Operating Systems

Lecture 5.1

Overview of Synchronization

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda

- Introduction to Synchronization
- Examples of Race Condition
- Key Concepts in Concurrency
- Structure of CS Problem Solution
- Characteristics of Good CSP Solution
- Peterson Algorithm (2-Process CSP Solution)
- Bakery Algorithm (N-Process CSP Solution)
- Busy Waiting and its Solution
- Overview of Concurrency Control Mechanisms

Introduction to Synchronization

Independent vs Cooperating Processes

Processes executing concurrently in the operating system can be:

- **Independent Process:** A process that cannot affect or cannot be affected by the execution of another process. A process that does not share data with another process is independent.
- **Cooperating Process:** A process that can affect or can be affected by the execution of another process. A process that share data with other process is a cooperating process.
- **Advantages of Cooperating processes:**
    - Information sharing.
    - Computation speed up.
    - Modularity.
    - Convenience.

Introduction to Synchronization

- **In Real Life:** Synchronization means coordinating actions so they occur in correct order or at the same time
- **In Computer Science:** Synchronization refers to relationships among events:

  Before → one event must happen before another.

  During → one event depends on another in progress.

  After → one event must follow another.

- **Key Synchronization Constraints:**
  - **Mutual Exclusion:** Event **A** and **B** must not happen at the same time.
  - **Serialization:** Event **A** must happen before event **B**, or vice versa.

- **In Real Life:** We often check and enforce synchronization constraints using a clock i.e. by comparing times.
- **In CS:** Too many operations occur in parallel, so we can't use clocks (due to distributed environments). Moreover, most of the time the computers cannot keep track of what time various events happen, as there are too many events happening, too fast, to record the exact time of every event.

Why Synchronization is needed?

If computers execute one instruction after another in sequence, the synchronization (serialization and Mutual execution) is trivial. If statement A comes before statement B, it will be executed first.

Problems with concurrency:

- **Multiple CPUs**

  It is not easy to know if an instruction on CPU1 is executed before an instruction on CPU2.
- **Single CPU and multithreaded system**

  The scheduler decides when each thread runs, the programmer has no control over when each thread runs.
- **Non-determinism**

  Concurrent programs are often **non deterministic**, the outcome of a program can change with each run, even with the same input.

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

Producer Consumer Pseudocode

```
#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;
```

Produces 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--;
}
```

Mutual Exclusion Violation (Producer-Consumer)

- 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:

Producer

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

Consumer

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

- Suppose the initial value of `ctr` is **5**
- Suppose both processes execute concurrently

Interleaving: There can be different ways in which the three instructions of producer and the three instructions of consumer can be interleaved by the scheduler.

- If Consumer runs last (P1, P2, C1, C2, P3 , C3) → `ctr` = 4
- If Producer runs last (P1, P2, C1, C2, C3, P3) → `ctr` = 6

Mutual Exclusion Violation (Deposit-Withdrawl)

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

Deposit

```
D1:    MOV   R1, balance
D2:    ADD   R1, depositAmount
D3:    MOV   balance, R1
```

Withdrawl

```
W1:  MOV   R2, balance
W2:  SUB   R2, wdrAmount
W3:  MOV   balance, R2
```

Sample Transaction

Current Balance: 100/-

Cheque deposited: 25/-

ATM withdrawal: 10/-

Interleaving. Calculate the result of following two possible interleaving of the above statements:

- $D_1, D_2, W_1, W_2, D_3, W_3$
- $D_1, D_2, W_1, W_2, W_3, D_3$

Example: race1.c

