Introduction

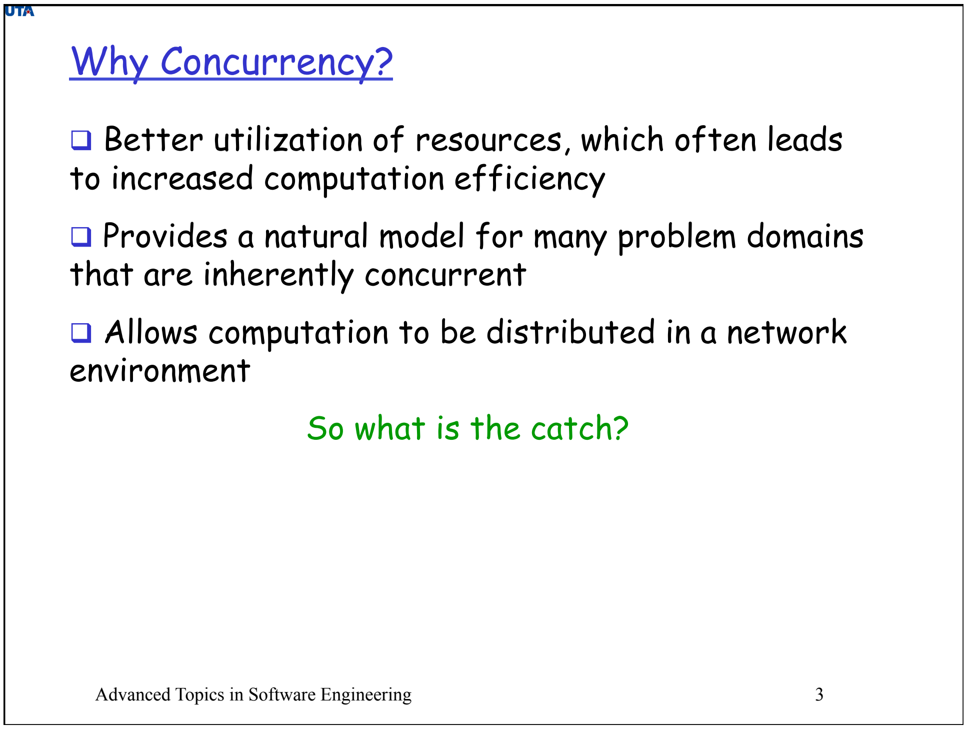
- ❑ Overview
- ❑ Motivating Examples
- ❑ Interleaving Model
- ❑ Semantics of Correctness
- ❑ Testing, Debugging, and Verification

Concurrent Programs

❑ Characterized by a set of cooperative processes or threads that can execute in parallel.

❑ Three types of concurrent programs

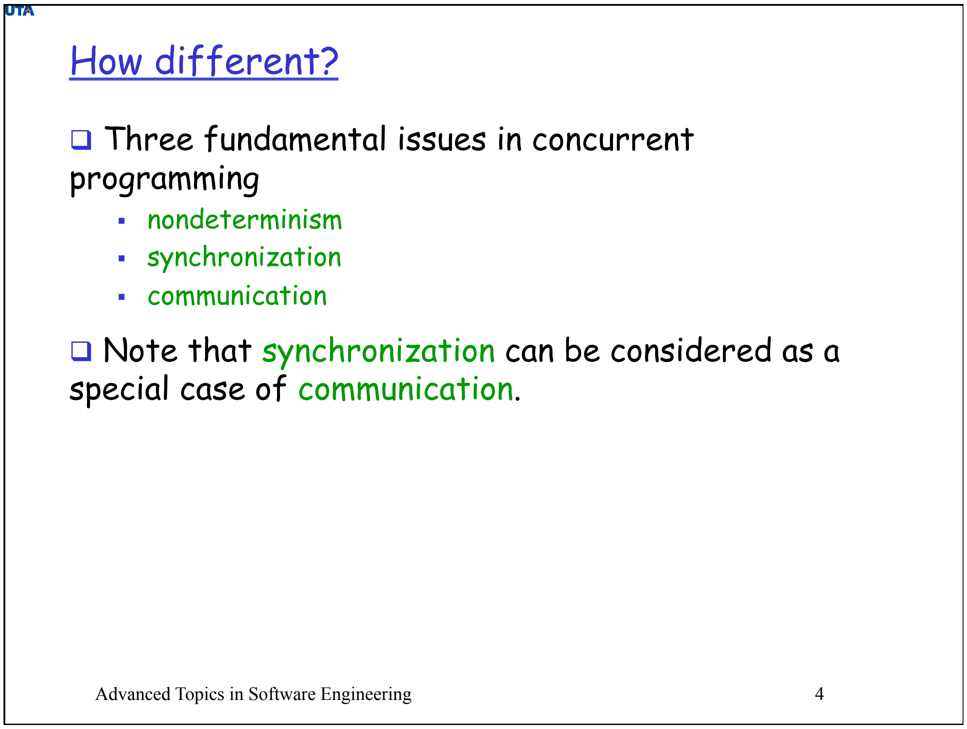
- multi-threaded programs: multiple threads running in a single process space
- distributed programs: multiple processes running in a distributed environment
- distributed multi-threaded programs: multiple processes running a distributed environment, and each process consists of multiple threads



Why Concurrency?

- ❑ Better utilization of resources, which often leads to increased computation efficiency
- ❑ Provides a natural model for many problem domains that are inherently concurrent
- ❑ Allows computation to be distributed in a network environment

So what is the catch?



How different?

- ❑ Three fundamental issues in concurrent programming
  - nondeterminism
  - synchronization
  - communication
- ❑ Note that synchronization can be considered as a special case of communication.

Nondeterminism

- An inherent property of concurrent programs - two executions of the same program with the same input may produce different, and sometimes unpredictable, results.
- Caused by one or more of the following factors:
  - the unpredictable rates of progress of threads
  - the unpredictable network latency
  - the explicit use of nondeterministic programming constructs

Synchronization

- Enforces a certain order on tasks that are performed by different threads/processes
  - Mutual exclusion ensures that critical sections in different threads do not execute at the same time.
  - Conditional synchronization delays a thread until a given condition is true.
  - Mutual exclusion is a special case of conditional synchronization.

Communication

❑ Exchange data between threads/processes that are running in parallel

- shared variables - a thread writes into a variable that is read by another thread.
- message passing - a thread sends a message that is received by another thread.

Process and Thread

In general, a process is a program in execution. A thread is a unit of control within a process.

When a program is executed, the operating system creates a process.

A process starts with a running thread, called the “main” thread, which executes the “main function” of the program.

Other threads can be created by the main thread, and these threads can create other threads.

Process vs. Thread

A process has its own stream of instructions and its own logical address space.

A thread has its own stream of instructions, but shares the same logic address space with other threads in the same process.

