**Lecture 04: Inter process communication**

**Software System Components**
Behzad Bordbar
School of Computer Science, University of Birmingham, UK

---

**Recap**
- Extending Thread or implementing Runnable
- Thread terminology
- Stopping Threads
- Sleep
- Interrupt
- multiple threads

---

**Contents**
- Interacting threads
- Example of a race condition
- Theory of concurrency
- Mutual exclusion
- Locks
- Semaphores
- Monitors

---

**Interacting threads**

**Aim:** study threads which interact with each other by working on the same data, ensuring they don’t modify the same piece of data simultaneously.

Mutual exclusion
Let us see it with the help of an example

---

**Flight booking system**

Consider a flight with 10 seats and a number of travel agents each trying to book two seats on the flight.
1) Write a class `Flight` with two integer value attributes for total number of seats and number of booked seats (make them static to simplify)
2) Write a class `TravelAgent` which is implemented as a thread. `TravelAgent` checks if there are two seats available and books them.
3) In `Flight`, create 10 `TravelAgent` and let them book. At the end make sure that all threads `join()`.
4) Print the number of booked seats. (see folder `flight_travelagent_01` for a solution)

---

**Flight booking system (ctd)**

- Although each travel agent checks for availability of seat, we end up with too many seats booked.
- Tidy up the code by refactoring the `run()` method of `TravelAgent` by extracting method. Then include some print statement (or use a debugger) to explore why. (see folder `flight_travelagent_2`)
Challenges of concurrency
- The problem is known as a race condition or data race.

How to achieve mutual exclusion:
- Lock variables
- Strict Alternation
- sleep and wake up
- Semaphore
- Monitors

Race condition
- The problem is known as a race condition or data race.
- It is caused because threads/processes can access shared resources (shared variables, files, …)
- The key to avoiding race condition is to ensure mutual exclusion.
- Mutual exclusion: no two thread can access the critical region simultaneously.
- Critical region (also section) part of the program where shared memory is accessed.

Question: what is the critical section in case of the Flight example?

Race condition (ctd)
Mutual exclusion solves the race condition problem. But to have concurrent programs with correct behaviour the following conditions must satisfy:
1. No two thread may be simultaneously inside their critical regions
2. No assumption may be made about speed or number of CPUs
3. No thread running outside its critical region may block other thread
4. No thread should have to wait forever to enter its critical region

Achieving mutual exclusion
We shall examine the following:
- Lock variables
- Strict Alternation
- sleep and wake up
- Semaphore
- Monitors

Lock variables
- consider having a shared (lock) variable, $v = 0$
- $v = 0$ means no thread in the critical region (CR)
- $v = 1$ means some thread is in CR

When a thread wants to enter the CR, it checks $v$
- If $v = 0$, threads sets $v$ to 1 (obtain locks) and enters CR. Threads sets $v$ to 0 before leaving CS (release lock)
- If $v = 1$, the thread waits.

Question: would this work?

Lock variables (ctd)
The same problem as before:
Assume a thread checks the value of lock and finds out it is 0. But before setting it to 1, another thread reads the value (which is still 0), sets the value to 1 and enters CR. Then, when the first thread starts again (it sets the value again to 1) ending up with both threads in the CR.

Question: how about we read the lock once and then checking it before storing? It won’t help
We need a mechanism that ensure things are atomic.
How about threads taking turn?
Strict alternation

- A lock that uses busy waiting (turn) is called a *spin lock*.
- Question: Are all four conditions for correct concurrent behaviour met?
- Answer: condition 3 can be violated. But it ensures the other three condition are met. Work it out!

Thread 1: Strict alternation

```c
While (true) {
    while (turn != 0) {
        /* loop here to discover when turn change to 0 */
        enter CR;
        turn = 1;
        execute non CR
    }
}
```

Thread 2:

```c
While (true) {
    while (turn != 1) {
        /* loop here to discover when turn change to 1 */
        enter CR;
        turn = 0;
        execute non CR
    }
}
```

- thread are taking turn in execution.
- Use an variable turn.
- To access the CR, threads continuously test value of turn. This is called *busy waiting*
- … waste of CPU time

Condition 3 can be violated.

- Thread1 leaves its CR, sets turn =1
- Thread2 enters its CR
- Thread2 finishes its CR
- Now, both threads in nonCR and turn = 0
- Thread1 execute its loop, exiting CR & setting turn= 1
- Now, both threads in nonCR and turn =1
- Thread1 finishes its nonCR fast, goes back to busy loop
- At this point, Thread1 is not permitted to enter CR because turn=1
- So thread2 which is not in its CR is blocking Thread1

Busy waiting

- All above methods require busy waiting.
- Other methods based on busy waiting
- Dekker’s mutual exclusion algorithm (60s)
- Peterson’s algorithm (1981)
- Test and Set Lock (TSL)
- Negative aspects of busy waiting
  - wasting CPU resources
  - Priority Inversion Problem
- Avoid waste of resources “Sleep and wakeup”

Priority inversion problem 4 processes/threads

- H: high priority process, *i.e. scheduled to run when ready*
- L: Low priority process
- Suppose L is executing its CR
- H becomes available (for example finishing I/O)
- H is now busy waiting and can not execute it CR, because L is in CR & L can not run to leave its CR because H has the priority and must run

Busy waiting (ctd)

- Avoid waste of resources *Sleep and wakeup*
- Sleep is a system call that allows the caller to block itself, until another thread wakes it up (wakeup call)
- We must also take care of the case that the thread is not sleep and it receives a wakeup call
- This leads to the idea of *semaphores*
Summary

- Example of race condition
- Four main conditions to ensure correct concurrent programs
- Using locks – requires ensuring atomic action
- strict alternation (not all four conditions are satisfied)
- Busy waiting and its problems