```
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);
}
```

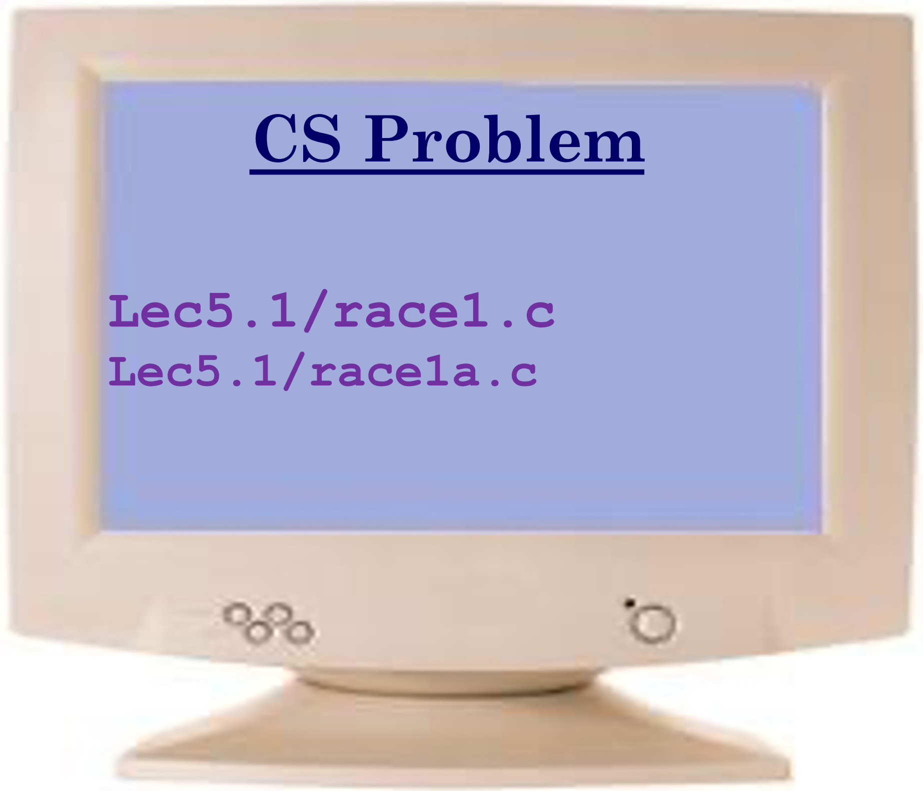
Example: race1a.c

```c
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;
}
```

```c
void * dec(void * arg){
   int temp = balance;
   usleep(10000);
   temp = temp - 1;
   usleep(10000);
   balance = temp;
   pthread_exit(NULL);
}
```

```c
void * inc(void * arg){
   int temp = balance;
   usleep(10000);
   temp = temp + 1;
   usleep(10000);
   balance = temp;
   pthread_exit(NULL);
}
```

Demonstration



CS Problem

```
Lec5.1/race1.c
Lec5.1/race1a.c
```

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Key Concepts in Concurrency

Key Concepts in Concurrency

Race Condition:

- A situation, where several threads or cooperating processes are updating some shared data concurrently, and the final value of the data depends on which thread/process finishes last - unpredictable and incorrect outcomes.

Critical Section:

- A piece of code in threads or cooperating processes in which the thread/process may update some shared data (variable, file, database)
- Regions of code that must be executed by only one process at a time
- Example: updating a global counter, writing to a file

Critical Section Problem:

- When multiple threads try to execute the critical section simultaneously, the result is data corruption. Solution is to allow only one thread to execute its critical section at a time

Key Concepts in Concurrency

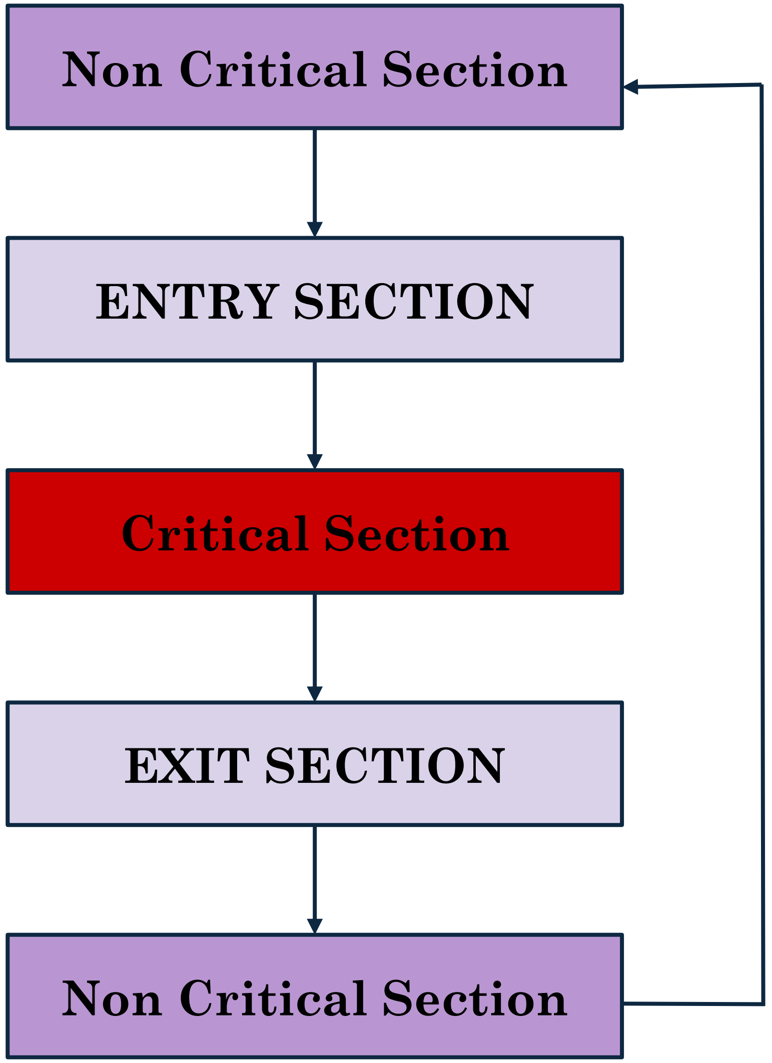
- **Atomic operation** is an operation that always runs to completion or not at all, as it is indivisible and uninterruptible by other threads or processes.
- Atomic operations are foundational for safe thread cooperation; without them, shared data access becomes unreliable and prone to race conditions.
- On most machines, memory references and assignments i.e. loads and stores of words are atomic.
- Simple aligned word-sized loads and stores (e.g., reading or writing an integer) are typically atomic on modern hardware. However, larger or misaligned operations—like double-precision floating-point stores or multi-word copies are not atomic.

Structure of CSP Solution

