Solutions for Practical Session N°3

Thread Synchronization with Semaphores (POSIX)

Objective: Understand and implement thread synchronization using POSIX semaphores to control execution order and coordination between concurrent threads in a Linux environment.

Basic Concepts:

- The POSIX semaphore manipulation services are provided in the **<semaphore.h>** library.
- The semaphore type is represented by the **sem_t** data type.
- This library provides the following functions:

Function	Description
int sem_init(sem_t *sem, int pshared, unsigned int value)	Initializes a semaphore. **sem** is a pointer to the semaphore to be initialized; **value** is the initial value of the semaphore; **pshared** specifies whether the semaphore is local to the process **(pshared = 0)** or shared between processes **(pshared ≠ 0)**. Currently, Linux does not support shared semaphores.
int sem_wait(sem_t *sem)	Equivalent to the **P** operation. Always returns **0**.
int sem_post(sem_t *sem)	Equivalent to the **V** operation. Returns **0** on success or **-1** on failure.
int sem_trywait(sem_t *sem)	Decrements the semaphore value if it is greater than 0; otherwise, it returns an error. This is an atomic (non-blocking) operation..
int sem_getvalue (sem_t* nom, int * sval)	Retrieves the current value of a semaphore. Always returns 0.
int sem_destroy (sem_t* nom)	Destroys a semaphore. Returns -1 if there are still threads waiting on it.

Exercise 1:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <sched.h>   // for sched_yield()

sem_t mutex; // semaphore used as a mutex

void* display(void* name) {
    int i, j;
```

```c
    for (i = 0; i < 20; i++) {
        sem_wait(&mutex); // enter critical section

        // print first 5 repetitions
        for (j = 0; j < 5; j++)
            printf("%s ", (char*)name);

        sched_yield(); // yield CPU to allow other thread to run

        // print next 5 repetitions
        for (j = 0; j < 5; j++)
            printf("%s ", (char*)name);

        printf("\n");

        sem_post(&mutex); // leave critical section
    }

    return NULL;
}

int main(void) {
    pthread_t threadA, threadB;

    // initialize semaphore (value = 1)
    if (sem_init(&mutex, 0, 1) != 0) {
        perror("sem_init");
        return 1;
    }

    // create threads
    if (pthread_create(&threadA, NULL, display, "AA") != 0) {
        perror("pthread_create threadA");
        return 1;
    }
    if (pthread_create(&threadB, NULL, display, "BB") != 0) {
        perror("pthread_create threadB");
        return 1;
    }

    // wait for threads to finish
    pthread_join(threadA, NULL);
    pthread_join(threadB, NULL);

    printf("Parent finished\n");

    sem_destroy(&mutex);