This difference has significant implications on how processes and threads communicate.

Introduction

- ❑ Overview
- ❑ Motivating Examples
- ❑ Interleaving Model
- ❑ Semantics of Correctness
- ❑ Testing, Debugging, and Verification

Example 1

Assume that integer x is initially 0.

Thread 1	Thread 2	Thread 3
(1) x = 1;	(2) x = 2;	(3) y = x;

What is the final value of y?

□ (1) (2) (3)

□ (2) (1) (3)

□ (3) (2) (1)

Example 2

Assume that y and z are initially 0.

Thread 1	Thread 2

x = y + z	y = 1;
	z = 2;

What is the final value of x?

Example 2

Thread 1

(1) load r1, y

(2) add r1, z

(3) store r1, x

Thread 2

(4) assign y, 1

(5) assign z, 2

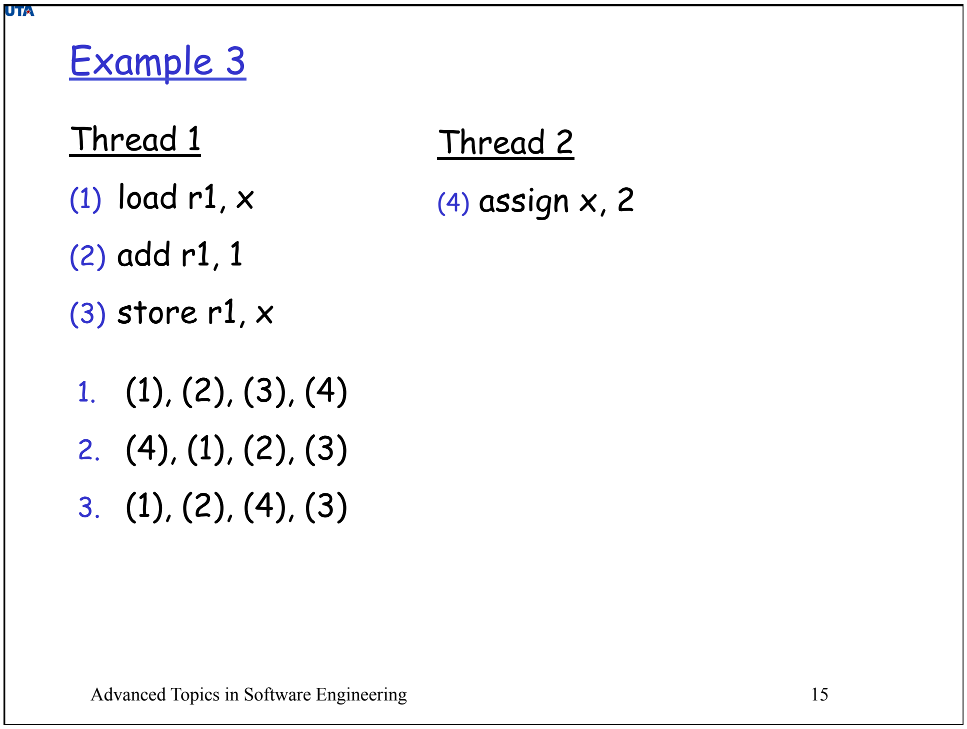
1. (1), (2), (3), (4), (5)
2. (4), (1), (2), (3), (5)
3. (1), (4), (5), (2), (3)
4. (4), (5), (1), (2), (3)

Example 3

Assume that the initial value of x is 0.

Thread 1	Thread 2
x = x + 1;	x = 2;

What is the final value of x?



Example 3

Thread 1

(1) load r1, x

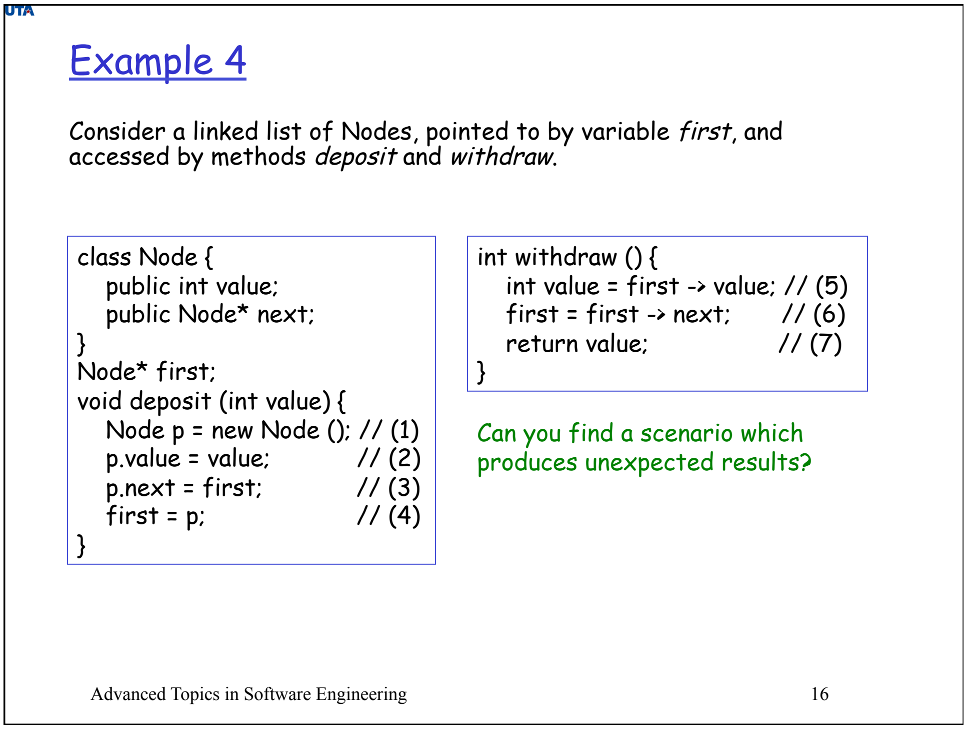
(2) add r1, 1

(3) store r1, x

Thread 2

(4) assign x, 2

1. (1), (2), (3), (4)
2. (4), (1), (2), (3)
3. (1), (2), (4), (3)



Example 4

Consider a linked list of Nodes, pointed to by variable *first*, and accessed by methods *deposit* and *withdraw*.

```
class Node {
   public int value;
   public Node* next;
}
Node* first;
void deposit (int value) {
   Node p = new Node (); // (1)
   p.value = value;      // (2)
   p.next = first;       // (3)
   first = p;            // (4)
}
```

```
int withdraw () {
   int value = first -> value; // (5)
   first = first -> next;      // (6)
   return value;               // (7)
}
```

Can you find a scenario which produces unexpected results?

