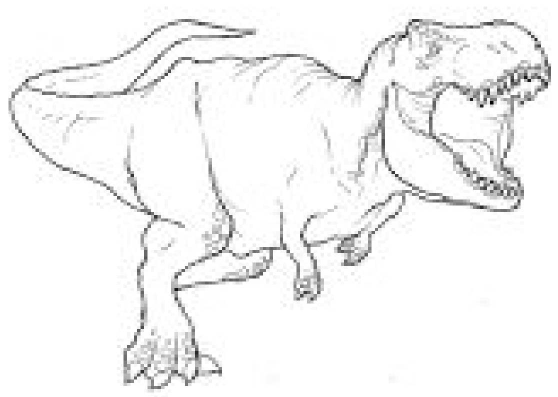


OPERATING SYSTEM

PROCESS SYNCHRONIZATION PROBLEM

Summarize

Details some process synchronization problems that popular.

Guide Teacher

DR. Pham Dang Hai

Student

Dam Minh Tien

20156599

INTRODUCTION

In computer science, **synchronization** refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. *Process synchronization* refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. *Data synchronization* refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization.

This document will intro some sychronization process problems:

- Railways in the Andes, a practical problem.

(Jerry Breecher – CS3013 – Operating Systems Process Synchronization -page 25)

- The Sleeping Teaching Assistant problem.

(Abraham Silberschatz, Peter Baer Galvin, Greg Gagne – Operating System Concepts 9th edition, page 251)

- The Cigarette Smokers' problem

(David Lorge. Parnas Carnegie Mellon University - On a solution to the cigarette smokers' problem)

- The Santa Claus problem

(William Stallings. Operating Systems: Internals and Design Principles. Prentice Hall, sixth edition)

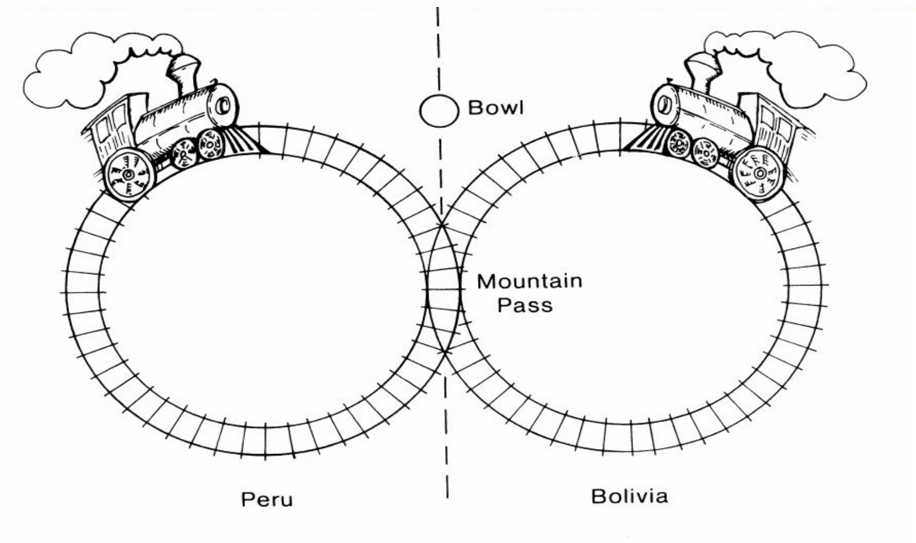
- Building H20 problem

(Gregory R. Andrews. Concurrent Programming: Principles and Practice. Addison-Wesley, 1991.)

1. **Railways in the Andes, a practical problem.**
    - Problem:

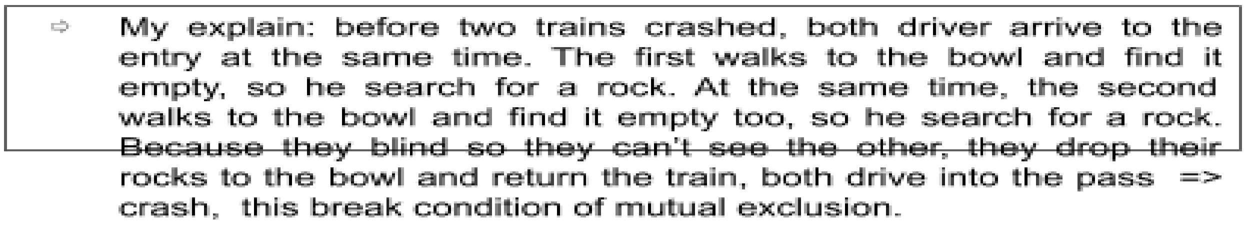
High in the Andes mountains, there are two circular railway lines. One line is in Peru, the other in Bolivia. They share a common section oftrack where the lines cross a mountain pass that lies on the internationalborder (near Lake Titicaca?).

Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the common section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains areboth blind and deaf, so they can neither see nor hear each other.



The two drivers agreed on the following method of preventing collisions. They set up a large bowl at the entrance to the pass. Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see it it contains a rock. If the bowl is empty, the driver finds a rock and drops it in the bowl, indicating that his train is entering the pass; once his train has cleared the pass, he must walk back to the bowl and remove his rock, indicating that the pass in no longer being used. Finally, he walks back to the train and continues down the line. If a driver arriving at the pass finds a rock in the bowl, he leaves the rock there; he repeatedly takes a siesta and rechecks the bowl until he finds it empty. Then he drops a rock in the bowl and drives his train into the pass. A smart graduate from the University of La Paz (Bolivia) claimed that subversive train schedules made up by Peruvian officials could block the train forever.

The Bolivian driver just laughed and said that could not be true because it never happened. Unfortunately, one day the two trains crashed.



⇨ My explain: before two trains crashed, both driver arrive to the entry at the same time. The first walks to the bowl and find it empty, so he search for a rock. At the same time, the second walks to the bowl and find it empty too, so he search for a rock. Because they blind so they can't see the other, they drop their rocks to the bowl and return the train, both drive into the pass => crash, this break condition of mutual exclusion.

This is an example of the test and set problem when the test is separated from the set.

Following the crash, the graduate was called in as a consultant to ensure that no more crashes would occur. He explained that the bowl was being used in the wrong way. The Bolivian driver must wait at the entry to the pass until the bowl is empty, drive through the pass and walk back to put a rock in the bowl. The Peruvian driver must wait at the entry until the bowl contains a rock, drive through the pass and walk back to remove the rock from the bowl. Sure enough, his method prevented crashes.

Prior to this arrangement, the Peruvian train ran twice a day and the Bolivian train ran once a day. The Peruvians were very unhappy with the new arrangement.

- ⇨ My solution: Use Dekker Algorithm to shown the prioriti process. Because, there are only
two process & one section so the algorithm will be not too complex.

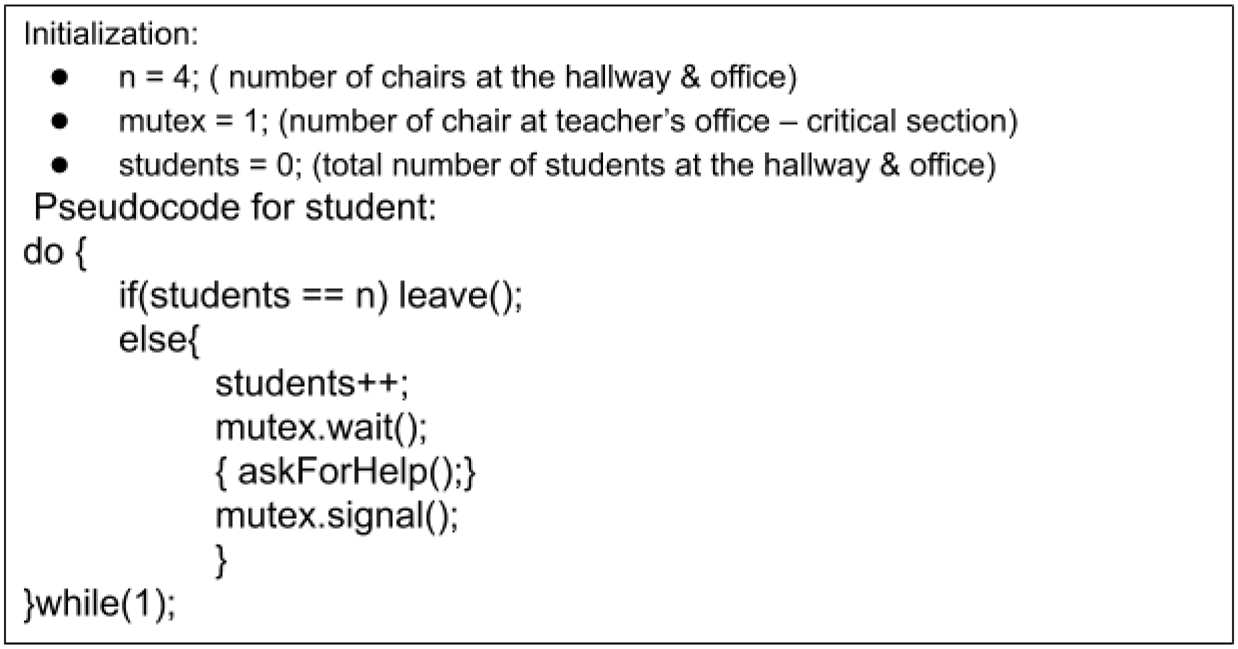
Explain The graduate was called in again and was told to prevent crashes while avoiding the problem of his previous method. He suggested that two bowls be used, one for each driver. When a driver reaches the entry, he first drops a rock in his bowl, then checks the other bowl to see if it is empty. If so, he drives his train through the pass. Stops and walks back to remove his rock. But if he finds a rock in the other bowl, he goes back to his bowl and removes his rock. Then he takes a siesta, again drops a rock in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty. This method worked fine until late in May, when the two trains were simultaneously blocked at the entry for many siestas.