    return 0;
}
```

1. Study this program, then run it several times and observe the results obtained.
2. Remove some important parts of the code (e.g., sem_post, pthread_join, …) and execute it multiple times to understand the purpose of each function.

SOLUTION

1- observations

- ✓ sem_wait(&mutex) → ensures **only one thread prints at a time**.
- ✓ sem_post(&mutex) → releases the semaphore for the other thread.
- ✓ pthread_join() → ensures **main waits for both threads** before printing "Parent finished".
- ✓ sched_yield() → improves fairness of CPU scheduling

2- Remove some important parts

Removed/Modified	Observed Behavior	Explanation
Remove sem_wait()	Output becomes **mixed**: AA and BB interleave randomly.	Threads no longer wait for each other → no mutual exclusion.
Remove sem_post()	Program **freezes** after first thread prints.	Semaphore never released → other thread blocks forever.
Remove pthread_join()	"Parent finished" may appear **before threads finish printing**, or threads may be terminated prematurely.	Main thread does not wait → program ends while threads are still running.
Remove sched_yield()	**one thread may dominate CPU**, producing several consecutive lines of AA before BB prints but only if we do not use semaphore. When we use semaphore the behavior stay correct.	used here to **visually confirm mutual exclusion**. T**hread scheduling** may differ, but semaphore ensures **critical section safety**.

Exercise 2: Alternation Problem Between Processes

Create a C program under POSIX that starts three threads: **ThreadA**, **ThreadB**, and **ThreadC**.

Each thread infinitely displays a character on the screen:

- **ThreadA:** while(true) { print("A"); }
- **ThreadB:** while(true) { print("B"); }
- **ThreadC:** while(true) { print("C"); }

Synchronize the three threads so that the output on the screen appears as follows:

1. ABCABCABCABC...
2. ABACABACABAC...

SOLUTION

Part 1 – ABCABCABC...

- ✓ Use three semaphores sA = 1, sB=0, sC=0
- ✓ sA starts at 1 (so A runs first).
- ✓ After printing "A", it signals sB.
- ✓ B then prints and signals sC.
- ✓ C prints and signals sA again.
- ✓ The cycle repeats indefinitely.

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

sem_t sA, sB, sC;

void* threadA(void* arg) {
    while (1) {
        sem_wait(&sA);
        printf("A");
        fflush(stdout);
        sem_post(&sB); // allow B next
    }
}

void* threadB(void* arg) {
    while (1) {
        sem_wait(&sB);
        printf("B");
        fflush(stdout);
        sem_post(&sC); // allow C next
    }
}

void* threadC(void* arg) {
    while (1) {
        sem_wait(&sC);
        printf("C");
        fflush(stdout);
        sem_post(&sA); // allow A next (loop)
    }
}

int main() {
    pthread_t tA, tB, tC;

    // Initialize semaphores
    sem_init(&sA, 0, 1);  // Start with A
    sem_init(&sB, 0, 0);
    sem_init(&sC, 0, 0);

    // Create threads
    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);
    pthread_create(&tC, NULL, threadC, NULL);

    // Wait (threads loop infinitely)
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);
    pthread_join(tC, NULL);

    // Cleanup
    sem_destroy(&sA);
    sem_destroy(&sB);
    sem_destroy(&sC);
    return 0;
}
```

Part 2 – ABACABAC

- ✓ Use three semaphores sA = 1, sB=1, sC=0
- ✓ A starts, prints "A", and signals **both** B and C once.
- ✓ Both B and C need **two signals** before they can print.
- ✓ This ensures alternation:

  A → B → A → C → A → B → A → C ...

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

sem_t sA, sB, sC;

void* threadA(void* arg) {
    while (1) {
        sem_wait(&sA);
        printf("A");
        fflush(stdout);
        sem_post(&sB); // give one token to B
        sem_post(&sC); // give one token to C
    }
}

void* threadB(void* arg) {
    while (1) {
        sem_wait(&sB);
        sem_wait(&sB); // needs two tokens before printing
        printf("B");
        fflush(stdout);
        sem_post(&sA); // allow A again
    }
}

void* threadC(void* arg) {
    while (1) {
        sem_wait(&sC);
        sem_wait(&sC); // needs two tokens before printing
        printf("C");
        fflush(stdout);
        sem_post(&sA); // allow A again
    }
}

int main() {
    pthread_t tA, tB, tC;

    // Initialize semaphores like in the Java version
    sem_init(&sA, 0, 1); // A starts first
    sem_init(&sB, 0, 1); // B initially has 1
    sem_init(&sC, 0, 0); // C starts blocked

    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);
    pthread_create(&tC, NULL, threadC, NULL);
```