```
do {
ENTRY SECTION
<Critical Section>
EXIT SECTION
<Remainder Section>
} while(true);
```

Controls the entry into critical section and gets lock on required resources

Removes lock on resources and let others know that its CS is over



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. This prevent corruption of shared data and inconsistencies.

2. **Progress:** If no process is executing in 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

   This ensures the system keeps moving forward (avoids deadlock)

3. **Bounded Wait:** If a process has requested to enter in the 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. This prevents starvation (where one process waits forever while others repeatedly enter)

Peterson’s Algorithm

2-Processes CSP Solution

Algorithm Using Strict Alternation

- Given by a Dutch mathematician Dekker for two processes P_0 and P_1
- Use a shared variable `turn` to enforce order, initialized to zero
- If `turn = i` then `P`$_i$ can enter in its CS otherwise it will wait

Process P_0

```
do {
   while (turn!=0);  //Entry Section
   <CS>
   turn = 1;          //Exit Section
   <RS>
} while (1);
```

Process P_1

```
do {
   while (turn!=1);  //Entry section
    <CS>
   turn = 0;          //ExitSection
   <RS>
} while (1);
```

- Guarantees mutual exclusion i.e.; only one process in CS
- If turn = 0 but P0 has no work, P1 is blocked unnecessarily
- Processes are forced to take turns, even if one is idle

Algorithm Using Flags

- The limitation of strict alternation is solved in this algo, (processes don't have to take turns).
- Instead of a single variable turn, take an array of two Boolean flags, `flag[0]` and `flag [1]`
- A process set its flag to true (showing its intention that it want to enter its CS) and check for the other process flag, if the other process flag is true keep spinning in loop.

Process P_0

```
do {
   flag[0] = true;
   while (flag[1]== true]);
   <CS>
   flag [0] = false; //Exit Section
   <RS>
} while (1);
```

Process P_1

```
do {
   flag[1] = true;
   while (flag[0]==true);
    <CS>
   flag [1] = false; //Exit Section
    <RS>
} while (1);
```

What if both set `flag[i] = true` at the same time → deadlock (both spin forever)

Use of Flags - Improved

Instead of waiting/spinning indefinitely in the while loop, the process set its flag to false, wait for a random period of time, set its flag back to true and then again try the while loop condition.

Process P_0

```
do {
    flag[0] = true;
    while (flag[1] == true){
        flag[0] = false;
        wait(randno());
        flag[0] = true;
    }
        <CS>
        flag [0] = false;
        <RS>
} while (1);
```

Process P_1

```
do {
        flag[1] = true;
        while (flag[0] == true){
        flag[1] = false;
        wait(randno());
        flag[1] = true;
    }
    <CS>
        flag [1] = false;
        <RS>
} while (1);
```

Inefficient: Processes waste CPU cycle in busy waiting

Peterson Algorithm

- The algorithm is given by Peterson, the person who wrote the first edition of our textbook "OS concepts" in 1984
- It combines the shared variables of previous algorithms
- Keep two **boolean flags** one for each process and a shared integer variable **turn**