- ⇨ My explain: This method of the graduate worked like the lock-in method (with two lock in this example).
- ⇨ Suppose both drive arrive to the entry of the pass at the same time. Both drop rock into the their bowl at the same time and check the other bowl, because both bowl contain a rock so no one can drive into the pass and must remove a rock of their bowl then take siestas.

If the supposition loops many time, two train locked at the entry for many siestas and this
break condition of progress.

2. **The Sleeping Teaching Assitant**:

The Sleeping Teaching Assitant is a variation of **The barbershop problem** appears in Silberschatz and Galvin's Operating Systems Concepts. A university computer sicence department has a teaching assistan (TA) who helps undergraduate students with their programing assignments during regular office hours. The TA's office is rather small and has room for only one desk with a chair and computer. There are three chairs in the hallway outside the office where student can sit and wait if the TA is currently helping another student. When there are no student who need help during office hours, the TA sits at the desk and takes a nap. If a student arrives during office hours and finds the TA sleeping, the student must awaken the TA to ask for help. If a student arrives and finds the TA currently helping another student, the student sits on one of the chairs in the hallway and waits. If no chair are available, the student will come back at a later time.



Initialization:

- n = 4; (number of chairs at the hallway & office)
- mutex = 1; (number of chair at teacher's office – critical section)
- students = 0; (total number of students at the hallway & office)

Pseudocode for student:

```
do {
        if(students == n) leave();
        else{
                students++;
                mutex.wait();
                { askForHelp();}
                mutex.signal();
                }
}while(1);
```

3. **The Cigarette Smokers' problem:**
    - Introduction:

  In a widely circulated and referenced memorandum , Suhas Patil has introduced a synchronization problem entitled "The Cigarette Smokers's Problem". He claims that the problem cannot be solved using the P and V primitives introduced by Dijkstra unless conditional statements are also used. He supports that claim with an elaborate proof in terms of Petri Nets. On the basis of that proof, Patil concludes that the P and V primitives are not sufficiently powerful and that more complex operations are needed. In this paper we present a solution to the Cigarette Smokerk's Problem, discuss the "flaw" in Patil1's proof, and discuss the need for additional operations.

    - Problem:

Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches. There are three smokers around a table, each of whom has an infinite supply of *one* of the three ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.

There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table. The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while. Once the smoker has finished his cigarette, the agent places two new random items on the table. This process continues forever.

Three semaphores are used to represent the items on the table; the agent increases the appropriate semaphore to signal that an item has been placed on the table, and smokers decrement the semaphore when removing items. Also, each smoker has an associated semaphore that they use to signal to the agent that they are done smoking; the agent has a process that waits on each smoker's semaphore to let it know that it can place the new items on the table.

A simple pseudocode implementation of the smoker who has the supply of tobacco might look like the following:

```
def tobacco_smoker():
    repeat:
        paper.wait()
        matches.wait()
        smoke()
        tobacco_smoker_done.signal()
```

However, this can lead to deadlock; if the agent places paper and tobacco on the table, the smoker with tobacco may remove the paper, leaving the smoker with matches unable to make their cigarette. The problem is to define additional processes and semaphores that prevent deadlock, without modifying the agent.

4. The Santa Claus problem:

- Introduction:

This problem is from William Stallings's Operating Systems , but he attributes it to John Trono of St. Michael's College in Vermont.

- Problem:

(Original problem refer at William Stallings's Operating Systems, page 261)

Stand Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

Here are some addition specifications:

- After the ninth reindeer arrives, Santa must invoke prepareSleigh, and then all nine reindeer must invoke getHitched.
- After the third elf arrives, Santa must invoke helpElves. Concurrently, all three elves should invoke getHelp.
- All three elves must invoke getHelp before any additional elves enter (increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

5. Building H2O problem:

- Introduction:

This problem has been a staple of the Operating Systems class at U.C. Berkeley for at least a decade. It seems to be based on an exercise in **Andrews's Concurrent Programming**.

