# CMP325
# Operating Systems
# Lecture 12

# Introduction to Synchronization

**Fall 2021**

**Arif Butt (PUCIT)**

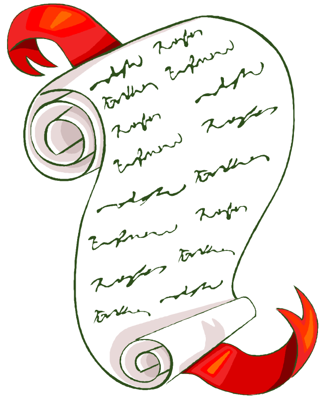
**Note:**

Some slides and/or pictures are adapted from course text book and Lecture slides of

- Dr Syed Mansoor Sarwar
- Dr Kubiatowicz
- Dr P. Bhat
- Dr Hank Levy
- Dr Indranil Gupta

For practical implementation of operating system concepts discussed in these slides, students are advised to watch and practice video lectures on the subject of **OS with Linux** by Arif Butt available on the following link:
http://www.arifbutt.me/category/os-with-linux/

## Today's Agenda



- Review of Previous Lecture
- What is Synchronization
- Concurrency Control
  - Producer Consumer Example
  - Deposit With Drawl Example
  - Spooling Example
- Race Conditions and Critical Section Problem
- Characteristics of Good CS Problem Solution
- A Comprehensive Example (Too much Milk)

# Introduction to Synchronization

- **In Real Life:** synchronization means making two things happen at the same time.
- **In CS:** synchronization refers to relationships among events, e.g. **before**, **during** and **after**.
- **Synchronization Constraints:**
  - **Serialization:** Event *A* must happen before event B.
  - **Mutual Exclusion:** Event *A* and B must not happen at the same time.
- **In Real Life:** we often check and enforce synchronization constraints using a clock (i.e. by comparing times).
- **In CS:** we can't use clocks (due to distributed environments) because most of the time the computer does not keep track of what time things happen, as there are too many things happening, too fast, to record the exact time of every thing.

## Introduction to Synchronization(…)

- If computers execute one instruction after another in sequence; the synchronization (serialization and ME) is trivial. If statement A comes before statement B, it will be executed first.
- **Problems**
  - In case of multiple CPUs, it is not easy to know if a statement on one CPU is executed before a statement on another.
  - In case of uni-Processor but multi threaded system, the programmer has no control over when each thread runs, the scheduler makes those decisions.
- Concurrent programs are often non deterministic, which means it is not possible to tell, by looking at the program, what will happen, when it executes.
- Example.

**"Two events are concurrent if we cannot tell by looking at the program which will happen first"**

# Example- Concurrent Program

```
#define BUFFER_SIZE 5
typedef struct{ ---- } item;
item buffer[BUFFER_SIZE];
int in = 0; //points to location where next item will be placed, will
             be used by producer process
int out = 0; //points to location from where item is to be consumed,
             will be used by consumer process
int ctr = 0;
```

**Producer Process**

```
item nextProduced;
while(1)
{
   nextProduced = getItem();
   while(ctr == BUFFER_SIZE)
        ; //do nothing
   buffer[in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   ctr++; }
```

**Consumer Process**

```
item nextConsumed;
while(1)
{
   while(ctr == 0)
        ; //do nothing
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   ctr--;
}
```

## Example (Race Condition)

```
long balance = 0;
void * inc(void * arg);
void * dec(void * arg);
int main(){
  pthread_t t1, t2;
  pthread_create(&t1, NULL, inc,NULL);
  pthread_create(&t2, NULL, dec,NULL);
  pthread_join(t1,NULL);   pthread_join(t2,NULL);
  printf("Value of balance is :%ld\n", balance);
  return 0;
}
```

```
void * inc(void * arg){
  for(long i=0;i<100000000;i++)
      balance++;
  pthread_exit(NULL);
}
```

```
void * dec(void * arg){
  for(long i=0;i<100000000;i++)
      balance--;
  pthread_exit(NULL);
}
```

# Concurrency Control

- We have just seen that cooperating processes (e.g. Producer Consumer Process) need to access common data; e.g. **buffer** or **ctr**.
- Concurrent access to shared data may result in data inconsistency. (HOW?)
- **Concurrency control is a mechanism using which multiple processes can access shared data w/o any conflict.**

