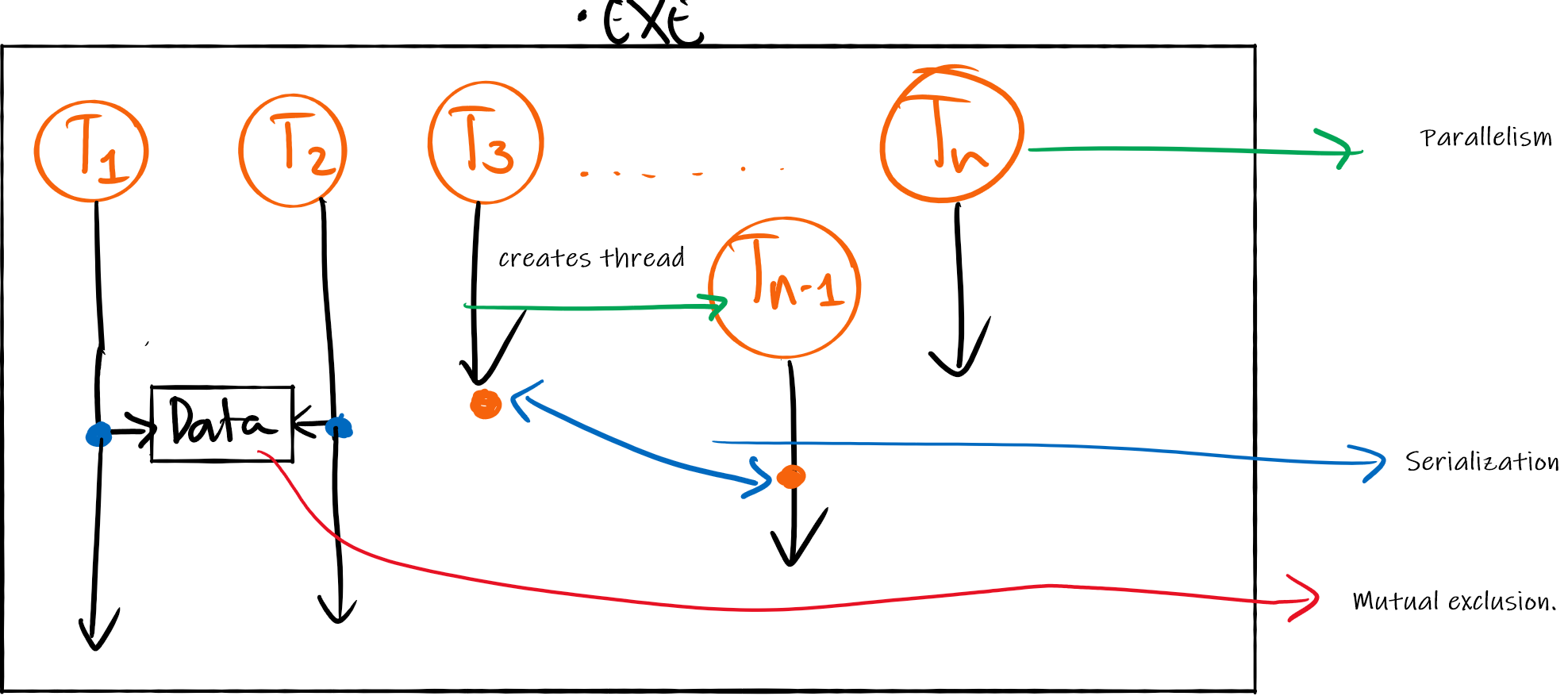
|  |  |
| --- | --- |
| **GDPARCM Lecture – Thread Synchronization** | Instructor: Neil Patrick Del Gallego |

The goal is to:

* Introduce students to concepts of process/thread synchronization.
* Understand characteristics of “concurrency”.
* Help students realize the difficulty of synchronizing processes and how to solve them correctly.

**Synchronization**

* Attempting to perform multiple tasks at the same time in an organized manner.
* Developers who create concurrent systems are primarily concerned with:
  + Parallelism – Task A and B should execute at the same time to speed up performance.
  + Serialization – Task A should execute before Task B.
  + Mutual exclusion – Datatype X should only be modified by one task, A or B.



**Why concurrency programming is difficult**

Consider this sample program with two threads, Alice and Bob.

|  |  |
| --- | --- |
|  |  |

* First possibility – The computer is parallel. Alice and Bob **MAY** execute line 2 and 9 at the same time. In that case it is not easy to know if a statement on one processor is executed before a statement on another.
* Second possibility – Single processor is running multiple threads of execution. If there are multiple threads inside a single processor, then the processor can work on one for a while, then switch to another, and so on.

**What is concurrency?**

* The developer has no control over when each thread runs. The operating system makes those decisions (depending on the CPU scheduling scheme).
* The developer can't tell when statements in different threads will be executed.
* Two events are **concurrent** if we cannot tell by looking at the program/code which will happen first.

