CMPT 300 D100
Operating System I
Synchronization Tools– Chapter 6

Dr. Hazra Imran

Learning goals

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores and condition variables can be used to solve the critical section problem

Background

P1 dirty

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution

R/W P1 x=5 P2 R/W

- Concurrent access to shared data may result in data inconsistency

x=5

x=x+5

P2 10

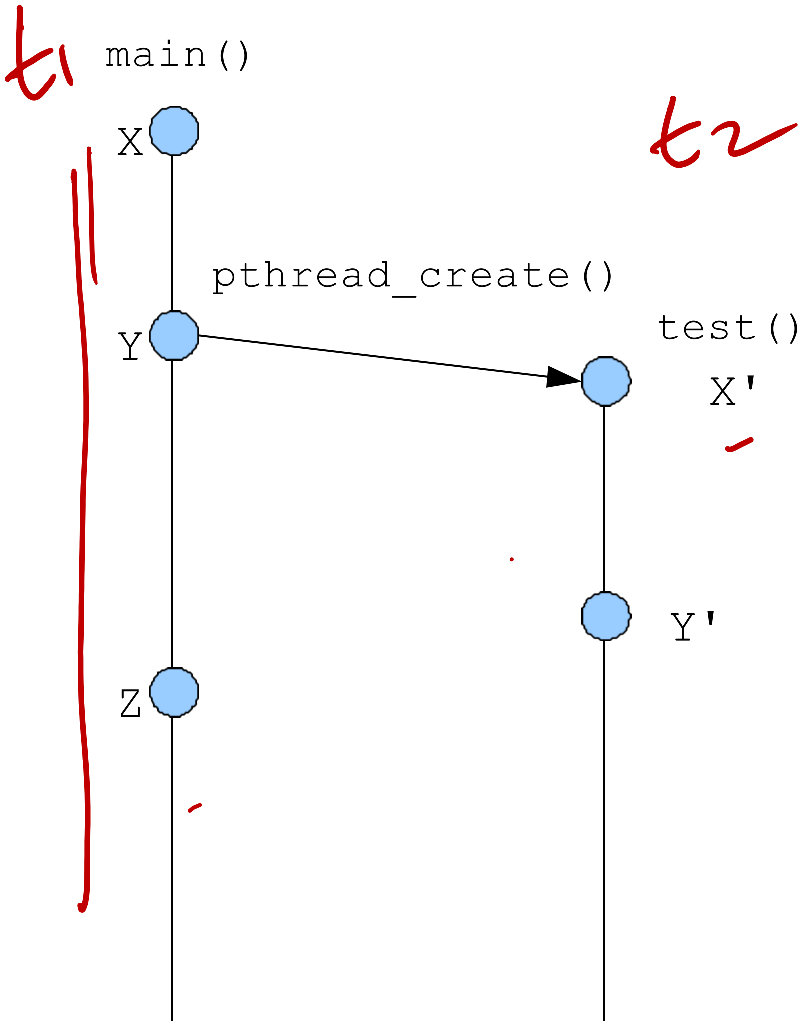
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

What if have multiple threads

- Instructions executed by a single thread are totally ordered
  - A then B then C and so on

- If there is no synchronization, then instructions executed by distinct threads will be unordered