Example 4

Suppose that the linked list pointed to by *first* is not empty and that methods deposit and withdraw are executed concurrently by two threads. In the following possible scenario, the deposited item has been lost.

```
int value = first -> value;     (5) in withdraw
Node p = new Node ();           (1) in deposit
p.value = value;                (2) in deposit
p.next = first;                 (3) in deposit
first = p;                      (4) in deposit
first = first -> next;          (6) in withdraw
return value;                   (7) in withdraw
```

Introduction

- ❑ Overview
- ❑ Motivating Examples
- ❑ Interleaving Model
- ❑ Semantics of Correctness
- ❑ Testing, Debugging, and Verification

They actually take turns …

A multithreaded process consists of a set of threads that execute simultaneously.

However, a single CPU can only execute one instruction (from one thread) at a time.

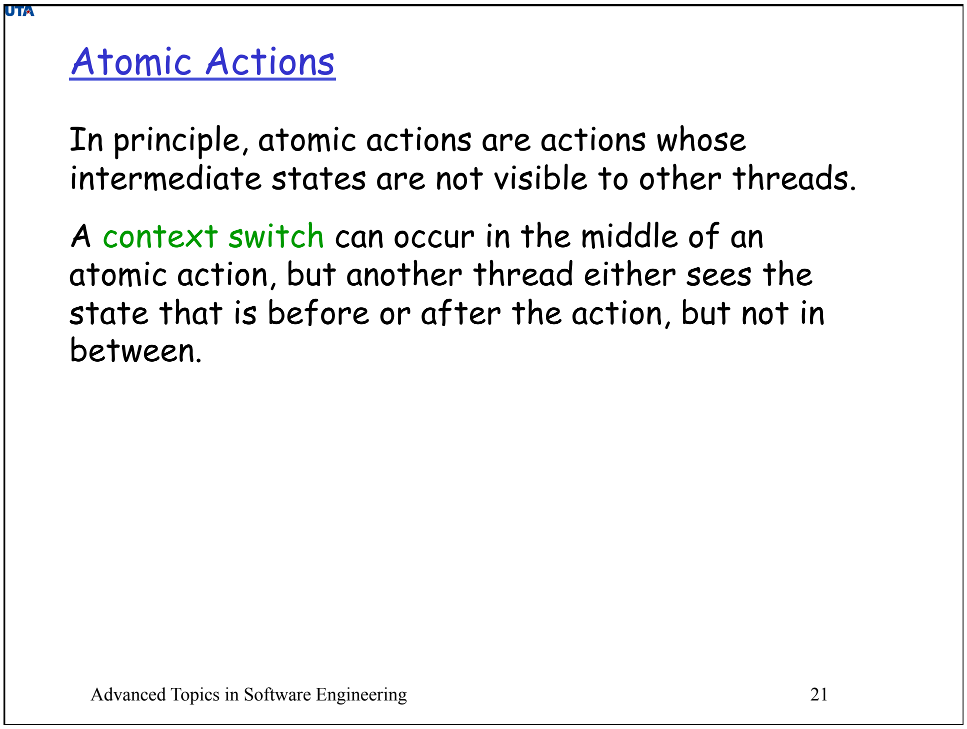
In general, each thread receives a time slice of the CPU. Threads that execute simultaneously actually take turns to run.

Scheduler

A hardware timer is used to generate interrupts at predetermined interval, which interrupts the execution of the current thread.

The operating system uses a scheduler program to determine which thread will take over the CPU for the next time slice.

A context switch is then performed between the current thread and the next thread.

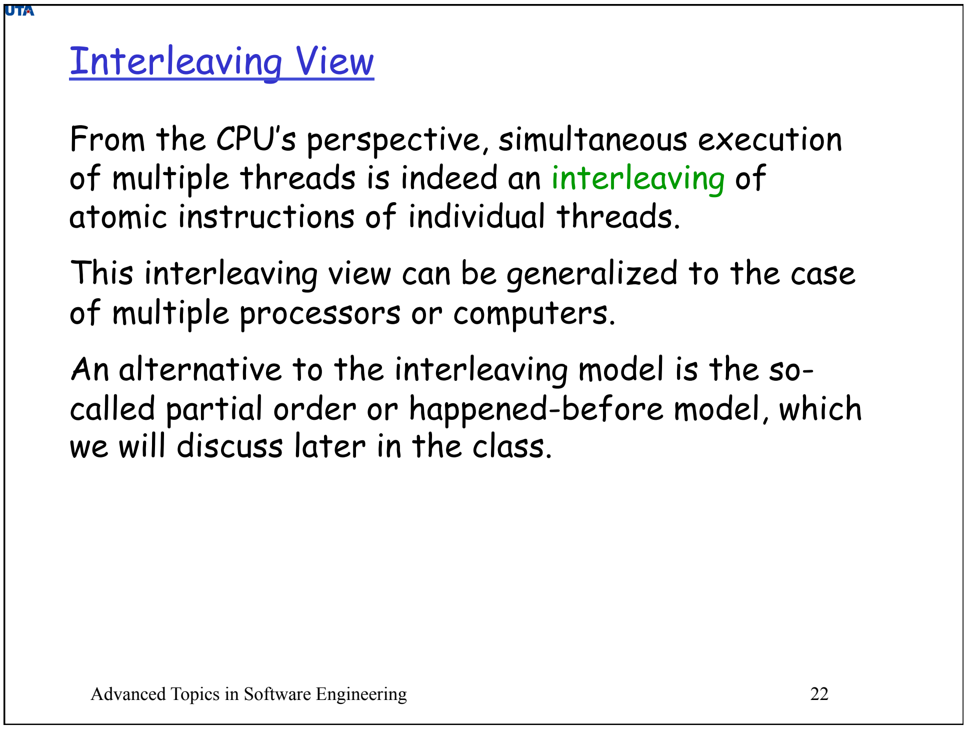


UTA

Atomic Actions

In principle, atomic actions are actions whose intermediate states are not visible to other threads.

A context switch can occur in the middle of an atomic action, but another thread either sees the state that is before or after the action, but not in between.



