Part III

Synchronization

Critical Section and Mutual Exclusion

The question of whether computers can think is just like the question of whether submarines can swim

Edsger W. Dijkstra

Spring 2019

1

Process Synchronization Topics

- **Why is synchronization needed?**
- **Race Conditions**
- **Critical Sections**
- **Pure Software Solutions**
- **Hardware Support**
- **Semaphores**
- **Race Conditions, Revisited**
- **Monitors**

Synchronization Needed! 1/6

```
int  a[3] = { 3, 4, 5};
```

Process 1   **Process 2**

```
a[1] = a[0] + a[1];          a[2] = a[1] + a[2];
```

```
a[3] = { 3, ?, ? }
```

Statement level execution interleaving

Synchronization Needed! 2/6

```
        int  a[3] = { 3, 4, 5};

      Process 1                      Process 2

a[1] = a[0] + a[1];          a[2] = a[1] + a[2];
```

▪ If process 1 updates `a[1]` first, `a[1]` is 7, and `a[ ]={3,7,5}`
▪ Then, process 2 uses the new `a[1]` to computes `a[2]`, and
   `a[ ]={3,7,12}`

▪ If process 2 uses `a[1]` first, now `a[2]` is 9, and `a[ ]={3,4,9}`
▪ Then, process 1 computes `a[1]`, and `a[ ]={3,7,9}`

Results are non-deterministic!
Situation can be much worse.

4

Synchronization Needed! 3/6

```
int  Count = 10;
```

Higher-level language statements
are *not* atomic

Process 1   **Process 2**

```
Count++; Count--;
```

Count = **9, 10 or 11?**

Synchronization Needed! 4|6

```
int   Count = 10;
```

Process 1   **Process 2**

```
LOAD   Reg, Count      LOAD   Reg, Count
ADD    #1              SUB    #1
STORE  Reg, Count      STORE  Reg, Count
```

The problem is that the execution flow may be switched in the middle. *Results become non-deterministic!*

instruction level execution interleaving

Synchronization Needed! 5/6

<table>
<tr><th colspan="3">Process 1</th><th colspan="3">Process 2</th></tr>
<tr><th>Inst</th><th>Reg</th><th>Memory</th><th>Inst</th><th>Reg</th><th>Memory</th></tr>
<tr><td>LOAD</td><td>10</td><td>10</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>LOAD</td><td>10</td><td>10</td></tr>
<tr><td></td><td></td><td></td><td>SUB</td><td>9</td><td>10</td></tr>
<tr><td>ADD</td><td>11</td><td>10</td><td></td><td></td><td></td></tr>
<tr><td>STORE</td><td>11</td><td>11</td><td colspan="3">overwrites the previous value 11</td></tr>
<tr><td></td><td></td><td></td><td>STORE</td><td>9</td><td>9</td></tr>
</table>

Always use instruction level interleaving to show race conditions

Synchronization Needed! 6/6

Process 1				Process 2		
Inst	**Reg**	**Memory**		**Inst**	**Reg**	**Memory**
LOAD	10	10				
ADD	11	10				
				LOAD	10	10
				SUB	9	10
				STORE	9	**9**
STORE	11	**11**		*overwrites*	*the previous*	*value 9*

Always use instruction level interleaving to show race conditions 8

Race Conditions

- **A *Race Condition* occurs, if**

  - ❖ **two or more processes/threads manipulate a shared resource concurrently, and**

  - ❖ **the outcome of the execution depends on the particular order in which the access takes place.**

- *Synchronization* **is needed to prevent race conditions from happening.**

- **Synchronization is a difficult topic. Don't miss classes; otherwise, you will miss a lot of things.**

Execution Sequence Notes: 1/3

- *You must always use instruction level interleaving to demonstrate the existence of race conditions*, because

  a) higher-level language statements are not atomic and can be switched in the middle of execution

  b) instruction level interleaving can show clearly the "sharing" of a resource among processes and threads.

  c) *two* execution sequences are needed to show the answer depends on order of execution.

Execution Sequence Notes: 2/3

```
int  a[3] = { 3, 4, 5};
```

Process 1   **Process 2**

```
a[1] = a[0] + a[1];        a[2] = a[1] + a[2];
```

Execution Sequence 1

Process 1	Process 2	Array `a[ ]`
`a[1]=a[0]+a[1]`		{ 3, 7, 5 }
	`a[2]=a[1]+a[2]`	{ 3, 7, 12 }

There is no concurrent sharing, not a valid example for a race condition.