Example



$X < \quad < Z$

$X' < Y'$

$X < X'$

$Z == X'$

$Z == Y'$

Different views

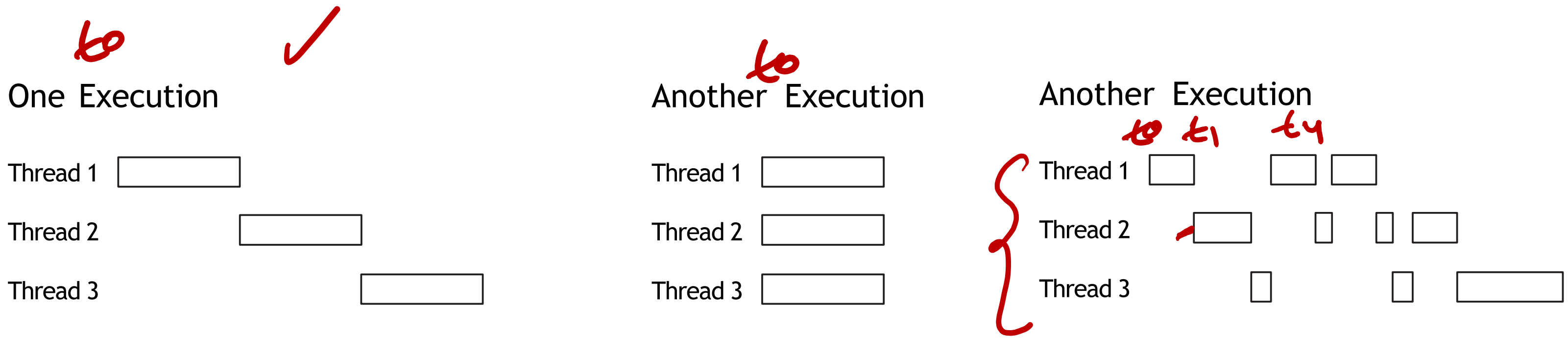
Best case

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
`x = x + 1;`	`x = x + 1;`	`x = x + 1;`	`x = x + 1;`
`y = y + x;`	`y = y + x;`	....	`y = y + x;`
`z = x + 5y;`	`z = x + 5y;`	Thread is suspended. Other thread(s) run. Thread is resumed.	....
.	.	....	Thread is suspended. Other thread(s) run. Thread is resumed.
.	.	`y = y + x;`	....
.	.	`z = x + 5y;`	`z = x + 5y;`

Atomic Memory Operations

- In most architectures, load and store operations on single-byte are atomic
- Threads cannot get context switched in the middle of load/store to/from a word
- Many instructions are not atomic . (context switch)

Different ways



Can this cause any problem?

(A) Yes

(B) No

(C) It depends!

Example

- When threads work on separate data, the order of scheduling does not change results

Thread A	Thread B
x = 1;	y = 2;

- Scheduling order matters when threads work on shared data

Thread A	Thread B
x = 1;	y = 2;
x = y + 1;	y = y * 2;

- What are possible values of x? (initially, y = 12)

	Thread A		Thread B
1	x = 1;		
2	x = y + 1;		
		3	y = 2;
		4	y = y * 2;

$\boxed{x = 13}$

Example

- What are possible values of x? (initially, y = 12)

	Thread A	Thread B
1		`y = 2;`
2		`y = y * 2;`
3	`x = 1;`	
4	`x = y + 1;`	

$\boxed{x = 5}$

- What are possible values of x? (initially, y = 12)

	Thread A	Thread B
1		`y = 2;`
2	`x = 1;`	
3	`x = y + 1;`	
4		`y = y * 2;`

$\boxed{x = 3}$

Race conditions

- A program has a **race condition** if the result of an execution depends on timing
  - its non-deterministic

- Example:
  - You run it on the same data, and sometimes it prints 0 and sometimes it prints 100

Race condition

- This often happens when threads share some data

data race.