```
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);
    pthread_join(tC, NULL);

    sem_destroy(&sA);
    sem_destroy(&sB);
    sem_destroy(&sC);
    return 0;
}
```

Exercise 3

For each of the following problems (covered in **chapter 2**), write a **C program** using **POSIX semaphores**, to simulate its behavior:

1) The **Rendezvous Problem:** one RDV and N processes

2) Producers/Consumers: assume the program ends after a fixed number of iterations (10 productions)

- **a)** Program for one producer and one customer,
- **b)** then generalize the program for multiple producers and customers

3) Readers/Writers:

- **a)** We must fix the number of readers and writers. The number of accesses must also be fixed (to avoid an infinite loop), for example `ACCES = 4`. The operations of reading the database, modifying the data, and writing to the database are simulated using `sleep(1)` for one second.
- **b)** Chapter 2's solution causes writer starvation; modify the program to ensure fairness

4) The Dining Philosophers Problem

SOLUTION

Note: Algorithmic solutions to all of these problems were covered in the course sessions (Chapter 2); therefore, the task now is simply to implement them in C using POSIX.

1) The Rendezvous Problem

- ✓ A process, called **RDV**, must wait until **N other processes** have reached a specific point before continuing its execution.
- ✓ The **N−1 processes** are blocked, waiting for the **last process** to arrive.
- ✓ The **last (Nth) process** will then wake up the **RDV process**.
- ✓ In turn, the **RDV process** will wake up all the other blocked processes.

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

#define N 5  // Number of processes that must reach the rendezvous

sem_t mutex;      // Protects count
sem_t waitRDV;    // RDV process waits on this
sem_t waitOthers; // Other processes wait for RDV

int count = 0;    // Number of processes arrived

void* RDV(void* arg) {
    sem_wait(&waitRDV);
    printf("RDV process starts\n");

    // Wake up all waiting processes
    int i;
    for (i = 1; i < N; i++) {
        sem_post(&waitOthers);
    }

    printf("RDV process finished waking others\n");
    return NULL;
}

// Instead of taking an argument, each thread uses its index
void* process(void* arg) {
    long id = (long)arg; // cast back from long (not pointer arithmetic!)

    printf("Process %ld reached rendezvous point\n", id);

    sem_wait(&mutex);
    count++;
    if (count == N) {
        printf("Process %ld is not blocked, it wakes up the RDV\n", id);
        sem_post(&mutex);
        sem_post(&waitRDV);
    } else {
        sem_post(&mutex);
        sem_wait(&waitOthers);
        printf("Process %ld continues after RDV\n", id);
    }

    return NULL;
}

int main() {
    pthread_t rdvThread;
    pthread_t procs[N];

    // Initialize semaphores
    sem_init(&mutex, 0, 1);
    sem_init(&waitRDV, 0, 0);
    sem_init(&waitOthers, 0, 0);

    // Create the RDV process
    pthread_create(&rdvThread, NULL, RDV, NULL);
```

```
    // Create N ordinary processes
    long i;
    for (i = 1; i <= N; i++) {
        pthread_create(&procs[i], NULL, process, (void*)i);
    }

    // Wait for all threads
    for (i = 1; i <= N; i++) {
        pthread_join(procs[i], NULL);
    }
    pthread_join(rdvThread, NULL);

    // Destroy semaphores
    sem_destroy(&mutex);
    sem_destroy(&waitRDV);
    sem_destroy(&waitOthers);

    return 0;
}
```

2) Producers/Consumers

- We consider two categories of processes: Producers and Consumers, which access a shared memory structure called the Buffer.
- The buffer has a fixed capacity of N, meaning it can store up to N items simultaneously.
- Producers generate items (of arbitrary values) and insert them into the buffer.
- Consumer processes retrieve and process the items stored in the buffer.

The operation of the processes must satisfy the following constraints:
- Producers do not place items into the buffer when it is full.
- Consumers do not remove items from the buffer when it is empty.
- Only one process may access the buffer at any given time.
- Items must not be lost or consumed more than once.

