Classical synchronization problems

Thang Nguyen Manh 20235426[1] **, Bao Nguyen Ngoc Gia 20235268**[2]

[1,2]School of Information and Communication Technology,
Hanoi University of Science and Technology,
[1]thang.nm235426@sis.hust.edu.vn,
[2]bao.nng235268@sis.hust.edu.vn

Abstract

This report investigates two classical synchronization problems in operating system: the Readers-Writers problem and the River Crossing problem. We present the historical origins and content of each problem, and various methods using semaphore-based algorithms. The analysis aims to provide a comprehensive understanding of these problems.

Keywords: synchronization problems, reader-writer problem, river crossing problem, semaphore.

1 Readers-writers problem[1]

1.1 Introduction

There is a situation in all computer systems that is: we have a database that can be accessed by a variety of processes or threads. However, there are only two types of job for these processes or threads: **reading** the database or **updating** the database. Throughout this report, we will refer to the reading processes as **readers** and the writing processes as **writers**. On the one hand, if there are a number of readers reading the database simultaneously, it will not have problems. On the other hand, if a writer updating the database and other readers or writers access the database simultaneously, chaos may ensue. Now, we must synchronize these processes: readers and writers to ensure that these difficulties do not arise. This synchronization problem is referred to as the **readers–writers problem**.

To be specific, the synchronization constraints are:

1. Any number of readers can be in the critical section (database) simultaneously when there was already a reader in the critical section. And a reader cannot access the critical section while a writer accesses.

2. Writers must have exclusive access to the critical section, no other processes either writer or reader may enter. And a writer cannot access the critical section while a reader accesses.

Based on these constraints, the readers–writers problem has several methods to solve, all involving priorities, such as: reader preference solution, writer preference solutions. The reader preference solution will be presented in **1.2 Methods** 1.2. However, the two solutions could lead to starvation for either the reader or the writer. For this reason, we also present a starvation-free solution to solve this problem.

1.2 Methods

In this section, we will present two solutions to address the readers–writers problem.

1.2.1 Reader preference solution

The idea of this solution is that no reader should wait for other readers to finish simply because a writer is waiting. To be more specific, when a reader is working in the critical section, other readers will also work simultaneously, whether the writers are waiting or not. When a writer is working in the critical section, other readers have to wait and it will implement the FIFO order to decide which is the next process. If the writers come first, it will be the next process and vice versa.

The variables and pseudocode will be demonstrated below:

Variables: Here is a set of variables that is sufficient to solve the problem.

```
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```

Variables

The `readers` variable is used to keep track how many readers in the critical section, and the semaphore variable `mutex` is used to ensure that there is only a reader could modify the `readers` variable at one time. `mutex` is 1 if there is no process using `readers`, and less than or equal to 0 otherwise. The semaphore variable `roomEmpty` is used to ensure that chaos cannot ensue. To be more specific, when a writer is in the critical section, there is no other process entering the critical section. Similarly, when readers is in the critical section, there is no writer process entering the critical section.

Pseudocode: Here is the pseudocode demonstrating the reader preference solution, it includes the writer solution and reader solution.

```
roomEmpty.wait()
...
// critical section for writers
...
roomEmpty.signal()
```

Writer solution

The writer solution is quite simple; if there is no other process/thread in the critical section, a writer process could enter the critical section. Of course, other processes can not access the critical section while a writer is working. Because the semaphore variable `roomEmpty` now is less than or equal to 0, other processes will be waited until the variable is 1 (a writer exits the critical section).

```
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()       # first in locks
mutex.signal()
...
// critical section for readers
...
mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()     # last out unlocks
mutex.signal()
```

Reader solution

The reader solution is a bit more complicated more than the writer solution. We will keep track of the number of readers in the room so that we can give a special assignment to the first to arrive and the last to leave. If the `roomEmpty` is less than or equal to 0, the first reader that arrives has to wait for `roomEmpty`. And other readers will be blocked by `mutex`, not by `roomEmpty`; writers will still be blocked by `roomEmpty`. Otherwise, if the `roomEmpty` is 1, then the reader proceeds and, at the same time, bars writers. Subsequent readers can still enter because none of them will try to wait on `roomEmpty`. After the critical section, the last reader to leave the room signals `roomEmpty`, possibly allowing a waiting writer to enter.

1.2.2 Starvation-free solution

The idea of this solution is that it will implement the FIFO order to decide which is the next process. To be more precise, when the critical section is occupied, if a writer is waiting, other processes both writers and readers that come after the writer will be waited until the writer finishes in the critical section. Otherwise, if other readers come first, they could still work simultaneously in the critical section while there was already a reader in the critical section.

The variables and pseudocode will be demonstrated below:

Variables: Here is a set of variables that is sufficient to solve the problem.

