Lab 6: Semaphore and its Applications

CS363 • Operating System (Lab)

Introduction to Semaphore

A semaphore is a fundamental synchronization primitive used in operating systems and concurrent programming to control access to shared resources by multiple processes or threads, helping prevent race conditions and ensuring mutual exclusion.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them.

Semaphore Basics

A semaphore is an integer variable that supports two atomic operations: wait (P) and signal (V).

Counting semaphores allow a resource to be shared among N processes, while binary (mutex) semaphores offer exclusive access.

They help coordinate process execution and protect critical sections in code.

Types of Semaphore

The two main types of semaphores are counting semaphores and binary semaphores, each serving different synchronization needs in operating systems and concurrent programming.

Counting Semaphore

- A counting semaphore can take any non-negative integer value (typically from 0 up to N).
- It is used to control access to a resource with multiple instances (e.g., managing access to a pool of 5 printers).
- The count represents the number of available resources. Whenever a process acquires a resource, the count decreases; when released, it increases.

Binary Semaphore

- A binary semaphore is a special case of the counting semaphore, restricted to only two values: 0 or 1.
- It is commonly used for achieving mutual exclusion (locking): only one process can be in the critical section at a time.
- The binary semaphore is also known as a mutex (mutual exclusion lock).

Summary Table

Type	Value Range	Typical Use
Counting Semaphore	0 to N	Multiple resource allocation
Binary Semaphore	0 or 1	Mutual exclusion, locks

Table 1: Summary of Semaphore Types

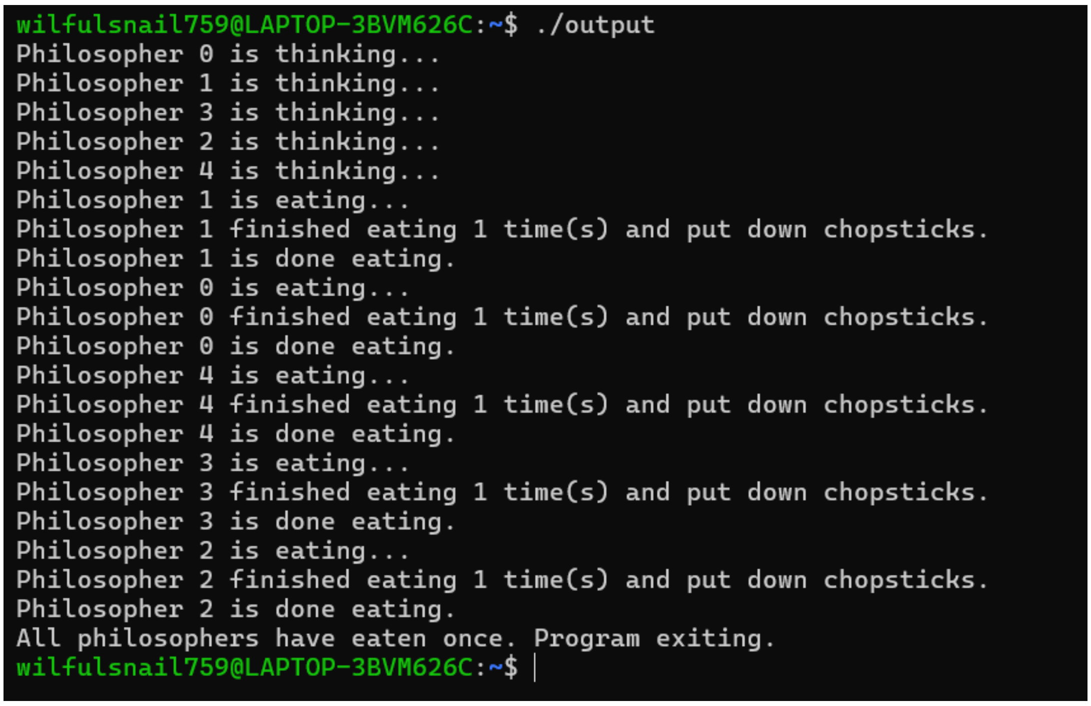
Dining Philosopher Problem

The **Dining Philosopher Problem** models five philosophers sitting at a table with five chopsticks (resources) between them. Each philosopher alternates between thinking and eating, but needs both left and right chopsticks to eat. The challenge is to design a system so that philosophers can eat without causing deadlock or starvation.

