# COMP 346 – FALL 2020
Tutorial # 3

## SYNCHRONIZATION

## TOPICS

- Synchronization in Details
- Examples

## SYNCHRONIZATION

- Improper synchronization leads to
  - a corruption to shared file / object / data.
  - Deadlock , one process holds a lock for long long time while other process is waiting for that lock.
- Proper synchronization helps to avoid race conditions.
- Our goal is to
  - Synchronize between process and
  - Maximize concurrency.
  - Make the CPU busy doing useful tasks most of the time

## Critical sections Introduction

- Objectives of synchronization.
  - To ensure that a set of concurrent processes execute in a proper sequence.
  - To ensure that the shared resources are orderly accessed by the concurrent processes.
    - The execution cycle of a process becomes dependent on the behavior of the other processes.
    - Unordered accesses may cause a race condition.
  - To ensure that the resources remain in a consistent state following sequences of concurrent accesses.

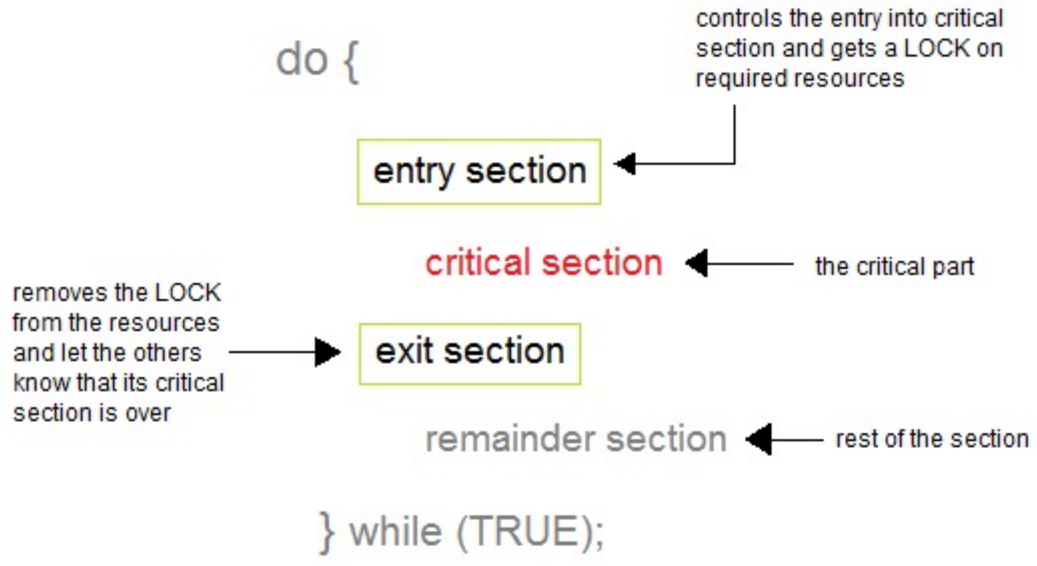
## Problems related to synchronization.

- Mutual exclusion.
- Deadlocks.
- Indefinite waiting (starvation).

## Critical sections Elements

- Entry section.
  - Process requests permission to access its critical section.
  - Process must wait its turn to enter in the critical section.
- Critical section.
  - Process executes code for its critical section.
- Exit section.
  - Process signals the completion of its critical section.
  - Another process will now be permitted to enter in the critical section.
- Remainder section.
  - Process is executing code that is independent of shared resources.

# Critical sections Elements


do {
entry section
controls the entry into critical section and gets a LOCK on required resources
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

## Critical sections Requirements

- Mutual exclusion.
  - Ensures that only one process at a time is executing in the critical section.
  - A process should remain in the critical section only for the finite time that is required to access the shared resource.
- Progress.
  - A new process should be selected to enter in the critical section as soon as the critical section becomes free.
  - The selection of the next process to enter in the critical section should involve only the waiting processes but no others.
- •Bounded waiting.
  - A waiting process should not wait indefinitely to enter in the critical section.
  - A process should wait only after a finite number of other processes to enter in the critical section.
  - No assumption should be made about the relative speed of the processes and the number of processors to determine the moment that a process will enter in the critical section.

