

Tutoriat 4 (Operating Systems)

Synchronization Examples

Contents

1	Bounded-buffer problem	2
2	Readers-writers problem	3
2.1	Reader priority	3
2.1.1	Writer process	3
2.1.2	Reader process	4
2.2	Writer priority	4
3	Dining-philosophers problem	4
3.1	Semaphore solution	5
3.1.1	Fixing the semaphore solution	6
3.2	Dijkstra/Monitor solution	6
3.3	Real-life analogous	7
4	Barbershop problem	7
4.1	Solution	7
4.2	Queue Based Barbershop problem	8

1 Bounded-buffer problem

Recall that the **bounded-buffer problem** (also known as the **producer-consumer** problem) describes a scenario where two or more threads/processes, known as the **producer process** and the **consumer process**, share a common data structure called the **buffer** where the producer generates data and stores them on the buffer, while the consumer takes data off the buffer and consumes it. The two processes share the following variables:

```
1 #define BUFFER_SIZE 10
2
3 typedef struct {
4     ...
5 } item;
6
7 item buffer[BUFFER_SIZE];
8 int in = 0;
9 int out = 0;
10 int count = 0;
```

The buffer is implemented as a *circular array*. The variable **in** points to the next free position in the buffer, whereas **out** points to the first occupied slot. The variable **count** keeps track of the number of items currently in the buffer. If the buffer is empty, **count == 0**; if the buffer is full, **count == BUFFER_SIZE**.

Producer process

```
1 item next_produced;
2
3 while (true) {
4     /* produce an item, store it in
5      next_produced */
6
7     while (count == BUFFER_SIZE); /* do
8     nothing */
9
10    buffer[in] = next_produced;
11    in = (in + 1) % BUFFER_SIZE;
12    count++;
13 }
```

Consumer process

```
1 item next_consumed;
2
3 while (true) {
4     while (count == 0); /* do nothing */
5
6     next_consumed = buffer[out];
7     out = (out + 1) % BUFFER_SIZE;
8     count--;
9
10    /* consume the item stored in
11    next_consumed */
12 }
```

The issue is that these are two processes that enter their critical sections without following any rules. They execute operations on shared data without a synchronization mechanism. The *count++* and *count--* instructions are critical and can corrupt the state of the buffer due to race conditions. The following shared data structures can solve this issue:

```
1 semaphore mutex = 1;
2 semaphore empty = n;
3 semaphore full = 0;
```

The binary semaphore *mutex* is used to ensure mutual exclusion. If one of the processes wishes to enter its critical section, it must first acquire the "lock" by calling *wait(mutex)*; once that is all set and done, it can enter its critical section and call *signal(mutex)* upon exiting. The counting semaphore *empty* represents the number of **empty** slots in the buffer. The counting semaphore *full* represents the number of items in the buffer. The new implementation can be found below.

The producer process must first wait for the buffer to have at least one empty slot before storing an item; thus, it calls *wait(empty)*. Once that condition is met, it tries to acquire the mutex. After storing an item, it releases the mutex and increments the number of items in the buffer by calling *signal(full)*. Notice the beginning and ending: it decrements *empty* and increments *full*.

The consumer process must first wait for the buffer to have at least one item before being able to consume; thus, it calls *wait(full)*. Once that condition is met, it tries to acquire the mutex. After consuming an item, it releases the mutex and increments the number of empty slots in the buffer by calling *signal(empty)*. In conclusion, it decrements *full* and increments *empty*.

Producer process

```
1 item next_produced;
2
3 while (true) {
4     /* produce an item, store it in
       next_produced */
5
6     wait(empty);
7     wait(mutex);
8
9     /* add next_produced to the buffer */
10
11     signal(mutex);
12     signal(full);
13 }
```

Consumer process

```
1 item next_consumed;
2
3 while (true) {
4     wait(full);
5     wait(mutex);
6
7     /* consume an item from the buffer */
8
9     signal(mutex);
10    signal(empty);
11 }
```

2 Readers-writers problem

The **readers-writers problem** deals with synchronization between multiple threads/processes accessing a shared resource, such as a file or a database. There are two types of processes: **readers** (processes that only **read** the data) and **writers** (processes that both **read** and **modify** the data). While a writer is modifying the data structure, it is necessary to bar other processes from executing, as to prevent them from reading inconsistent data. The proposed synchronization protocol must enforce the following rules:

- **Multiple readers allowed:** any number of readers can be in their critical sections simultaneously.
- **Exclusive writer access:** if a writer is accessing the data, no other process can access it at the same time.

