Tutorial 5 - Concurrent programming problems

Gidon Rosalki

2025-04-29

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

1 The general critical section problem

You have n processes $p_0, \dots, p_{n-1}$, and the problem is to implement a general mechanism for entering and leaving critical sections. This has the following success criteria:

1. Mutual Exclusion: Only one process is in the critical section at a time
2. Progress: If processes want to get into the critical section one of them will eventually get into the critical section.
3. Starvation free: No process will wait indefinitely while other processes continuously enter the critical section (also called "Bounded waiting").
4. Generality: It works for n processes, and m processors
5. No blocking in the remainder: No process running outside its critical section may block other processes

1.1 Dining philosophers problem

Philosophers spend their life alternating between eating and thinking. Eating requires 2 chopsticks, and the philosophers, who are all sitting around a circular table, all have one chopstick to the right, and one to the left. So, the order goes:

1. Pick up chopsticks
2. Eat
3. Put down chopsticks
4. Think

The philosophers pick up 1 chopstick at once, and 2 philosophers cannot simultaneously hold a chopstick. We have the following proposed solution: We use chopstick 5 as a semaphore, and initialise `chopstick[5] = 1` and may then carry out the following algorithm:

```
while (true) {
    down(chopstick[(i + 1) % 5)
    down(chopstick[i])
    eat
    up(chopstick[(i + 1) % 5)
    up(chopstick[i])
    think
}
```

However, this can cause deadlocks. Consider when they all simultaneously reach for the left chopstick, then they are all stuck waiting on the right chopstick. This problem demonstrates some of the fundamental limitations of deadlock free synchronisation Here, there is no symmetric solution (no solution where every philosopher has the same code), only asymmetric solutions.
For example, we can execute different code for odd / even, or even randomly, for example Lehmann-Rabin Algorithm. This does not solve the problem, but it does on average solve the problem:

```
repeat
   if coinflip() == 0 then           // randomly decide on a first chopstick
      first = left
   else
      first = right
   end if
   wait until chopstick[first] == false
   chopstick[first] = true           // wait until it is available
   if chopstick[~first] == false then // if second chopstick is available
      chopstick[~first] = true       // take it
      break
   else
      chopstick[first] = false       // otherwise drop first chopstick
   end if
end repeat
eat
chopstick[left] = false
chopstick[right] = false
```

This can also cause deadlocks, thanks to starvation, and there can be a case where progress is not met, but as we increase the number of philosophers, the probability of this tends to 0. On average, given enough time, this code works.
Here, every thread begins by flipping a coin. 0 indicates starting with the left chopstick, and 1 with the right. It waits until the first chopstick is available, and then if the second is not available, it drops the first chopstick, and starts again. If the second is available, then it takes it, eats, and then stops eating releasing the chopstick. This way, if the second chopstick is not available, then it will ensure it is not stopping another philosopher from taking their first chopstick.

1.2 Bounded Buffer problem

Here we have a cyclic buffer that can hold up to n items. There are both *producers* and *consumers* using the buffer. We need to ensure that a producer cannot add data to the buffer if it is full, and a consumer cannot consume data from the buffer if it is empty. Since the buffer is a shared resource, protection is required. We will achieve this using counting semaphores, where the number in the semaphore represents the number of resources of some type.
Our solution will use 3 semaphores:

1. Semaphore `mutex` initialised to the value 1. This protects the buffer (only one thread may access it at once)
2. Semaphore `fill_count` initialised to the value 0. This indicates how many items in the buffer are available to be read
3. Semaphore `empty_count` initialised to the value n. This indicates how many items in the buffer are available to be written.

We then have the following producer, and consumer, code:

Producer:

```
while (true) {
   produce_item();
   down (emptyCount);
   down (mutex);

   add_item();
   up (mutex);
   up (fillCount);
}
```

Consumer:

```
while (true) {
   down (fillCount);
   down (mutex);
   pop_item();

   up (mutex);
   up (emptyCount);
   consume_item();
}
```

1.3 Reader-writers problem

A data structure is shared among a number of concurrent processes:

- Readers that only read the data, and do not perform updates
- Writers that can both read **and** write

One must allow multiple readers to read at the same time, but only **one** writer can access the shared data at any time. While a writer is writing to the data structure, then the readers should be blocked from reading. This follows since we don't want the data to change while reading is occurring, or for one writer to be working with outdated data.

1.3.1 Solution 1

We use the following shared data:

- An integer `read_count` initialised to 0. This represents the number of readers
- Semaphore `read_count_mutex`, initialised to 1. This protects read_count
- Semaphore `write` initialised to 1. This makes sure that the writer doesn't use data at the same time as any readers.

Writer:

```
while (true) {
    down (write);
    perform_writing();
    up (write);
}
```

Reader:

```
while (true)
    {
        down (read_count_mutex);
        read_count++;
        if (read_count == 1)    // The first reader
            blocks writing
            {
                down (write);   // lock from writers
            }
        up (read_count_mutex);

        perform_reading ();     // CS

        down (read_count_mutex);
        read_count--;
        if (read_count == 0)    // The last reader
            unblocks writing
            {
                up (write);
            }
        up (read_count_mutex);
    }
```

However, this solution is imperfect. What if a writer is waiting to write, but there are always readers that are reading? The `write` semaphore will thus never be released, and the writer will thus never be able to write. Writers are therefore subject to starvation.

1.3.2 Solution 2 - Writer priority

We use extra semaphores, and variables:

- Semaphore `read` initialised to 1. This inhibits readers when a writer wants to write.
- Integer `write_count` initialised to 0. This counts waiting writers.
- Semaphore `write_count_mutex` initialised to 1. This controls the updating of `write_count`
- `queue` semaphore, initialised to 1, and used only in the reader.

The writer:

```
while (true)
   {
      down (write_count_mutex);
      write_count++; // Counts number of waiting
          writers
      if (write_count == 1)
         {
            down (read);
         }
      up (write_count_mutex);

      down (write);
      write ();     // Writing is performed -one
          writer at a time
      up (write);

      down (write_count_mutex);
      write_count--;
      if (write_count == 0)
         {
            up (read);
         }
      up (write_count_mutex);
   }
```

The reader:

```
while (true)
   {
      down (queue);
      down (read);
      down (read_count_mutex);
      read_count++;
      if (read_count == 1)
         {
            down (write);
         }
      up (read_count_mutex);
      up (read);
      up (queue);

      read ();

      down (read_count_mutex);
      read_count--;
      if (read_count == 0)
         {
            up (write);
         }
      up (read_count_mutex);
   }
```

There are now some questions from previous exams. They are not replicated here. It is important to remember that questions on critical sections, and all this content is common exam content.