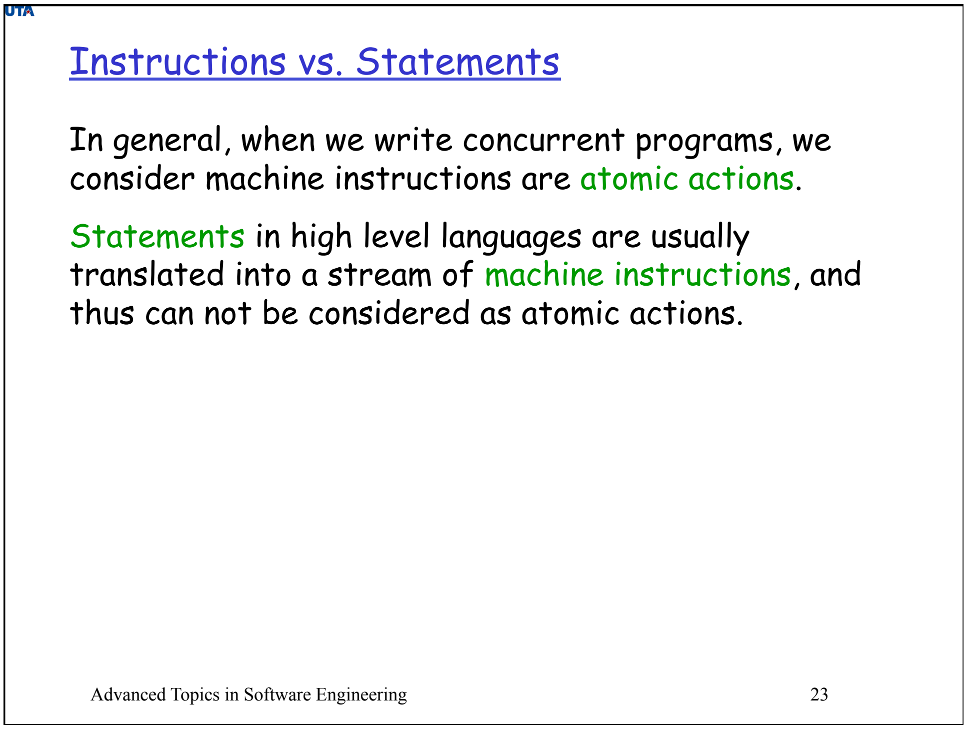
UTA

Interleaving View

From the CPU's perspective, simultaneous execution of multiple threads is indeed an interleaving of atomic instructions of individual threads.

This interleaving view can be generalized to the case of multiple processors or computers.

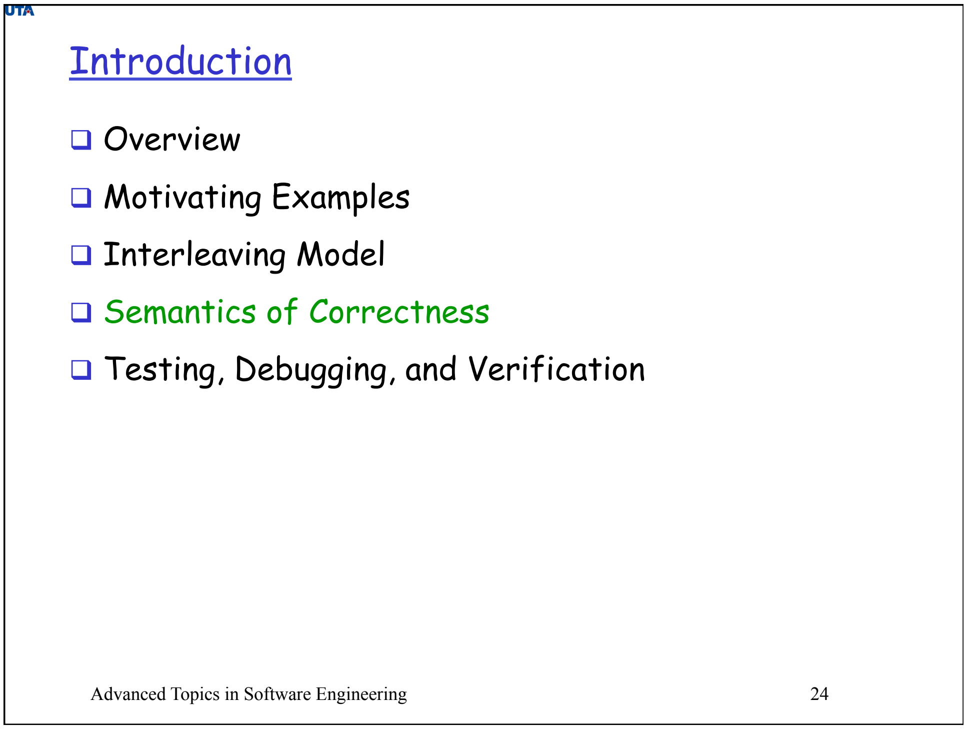
An alternative to the interleaving model is the so-called partial order or happened-before model, which we will discuss later in the class.



Instructions vs. Statements

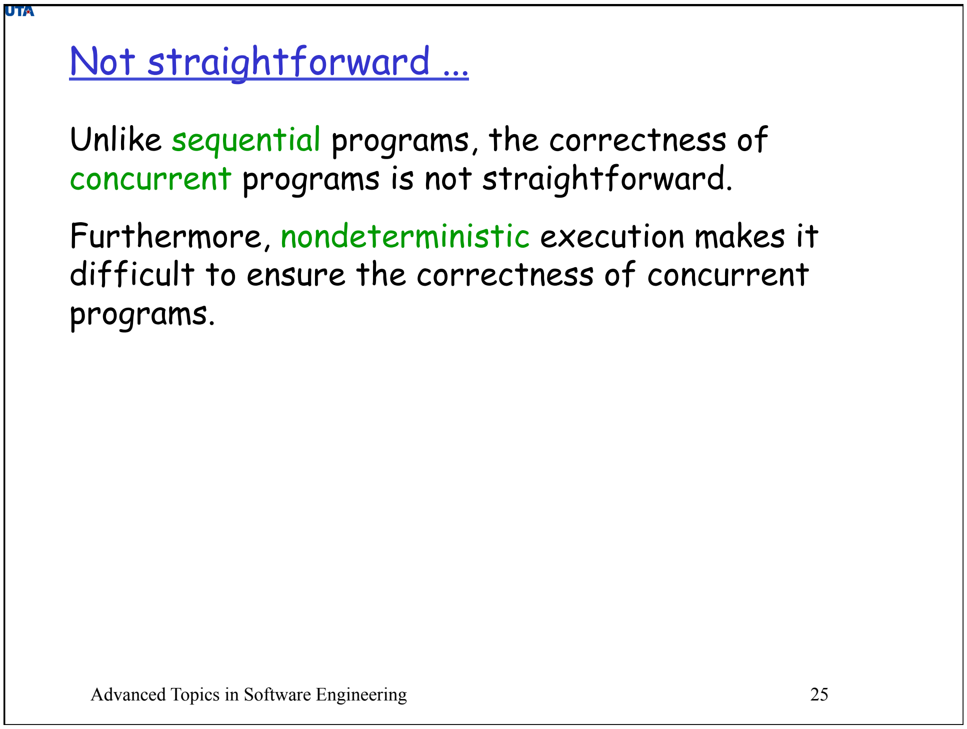
In general, when we write concurrent programs, we consider machine instructions are atomic actions.

Statements in high level languages are usually translated into a stream of machine instructions, and thus can not be considered as atomic actions.



