};

    // creating the reader and writer processes randomly
    pthread_create(&read[0], NULL, (void *)reader, (void
*)&a[0]);
    pthread_create(&read[1], NULL, (void *)reader, (void
*)&a[1]);
    pthread_create(&write[0], NULL, (void *)writer, (void
*)&a[0]);
    pthread_create(&read[2], NULL, (void *)reader, (void
*)&a[2]);
    pthread_create(&write[1], NULL, (void *)writer, (void
*)&a[1]);
    pthread_create(&read[3], NULL, (void *)reader, (void
*)&a[3]);

    for(int i = 0; i < READERS; i++){
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < WRITERS; i++){
        pthread_join(write[i], NULL);
```

```
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&db);
    return 0;
}
```

Definition:

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers), whereas others may want to update (writers) the database. Here if we two readers want to access the database simultaneously there will be no issue. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue. This synchronization problem is referred to as the readers-writers problem.

This problem of synchronization can be solved using semaphores.

1. **semaphore mutex**: semaphore mutex is used to ensure mutual exclusion when reader_count is updated i.e. when any reader enters or exit from the critical section and semaphore wrt is used by both readers and writers
2. **int reader_count:** reader_count tells the number of processes performing read in the critical section, initially : the value of reader count is 0

Reader process:

It increments the count of the number of readers inside the critical section. If this reader is the first reader entering, it restricts entry of writers if any reader is inside. After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the writer can enter the critical section.

Writer process:

Writer requests the entry to the critical section. If allowed it enters and performs the write. If not allowed, it keeps on waiting. After performing the write It exits the critical section.