Listing 1: Example Program for Dining Philosopher

```c

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

#define N 5 // Number of philosophers
#define EAT_LIMIT 1 // Number of times each philosopher eats

sem_t chopstick[N];
int eat_count[N] = {0}; // Track how many times each philosopher has eaten

void* philosopher(void* num) {
    int id = *(int*)num;

    while (eat_count[id] < EAT_LIMIT) {
        printf("Philosopher %d is thinking...\n", id);
        sleep(1);

        sem_wait(&chopstick[id]);
        sem_wait(&chopstick[(id + 1) % N]);

        printf("Philosopher %d is eating...\n", id);
        sleep(2);

        eat_count[id]++;
        printf("Philosopher %d finished eating %d time(s) and put down
            chopsticks.\n", id, eat_count[id]);

        sem_post(&chopstick[id]);
        sem_post(&chopstick[(id + 1) % N]);
    }
    printf("Philosopher %d is done eating.\n", id);
    return NULL;
}

int main() {
    pthread_t tid[N];
    int philosopher_ids[N];

    for (int i = 0; i < N; i++)
        sem_init(&chopstick[i], 0, 1);

    for (int i = 0; i < N; i++) {
        philosopher_ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &philosopher_ids[i]);
    }

    for (int i = 0; i < N; i++)
        pthread_join(tid[i], NULL);

    printf("All philosophers have eaten once. Program exiting.\n");
    return 0;
}
```

Output



```
wilfulsnail759@LAPTOP-3BVM626C:~$ ./output
Philosopher 0 is thinking...
Philosopher 1 is thinking...
Philosopher 3 is thinking...
Philosopher 2 is thinking...
Philosopher 4 is thinking...
Philosopher 1 is eating...
Philosopher 1 finished eating 1 time(s) and put down chopsticks.
Philosopher 1 is done eating.
Philosopher 0 is eating...
Philosopher 0 finished eating 1 time(s) and put down chopsticks.
Philosopher 0 is done eating.
Philosopher 4 is eating...
Philosopher 4 finished eating 1 time(s) and put down chopsticks.
Philosopher 4 is done eating.
Philosopher 3 is eating...
Philosopher 3 finished eating 1 time(s) and put down chopsticks.
Philosopher 3 is done eating.
Philosopher 2 is eating...
Philosopher 2 finished eating 1 time(s) and put down chopsticks.
Philosopher 2 is done eating.
All philosophers have eaten once. Program exiting.
wilfulsnail759@LAPTOP-3BVM626C:~$
```

Figure 1: Output of Dining Philosopher problem...

