CHAPTER 7

Synchronization Examples

CHAPTER OBJECTIVES

- Explain the bounded-buffer, readers–writers, and dining–philosophers synchronization problems.

Bounded Buffer Problem

7.1.1 The Bounded-Buffer Problem

The *bounded-buffer problem* was introduced in Section 6.1; it is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of `n` buffers, each capable of holding one item. The `mutex` binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore `empty` is initialized to the value `n`; the semaphore `full` is initialized to the value 0.

The code for the producer process is shown in Figure 7.1, and the code for the consumer process is shown in Figure 7.2. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The bounded-buffer problem is a classic synchronization scenario in concurrent programming where a producer process generates data and places it into a fixed-size buffer (of size *n*), and a consumer process removes data from that buffer. The problem ensures that the producer does not add data to a full buffer and the consumer does not remove data from an empty buffer, while both processes may operate concurrently.

Synchronization Challenges:

1. **Mutual Exclusion:** Only one process (producer or consumer) should access the buffer at a time to prevent data corruption or race conditions. This is managed using a binary semaphore (*mutex*).
2. **Buffer Overflow:** The producer must wait if the buffer is full (i.e., no empty slots remain). The *empty* semaphore tracks available spaces and blocks the producer when it reaches zero.
3. **Buffer Underflow:** The consumer must wait if the buffer is empty (i.e., no full slots exist). The *full* semaphore tracks filled slots and blocks the consumer when it reaches zero.
4. **Coordination:** The producer and consumer must signal each other appropriately—after producing, the producer increments *full* to notify the consumer, and after consuming, the consumer increments *empty* to notify the producer—ensuring smooth alternation and avoiding deadlock or starvation.

7.1.1 The Bounded-Buffer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

See Dry Runs Handout

Figure 7.1 The structure of the producer process.

```
while (true) {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
}
```

Figure 7.2 The structure of the consumer process.

```
while (true) {
   wait(full);
   wait(mutex);
    . . .
   /* remove an item from buffer to next_consumed */
    . . .
   signal(mutex);
   signal(empty);
    . . .
   /* consume the item in next_consumed */
    . . .
}
```

7.1.1 The Bounded-Buffer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

See Dry Runs Handout

Figure 7.1 The structure of the producer process.

```
while (true) {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
}
```

7.1.1 The Bounded-Buffer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0


 while (true) {
     . . .
    /* produce an item in next_produced */
     . . .
    wait(empty);
    wait(mutex);
     . . .
    /* add next_produced to the buffer */
     . . .
    signal(mutex);
    signal(full);
 }
```

Figure 7.1 The structure of the producer process.

Initial State:

- *mutex = 1*, *empty = 2*, *full = 0*, Buffer: [-, -].

1. **Producer: Adds "A"**
   - *wait(empty)*: *empty = 2 → 1*.
   - *wait(mutex)*: *mutex = 1 → 0*.
   - Buffer: [A, -].
   - *signal(mutex)*: *mutex = 0 → 1*.
   - *signal(full)*: *full = 0 → 1*.
   - State: *mutex = 1*, *empty = 1*, *full = 1*, Buffer: [A, -].
2. **Producer: Adds "B"**
   - *wait(empty)*: *empty = 1 → 0*.
   - *wait(mutex)*: *mutex = 1 → 0*.
   - Buffer: [A, B].
   - *signal(mutex)*: *mutex = 0 → 1*.
   - *signal(full)*: *full = 1 → 2*.
   - State: *mutex = 1*, *empty = 0*, *full = 2*, Buffer: [A, B].

7.1.1 The Bounded-Buffer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```

Figure 7.2 The structure of the consumer process.

3. **Consumer: Removes "A"**
   - *wait(full)*: *full = 2 → 1*.
   - *wait(mutex)*: *mutex = 1 → 0*.
   - Buffer: [B, -].
   - *signal(mutex)*: *mutex = 0 → 1*.
   - *signal(empty)*: *empty = 0 → 1*.
   - State: *mutex = 1*, *empty = 1*, *full = 1*, Buffer: [B, -].

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

#define BUFFER_SIZE 10

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

int main() {
    pthread_t prod, cons;
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}
```

```c
void* producer(void* arg) {
    int item = 0;
    while (1) {
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item++;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&full);
        printf("Produced: %d\n", item - 1);
    }
    return NULL;
}

void* consumer(void* arg) {
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&empty);
        printf("Consumed: %d\n", item);
    }
    return NULL;
}
```

Readers-Writers Problem

7.1.2 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, read and write) the database. We distinguish between these two types of processes by referring to the former as *readers* and to the latter as *writers*. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers–writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive.