UNIVERSITÉ Concordia UNIVERSITY

## What's the Problem?

Example 1:

- A cup of coffee
- A "pourer" (producer)
- A "drinker" (consumer)

```
Pourer:
while (true)
{
        pour();
   }
```

```
Drinkers:
while (true)
{
        drink();
   }
```

## What's the Problem?

Example 1:

- A cup of coffee
- A "pourer" (producer)
- A "drinker" (consumer)

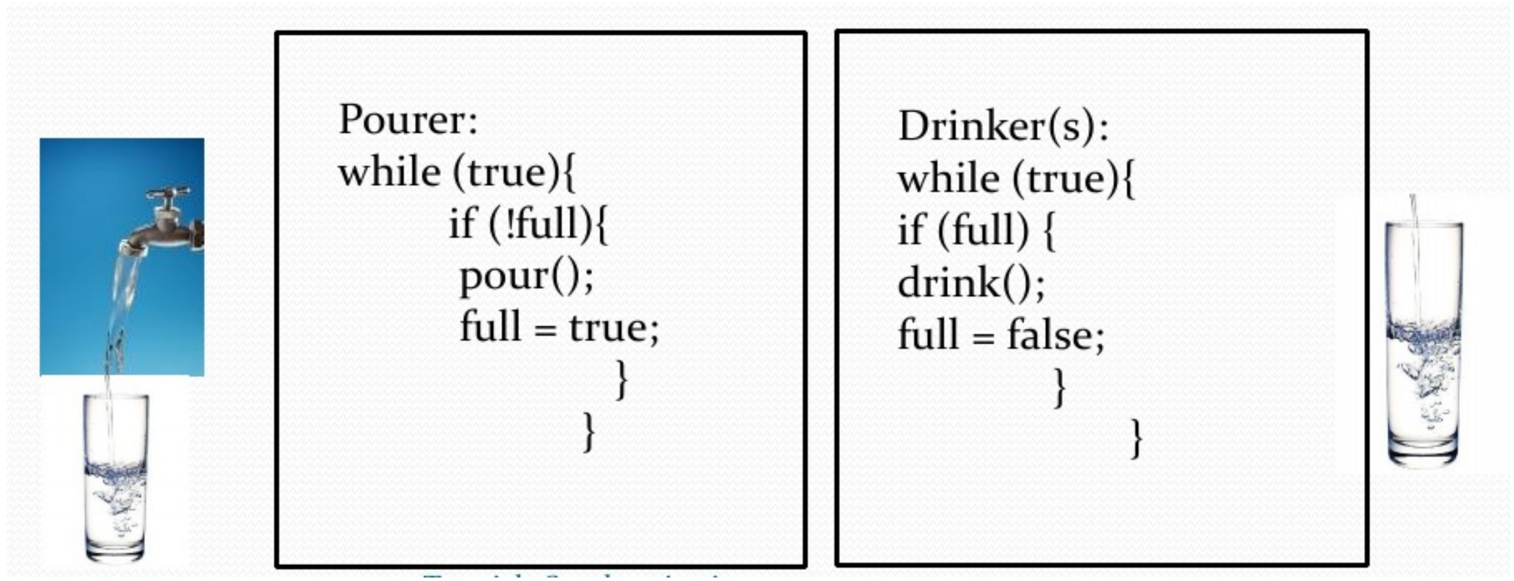
```
pourer:
while (true)
{
     pour();
 }
```

```
Drinkers:
while (true)
{
        drink();
   }
```

**Result: A mess!**

# So, what do we want?

- Simple Protocol:
- Pourer pours only when cup is empty
- Drinker drinks only when cup is full