```
boolean flag[2];   // initialized to false
int turn = 0;      // initialized to 0
```

- Before entering CS, each process:
  - Set its flag to true (I want to enter CS)
  - Sets `turn` equal to other process ID (but if you want to enter, you go first).
  - It then checks whether the other process wants to enter its CS and is it his turn.
  - If yes, it waits. Otherwise enters CS safely.

Peterson Algorithm

Process P_0

```
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1); //spin
    <CS>
   flag [0] = false;
    <RS>
} while (1);
```

Process P_1

```
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0); //spin
    <CS>
    flag [1] = false;
    <RS>
} while (1);
```

- If both processes want to enter, the turn variable ensures only one gets in
- After leaving the CS, a process resets its flag
- Works only for two processes. Try extending the above pseudocode for three or more processes

Bakery Algorithm

N-Processes CSP Solution

Bakery Algorithm

Think of a bakery having two doors, each with a separate Token number dispenser:

- Whenever a person enters the bakery, he is given a token number (TNumber)
- The customer with the smallest token number is served first
- If two customers have same token number because they entered the bakery at the same time, who is to be served first?
    - Ladies first
    - Senior Citizen first
    - In OS, smaller PID first

Bakery Algorithm

In concurrent systems:

- Every process gets a token number before entering the CS
- Process with the smallest token number enters the CS. If process P_i and P_j gets the same number then

```
if i < j then
    P_i is served first;
else
    P_j is served first;
```

- Process with the smallest PID enters the CS

Bakery Algorithm - Example

Consider following six cooperating processes, with a token number assigned to each. Keeping the Bakery algorithm can you give the sequence in which they will enter their CSs.

PID	Token number
P_0	8
P_1	5
P_2	0
P_3	5
P_4	6
P_5	2

Sequence of CS entry will be: **< P_5 P_1 P_3 P_4 P_0 >**

Bakery Algorithm - Algorithm Semantics

- Ticket numbering are monotonically increasing 1, 2, 3, 3, 4, 5, …
- Every upcoming process gets a number larger than or equal to existing ones
- **Notations**
    - `(Tnumber, PID)` is an ordered pair
    - `(a, b) < (c, d)  if`

      `(a < c) OR (a == c & b < d)`
    - `Max(…)` is a function that returns the largest ticket number currently assigned
- **Data Structures**
    - `Boolean choosing[n]`: initialized to false, true if process is picking a ticket
    - `int TNumber[n]`: initialized to zero, ticket number for each process

Bakery Algorithm

```
do {

    Tnumber[i] = 1 + max(Tnumber[0, Tnumber[1],...Tnumber(n-1)]);

    for (j = 0; j < n; j++) {

    while(Tnumber[j]!=0 && (Tnumber[j],j) < (Tnumber[i],i) );}
```

This loop is going to compare the (no, id) pair of P_i with (no, id) pair of all other processes and finally selects which process goes to the CS

If P_j is having a Tnumber equal to 0, i.e. P_j is not interested to enter its CS. So break this while loop, go back to for loop, increment j and check next process

If P_j is interested to go to its CS, then check its ordered pair, with ordered pair of P_i. If P_j's (no, id) pair is less than P_i's pair then P_i wait in this loop, else move up to for loop, increment j and check next process.

```
    <CS>
    Tnumber[i] = 0;
    <RS>
} while (1);
```

After leaving the CS, P_i set its Tnumber to 0, showing that it is now not interested to enter its CS

Bakery Algorithm

```
do {
    choosing[i] = true;
    Tnumber[i] = 1 + max(Tnumber[0], Tnumber[1],...);
    choosing[i] = false;
    for (j = 0; j < n; j++) {
    while (choosing[j]);
    while(Tnumber[j]!=0 && (Tnumber[j],j) < (Tnumber[i],i) );}
```

If P_j is having a Tnumber equal to 0, i.e. P_j is not interested to enter its CS. So break this while loop, go back to for loop, increment j and check next process

If P_j is interested to go to its CS, then check its ordered pair, with ordered pair of P_i. If P_j's (no, id) pair is less than P_i's pair then P_i wait in this loop, else move up to for loop, increment j and check next process.