```
int readers
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
waitingRoom = Semaphore(1)
```

Variables

The `readers` variable is used to keep track how many readers in the critical section, and the semaphore variable `mutex` is used to ensure that there is only a reader could modify the `readers` variable at one time. `mutex` is 1 if there is no process using `readers`, and less than or equal to 0 otherwise. The semaphore variable `roomEmpty` is used to ensure that chaos cannot ensue. To be more specific, when a writer is in the critical section, there is no other process entering the critical section. Similarly, when readers is in the critical section, there is no writer process entering the critical section. Here, we also added a semaphore variable `waitingRoom`, both readers and writers need to pass the variable `waitingRoom` to access the critical section. The `waitingRoom` will ensure that if a writer is waiting, other processes both writers and readers that come after the writer will be waited until the writer finishes in the critical section (no starvation).

Pseudocode: Here is the pseudocode demonstrating the starvation free solution, it includes the writer solution and reader solution

```
waitingRoom.wait()
    roomEmpty.wait()
waitingRoom.signal()
...
// critical section for writers
...
roomEmpty.signal()
```

Writer solution

```
waitingRoom.wait()
waitingRoom.signal()

mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()      // first in locks
mutex.signal()
```

```
...
// critical section for readers
...
mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()    // last out unlocks
mutex.signal()
```

Reader solution

If a writer is waiting, it will be blocked by `roomEmpty`, and the `waitingRoom` will locked. Thus, other processes, both readers and writers, will be queued at the `waitingRoom`. When the writer accesses the critical section, the `waitingRoom` will be unlocked. Then, the next processes continue to pass, if these are readers, it will work like the *Reader solution* in the **1.2.1 Reader preference solution** 1.2.1. If the next process is another writer, it will keep locking the `waitingRoom`, other processes will queue at the `waitingRoom`.

2 River crossing problem[2]

2.1 Introduction

This is from a problem set written by Anthony Joseph at U.C. Berkeley. Somewhere near Redmond, Washington there is a rowboat that is used by both Linux **hackers** and Microsoft employees (**serfs**) to cross a river. The ferry can hold exactly **four** people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

As each thread boards the boat it should invoke a function called `board`. It may be understood as each thread will prepare its 'luggage' before `rowBoat`. After all four threads have invoked `board`, exactly one of them should call a function named `rowBoat`, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

2.2 Methods

In this section, we will present a solution to address the river crossing problem. The basic idea of this solution is that each arrival updates one of the counters and then checks whether it makes a full complement, either by being the fourth of its kind or by completing a mixed pair of pairs.

Variables Here is a set of variables that is sufficient to solve the problem.

```
mutex = Semaphore(1)
hackers = 0
serfs = 0
barrier = Barrier(4)
hackerQueue = Semaphore(0)
serfQueue = Semaphore(0)
isCaptain = false
```

`hackers` and `serfs` count the number of hackers and serfs waiting to board. Since they are both protected by `mutex`, we can check the condition of both variables without worrying about an untimely update.

`hackerQueue` and `serfQueue` allow us to control the number of hackers and serfs that pass.

The `barrier` is an object, that contains: `int` variables and `semaphore` variables. It is used to ensure the requirement: 'no thread executes the critical section until after all threads are ready to execute the critical section' is satisfied. It means these threads will be entered the critical section, if the number of waiting threads is enough; in this problem it is four. Thus, the `barrier` makes sure that all four threads have invoked `board` before the captain invokes `rowBoat`. The code for `barrier` will be desmonstrated in the **Appendix A** 3.1.

`isCaptain` is a local variable that indicates which thread should invoke `rowBoat`.

Pseudocode Here is the pseudocode demonstrating the reader preference solution, it includes the hacker solution and serf solution.