Introduction

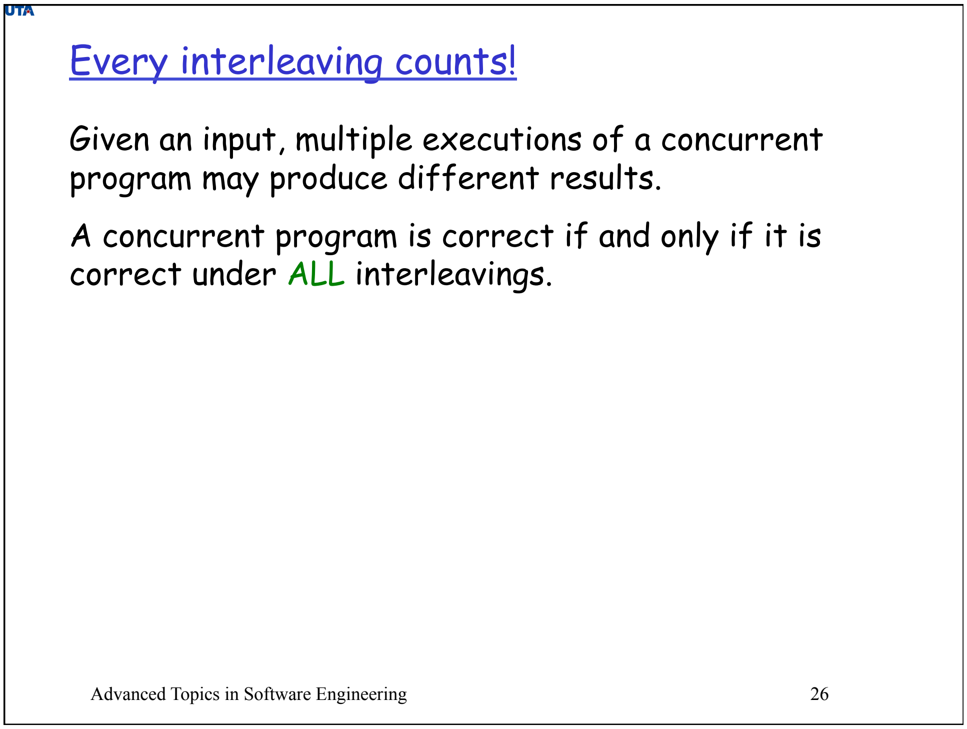
- ❑ Overview
- ❑ Motivating Examples
- ❑ Interleaving Model
- ❑ Semantics of Correctness
- ❑ Testing, Debugging, and Verification



Not straightforward ...

Unlike sequential programs, the correctness of concurrent programs is not straightforward.

Furthermore, nondeterministic execution makes it difficult to ensure the correctness of concurrent programs.



Every interleaving counts!

Given an input, multiple executions of a concurrent program may produce different results.

A concurrent program is correct if and only if it is correct under ALL interleavings.

Safety & Liveness

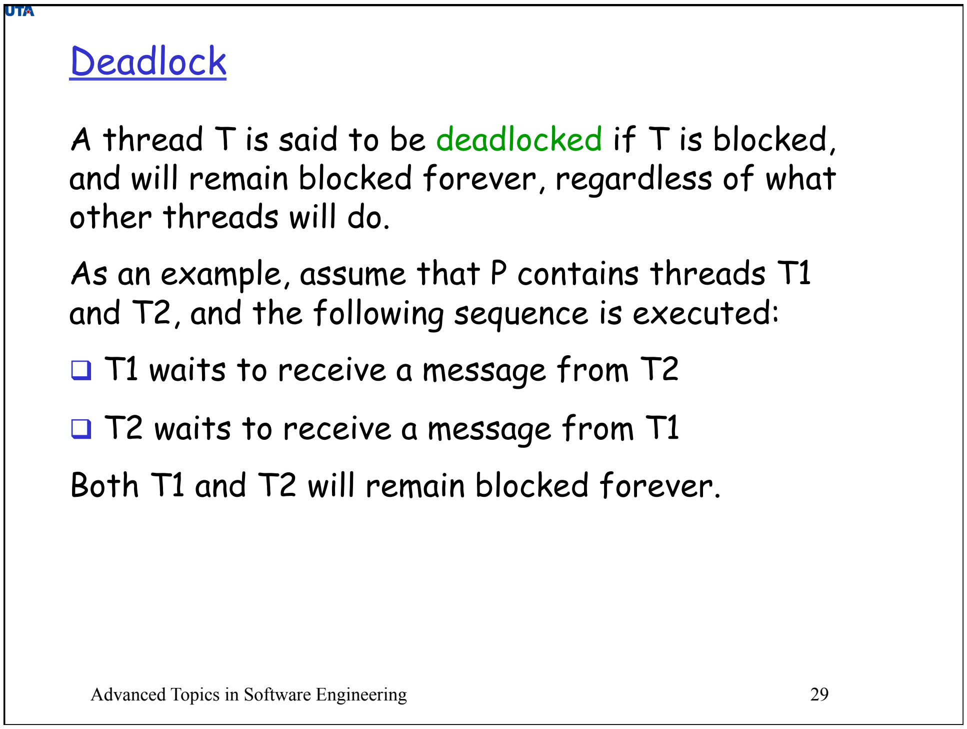
There are two types of properties:

- Safety: Nothing bad will happen.
- Liveness: Something good will eventually happen.

Usually domain specific …

Correctness is usually a domain specific concept. It depends on the functional requirements of a program.

However, deadlock, livelock, and starvation are properties that are important to all concurrent programs.