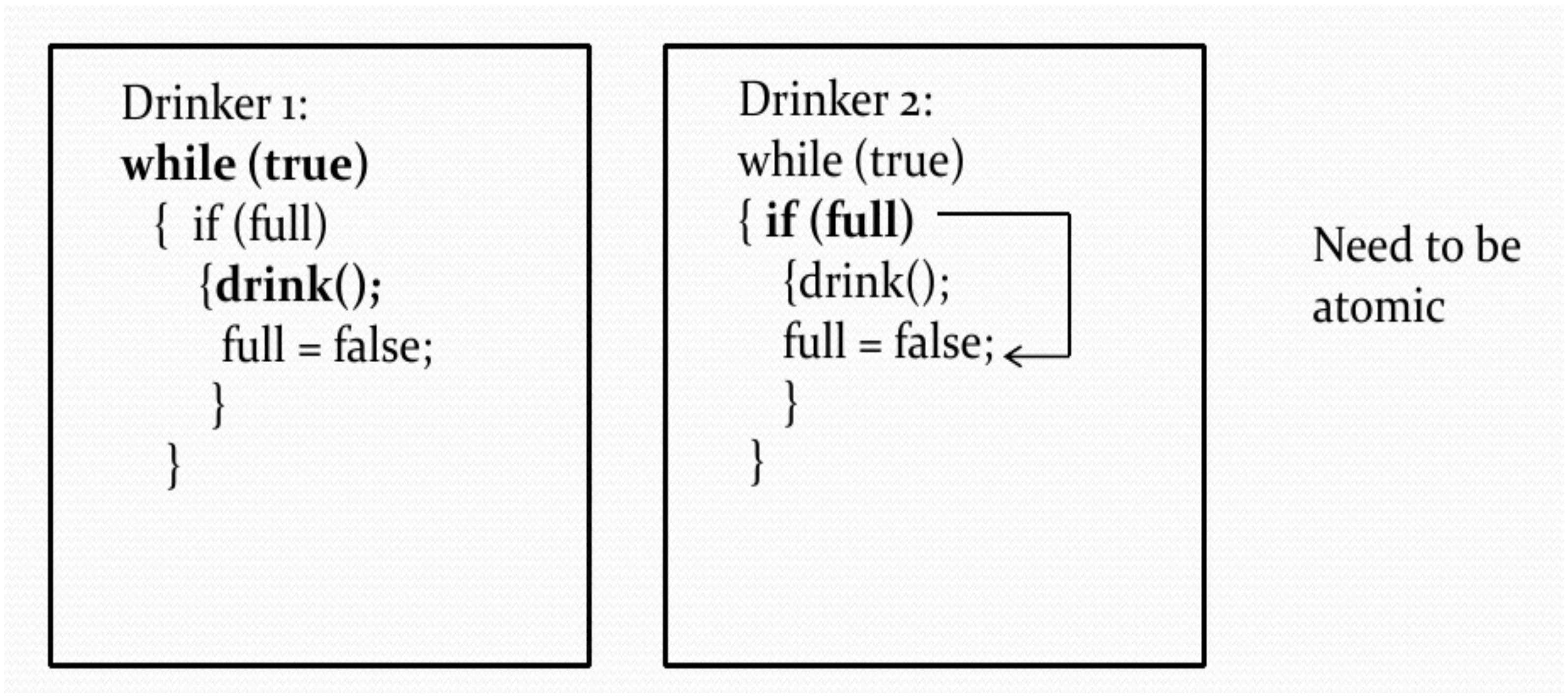


```
Pourer:
while (true){
        if (!full){
         pour();
         full = true;
                }
        }
```

```
Drinker(s):
while (true){
if (full) {
drink();
full = false;
        }
            }
```

## Ok, but .....

- ....... What if we have two drinkers?
- Both Drinkers can drink at the same time!



```
Drinker 1:
while (true)
  { if (full)
     {drink();
      full = false;
     }
  }
```

```
Drinker 2:
while (true)
{ if (full)
    {drink();
    full = false;
    }
}
```

Need to be atomic

## Example 2

- Shared account (between spouses John and Jane)
- Current account balance: $400
- John and Jane happened to be at the ATM in different places, but at almost the same time.
- Let say John wants to withdraw $200, and Jane wants to withdraw $300.
- First scenario: both of them see the current balance of $400 and relatively at the same time perform request to withdraw. John goes first…

## Example 2

- A DB system takes $400, subtracts $200, and gets interrupted (e.g. network congestion), thus the remaining balance of $200 wasn’t recorded yet.
- Jane’s request goes through and the system writes down $100 balance remaining.
- Then John’s request finally goes through, and system updates the balance to $200.
- The bank is a victim in that case because John and Jane were able to withdraw $500 and have $200 remaining, when initially account’s balance was $400!

## A Typical Example

```
class John extends Thread {
  run() {
    balance = ATM.getBalance();
    if(balance >= $200)
       ATM.withdraw($200);
  }
}
```

```
class Jane extends Thread {
  run() {
    balance = ATM.getBalance();
    if(balance >= $300)
       ATM.withdraw($300);
  }
}
```

```
class ATM{ …
  int withdraw(amount){
     if(amount <= balance) {
       balance = balance – amount;
       return amount;
  }
}
```

A trouble may occur at the points marked with the red arrows. The code **MUST NOT** be interrupted at those places.

## Race Condition

- This kind of bug, which only occurs under certain timing conditions, is called a ***race condition***.
- Output depends on ordering of thread execution
- More concretely:
  1) two or more threads access a shared variable with no synchronization, and
  2) at least one of the threads writes to the variable

Université Concordia University

## CRITICAL SECTION

- Section of code that:
  - Must be executed by one thread at a time
  - If more than one thread executes at a time, have a race condition
  - Ex: linked list from before
    - Insert/Delete code forms a critical section
    - What about just the Insert or Delete code?

## Solution: Use Semaphores

- Obvious: make the critical section part atomic.
- One way of doing it: **Semaphores**
- Semaphores are **system-wide** OS objects (also resources) used to
  - Protect critical section (mutexes – for Mutual Exclusion),
  - Coordinate other process' activities.
- Semaphores are NOT shared memory segments! But they both are often used together.