The exclusion pattern here can be called **categorical mutual exclusion**. One process in its critical section does not necessarily exclude other processes, but the presence of one category of processes in the critical section excludes other categories. In terms of priority, there are two versions of this problem:

- **Reader priority:** if a reader is in its critical section, other readers are immediately allowed to enter as well, even if a writer is waiting; the writer must wait for all current readers to finish. **Drawback:** this can lead to **writer starvation**.
- **Writer priority:** once a writer is ready, no new reader may start reading; they must wait until the writer has finished executing in its critical section. **Drawback:** this can lead to **reader starvation**.

The three following variables will be used when designing a solution:

```
1 int readers = 0;
2 mutex = Semaphore(1);
3 empty = Semaphore(1);
```

The **readers** variable keeps track of the amount of readers currently active. The **mutex** variable is a binary semaphore used to increment and decrement **readers**. The **empty** variable is a binary semaphore that is 1 if there are no processes (readers or writers) in their critical sections, and 0 otherwise.

2.1 Reader priority

2.1.1 Writer process

A writer can enter its critical section when there are no other processes executing in their critical sections. The code for the writers would be as follows:

```
1 wait(empty);
2 /* critical section (writing operations) */
3 signal(empty);
```

2.1.2 Reader process

There can be multiple readers executing in their critical sections at the same time. When the first reader wishes to enter its critical section, it must first check with the **empty** semaphore to see if there is already a writer process executing. Once there is no writer active, the reader is allowed to enter its critical section. It uses the **mutex** semaphore to safely increment the variable **readers**. Once the first reader has successfully made its appearance, subsequent readers do not have to check with the **empty** semaphore; as long as there is one reader active, there is no writer, so they only have to increment the variable **readers**.

When a reader exits its critical section, it has to make use of the **mutex** semaphore to decrement the variable **readers**. When the last reader exits its critical section, it has the additional responsibility of updating the **empty** semaphore, signaling that there are no processes left executing in their critical sections, giving the green light for any waiting writer.

```
1  /* 1. increment readers and check if it's the FIRST reader */
2  wait(mutex);
3  readers++;
4  if (readers == 1) {
5      wait(empty);
6  }
7  signal(mutex);
8
9  /* 2. critical section (reading operations) */
10
11 /* 3. decrement readers and check if it's the LAST reader */
12 wait(mutex);
13 readers--;
14 if (readers == 0) {
15     signal(empty);
16 }
17 signal(mutex);
```

This solution can lead to **writer starvation**, since reader processes can always keep entering their critical sections, leaving writer processes to wait indefinitely.

2.2 Writer priority

To ensure that writer processes have priority over the readers, some additional variables must be introduced:

```
1  int readers = 0;
2  int writers = 0;
3
4  rmutex = Semaphore(1);
5  wmutex = Semaphore(1);
6  read = Semaphore(1);
```

test

3 Dining-philosophers problem

Problem definition

Suppose we have 5 philosophers at a table that can either eat or think. Each of them has bowl of rice in front of them, a chopstick to their left and another to their right. There are only five chopsticks on the table. In order to eat, the philosopher has to first pickup both chopsticks (one at a time). A philosopher can't take a chopstick already picked up by someone else. When done eating, the philosopher puts down both chopsticks and goes back to thinking. And the cycle repeats.

Note: The philosopher's can't communicate with each other

Here is the pseudocode for philosopher i:

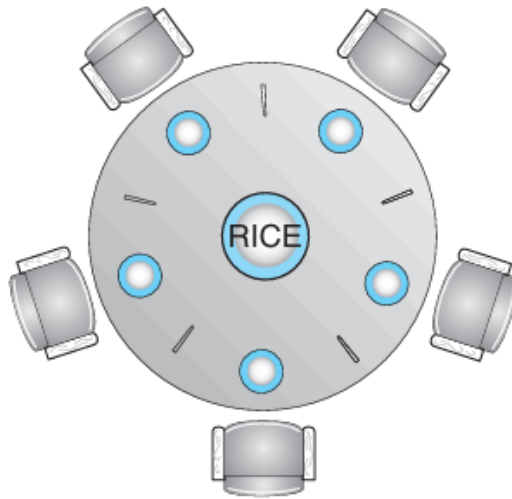


Figure 1: Dining philosophers problem

Listing 1: Pseudocode philosopher i

```

1  while (true){
2      // Pickup in any order
3      pickup(chopsticks[i], chopsticks[(i + 1) % chopstick_count]);
4
5      eat();
6
7      // Putdown in any order
8      putdown(chopsticks[i], chopsticks[(i + 1) % chopstick_count]);
9
10     think();
11 }
12

```

Questions

1. What are the shared resources?
2. What is the critical section?
3. How could we provide mutual exclusion?

3.1 Semaphore solution

This answers exactly question 3, using a semaphore to lock each chopstick before eating and releasing them when done.