**Introduction to Serialization**

* Task A should execute before task B.
* callBob() passes a message to Bob, which is received by waitForCall(). As long as waitForCall() does not receive any message from callBob(), it cannot continue executing (thread sleeps).
* QUESTION: Where should we put waitForCall() to guarantee that Alice calls eatLunch() first?

|  |  |
| --- | --- |
|  |  |

Adding waitForCall at line 10 puts the Bob thread to sleep, until callBob() is called by Alice thread. This is called **message passing**.

Alice thread execution – line A2 < A3 < A4 < A5. where **X < Y** means task X happens BEFORE task Y. **X > Y** means X happens AFTER task Y.

Bob thread execution – line B9 < B10 < B11

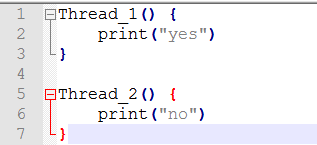
With message passing, it guarantees that: A4 < B11.

**Activity:**

Are the following statements true/false? Explain why.

* A2 < B9? – False. B9 can happen first.
* B9 < A3? – False. A3 can happen first.
* B11 > B10 > A5 > A4? – True. waitForCall() guarantees this.

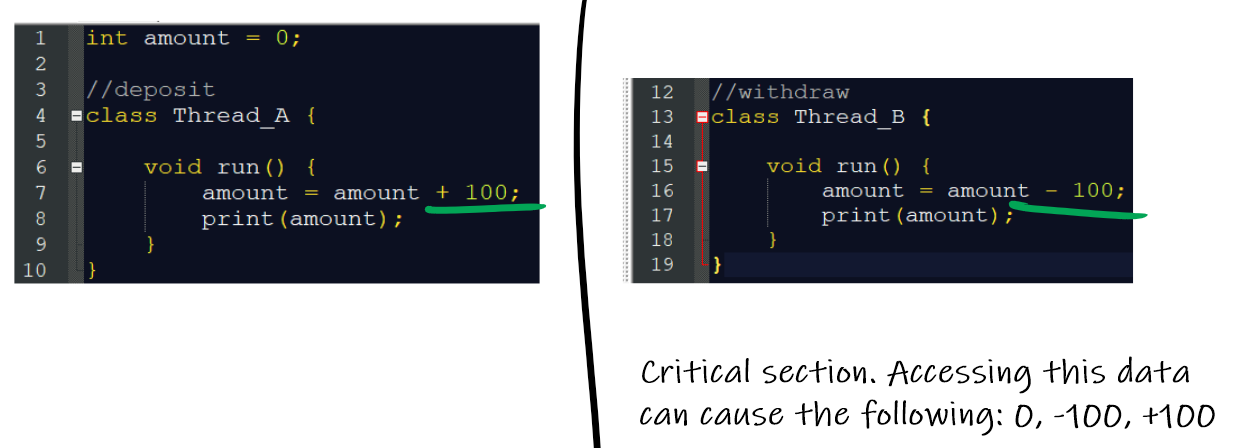
**Threads are non-deterministic!**

* Concurrent programs are **non-deterministic.** It is impossible to tell by looking at the code, which will execute first.
* Non-determinism is one of the things that makes concurrent programs hard to debug. A program might work correctly 1000 times in a row, and then crash on the 1001st run, depending on the particular decisions of the scheduler.
* These kinds of bugs are almost impossible to find by testing; they can only be avoided by careful programming.

**Problem: Withdraw-Deposit**

Consider this problem below.

|  |  |
| --- | --- |
| int amount = 0; | |
| **//deposit**  class Thread\_A {    void run() {  amount = amount + 100;  print(amount);  }  } | **//withdraw**  class Thread\_B {    void run() {  amount = amount - 100;  print(amount);  }  } |



The three essential criteria for solving the critical section problem:

1. **Mutual Exclusion** - If process ***Pi*** is executing in its critical section, then no other processes can be executing in their critical sections
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 processes that will enter the critical section next cannot be postponed indefinitely
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

Peterson’s Solution:

|  |  |
| --- | --- |
|  | |
|  |  |

Explanation why it meets all three criteria:

1. Mutual exclusion – If Thread\_A is in the critical section, then flag[0] is true. Thread\_B will have turn = 0, which causes it to perform busy waiting. Vice versa, if Thread\_B is in the critical section, then flag[1] is true. Thread\_A will have turn = 1 which will cause Thread\_A to perform busy waiting.
2. Progress – Once Thread\_A finishes, it guarantees that Thread\_B becomes the next to access the critical section because flag[0] becomes false, and vice versa.
3. Bounded waiting – Since flag[0]/flag[1] will become false in the future, this guarantees bounded waiting on the thread executing the while loop.

**Activity:**

Consider the pseudocode of Petearson’s solution being used by two threads.

|  |  |
| --- | --- |
|  | |
|  |  |

What do you think will happen from the above code?