The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers–writers problem. See the bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers–writers problem.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The binary semaphores `mutex` and `rw_mutex` are initialized to 1; `read_count` is a counting semaphore initialized to 0. The semaphore `rw_mutex` is common to both reader and writer processes. The `mutex` semaphore is used to ensure mutual exclusion when the variable `read_count` is updated. The `read_count` variable keeps track of how many processes are currently reading the object. The semaphore `rw_mutex` functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

The code for a writer process is shown in Figure 7.3; the code for a reader process is shown in Figure 7.4. Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on `rw_mutex`, and $n - 1$ readers are queued on `mutex`. Also observe that, when a writer executes `signal(rw_mutex)`, we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers–writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data, it requests the reader–writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader–writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

The readers-writers problem is a synchronization challenge in concurrent systems where multiple processes access a shared database. "Readers" only read the data, while "writers" read and modify it. Multiple readers can access the database simultaneously without issues, but writers require exclusive access to prevent data inconsistency. The problem ensures proper coordination between readers and writers, with variations like the first readers-writers problem (prioritizing readers) and the second (prioritizing writers).

Synchronization Challenges:

1. **Mutual Exclusion for Writers:** Writers must have exclusive access to the database, enforced by a semaphore like *rw_mutex*, preventing simultaneous access by other writers or readers.
2. **Concurrent Reader Access:** Multiple readers should read simultaneously without interference, requiring a mechanism (e.g., *read_count* and *mutex*) to track active readers and manage access to *rw_mutex*.
3. **Priority Handling:** In the first readers-writers problem, readers must not wait unless a writer is active, while in the second, a waiting writer should proceed as soon as possible, delaying new readers. This introduces trade-offs risking starvation (writers in the first case, readers in the second).
4. **Critical Section Coordination:** The first/last reader must lock/unlock *rw_mutex* to allow writers, while intermediate readers avoid unnecessary synchronization overhead, managed via *mutex* and *read_count*.
5. **Starvation Risk:** Solutions must balance reader and writer access to avoid indefinite delays, as scheduler decisions (e.g., resuming readers or a writer after *signal(rw_mutex)*) can exacerbate this issue.

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

Initial State

- *rw_mutex = 1* (unlocked, available for writers or first reader)
- *mutex = 1* (unlocked, for updating *read_count*)
- *read_count = 0* (no active readers)
- Shared database: Accessible, no process currently using it

1. **wait(rw_mutex)**
   - *rw_mutex = 1* (unlocked), so decrement *rw_mutex*.
   - New state: *rw_mutex = 0* (locked by writer).
   - Writer gains exclusive access to the database.
2. **Writing is performed**
   - Writer modifies the database (e.g., updates a value).
   - State: *rw_mutex = 0*, *mutex = 1*, *read_count = 0*.
3. **signal(rw_mutex)**
   - Increment *rw_mutex*.
   - New state: *rw_mutex = 1* (unlocked).
   - Writer releases exclusive access.
   - Final state: *rw_mutex = 1*, *mutex = 1*, *read_count = 0*.

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

1. **wait(mutex)**
   - *mutex = 1* (unlocked), so decrement *mutex*.
   - New state: *mutex = 0*.
2. **read_count++**
   - *read_count = 0 → 1*.
   - State: *read_count = 1*, *mutex = 0*.
3. **if (read_count == 1) wait(rw_mutex)**
   - *read_count = 1*, condition true.
   - *rw_mutex = 1* (unlocked), so decrement *rw_mutex*.
   - New state: *rw_mutex = 0* (locked by first reader).
4. **signal(mutex)**
   - Increment *mutex*.
   - New state: *mutex = 1*.
   - State: *rw_mutex = 0*, *mutex = 1*, *read_count = 1*.
5. **Reading is performed**
   - Reader 1 reads the database.
   - State unchanged.
6. **wait(mutex)**
   - *mutex = 1*, so decrement *mutex*.
   - New state: *mutex = 0*.
7. **read_count--**
   - *read_count = 1 → 0*.
   - State: *read_count = 0*, *mutex = 0*.
8. **if (read_count == 0) signal(rw_mutex)**
   - *read_count = 0*, condition true.
   - Increment *rw_mutex*.
   - New state: *rw_mutex = 1* (unlocked).
9. **signal(mutex)**
   - Increment *mutex*.
   - New state: *mutex = 1*.
   - Final state: *rw_mutex = 1*, *mutex = 1*, *read_count = 0*.