Reader-Writer Problem

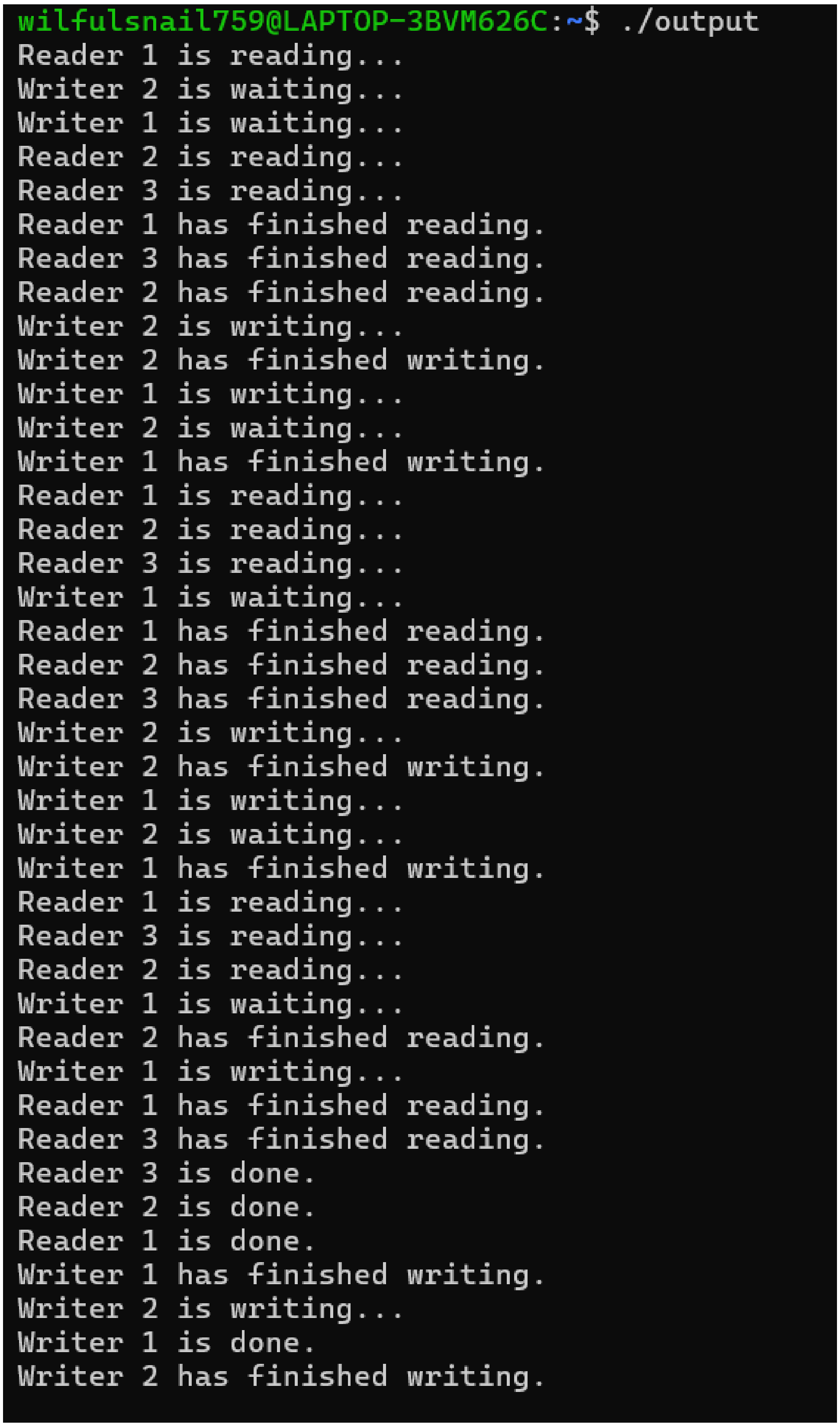
The **Reader-Writer Problem** deals with processes (readers and writers) accessing shared data. Multiple readers can read simultaneously, but writers need exclusive access. The challenge is to allow maximum concurrency without starvation.

Listing 2: Example Program for Reader Writer

```

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

#define NUM_READS 3
#define NUM_WRITES 3

sem_t x, wrt;
int readcount = 0;

void* reader(void* arg) {
    int id = *((int*)arg);
    for (int i = 0; i < NUM_READS; i++) {
        sem_wait(&x);
        readcount++;
        if (readcount == 1)
            sem_wait(&wrt);
        sem_post(&x);

        printf("Reader %d is reading...\n", id);
        sleep(1);

        sem_wait(&x);
        readcount--;
        if (readcount == 0)
            sem_post(&wrt);
        sem_post(&x);

        printf("Reader %d has finished reading.\n", id);
        sleep(1);
    }
    printf("Reader %d is done.\n", id);
    return NULL;
}

void* writer(void* arg) {
    int id = *((int*)arg);
    for (int i = 0; i < NUM_WRITES; i++) {
        printf("Writer %d is waiting...\n", id);
        sem_wait(&wrt);
        printf("Writer %d is writing...\n", id);
        sleep(2);
        sem_post(&wrt);
        printf("Writer %d has finished writing.\n", id);
        sleep(1);
    }
```

```c
    printf("Writer %d is done.\n", id);
    return NULL;
}

int main() {
    pthread_t rtid[3], wtid[2];
    int r[3] = {1, 2, 3};
    int w[2] = {1, 2};

    sem_init(&x, 0, 1);
    sem_init(&wrt, 0, 1);

    for (int i = 0; i < 3; i++)
        pthread_create(&rtid[i], NULL, reader, &r[i]);

    for (int i = 0; i < 2; i++)
        pthread_create(&wtid[i], NULL, writer, &w[i]);

    for (int i = 0; i < 3; i++)
        pthread_join(rtid[i], NULL);

    for (int i = 0; i < 2; i++)
        pthread_join(wtid[i], NULL);

    printf("All readers and writers finished. Program exiting.\n");
    return 0;
}
```