```
# Begin - request access to shared region
mutex.wait()
# Increase the number of waiting hackers
     hackers += 1
# If 4 hackers form a valid group
     if hackers == 4:
# Signal 4 hackers in the queue to board the boat
         hackerQueue.signal(4)
# Reset hacker counter
         hackers = 0
# Current hacker becomes the captain
         isCaptain = True
# If there are 2 hackers and 2 serfs
     elif hackers == 2 and serfs >= 2:
# Signal 2 hackers
         hackerQueue.signal(2)
# Signal 2 serfs
         serfQueue.signal(2)
```

```
# Reset hacker counter
        hackers = 0
# Decrease number of waiting serfs by 2
        serfs -= 2
# Current hacker becomes the captain
        isCaptain = True

    else:
# Not enough passengers, release the mutex for others
        mutex.signal()
# This hacker waits until the captain signals boarding
hackerQueue.wait()
# Mark that the hacker has boarded the boat
board()
# Wait for all 4 passengers to board
barrier.wait()
# If this hacker is the captain, row the boat
if isCaptain:
    rowBoat()
# Captain releases the mutex after rowing
    mutex.signal()
```

Hacker solution

As each thread files through the mutual exclusion section, it checks whether a complete crew is ready to board. If so, it signals the appropriate threads, declares itself captain, and holds the `mutex` in order to bar additional threads until the boat has sailed.
The barrier keeps track of how many threads have boarded. When the last thread arrives, all threads proceed. The captain invoked `row` and then (finally) releases the `mutex`.
The code for serfs is completely symmetric.

```
# Begin - request access to shared region
mutex.wait()
# Increase the number of waiting serfs
    serfs += 1
# If 4 serfs form a valid group
    if serfs == 4:
# Signal 4 serfs in the queue to board the boat
        serfQueue.signal(4)
# Reset serf counter
        serfs = 0
# Current serf becomes the captain
```

```
            isCaptain = True
# If there are 2 serfs and 2 serfs
        elif serfs == 2 and hackers >= 2:
# Signal 2 serfs
            serfQueue.signal(2)
# Signal 2 serfs
            hackerQueue.signal(2)
# Reset serf counter
            serfs = 0
# Decrease number of waiting hackers by 2
            hackers -= 2
# Current serf becomes the captain
            isCaptain = True

        else:
# Not enough passengers, release the mutex for others
            mutex.signal()
# This serf waits until the captain signals boarding
serfQueue.wait()
# Mark that the serf has boarded the boat
board()
# Wait for all 4 passengers to board
barrier.wait()
# If this serf is the captain, row the boat
if isCaptain:
    rowBoat()
# Captain releases the mutex after rowing
    mutex.signal()
```

Serf solution

3 Appendix

3.1 Appendix A: Barrier object

Here we will use Python syntax defining the class:

```
class Barrier:
    def __init__(self, n):
        self.n = n
        self.count = 0
        self.mutex = Semaphore(1)
        self.turnstile = Semaphore(0)
```

```
        self.turnstile2 = Semaphore(0)

    def phase1(self):
        self.mutex.wait()
        self.count += 1
        if self.count == self.n:
            self.turnstile.signal(self.n)
        self.mutex.signal()
        self.turnstile.wait()

    def phase2(self):
        self.mutex.wait()
        self.count -= 1
        if self.count == 0:
            self.turnstile2.signal(self.n)
        self.mutex.signal()
        self.turnstile2.wait()

    def wait(self):
        self.phase1()
        self.phase2()
```

Barrier class

The `init` method runs when we create a new Barrier object, and initializes the instance variables. The parameter `n` is the number of threads that have to invoke wait before the Barrier opens.

The variable `self` refers to the object the method is operating on. Since each barrier object has its own `mutex` and `turnstiles`, `self.mutex` refers to the specific `mutex` of the current object.

Here is an example that creates a Barrier object and waits on it:

```
barrier = Barrier(n)       # initialize a new barrier
barrier.wait()             # wait at a barrier
```

References

[1] Allen B. Downey, *The Little Book of Semaphores version 2.2.1*, Green Tea Press, 2016.

[2] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts 10th edition*, John Wiley & Sons, Inc, 2018.

[3] Andrew S. Tanenbaum, Herbert Bos, *Modern Operating Systems 4th edition*, Pearson Education, 2014.