Execution Sequence 2

Process 1	Process 2	Array `a[ ]`
	`a[2]=a[1]+a[2]`	{ 3, 4, 9 }
`a[1]=a[0]+a[1]`		{ 3, 7, 9 }

11

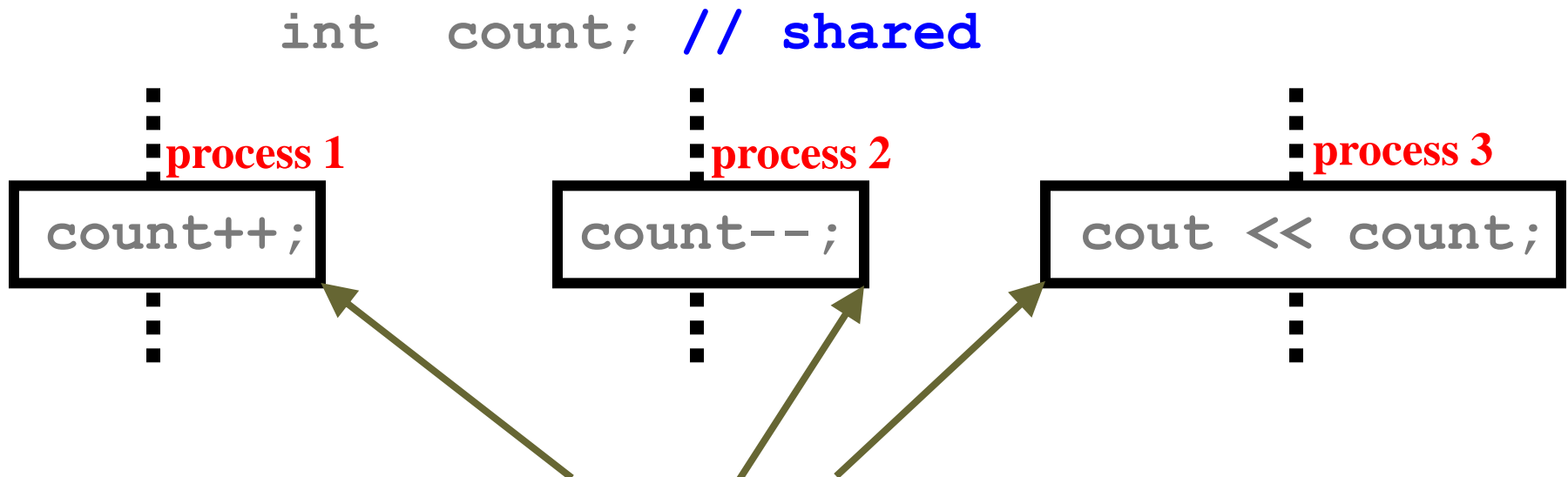# Execution Sequence Notes: 3/3

```
int  Count = 10;


Process 1                Process 2
LOAD  Reg, Count         LOAD  Reg, Count
ADD   #1                 SUB   #1
STORE Reg, Count         STORE Reg, Count
```

Process 1	Process 2	Memory
LOAD Reg, Count		10
	LOAD Reg, Count	10
	SUB #1	10
ADD #1		10
STORE Reg, Count		11
	STORE Reg, Count	9

variable count is shared concurrently here

Critical Section

- **A *critical section*, CS, is a section of code in which a process accesses shared resources.**

```
int  count; // shared
```

process 1

```
count++;
```

process 2

```
count--;
```

process 3

```
cout << count;
```

These are critical sections since `count` is a shared resource

13

Mutual Exclusion

- **To avoid race conditions, the execution of critical sections must be *mutually exclusive* (e.g., at most one process can be in its critical section at any time).**

- **The *critical-section problem* is to design a protocol with which processes can use to cooperate and ensure mutual exclusion.**

The Critical Section Protocol

```
do {
```

entry section

critical section

exit section

```
} while (1);
```
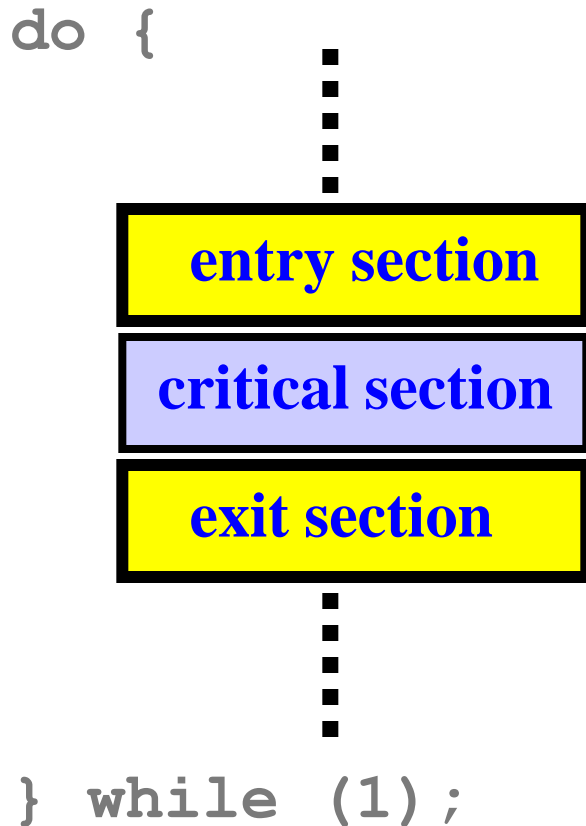
- **A critical section protocol consists of two parts: an *entry section* and an *exit section*.**