Output



```
wilfulsnail759@LAPTOP-3BVM626C:~$ ./output
Reader 1 is reading...
Writer 2 is waiting...
Writer 1 is waiting...
Reader 2 is reading...
Reader 3 is reading...
Reader 1 has finished reading.
Reader 3 has finished reading.
Reader 2 has finished reading.
Writer 2 is writing...
Writer 2 has finished writing.
Writer 1 is writing...
Writer 2 is waiting...
Writer 1 has finished writing.
Reader 1 is reading...
Reader 2 is reading...
Reader 3 is reading...
Writer 1 is waiting...
Reader 1 has finished reading.
Reader 2 has finished reading.
Reader 3 has finished reading.
Writer 2 is writing...
Writer 2 has finished writing.
Writer 1 is writing...
Writer 2 is waiting...
Writer 1 has finished writing.
Reader 1 is reading...
Reader 3 is reading...
Reader 2 is reading...
Writer 1 is waiting...
Reader 2 has finished reading.
Writer 1 is writing...
Reader 1 has finished reading.
Reader 3 has finished reading.
Reader 3 is done.
Reader 2 is done.
Reader 1 is done.
Writer 1 has finished writing.
Writer 2 is writing...
Writer 1 is done.
Writer 2 has finished writing.
```

Figure 2: Output of Reader Writer problem...

Producer-Consumer Problem

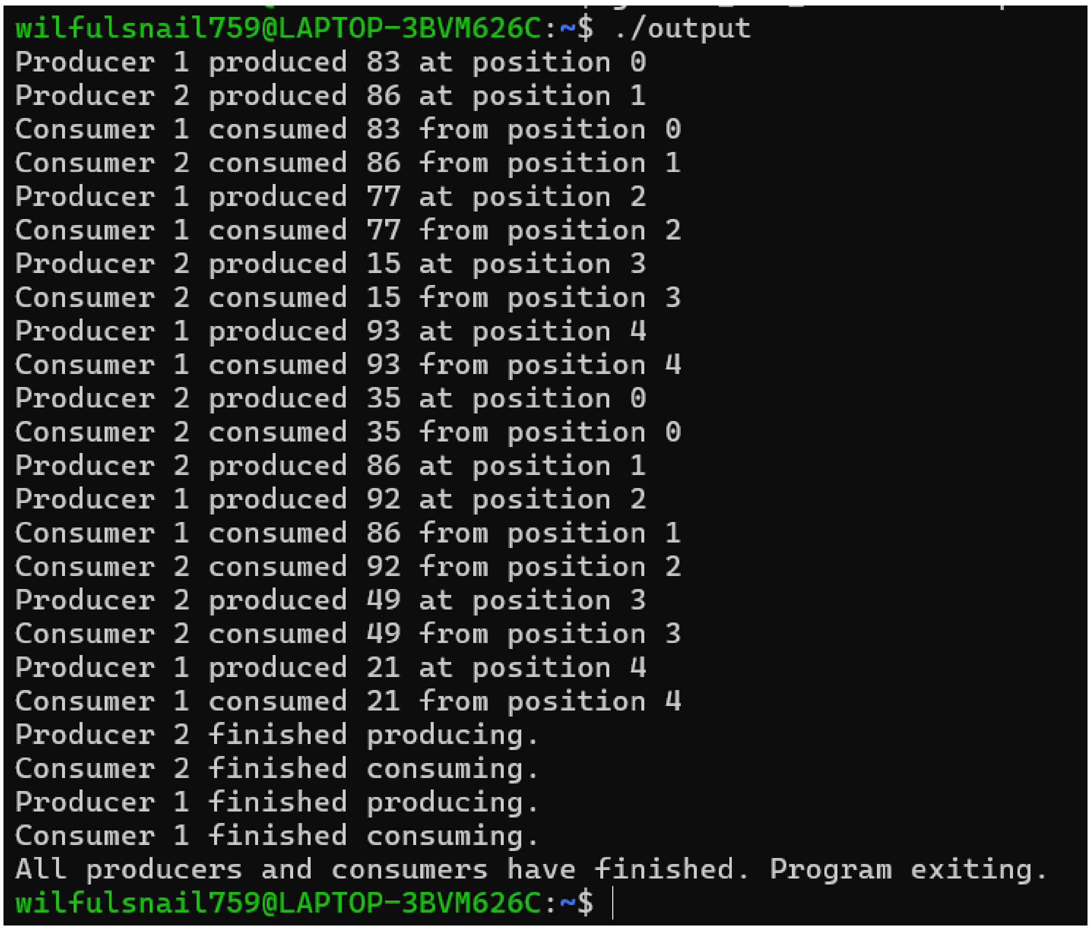
The Producer-Consumer Problem involves two types of processes: producers (which generate data and place them in a buffer) and consumers (which take data from the buffer). The buffer has a fixed size. Proper synchronization ensures that producers do not add to a full buffer and consumers do not remove from an empty buffer.