C program for the first readers-writers problem

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

sem_t rw_mutex, mutex;
int read_count = 0;

void* writer(void* arg) {
    while (1) {
        sem_wait(&rw_mutex);
        printf("Writer is writing\n");
        sem_post(&rw_mutex);
    }
    return NULL;
}
```

```c
void* reader(void* arg) {
    while (1) {
        sem_wait(&mutex);
        read_count++;
        if (read_count == 1)
            sem_wait(&rw_mutex);
        sem_post(&mutex);
        printf("Reader %ld is rea
        sem_wait(&mutex);
        read_count--;
        if (read_count == 0)
            sem_post(&rw_mutex);
        sem_post(&mutex);
    }
    return NULL;
}
```

C program for the first readers-writers problem

```c
int main() {
    pthread_t w, r1, r2;
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);
    pthread_create(&w, NULL, writer, NULL);
    pthread_create(&r1, NULL, reader, (void*)1);
    pthread_create(&r2, NULL, reader, (void*)2);
    pthread_join(w, NULL);
    pthread_join(r1, NULL);
    pthread_join(r2, NULL);
    sem_destroy(&rw_mutex);
    sem_destroy(&mutex);
    return 0;
}
```

Dining Philosophers’ Problem

7.1.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The **dining-philosophers problem** is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

Use this picture to workout different chopsticks pickup scenarios on paper.

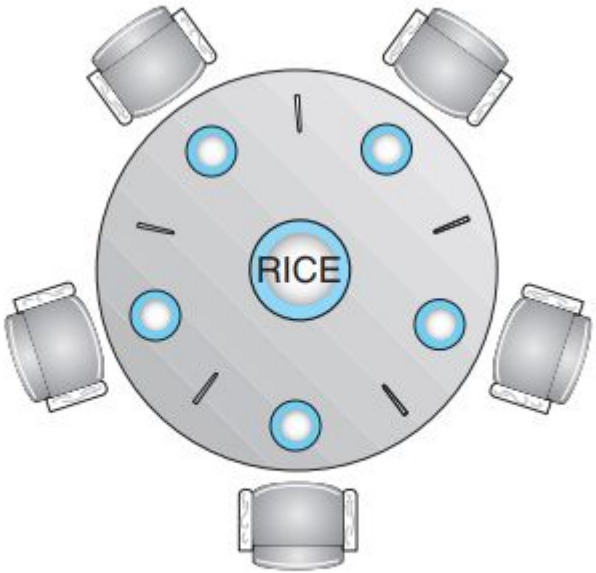

Figure 7.5 The situation of the dining philosophers.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Figure 7.6 The structure of philosopher *i*.

7.1.3.1 Semaphore Solution

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of `chopstick` are initialized to 1. The structure of philosopher *i* is shown in Figure 7.6.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of `chopstick` will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

How Deadlock Happens in the Given Code

The code represents the **naive solution** to the Dining Philosophers problem, where each philosopher:

1. Pick up the left chopstick (`wait(chopstick[i])`).
2. Pick up the right chopstick (`wait(chopstick[(i+1) % 5])`).
3. Eats, then releases both chopsticks (`signal`).

Deadlock Scenario

A deadlock occurs when **all philosophers pick up their left chopstick simultaneously**, leading to:

- Each philosopher holds **one chopstick** and waits indefinitely for the **second one**.
- **Circular dependency**:
  - Philosopher 0 waits for chopstick 1 (held by Philosopher 1).
  - Philosopher 1 waits for chopstick 2 (held by Philosopher 2).
  - ...
  - Philosopher 4 waits for chopstick 0 (held by Philosopher 0).
- **System halts permanently** unless external intervention occurs.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

How Starvation Happens in the Given Code

Starvation occurs when **some philosophers wait indefinitely** while others repeatedly eat.

Starvation Scenario

- Suppose Philosophers 0, 2, and 4 are **fast**, while 1 and 3 are **slow**.
- The fast philosophers might **repeatedly acquire both chopsticks**, while the slow ones **never get a chance**.
- **Unfair scheduling**: The OS scheduler might prioritize certain threads, exacerbating the issue.

Why Starvation Occurs

- **No enforced fairness**: The naive solution lacks a mechanism to ensure all philosophers get turns.
- **Greedy philosophers**: Some may **release and reacquire chopsticks faster**, monopolizing resources.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

How to solve deadlocks in Dining philosophers' problem?

Several possible remedies to the deadlock problem are the following:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.