Listing 2: Pseudocode philosopher i using semaphores

```

1  while (true){
2      // Take left chopstick, then right one
3      wait(chopsticks[i]);
4      wait(chopsticks[(i + 1) % chopstick_count]);
5
6      eat();
7
8      // Put down chopsticks
9      signal(chopsticks[i]);
10     signal(chopsticks[(i + 1) % chopstick_count]);
11
12     think();
13 }

```

This solution provides mutual exclusion, but gives the possibility of a deadlock if all philosophers pick the left chopstick at the same time.

3.1.1 Fixing the semaphore solution

Resource hierarchy solution

One way of avoiding deadlocks is to create an ordering of resources, making sure the lower index chopstick is picked up before the higher index one.

Asymmetric allocation solution

Another solution is giving different pickup strategies based on the philosopher index. Odd index philosophers pickup left and then right chopstick, even ones pick them in reverse.

Arbitrator solution

We could also impose that both chopsticks are picked up at the same time or none. This is done using a mutex, locking before picking the first one up, releasing after picking the second.

Note: This results in decreased parallelism.

3.2 Dijkstra/Monitor solution

This solution uses a monitor that encapsulates the state of each philosopher (THINKING, HUNGRY, EATING) and a condition variable for each.

Philosopher i picks both chopsticks at the same time only if his neighbors are not eating and if he is hungry, if he can't pick them up he goes to sleep.

When done eating, he puts the chopsticks down and tries to wake up a neighbor.

Listing 3: Monitor pseudocode

```

1
2 monitor DiningPhilosophers
3 {
4     enum {THINKING, HUNGRY, EATING} state [5];
5     condition self [5];
6
7     void pickup (int i) {
8         state[i] = HUNGRY;
9         test(i);
10        if (state[i] != EATING) self[i].wait;
11    }
12
13    void putdown (int i) {
14        state[i] = THINKING;
15
16        // test left and right neighbors
17        test((i + 4) % 5);
18        test((i + 1) % 5);
19    }
20
21    void test (int i) {
22        if ((state[(i + 4) % 5] != EATING) &&
23            (state[i] == HUNGRY) &&
24            (state[(i + 1) % 5] != EATING)) {
25
26            state[i] = EATING;
27            self[i].signal();
28        }
29    }
30
31    initialization_code() {
32        for (int i = 0; i < 5; i++)
33            state[i] = THINKING;
34    }
35 }
```

Listing 4: Philosopher i pseudocode

```

1
2 while (true){
3     DiningPhilosophers.pickup(i);
4
5     eat();
6
7     DiningPhilosophers.putdown(i);
8
9     think();
10 }

```

Final Question: Does any of these solutions guarantee that no philosopher will starve?

3.3 Real-life analogous

Let's say we have a database handling financial transactions between users. A transaction between user A and B needs to lock the rows corresponding to user A and B. If both of the users try to make a transaction of each other a deadlock might arise.

In this case, the philosophers are the user accounts, the chopsticks are the row locks, eating is the actual transaction, thinking could be sending a success/failure message back to the user.

4 Barbershop problem

Note: All of the problems and solutions are not mine, they can be found [here](#)

Problem Statement

A barbershop consists of a waiting room with n chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber

4.1 Solution

Listing 5: Barber pseudocode

```

1
2 while (true){
3     wait(customer)
4     signal(barber)
5
6     cutHair()
7
8     signal(barberDone)
9     wait(customerDone)
10 }

```

Listing 6: Customer pseudocode

```

1
2 while (true){
3     if customerCount == maxCustomerCount
4         continue
5     customerCount ++
6
7     signal(customer)
8     wait(barber)
9
10    getHaircut()
11
12    signal(customerDone)
13    wait(barberDone)
14
15    customerCount --
16
17    liveLifeUntilNextCut()
18 }

```

Question: What is the problem with the solution above and how do we solve it?

4.2 Queue Based Barbershop problem

In the above solution customers are not necessarily chosen in arrival order. How can we modify the solution to accommodate for that?

Listing 7: Barber pseudocode

```

1
2 while (true){
3     wait(customer)
4     signal(queue.pop())
5
6     cutHair()
7
8     signal(barberDone)
9     wait(customerDone)
10 }

```

Listing 8: Customer pseudocode

```

1
2 while (true){
3     if customerCount == maxCustomerCount
4         continue
5
6     customerCount ++
7     mySem = Semaphore(0)
8     queue.add(mySem)
9
10    signal(customer)
11    wait(mySem)
12
13    getHaircut()
14
15    signal(customerDone)
16    wait(barberDone)
17
18    customerCount --
19
20    liveLifeUntilNextCut()
21 }

```