```
    <CS>
    Tnumber[i] = 0;
    <RS>
} while (1);
```

Before getting a Tnumber, every process will first set its choosing to true and later will set it to false.

This loop is going to compare the (no, id) pair of P_i with (no, id) pair of all other processes and finally selects which process goes to the CS

If Process P_j is in the process of getting a ticket number lets wait.

After leaving the CS, P_i set its Tnumber to 0, showing that it is now not interested to enter its CS

$100 QUESTION

Why does Lamport used a variable called choosing? What will happen if we don’t use it?

Busy Waiting

Busy-Waiting Problem

- Busy waiting means that a process is waiting for a condition to be satisfied, sitting in a tight loop, without relinquishing the CPU.
- Lets see a bigger picture:
  - Imagine there are 100 cooperating processes.
  - One process is executing in its CS.
  - 50 out of remaining 99 wants to get inside their CS.
  - These 50 processes are all spinning in their entry section.
  - Whenever the CPU is scheduled to them they keep spinning for the allocated time quantum, instead of doing any useful work.
  - Thus wasting CPU cycles.

Busy-Waiting Problem

Busy waiting not only waste precious CPU cycles, but it can also have effects like **priority inversion:**

- Consider a system with two cooperating processes, **H** (high priority) and **L** (low priority).
- The scheduling rules are such that **H** is run whenever it is in ready state.
- At a certain moment, **L** is in its CS, and **H** becomes ready to run.
- **L** is preempted from it CS, and **H** executes.
- **H** now begins busy waiting, now due to low priority **L** is never scheduled while **H** is running.
- Process **L** never gets the chance to leave its CS and execute the Exit section, so **H** loops forever.
- This situation is referred to as the priority inversion problem.

Solution: Sleep and Wakeup

- In busy waiting whenever a process wants to enter its CS, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is
- Busy waiting can be avoided, instead of spinning:
    1. Block the process → put it in a waiting queue
    2. Relinquishing the CPU (in a queue) → let other process run
    3. Wait to be awakened at some appropriate time in the future
- Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep (in a queue) and having to wake it up when the appropriate program state is reached.
- **`sleep()`** is a system call that causes the caller to block, that is, be suspended until another process wakes it up
- The **`wakeup(processID)`** signal waiting process to continue
- Next slide shows the producer consumer problem that uses these calls

Sleep and Wakeup

```
#define N 100     // number of slots in the buffer
int count = 0;    // number of items in the buffer
void producer(){
  int item;
  while (TRUE) {
    item = produce_item(); // generate next item
    if (count == N) sleep(); //if buffer is full, go to sleep
    insert_item(item); //put item in buffer
    count = count + 1; // increment count of items in buffer
    if (count == 1) // if this is the first item in buffer
        wakeup(consumer);
  }
}
```

```
void consumer(){
int item;
while (TRUE) {
    if (count == 0) sleep(); // if buffer is empty, go to sleep
    item = remove_item(); // take item out of buffer
    count = count - 1; // decrement count of items in buffer
    if (count == N - 1) // if the buffer was full
        wakeup(producer);
     consume_item(item);
 }
}
```

Sleep and Wakeup

$100 QUESTION

Is there a race condition in the Producer Consumer code shown on previous slide?

Overview of Concurrency Control Mechanisms

Concurrency Control Mechanisms

Concurrency control mechanisms are synchronization primitives and techniques used to coordinate access to shared resources among multiple concurrent processes or threads, ensuring data consistency, preventing race conditions, and maintaining system correctness in multi-threaded/multi-process environments.

Overview of Concurrency Control Mechanisms

- **Software-level Synchronization Primitives:** Implemented through collaboration between C runtime libraries and the OS kernel. Examples include mutexes, spinlocks, condition variables, barriers, semaphores, and read-write locks.
- **Hardware-level Synchronization Primitives:** Hardware-based concurrency control mechanisms rely on special atomic CPU instructions (compare-and-swap and test-set-lock), that perform indivisible/atomic operations. These instructions enable synchronization directly at the hardware level without the need for kernel intervention or traditional locking mechanisms. Hardware atomics are commonly used in high-performance, low-latency applications where the overhead of locks is prohibitive.
- **Compiler-level Synchronization Primitives:** Higher-level concurrency control mechanisms are provided by programming languages, compilers, and frameworks to simplify thread synchronization. Instead of writing complex locking code manually, developers can use simple annotations or directives. These tools automatically generate the necessary synchronization logic behind the scenes. Examples include:
  - ➢ Java's `synchronized` keyword, which locks objects implicitly to prevent concurrent access.
  - ➢ GCC's `__transaction_atomic`, which allows blocks of code to run atomically, and rolling back changes if a conflict is detected.
  - ➢ C#'s `lock(object)` statement, which generates `Monitor.Enter()/Monitor.Exit()` calls wrapped in try-finally blocks to guarantee lock release even during exceptions.

To Do

Coming to office hours does NOT mean that you are academically weak!