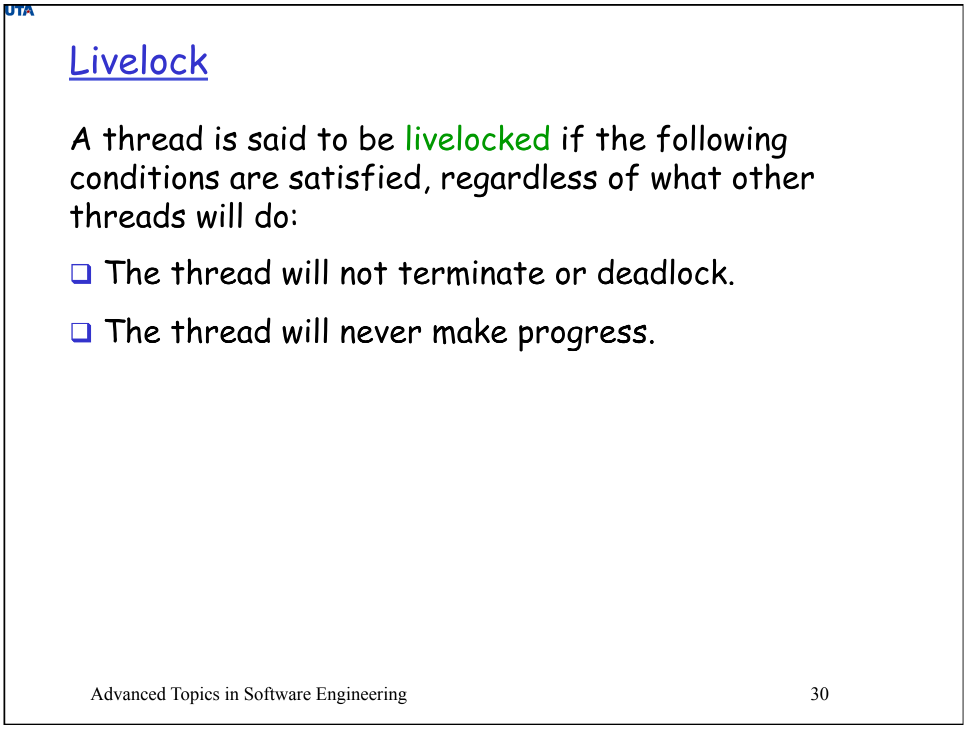
Deadlock

A thread T is said to be deadlocked if T is blocked, and will remain blocked forever, regardless of what other threads will do.

As an example, assume that P contains threads T1 and T2, and the following sequence is executed:

- T1 waits to receive a message from T2
- T2 waits to receive a message from T1

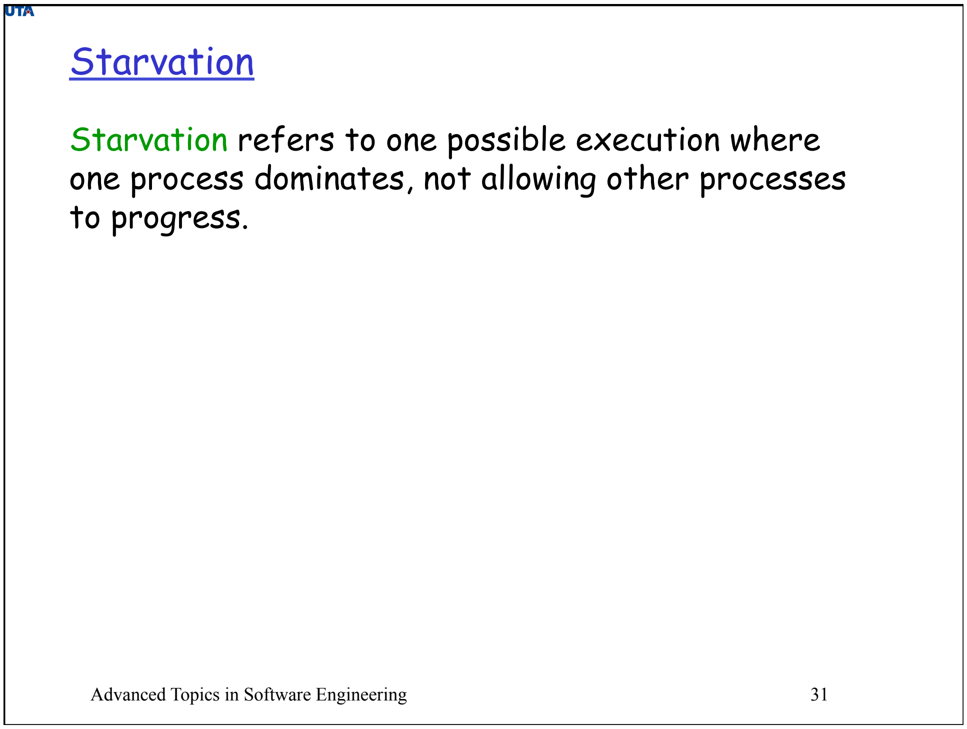
Both T1 and T2 will remain blocked forever.



Livelock

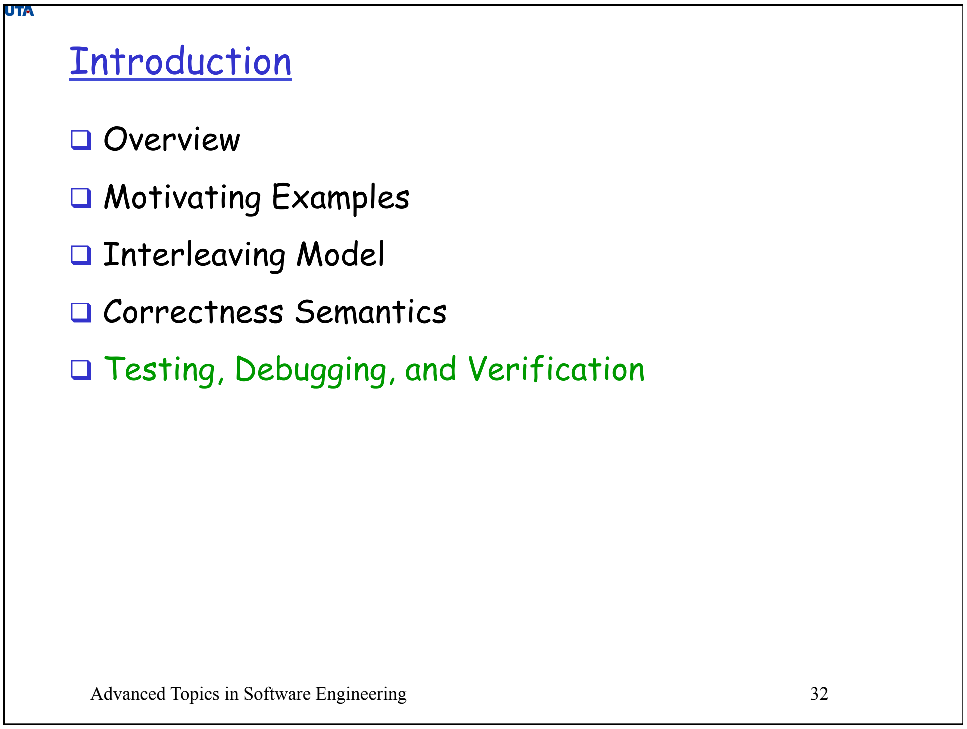
A thread is said to be livelocked if the following conditions are satisfied, regardless of what other threads will do:

- The thread will not terminate or deadlock.
- The thread will never make progress.



Starvation

Starvation refers to one possible execution where one process dominates, not allowing other processes to progress.