## Properties of Semaphores

- Simple
- Works with many processes
- Can have many different critical sections with different semaphores
- Each critical section has unique access semaphores
- Can permit multiple processes into the critical section at once, if desirable.

## SEMAPHORES

- We will talk about the semaphore with more details in the next tutorial

## Classical Synchronization Problems

- **The Producer-Consumer Problem**
- **The Readers-Writers Problem**
- **The Dining Philosophers Problem**

## The Producer-Consumer Problem

- In this problem, two processes, one called the *producer* and the other called the *consumer*, run concurrently and share a common buffer.
- The producer generates items that it must pass to the consumer, who is to consume them. The producer passes items to the consumer through the buffer.
- However, the producer must be certain that it does not deposit an item into the buffer when the buffer is full, and the consumer must not extract an item from an empty buffer.
- The two processes also must not access the buffer at the same time, for if the consumer tries to extract an item from the slot into which the producer is depositing an item, the consumer might get only part of the item.
- Any solution to this problem must ensure none of the above three events occur.

## The Producer-Consumer Problem

- A practical example of this problem is electronic mail.
- The process you use to send the mail must not insert the letter into a full mailbox (otherwise the recipient will never see it);
- similarly, the recipient must not read a letter from an empty mailbox (or he might obtain something meaningless but that looks like a letter).
- Also, the sender (producer) must not deposit a letter in the mailbox at the same time the recipient extracts a letter from the mailbox; otherwise, the state of the mailbox will be uncertain.
- Because the buffer has a maximum size, this problem is often called the *bounded buffer problem*.

# The Readers-Writers Problem

- In this problem, a number of concurrent processes require access to some object (such as a file.)
- Some processes extract information from the object and are called *readers*;
- others change or insert information in the object and are called *writers*.
- The Bernstein conditions state that many readers may access the object concurrently, but if a writer is accessing the object, no other processes (readers or writers) may access the object.
- There are two possible policies for doing this:
- ***First Readers-Writers Problem.***
  - Readers have priority over writers;
  - that is, unless a writer has permission to access the object, any reader requesting access to the object will get it.
  - Note this may result in a writer waiting indefinitely to access the object.
- ***Second Readers-Writers Problem.***
  - Writers have priority over readers;
  - that is, when a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access;
  - any readers that request access after the writer has done so must wait until the writer is done.
  - Note this may result in readers waiting indefinitely to access the object.

UNIVERSITÉ Concordia UNIVERSITY

# The Readers-Writers Problem

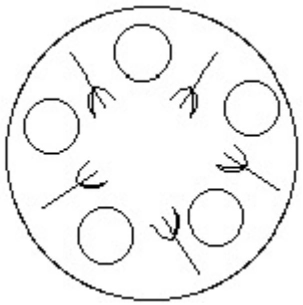
- **So there are two concerns:**
- **first**, enforce the Bernstein conditions among the processes,
- and **secondly**, enforcing the appropriate policy of whether the readers or the writers have priority.
- A typical example of this occurs with databases, when several processes are accessing data;
- some will want only to read the data, others to change it.
- The database must implement some mechanism that solves the readers-writers problem.

## The Dining Philosophers Problem

- In this problem, five philosophers sit around a circular table eating spaghetti and thinking.
- In front of each philosopher is a plate and to the left of each plate is a fork (so there are five forks, one to the right and one to the left of each philosopher's plate; see the figure).
- When a philosopher wishes to eat, he picks up the forks to the right and to the left of his plate.
- When done, he puts both forks back on the table.
- The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

## THE DINING PHILOSOPHERS PROBLEM

- **There are two issues here:**
- **first**, deadlock (where each philosopher picks up one fork so none can get the second) must never occur;
- and **second**, no set of philosophers should be able to act to prevent another philosopher from ever eating.
- A solution must prevent both.



# References

- [1] http://users.encs.concordia.ca/~mokhov/comp346/
- [2] Operating System, 3rd Edition, Gary J. Nutt, Addison Wesley, 2003
- [3] Operating Systems – Internals and Design Principles, 8th edition, William Stallings
- [4] Operating System Concepts, 7th Edition, A. Silberschatz, P. Galvin, G. Gagne, John Wiley & Sons, 2005
- [5] Operating systems – COMP346, Lesson 4 : Process synchronization, by Kerly Titus
- [6] http://www.studytonight.com/operating-system/process-synchronization
- [7] http://nob.cs.ucdavis.edu/classes/ecs150-2008-02/handouts/sync/sync-problems.html

UNIVERSITÉ Concordia UNIVERSITY