- **Between them is the critical section that must run in a mutually exclusive way.**

Good Solutions to the CS Problem

- **A good solution to the critical section problem must satisfy the following three conditions:**
  - ❖**Mutual Exclusion**
  - ❖**Progress**
  - ❖**Bounded Waiting**
- **Moreover, the solution cannot depend on CPU's relative speed, timing, scheduling policy and other external factors.**

Mutual Exclusion

- **If a process P is executing in its critical section, *no* other processes can be executing in their corresponding critical sections.**

- **The entry protocol should be able to block processes that wish to enter but should not.**

- **When the process that is executing in its critical section exits, the entry protocol must be able to know this fact and allows a waiting process to enter.**

Progress

- **If *no* process is executing in its critical section and some processes want to enter their corresponding critical sections, then**

  1. **Only those processes that are waiting to enter can participate in the competition (to enter their critical sections) and no other processes can influence this decision.**

  2. **This decision cannot be postponed indefinitely (i.e., finite decision time). Thus, one of the waiting processes can enter its critical section.**

Bounded Waiting

- **After** a process made a request to enter its critical section and **before** it is granted the permission to enter, there exists a *bound* on the **number of turns** that other processes are allowed to enter.

- *Finite is not the same as bounded*. The former means any value you can write down (e.g., billion, trillion, etc) while the latter means this value has to be no larger than a particular one (i.e., the bound).

Progress vs. Bounded Waiting

- *Progress* <u>does not</u> imply *Bounded Waiting*: *Progress* says a process can enter within a finite decision time. It does not say which process can enter, and there is no guarantee for bounded waiting.

- *Bounded Waiting* <u>does not</u> imply *Progress*: Even through we have a bound, all processes may be locked up in the enter section (i.e., infinite decision time).

- Therefore, *Progress* and *Bounded Waiting* are independent of each other.

A Few Related Terms: 1/7

- ***Deadlock-Freedom***: **If two or more processes are trying to enter their critical sections, one of them will eventually enter. This is *Progress* without the "outsiders having no influence" condition.**

- **Since the enter section is able to select a process to enter, the decision time is certainly finite.**

A Few Related Terms: 2/7

- *r-Bounded Waiting*: **There exists a fixed value r such that after a process made a request to enter its critical section and before it is granted the permission to enter, no more than r other processes are allowed to enter.**

- **Therefore, bounded waiting means there is a r such that the waiting is r-bounded.**

A Few Related Terms: 3/7

- ***FIFO***: **No process that is about to enter its critical section can pass an already waiting process. *FIFO* is usually referred to as *0-bounded*.**

- ***Linear-Waiting*** (***1-Bounded Waiting***): **No process can enter its critical section twice while there is a process waiting.**

A Few Related Terms: 4/7

- *Starvation-Freedom*: **If a process is trying to enter its critical section, it will eventually enter.**

- *Questions*:

  1. **Does starvation-freedom imply deadlock-freedom?**
  2. **Does starvation-freedom imply bounded-waiting?**
  3. **Does bounded-waiting imply starvation-freedom?**
  4. **Does bounded-waiting *AND* deadlock-freedom imply starvation-freedom?**

A Few Related Terms: 5/7

- *Question (1)*: **Does starvation-freedom imply deadlock-freedom?**

- *Yes!* **If every process can eventually enter its critical section, although waiting time may vary, it means the decision time of selecting a process is finite. Otherwise, all processes would wait in the enter section.**

A Few Related Terms: 6/7

- *Question (2)*: **Does starvation-freedom imply bounded-waiting?**
- *No!* **This is because the waiting time may not be bounded even though each process can enter its critical section.**

A Few Related Terms: 7/7

- *Question (3)*: **Does bounded-waiting imply starvation-freedom?**

- *No.* **Bounded-Waiting does not say if a process can actually enter. It only says there is a bound. For example, all processes are locked up in the enter section (i.e., failure of *Progress*).**

- **We need *Progress* + *Bounded-Waiting* to imply *Starvation-Freedom* (*Question (4)*). In fact, *Progress* + *Bounded-Waiting* is stronger than *Starvation-Freedom*. *Why?***

The End