# TASK 1:

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define NUM 5 //No of philosphers

pthread\_t philosophers[NUM];

sem\_t chopsticks[NUM];

void\* philosopher(void\* arg)

{

int id = \*((int\*)arg);

int left\_chopstick = id;

int right\_chopstick = (id + 1) % NUM;

while (1)

{

// Try to pick up left chopstick

sem\_wait(&chopsticks[left\_chopstick]);

// Try to pick up right chopstick

if (sem\_trywait(&chopsticks[right\_chopstick]) == 0)

{

printf("Philosopher %d is eating.\n", id);

usleep(rand() % 1000);

// Put down right chopstick

sem\_post(&chopsticks[right\_chopstick]);

}

// Put down left chopstick

sem\_post(&chopsticks[left\_chopstick]);

printf("Philosopher %d is not eating.\n", id);

usleep(rand() % 1000);

}

return NULL;

}

int main()

{

int id[NUM];

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

{

sem\_init(&chopsticks[i], 0, 1);

}

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

{

id[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &id[i]);

}

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

{

pthread\_join(philosophers[i], NULL);

}

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

{

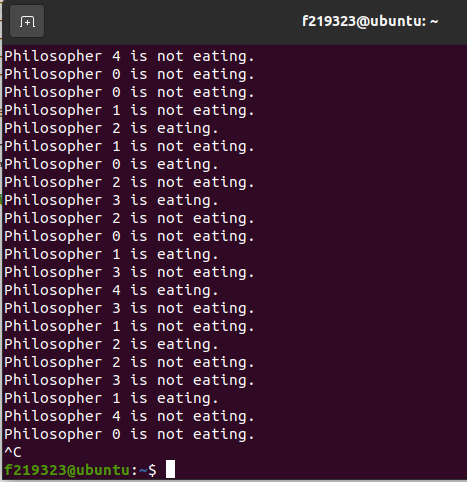
sem\_destroy(&chopsticks[i]);

}

return 0;

}

# OUTPUT:



# TASK 2:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_JOBS 10

typedef struct {

char\* text;

} Job;

typedef struct {

Job jobs[MAX\_JOBS];

int front;

int rear;

int count;

} PrinterSpooler;

void initialize(PrinterSpooler\* spooler) {

spooler->front = 0;

spooler->rear = -1;

spooler->count = 0;

}

void enqueue(PrinterSpooler\* spooler, char\* text) {

if (spooler->count >= MAX\_JOBS) {

printf("Printer spooler is full. Cannot add job.\n");

return;

}

spooler->rear = (spooler->rear + 1) % MAX\_JOBS;

Job newJob;

newJob.text = text;

spooler->jobs[spooler->rear] = newJob;

spooler->count++;

}

Job dequeue(PrinterSpooler\* spooler) {

if (spooler->count <= 0) {

printf("Printer spooler is empty. Cannot dequeue job.\n");

Job nullJob;

nullJob.text = "";

return nullJob;

}

Job dequeuedJob = spooler->jobs[spooler->front];

spooler->front = (spooler->front + 1) % MAX\_JOBS;

spooler->count--;

return dequeuedJob;

}

void printJob(Job job) {

printf("Printing job: %s\n", job.text);

}

int main() {

PrinterSpooler spooler;

initialize(&spooler);

enqueue(&spooler, "Job 1");

enqueue(&spooler, "Job 2");

enqueue(&spooler, "Job 3");

while (spooler.count > 0) {

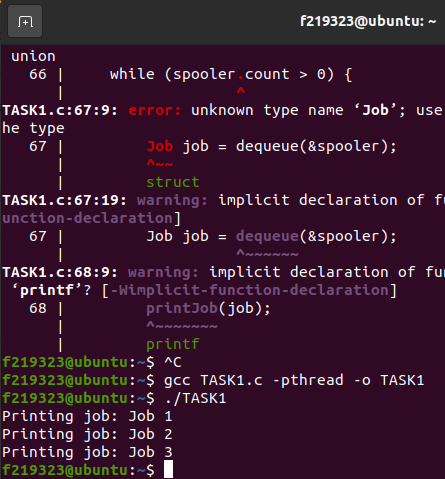
Job job = dequeue(&spooler);

printJob(job);