- Problem:

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed. As each thread passes the barrier, it should invoke bond. You must guarantee that all the threads from one molecule invoke bond before any of the threads from the next molecule do. In other words:

• If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.

• If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

We don't have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets; thus, if we examine the sequence of threads that invoke bond and divide them into groups of three, each group should contain one oxygen and two hydrogen threads. Puzzle: Write synchronization code for oxygen and hydrogen molecules that enforces these constraints.

- Solution:

Here are the variables I used in my solution:

```
mutex = Semaphore (1)

oxygen = 0
```

```
hydrogen = 0
barrier = Barrier (3)
oxyQueue = Semaphore (0)
hydroQueue = Semaphore (0)
```

Oxygen and hydrogen are counters, protected by mutex. barrier is where each set of three threads meets after invoking bond and before allowing the next set of threads to proceed. oxyQueue is the semaphore oxygen threads wait on; hydroQueue is the semaphore hydrogen threads wait on. I am using the naming convention for queues, so oxyQueue.wait() means “join the oxygen queue” and oxyQueue.signal() means “release an oxygen thread from the queue.

Initially hydroQueue and oxyQueue are locked. When an oxygen thread arrives it signals hydroQueue twice, allowing two hydrogens to proceed. Then the oxygen thread waits for the hydrogen threads to arrive. <Oxygen code>

```
mutex . wait () ;
oxygen += 1 ;
if hydrogen >= 2:
    hydroQueue . signal (2)  ;
    hydrogen -= 2 ;
    oxyQueue . signal ()  ;
    oxygen -= 1  ;
else :
    mutex . signal () ;
oxyQueue . wait () ;
```

```
bond ();
barrier . wait () ;
mutex . signal () ;
```

As each oxygen thread enters, it gets the mutex and checks the scoreboard. If there are at least two hydrogen threads waiting, it signals two of them and itself and then bonds. If not, it releases the mutex and waits. After bonding, threads wait at the barrier until all three threads have bonded, and then the oxygen thread releases the mutex. Since there is only one oxygen thread in each set of three, we are guaranteed to signal mutex once. The code for hydrogen is similar:

```
<Hydrogen code>
mutex . wait () ;
hydrogen += 1 ;
if hydrogen >= 2 and oxygen >= 1:
    hydroQueue . signal (2)
    hydrogen -= 2 ;
    oxyQueue . signal () ;
     oxygen -= 1 ;
else :
    mutex . signal () ;
hydroQueue . wait () ;
bond ();
barrier . wait ();
```

An unusual feature of this solution is that the exit point of the mutex is ambiguous. In some cases, threads enter the mutex, update the counter, and exit the mutex. But when a thread arrives that forms a complete set, it has to keep the mutex in order to bar subsequent threads until the current set have invoked bond.

After invoking bond, the three threads wait at a barrier. When the barrier opens, we know that all three threads have invoked bond and that one of them holds the mutex. We don't know which thread holds the mutex, but it doesn't matter as long as only one of them releases it. Since we know there is only one oxygen thread, we make it do the work.

This might seem wrong, because until now it has generally been true that a thread has to hold a lock in order to release it. But there is no rule that says that has to be true. This is one of those cases where it can be misleading to think of a mutex as a token that threads acquire and release.

Bibliograpphy:

- Allen B. Downey - The Little Book of Semaphores - Version 2.2.1 http://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf (Most used)

- Jerry Breecher – CS3013 – Operating Systems Process Synchronization - https://web.cs.wpi.edu/~cs3013/c07/lectures/Section06-Sync.pdf

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne – Operating System Concepts 7th edition http://www.uoitc.edu.iq/images/documents/informatics-institute/exam_materials/operating_system_concepts.pdf

- David Lorge. Parnas Carnegie Mellon University - On a solution to the cigarette smokers' problem http://repository.cmu.edu/cgi/viewcontent.cgi?article=2992&context=compsci

- William Stallings. Operating Systems: Internals and Design Principles. Prentice Hall, sixth edition http://tandon-books.com/Computer%20Science/CS6233%20-%20Introduction%20to%20Operating%20Systems/(CS6233)%20Stallings%20-%20Operating%20Systems%20Internals%20and%20Design%20Principles%206e.pdf

- Gregory R. Andrews. Concurrent Programming: Principles and Practice. Addison-Wesley, 1991. https://drive.google.com/file/d/0B_HSa629bKypcUtYX2NGWWZFZE0/view?usp=sharing)