Too Much Milk Example

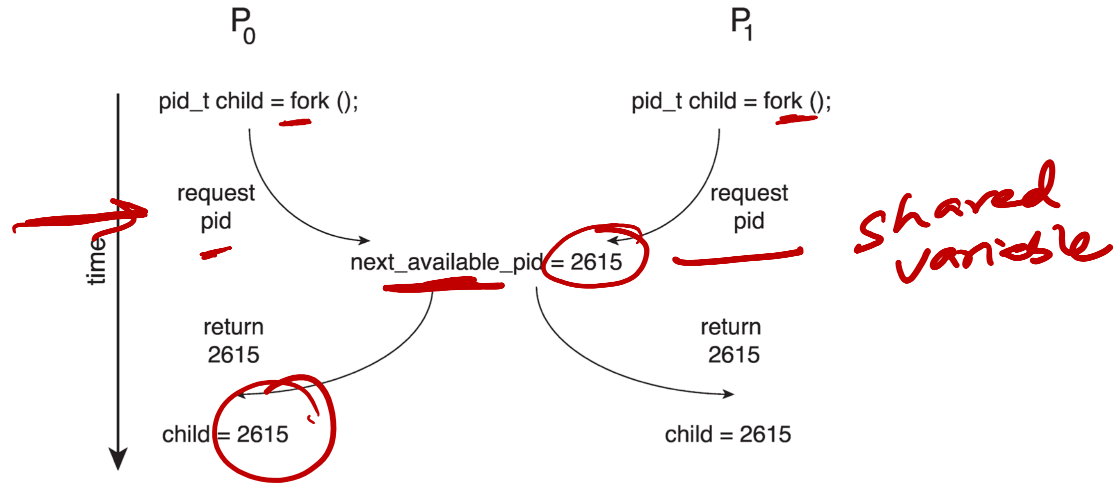
	Roommate A	Roommate B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
01:00		Arrive home, put milk away. Oh no!

- What are correctness properties of "too much milk" problem?
  - At most one roommate should buy milk
  - Someone should eventually buy milk

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call

- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Mutual Exclusion and Critical Sections