Listing 3: Example Program for Producer Consumer

```c

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define SIZE 5
#define ITEMS_TO_PRODUCE 5

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

void* producer(void* arg) {
    int item, id = *((int*)arg);
    for (int i = 0; i < ITEMS_TO_PRODUCE; i++) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        printf("Producer %d produced %d at position %d\n", id, item, in);
        in = (in + 1) % SIZE;
        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
    printf("Producer %d finished producing.\n", id);
    return NULL;
}

void* consumer(void* arg) {
    int item, id = *((int*)arg);
    for (int i = 0; i < ITEMS_TO_PRODUCE; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        printf("Consumer %d consumed %d from position %d\n", id, item, out)
            ;
        out = (out + 1) % SIZE;
        sem_post(&mutex);
        sem_post(&empty);
        sleep(1);
    }
    printf("Consumer %d finished consuming.\n", id);
    return NULL;
}

```

```c
int main() {
    pthread_t ptid[2], ctid[2];
    int pid[2] = {1, 2}, cid[2] = {1, 2};

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

    for (int i = 0; i < 2; i++)
        pthread_create(&ptid[i], NULL, producer, &pid[i]);
    for (int i = 0; i < 2; i++)
        pthread_create(&ctid[i], NULL, consumer, &cid[i]);

    for (int i = 0; i < 2; i++)
        pthread_join(ptid[i], NULL);
    for (int i = 0; i < 2; i++)
        pthread_join(ctid[i], NULL);

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

    printf("All producers and consumers have finished. Program exiting.\n")
       ;
    return 0;
}
```

Output



```
wilfulsnail759@LAPTOP-3BVM626C:~$ ./output
Producer 1 produced 83 at position 0
Producer 2 produced 86 at position 1
Consumer 1 consumed 83 from position 0
Consumer 2 consumed 86 from position 1
Producer 1 produced 77 at position 2
Consumer 1 consumed 77 from position 2
Producer 2 produced 15 at position 3
Consumer 2 consumed 15 from position 3
Producer 1 produced 93 at position 4
Consumer 1 consumed 93 from position 4
Producer 2 produced 35 at position 0
Consumer 2 consumed 35 from position 0
Producer 2 produced 86 at position 1
Producer 1 produced 92 at position 2
Consumer 1 consumed 86 from position 1
Consumer 2 consumed 92 from position 2
Producer 2 produced 49 at position 3
Consumer 2 consumed 49 from position 3
Producer 1 produced 21 at position 4
Consumer 1 consumed 21 from position 4
Producer 2 finished producing.
Consumer 2 finished consuming.
Producer 1 finished producing.
Consumer 1 finished consuming.
All producers and consumers have finished. Program exiting.
wilfulsnail759@LAPTOP-3BVM626C:~$
```

Figure 3: Output of Producer Consumer problem...