# Example 1-Producer Consumer Problem

- In the solution of Producer Consumer Problem on previous slide, **ctr** is a shared variable that is used by both the producer and the consumer process to check whether the buffer is full or empty.
- In Producer and Consumer the single instruction of ctr++ and ctr-- can be written in Assembly as

ctr++;

```
P1:  MOV  R1, ctr
P2:  INC  R1
P3:  MOV  ctr, R1
```

ctr--;

```
C1:  MOV  R2, ctr
C2:  DEC  R2
C3:  MOV  ctr, R2
```

- Suppose both instrs execute concurrently and assume that the value of ctr is 5 before the execution of above instructions.
- **Interleaving**. One possible interleaving of the above statements is:
  - $P_1$, $P_2$ , $C_1$, $C_2$, $P_3$ , $C_3$. With this sequence of interleaving, the final value of ctr will be 4, where as the actual result should have been 5.
  - What is the final result if the interleaving sequence is $P_1$, $P_2$ , $C_1$, $C_2$, $C_3$ , $P_3$?
- If Consumer runs last -> ctr = 4.
- If Producer runs last -> ctr = 6.

# Example 2-Deposit and Withdrawl

- Consider a Bank Transaction. The Deposit Process deposits a particular amount in the bank via a cheque. The Withdrawl Process withdraws a particular amount from the bank via ATM.
- In Deposit and Withdrawl process the instruction that updates the balance variable can be written in Assembly as

## Deposit

$D_1$: MOV R1, balance

$D_2$: ADD R1, depositAmount

$D_3$: MOV balance, R1

## Withdrawl

$W_1$: MOV R2, balance

$W_2$: SUB R2, wdrAmount

$W_3$: MOV balance, R2

**Sample Transaction**

Current Balance: 100/-

Cheque deposited: 25/-

ATM with drawl: 10/-

- **Interleaving**. One possible interleaving of the above statements is:
  - $D_1$, $D_2$ , $W_1$, $W_2$, $D_3$ , $W_3$. With this sequence of interleaving, the final value of balance will be 90/-
  - What is the final result if the interleaving sequence is $D_1$, $D_2$ , $W_1$, $W_2$, $W_3$ , $D_3$?

# Important Concepts

- **Race Condition:** The situation where several threads/cooperating processes are updating some shared data concurrently and the final value of the data depends on which thread finishes last

- **Critical Section:** A piece of code in threads/cooperating processes in which the threads may update some shared data (variable, file, database)

- **Critical Section Problem.** If multiple threads try to execute their ***CS*** simultaneously, we need to execute them one by one completely

## Important Concepts (cont…)

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- **Atomic Operation**: An operation that always runs to completion or not at all
  - It is **indivisible**: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store are often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

# Structure of CS Solution

```
do {

      ENTRY SECTION

       <CS>               //Access shared variables

      EXIT SECTION

        <RS>              //Do other work

 }while(1);
```

## Characteristics of Good CS Problem Solution

1. **Mutual Exclusion**. If a process is executing in its CS, no other cooperating processes can execute in their CS

2. **Progress**. If no process is executing in its CS and some processes wish to enter in their CS, two things need to happen:
   - No process in <RS> should participate in the decision
   - This decision has to be taken in a finite time

## Characteristics of Good CS Problem Solution

3. **Bounded Waiting**. If a process has requested to enter in its CS, a bound must exist on the number of times that other processes are allowed to enter in their CS, before the request is granted.
   - **Example.**
     - Consider three processes P1, P2 and P3; Suppose P2 has made a request to enter its CS.
     - Let the bound is set to 1.
     - Now if P1 and P3 also request to enter their CS they can be allowed only once.
     - Before giving them a second chance P2, should be given a chance to go to its CS.

## Synchronization Example

## Too Much Milk

# "Too much milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- **Example**: People need to coordinate:

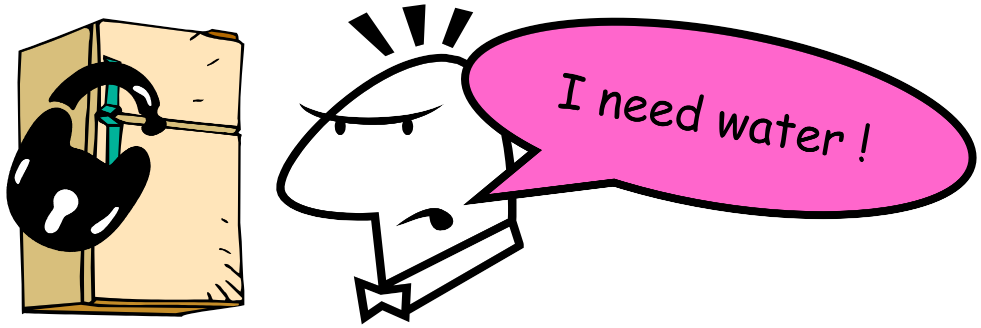
| **Time** | **Person A** | **Person B** |
|---|---|---|
| 3:00 | Look in Fridge. Out of milk | |
| 3:05 | Leave for store | |
| 3:10 | Arrive at store | Look in Fridge. Out of milk |
| 3:15 | Buy milk | Leave for store |
| 3:20 | Arrive home, put milk away | Arrive at store |
| 3:25 | | Buy milk |
| 3:30 | | Arrive home, put milk away |