We use three semaphores:

- The first, named **full**, counts the number of occupied slots (initialized to 0).
- The second, named **empty**, counts the number of free slots (initialized to N, the buffer size).
- The last, named **mutex**, ensures **mutual exclusion** for accessing the buffer (when one process is using the buffer, others cannot access it).

a) One Producer and One Consumer

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

#define BUFFER_SIZE 3
#define ITEMS_TO_PRODUCE 10

sem_t full, empty, mutex;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;
```

```c
void* producer(void* arg);
void* consumer(void* arg);

int main() {
    pthread_t prod, cons;

    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&mutex, 0, 1);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&full);
    sem_destroy(&empty);
    sem_destroy(&mutex);

    printf("Program finished.\n");
    return 0;
}

void* producer(void* arg) {
    int item = 0;
    int producedCount = 0;
    int i;  // loop counter declared before loop

    for (i = 0; i <= ITEMS_TO_PRODUCE; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Producer: buffer[%d] = %d\n", in, item);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);

        item++;
        producedCount++;
        usleep(100000);
    }
    return NULL;
}

void* consumer(void* arg) {
    int item;
    int consumedCount = 0;
    int i;  // loop counter declared before loop

    for (i = 0; i <= ITEMS_TO_PRODUCE; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumer: buffer[%d] = %d\n", out, item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
```

```
        sem_post(&empty);

        consumedCount++;
        usleep(150000);
    }
    return NULL;
}
```

b) Multiple Producers and Multiple Consumers

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

#define BUFFER_SIZE 5
#define NUM_PRODUCERS 2
#define NUM_CONSUMERS 3
#define ITEMS_TO_PRODUCE 10

sem_t full, empty, mutex;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void* producer(void* arg);
void* consumer(void* arg);

int main() {
    pthread_t producers[NUM_PRODUCERS];
    pthread_t consumers[NUM_CONSUMERS];
    int ids[NUM_PRODUCERS > NUM_CONSUMERS ? NUM_PRODUCERS : NUM_CONSUMERS];
    int i;  // loop counter declared before loops

    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&mutex, 0, 1);

    // Create producer threads
    for (i = 0; i < NUM_PRODUCERS; i++) {
        ids[i] = i + 1;
        pthread_create(&producers[i], NULL, producer, &ids[i]);
    }

    // Create consumer threads
    for (i = 0; i < NUM_CONSUMERS; i++) {
        ids[i] = i + 1;
        pthread_create(&consumers[i], NULL, consumer, &ids[i]);
    }

    // Wait for all threads
    for (i = 0; i < NUM_PRODUCERS; i++)
        pthread_join(producers[i], NULL);
    for (i = 0; i < NUM_CONSUMERS; i++)
        pthread_join(consumers[i], NULL);

    sem_destroy(&full);
    sem_destroy(&empty);
    sem_destroy(&mutex);

    printf("All producers and consumers finished.\n");
```

```
    return 0;
}

void* producer(void* arg) {
    int id = *(int*)arg;
    int item = 0;
    int i;  // loop counter declared before loop

    for (i = 0; i < ITEMS_TO_PRODUCE; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Producer %d: buffer[%d] = %d\n", id, in, item);
        in = (in + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&full);

        item++;
        usleep(100000);
    }
    return NULL;
}

void* consumer(void* arg) {
    int id = *(int*)arg;
    int item;
    int i;  // loop counter declared before loop

    int items_to_consume = (NUM_PRODUCERS * ITEMS_TO_PRODUCE) / NUM_CONSUMERS;
    for (i = 0; i < items_to_consume; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumer %d: buffer[%d] = %d\n", id, out, item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);

        usleep(150000);
    }
    return NULL;
}
```

3) Readers/Writers

Readers are not mutually exclusive among themselves.so, It is necessary to **count the number of readers** using the resource simultaneously. We use of two semaphores:

- ✓ **Write:** ensures mutual exclusion among writers, for allowing only **one writer to access the resource at a time**. So, it is initialized to 1
- ✓ **Mutex:** semaphore for mutual exclusion when updating the reader count, for allowing only **one reader to access the counter at a time**. It is initialized to 1