Introduction

- ❑ Overview
- ❑ Motivating Examples
- ❑ Interleaving Model
- ❑ Correctness Semantics
- ❑ Testing, Debugging, and Verification

Failure & Faults

A failure is an observed departure of the external result of software operation from software requirements or user expectations

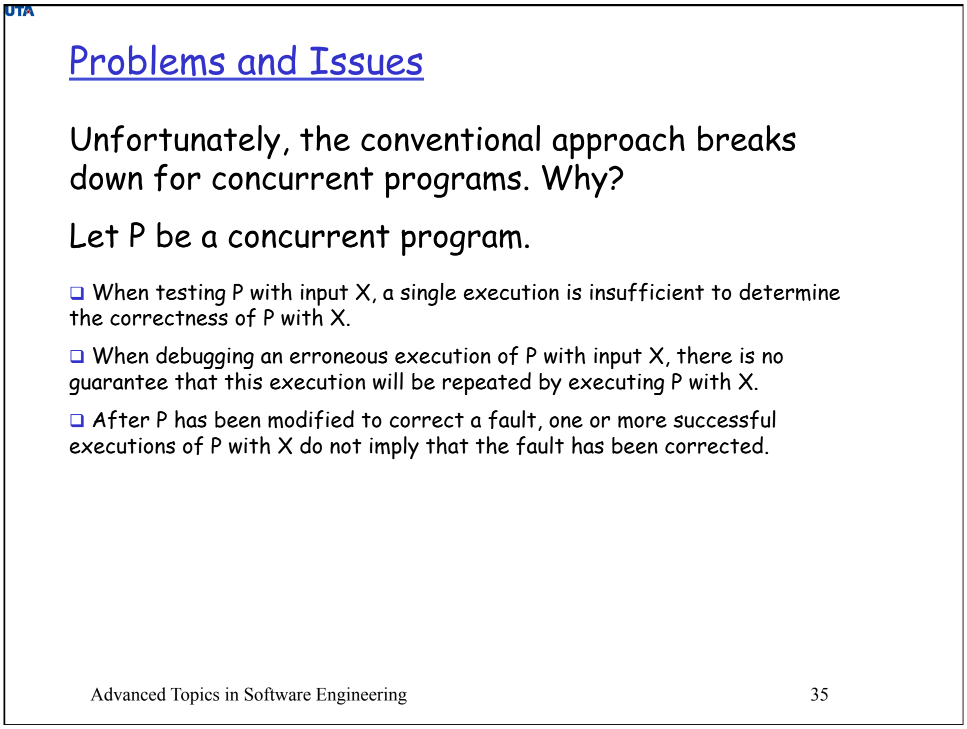
A fault is a defective, missing, or extra instruction or a set of related instructions that is the cause of one or more actual or potential failures.

Testing & Debugging

Testing is to find program failures; debugging is to locate and correct program faults.

The conventional approach to testing and debugging a program is as follows:

1. Select a set of test inputs.
2. Execute the program once with each input and check the test results.
3. If a test input finds a failure, execute the program again with the same input to locate and correct the fault that caused the failure
4. After the fault has been located and corrected, execute the program again with each of the test inputs to verify that the fault has been corrected and that no new faults have been introduced.



Problems and Issues

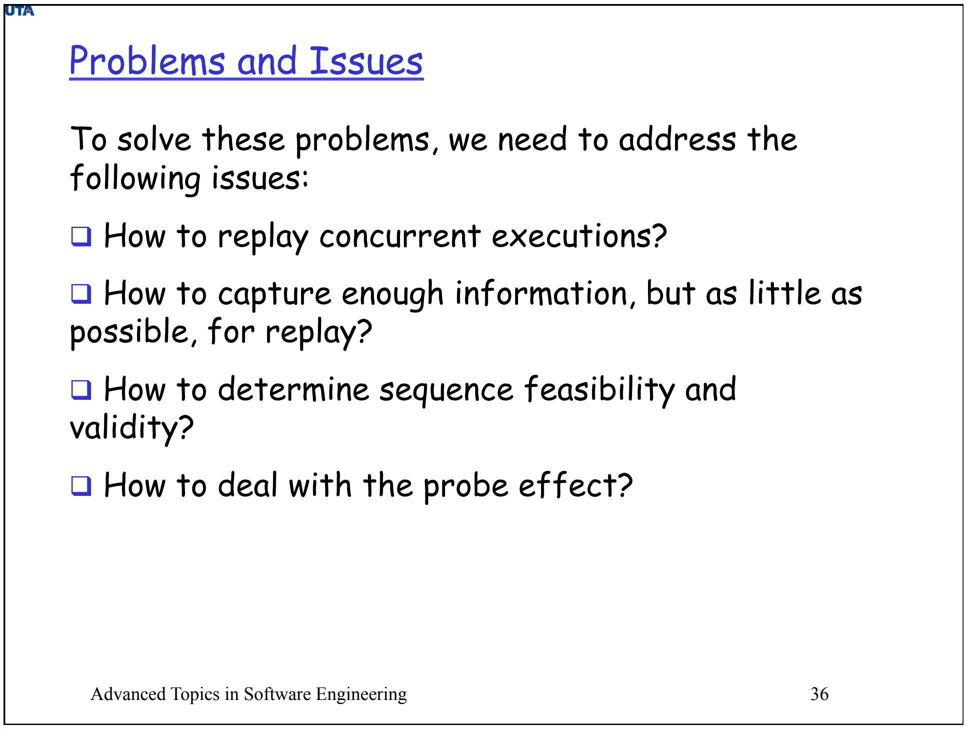
Unfortunately, the conventional approach breaks down for concurrent programs. Why?

Let P be a concurrent program.

❑ When testing P with input X, a single execution is insufficient to determine the correctness of P with X.

❑ When debugging an erroneous execution of P with input X, there is no guarantee that this execution will be repeated by executing P with X.

❑ After P has been modified to correct a fault, one or more successful executions of P with X do not imply that the fault has been corrected.



Problems and Issues

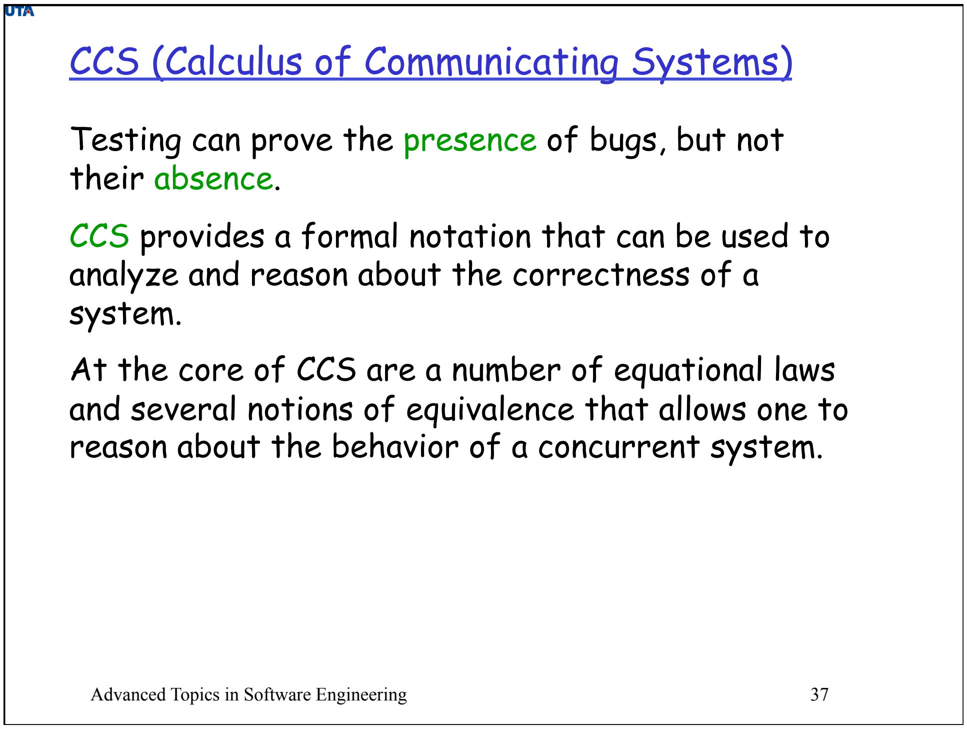
To solve these problems, we need to address the following issues:

❑ How to replay concurrent executions?

❑ How to capture enough information, but as little as possible, for replay?

❑ How to determine sequence feasibility and validity?

❑ How to deal with the probe effect?

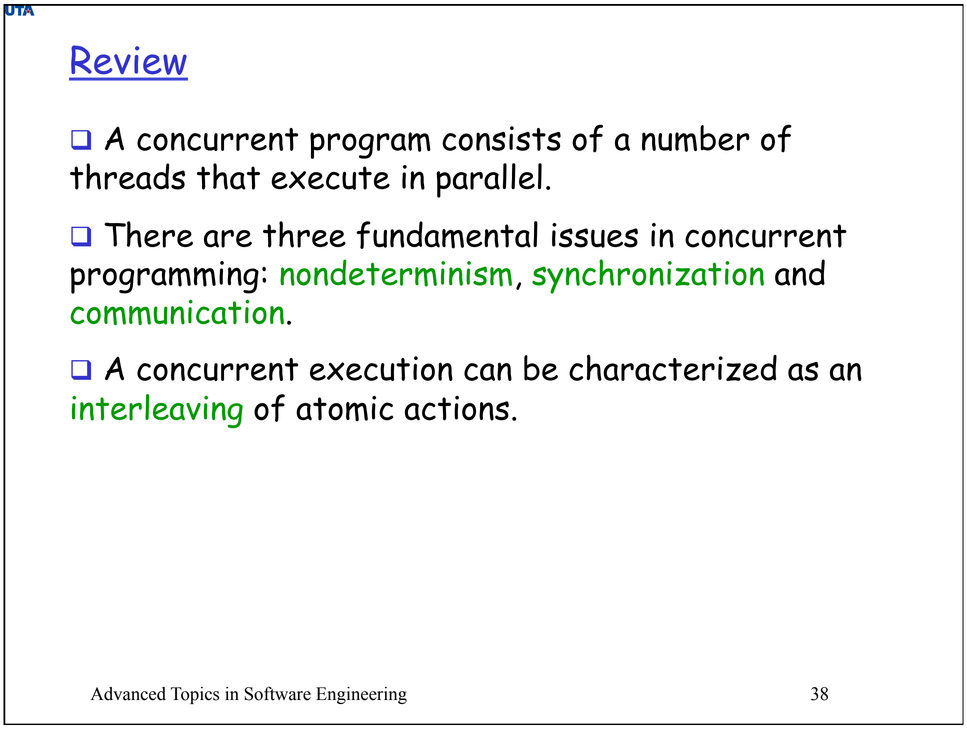


CCS (Calculus of Communicating Systems)

Testing can prove the presence of bugs, but not their absence.

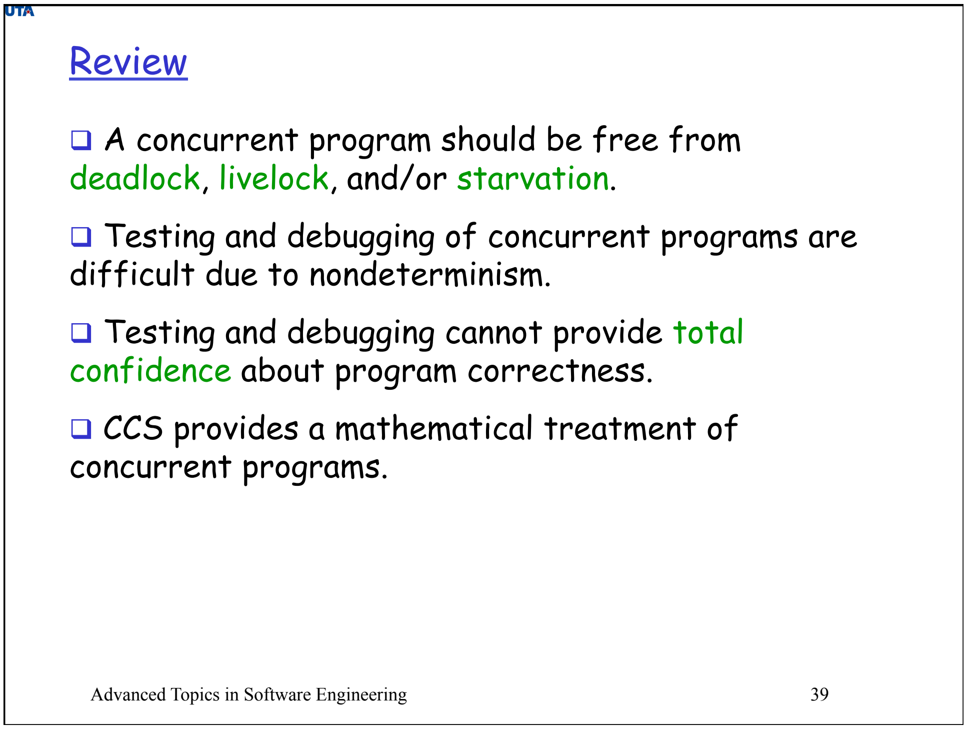
CCS provides a formal notation that can be used to analyze and reason about the correctness of a system.

At the core of CCS are a number of equational laws and several notions of equivalence that allows one to reason about the behavior of a concurrent system.



Review

- ❑ A concurrent program consists of a number of threads that execute in parallel.
- ❑ There are three fundamental issues in concurrent programming: nondeterminism, synchronization and communication.
- ❑ A concurrent execution can be characterized as an interleaving of atomic actions.



Review

- ❑ A concurrent program should be free from deadlock, livelock, and/or starvation.

- ❑ Testing and debugging of concurrent programs are difficult due to nondeterminism.

- ❑ Testing and debugging cannot provide total confidence about program correctness.

- ❑ CCS provides a mathematical treatment of concurrent programs.