# Too much Milk (cont...)

- **Lock:** Prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked

**Important idea: all synchronization involves waiting**

- **For example:** Lets fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk

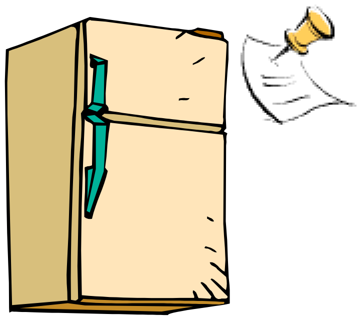


  - Fixes too much: roommate angry if only wants cold water

## Too much Milk-Solution # 1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



- **Result?**
  - Still too much milk but only occasionally! (HOW?)
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

# Too much Milk-Solution # 1 ½

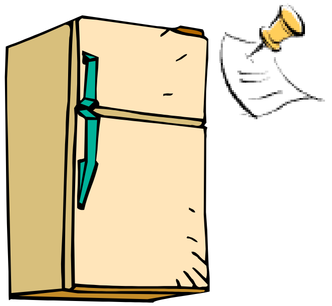
- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

# Too much Milk-Solution # 2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:



Thread A

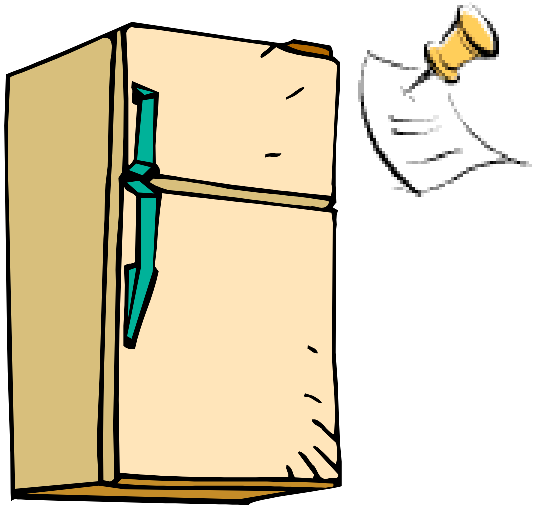
```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time

# Too much Milk-Solution # 2: Problem



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

## Too much Milk-Solution # 3

- **Here is a possible two-note solution:**

**Thread A**

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

**Thread B**

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- **Does this work?**
  - Yes.
- **At X:**
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- **At Y:**
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

## Discussion-Solution # 3

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
    buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

# Too much Milk-Solution # 4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - Lock.Acquire() – wait until lock is free, then grab
  - Lock.Release() – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between Acquire() and Release() called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.

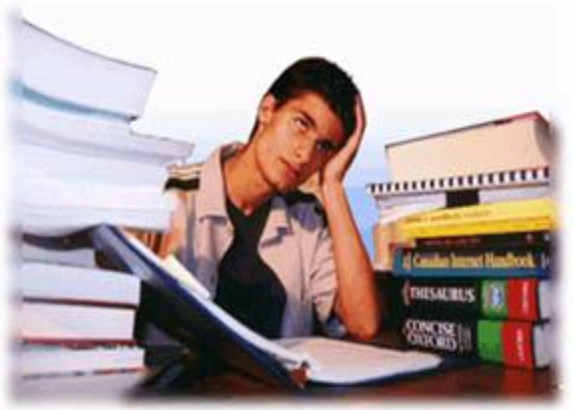
## Where R we going with Synchronization

| | | | | |
|---|---|---|---|---|
| Programs | Shared Programs | | | |
| Higher-level API | Locks | Semaphores | Monitors | Send/Receive |
| Hardware | Load/Store | Disable Ints | Test&Set | Comp&Swap |

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

## SUMMARY

# We're done for now, but Todo's for you after this lecture…



- Go through the slides and Book Sections: 5.1, 5.2
- Try to make an understanding about the non determinism of concurrent multi threaded programs
- Try hand/mind execution of the sample programs discussed in class

If you have problems visit me in counseling hours. . . .