a) We fix the number of readers and writers. The number of accesses is fixed (to avoid an infinite loop), `ACCES = 4`. The operations of reading the database, modifying the data, and writing to the database are simulated using `sleep(1)` for one second.

```c
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#define NUM_READERS 3
#define NUM_WRITERS 2
#define ACCES 4
sem_t mutex; // protects read_count
sem_t write; // ensures exclusive writer access
int read_count = 0; // number of active readers
void* reader(void* arg);
void* writer(void* arg);
int main() {
  pthread_t readers[NUM_READERS];
  pthread_t writers[NUM_WRITERS];
  int reader_ids[NUM_READERS];
  int writer_ids[NUM_WRITERS];
  int i;
  // Initialize semaphores
  sem_init(&mutex, 0, 1);
  sem_init(&write, 0, 1);
  // Create reader threads
  for (i = 0; i < NUM_READERS; i++) {
    reader_ids[i] = i + 1;
    pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
  }
  // Create writer threads
  for (i = 0; i < NUM_WRITERS; i++) {
    writer_ids[i] = i + 1;
    pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
  }
  // Wait for all threads
  for (i = 0;i < NUM_READERS;i++) pthread_join(readers[i],NULL);
  for (i = 0; i < NUM_WRITERS;i++) pthread_join(writers[i],NULL);

  // Destroy semaphores
  sem_destroy(&mutex);
  sem_destroy(&write);
  printf("All readers and writers finished.\n");
  return 0;
}
void* reader(void* arg) {
  int id = *(int*)arg;
  int i;
  for (i = 0; i < ACCES; i++) {
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1){
      printf("Reader %d is the first one,he wants to read\n", id);
      sem_wait(&write); // first reader locks writers
    }
    sem_post(&mutex);
    // Reading section
    printf("Reader %d is reading\n", id);
    sleep(1); // simulate reading
    sem_wait(&mutex);
    read_count--;
```

```
    if (read_count == 0){
      printf("Reader %d wakes up the next writer\n", id);
      sem_post(&write); // last reader releases writers
    }
    sem_post(&mutex); // allow other threads to run
    sleep(1);
  }
  return NULL;
}
void* writer(void* arg) {
  int id = *(int*)arg;
  int i;
  for (i = 0; i < ACCES; i++) {
    printf("Writer %d wants to write\n", id);
    sem_wait(&write); // writer locks exclusive access
    // Writing section
    printf("Writer %d is writing\n", id);
    sleep(1); // simulate writing
    printf("Writer %d wakes up an other process\n", id);
    sem_post(&write); // allow other threads to run
    sleep(1);
  }
  return NULL;
}
```

b) fair program:

- ✓ Use an extra semaphore readTry to prevent new readers from entering when a writer is waiting.
- ✓ Readers can only start reading if no writer is waiting.
- ✓ Writers get exclusive access but don't block active readers until they finish.

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
#define NUM_READERS 3
#define NUM_WRITERS 2
#define ACCES 4
sem_t mutex; // protects read_count
sem_t write; // ensures exclusive writer access
sem_t readTry; // blocks new readers if a writer is waiting
int read_count = 0; // number of active readers
void* reader(void* arg);
void* writer(void* arg);
int main() {
  pthread_t readers[NUM_READERS];
  pthread_t writers[NUM_WRITERS];
  int reader_ids[NUM_READERS];
  int writer_ids[NUM_WRITERS];
  int i;
  sem_init(&mutex, 0, 1);
  sem_init(&write, 0, 1);
  sem_init(&readTry, 0, 1);
  // Create reader threads
  for (i = 0; i < NUM_READERS; i++) {
    reader_ids[i] = i + 1;
    pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
  }
  // Create writer threads
```