- Mutual exclusion: only one thread runs a particular code at any given time
  - One thread excludes others while doing its task
  - Mutual exclusion means "not simultaneous"
    - A < B or B < A (we don't care which)

code

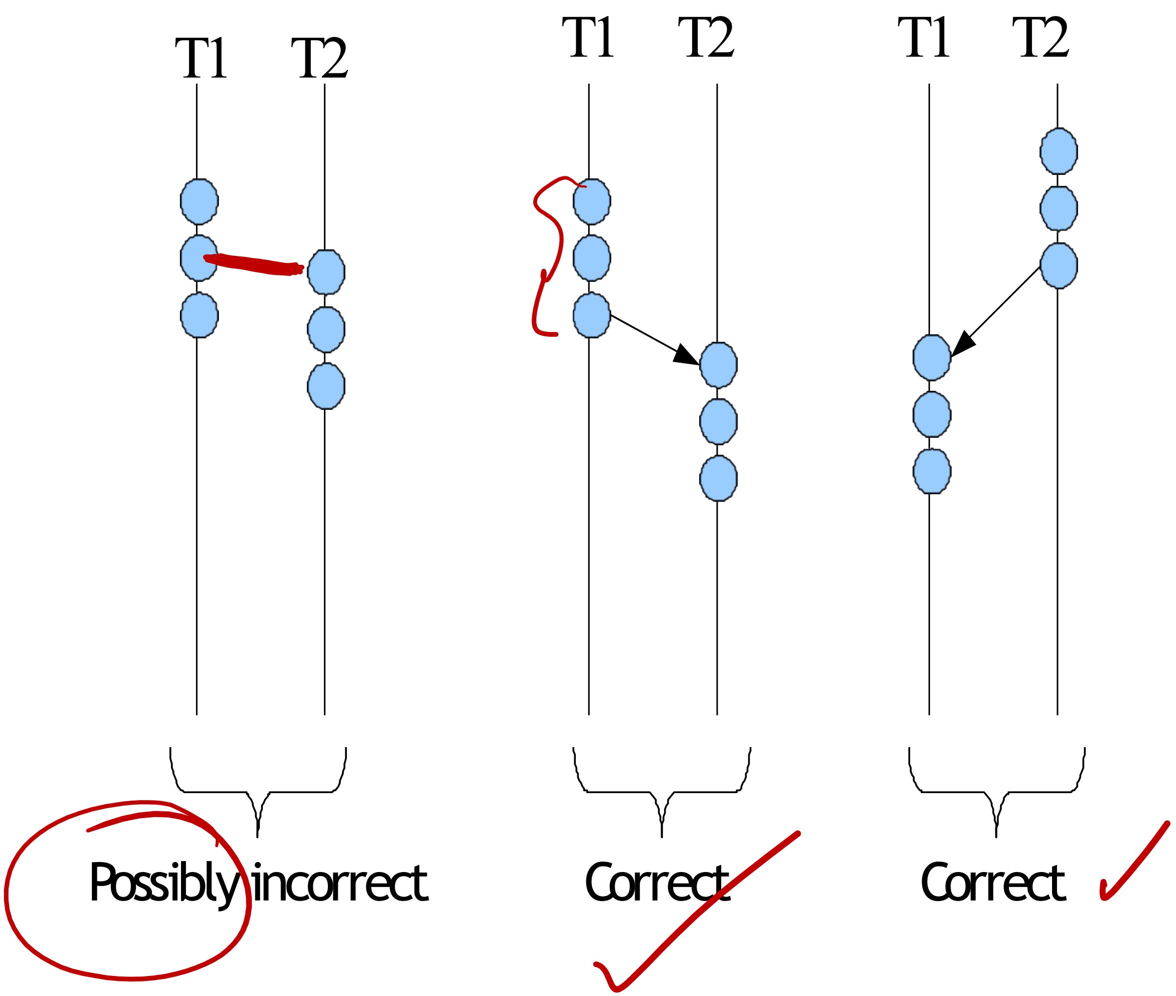
- Critical section: a particular code that only one thread can execute at once
  - Sequences of instructions that may get incorrect results if executed simultaneously are called **critical sections**

