**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)**:

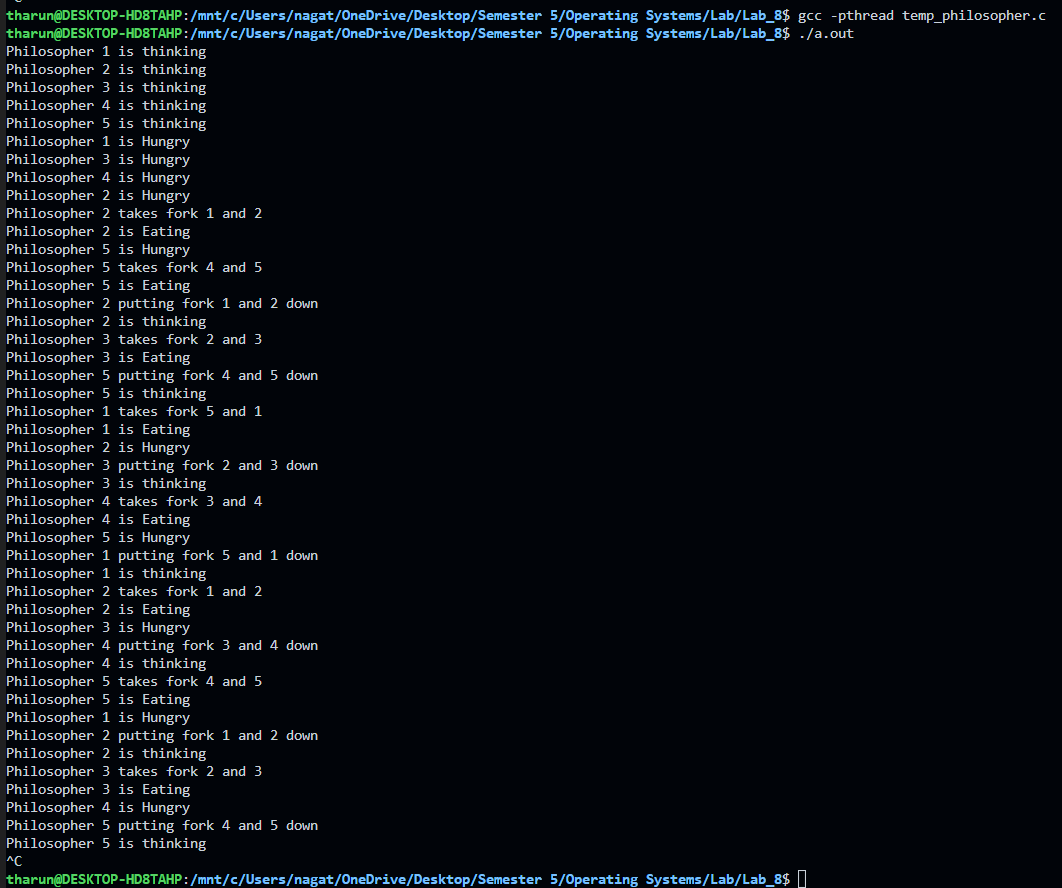
We first check the state of the current philosopher (whether he is hungry).

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.



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.