```
  for (i = 0; i < NUM_WRITERS; i++) {
    writer_ids[i] = i + 1;
    pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
  }
  // Wait for all threads
  for (i = 0; i < NUM_READERS;i++) pthread_join(readers[i],NULL);
  for (i = 0; i < NUM_WRITERS;i++) pthread_join(writers[i],NULL);
  sem_destroy(&mutex);
  sem_destroy(&write);
  sem_destroy(&readTry);
  printf("All readers and writers finished.\n");
  return 0;
}
void* reader(void* arg) {
  int id = *(int*)arg;
  int i;
  for (i = 0; i < ACCES; i++) {
    printf("Reader %d wants to read\n", id);
    sem_wait(&readTry); // wait if a writer is waiting
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) sem_wait(&write); // first reader locks writers
    sem_post(&mutex);
    sem_post(&readTry);
    printf("Reader %d is reading\n", id);
    sleep(1);
    sem_wait(&mutex);
    read_count--;
    if (read_count == 0){
      printf("Reader %d wakes up an other writer\n", id);
      sem_post(&write); // last reader releases writers
    }
    sem_post(&mutex);
    sleep(1);
  }
  return NULL;
}
void* writer(void* arg) {
  int id = *(int*)arg;
  int i;
  for (i = 0; i < ACCES; i++) {
    printf("Writer %d wants to write\n", id);
    sem_wait(&readTry); // block new readers
    sem_wait(&write); // wait for active readers/writers
    printf("Writer %d is writing\n", id);
    sleep(1);
    sem_post(&write); // allow readers again
    sem_post(&readTry);
    sleep(1);
  }
  return NULL;
}
```

4) The Dining Philosophers Problem

We use **N semaphores**, one for each fork, initialized to 0. When a philosopher *i* is unable to pick up the forks, they block by calling sem_wait(&S[i]). After finishing their meal, a philosopher checks whether their neighbors are waiting (via the **Test** procedure). If a neighbor can now eat, it is signaled using sem_post(&S[i]). Philosophers can be in one of three states: **thinking**, **hungry**, or **eating**.

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>
#define N 5 // Number of philosophers
// Philosopher states
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];// Array to keep track of philosopher states
sem_t mutex;// Mutex for critical section
sem_t S[N];// Semaphore for each philosopher
// Get the index of the left and right neighbors
#define LEFT(i) ((i + N - 1) % N)
#define RIGHT(i) ((i + 1) % N)
// Function to test if philosopher i can eat
void test(int i) {
  if (state[i] == HUNGRY &&
    state[LEFT(i)] != EATING &&
    state[RIGHT(i)] != EATING) {
    state[i] = EATING;
  sem_post(&S[i]); // Allow philosopher i to eat
}
}
// Function to take forks
void take_forks(int i) {
  sem_wait(&mutex);// Enter critical section
  state[i] = HUNGRY;
  printf("Philosopher %d is hungry\n", i);
  test(i);// Try to acquire forks
  sem_post(&mutex);// Exit critical section
  sem_wait(&S[i]);// Wait if forks not available
}

// Function to put down forks
void put_forks(int i) {
  sem_wait(&mutex);//Enter critical section
  state[i] = THINKING;
  printf("Philosopher %d ends eating \n", i);
  printf("Philosopher %d tries to wake up left neighbor\n", i);
  test(LEFT(i));//Check if left neighbor can eat
  printf("Philosopher %d tries to wake up right neighbor\n", i);
  test(RIGHT(i));//Check if right neighbor can eat
  sem_post(&mutex);//Exit critical section
}

// Philosopher thread function
void* philosopher(void* num) {
  int i = *(int*)num;
  while (1) {
    printf("Philosopher %d is thinking\n", i);
    sleep(rand() % 3 + 1); // Thinking
    take_forks(i);
    printf("Philosopher %d is eating\n", i);
    sleep(rand() % 2 + 1); // Eating
    put_forks(i);
  }
}
```

```c
int main() {
  int i;
  pthread_t thread_id[N];
  int phil_num[N];
  sem_init(&mutex, 0, 1); // Initialize mutex
  for (i = 0; i < N; i++) sem_init(&S[i], 0, 0); // Initialize philosophers semaphores
  for (i = 0; i < N; i++) {
    phil_num[i] = i;
    pthread_create(&thread_id[i], NULL, philosopher, &phil_num[i]);
  }
  for (i = 0; i < N; i++) pthread_join(thread_id[i], NULL);
  for (i = 0; i < N; i++) sem_destroy(&S[i]);
  sem_destroy(&mutex);
  return 0;
}
```