- Critical section and mutual exclusion are two ways of describing the same thing

① guarantee ordering

② ensure correct execution

Critical sections



When do critical sections arise?

Think of a common pattern between multiple threads

1 read - modify - write shared value

global heap allocated var

local var ×

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
while (true) {

        entry section

            critical section

        exit section

            remainder section

}
```

wait()
(common)

Correct critical section requirements

A solution to the critical-section problem must satisfy the following three requirements:

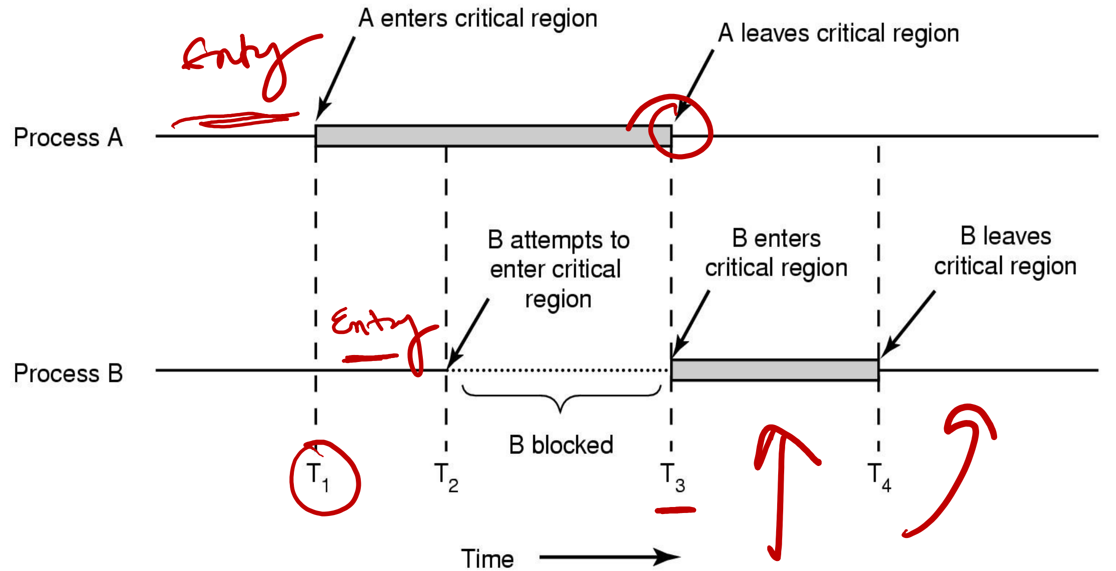
1. Mutual Exclusion

2. Progress

3. Bounded Waiting

Correct critical section requirements

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



c X

atmost one process is in CS.

Correct critical section requirements

P1
P2
P3
CS

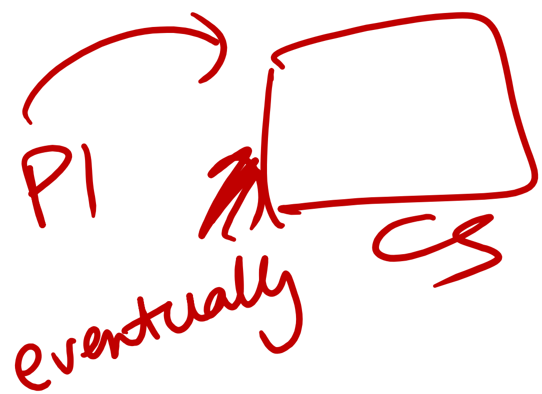
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely.

Correct critical section requirements

no starvation

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes



Interrupt-based solution

- Entry section: *disable interrupts*

CS ✓

- Exit section: *enable interrupts*

single core

- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve (never enter their critical section)?
  - What if there are two CPUs?

expensive / time consuming

Implementing Mutual Exclusion

How do we do it?

- via hardware: special machine instructions — OS
- via OS support: OS provides primitives via system call
- via software: entirely by user code

Peterson's Solution

Humble

- Restricted to two processes (Lets assume Pi and Pj)
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

int turn	Boolean flag [2]
indicates whose turn it is to enter its critical section	used to indicate if a process is ready to enter the critical section
If process Pi wants to enter CS then it will set turn equal to j	**flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

Idea: Though process wants to enter CS, it gives turn to other process if other process also wants to enter CS.

Algorithm for Process P_i

i is ready to enter cs

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

        /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

1 2 T looping F

Process i is ready to enter in CS but asserting if the other process (j) wishes to enter the CS.

Peterson's Solution

Process P0

```
while (true) {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn == 1);
    CRITICAL SECTION
    flag[0] = FALSE;

    REMAINDER SECTION
}
```

Process P1

```
while (true) {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0;
    CRITICAL SECTION
    flag[1] = FALSE;
    REMAINDER SECTION
}
```

Correctness of Peterson's Solution

Ques: Does this algorithm satisfy the three requirements (mutual exclusion, progress, bounded wait)?

Process P0

```
while (true) {
1   flag[0] = TRUE;
2   turn = 1;
3   while (flag[1] && turn == 1);
    CRITICAL SECTION
4   flag[0] = FALSE;
    REMAINDER SECTION
}
```

Process P1

```
while (true) {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
    CRITICAL SECTION
    flag[1] = FALSE;
    REMAINDER SECTION
}
```

Mutual exclusion? ✓

Progress requirement is satisfied

Bounded-waiting requirement is met ✓

Flag | F | F |

Turn | 0 |

Process	Line #1	Line #2	Line #3	CS	Line #4
P0	T F	T= 1 turn	F	Enter	

Now P1 is interested to enter

P1	T T	T= 0	T	Stuck into while loop

Now P0 is exiting from CS; so now Flag will be | F | T |

Now P1 can come out of while loop and enter in CS

Peterson's Solution

Disadvantages

1 busy waiting. solution

2 limited to 2 process

Busy waitig

Spinning

busy looping

NC ↓