}

return 0;

}  
  
OUTPUT:



# TASK 3:

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

sem\_t mutex, wrt;

int readCount = 0, data = 0;

void\* reader(void \*arg) {

int id = \*(int\*)arg;

while (1) {

sem\_wait(&mutex);

readCount++;

if (readCount == 1) {

sem\_wait(&wrt);

}

sem\_post(&mutex);

printf("Reader %d read data: %d\n", id, data);

sem\_wait(&mutex);

readCount--;

if (readCount == 0) {

sem\_post(&wrt);

}

sem\_post(&mutex);

sleep(1);

}

}

void\* writer(void \*arg) {

int id = \*(int\*)arg;

while (1) {

sem\_wait(&wrt);

data++;

printf("Writer %d wrote data: %d\n", id, data);

sem\_post(&wrt);

sleep(1);

}

}

int main() {

pthread\_t r1, r2, w1, w2;

sem\_init(&mutex, 0, 1);

sem\_init(&wrt, 0, 1);

int r1\_id = 1, r2\_id = 2, w1\_id = 1, w2\_id = 2;

pthread\_create(&r1, NULL, reader, &r1\_id);

pthread\_create(&r2, NULL, reader, &r2\_id);

pthread\_create(&w1, NULL, writer, &w1\_id);

pthread\_create(&w2, NULL, writer, &w2\_id);

pthread\_join(r1, NULL);

pthread\_join(r2, NULL);

pthread\_join(w1, NULL);

pthread\_join(w2, NULL);

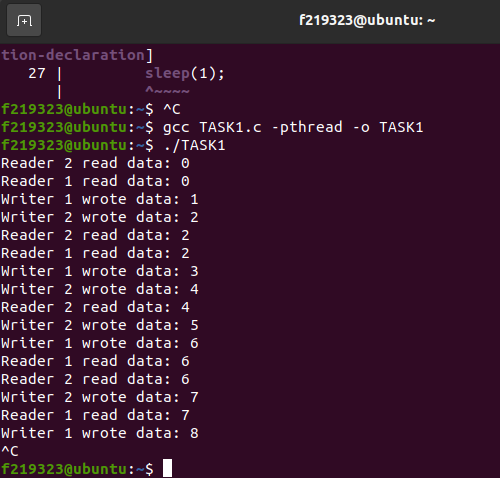
sem\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

OUTPUT:



TASK 4:  
#include <stdio.h>

#include <stdbool.h>

#include <pthread.h>

#define NUM\_THREADS 2

bool flag[2] = {false, false};

int turn = 0;

void enter\_critical\_section(int thread\_num)

{

int other\_thread = 1 - thread\_num;

flag[thread\_num] = true;

turn = other\_thread;

while (flag[other\_thread] && turn == other\_thread);

}

void leave\_critical\_section(int thread\_num)

{

flag[thread\_num] = false;

}

void\* thread\_function(void\* arg)

{

int thread\_num = \*(int\*) arg;

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

{

enter\_critical\_section(thread\_num);

printf("Thread %d is in critical section.\n", thread\_num);

leave\_critical\_section(thread\_num);

printf("Thread %d is out of critical section.\n", thread\_num);

}

pthread\_exit(NULL);

}

int main()

{

pthread\_t threads[NUM\_THREADS];

int thread\_args[NUM\_THREADS];

for (int i = 0; i < NUM\_THREADS; i++)

{

thread\_args[i] = i;

pthread\_create(&threads[i], NULL, thread\_function, &thread\_args[i]);

}

for (int i = 0; i < NUM\_THREADS; i++)

{

pthread\_join(threads[i], NULL);

}

return 0;

}

# OUTPUT:

