

# Races and Synchronization

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonse

Department of Computer Science and Engineering  
University at Buffalo



# Races

**Races**, or **race conditions**, are situations where:

- Two or more events are **dependent upon each other**
- Some of the events **may happen in more than one order**, or even simultaneously
- There exists some ordering of the events that is **incorrect**

For example:

- Some state will be updated multiple times
- Output will be produced based on the state

If some order of updates results in invalid output, **this is a race**.

# Synchronization

**Synchronization**, in the context of a computer program, is the **deliberate ordering of events** via some mechanism.

There are many synchronization mechanisms, working in different ways.

Synchronization mechanisms may:

- Directly **order** events
- Simply ensure that events **do not happen simultaneously**
- Ensure that two events **begin** at the same time
- ...

Synchronization is how we **avoid races**.

# Lecture Question

Ask a review question!

# Race Conditions

CS:APP [2] defines a **race** as:

*[...] when the correctness of a program depends on one thread reaching point  $x$  in its control flow before another thread reaches point  $y$ .*

Note that there may be **many points**  $x$  and  $y$  in a program!

The relationship between  $x$  and  $y$  **may change over time**, as well.

For example, “**once thread  $T_1$  has reached point  $p$** , it must reach point  $x$  before **any other thread** reaches point  $y$ .”

# Data Races

While **data races**, or races involving **modification of data**, are not the only kind of race, they are **very common**.

A data race occurs when:

- Two or more **concurrent flows** **access shared state**
- One or more of these flows **modifies** the state
- The **order** of the accesses/modifications **is important**
- The **synchronization** in use is **insufficient** to preserve the necessary order

Races among **any number of concurrent flows** for the **same data** may be reduced to **a set of pairwise races**.

At least one access in each pair **must be a modifying operation**.

# Example Race

Consider two threads running the following code:

```
char *strings[4];  
int nstrings;  
  
void setstring(char *str) {  
    int index = nstrings;  
    strings[index] = str;  
    nstrings++;  
}
```

$T_1$  index:

$T_2$  index:

nstrings: 0

strings: 

NULL
NULL
NULL
NULL

 ←

# Example Race

Consider two threads running the following code:

```
char *strings[4];  
int nstrings;
```

```
void setstring(char *str) {  
T1 → int index = nstrings;  
      strings[index] = str;  
      nstrings++;  
}
```

T<sub>1</sub> index: 0

T<sub>2</sub> index:

nstrings: 0

strings: 

NULL
NULL
NULL
NULL

 ←



# Example Race

Consider two threads running the following code:

```
char *strings[4];
int nstrings;
```

```
void setstring(char *str) {
    int index = nstrings;
    strings[index] = str;
    nstrings++;
}
```

$T_1 \rightarrow$  int index = nstrings;  $\leftarrow T_2$

$T_1$  index: 0

$T_2$  index: 0

nstrings: 0

strings: 

NULL
NULL
NULL
NULL

 $\leftarrow$

# Example Race

Consider two threads running the following code:

```
char *strings[4];  
int nstrings;
```

```
void setstring(char *str) {
```

```
    T1 → int index = nstrings;  
    strings[index] = str; ← T2  
    nstrings++;  
}
```

T<sub>1</sub> index: 0

T<sub>2</sub> index: 0

nstrings: 0

strings: 

T <sub>2</sub>
NULL
NULL
NULL

 ←

# Example Race

Consider two threads running the following code:

```
char *strings[4];  
int nstrings;
```

```
void setstring(char *str) {  
    int index = nstrings;  
    strings[index] = str;  
    nstrings++;  
}
```

$T_1 \rightarrow$  strings[index] = str;  $\leftarrow T_2$

$T_1$  index: 0

$T_2$  index: 0

nstrings: 0

strings: 

$T_1$
NULL
NULL
NULL

 ←

# Example Race

Consider two threads running the following code:

```
char *strings[4];
int nstrings;
```

```
void setstring(char *str) {
    int index = nstrings;
    strings[index] = str;
    nstrings++;
}
```

$T_1 \rightarrow$

$\leftarrow T_2$

$T_1$  index: 0

$T_2$  index: 0

nstrings: 1

strings:

$T_1$	
NULL	
NULL	
NULL	

←

# Example Race

Consider two threads running the following code:

```
char *strings[4];
int nstrings;
```

```
void setstring(char *str) {
    int index = nstrings;
    strings[index] = str;
    nstrings++;
}
```

$T_1 \rightarrow$

$\leftarrow T_2$

$T_1$  index: 0

$T_2$  index: 0

nstrings: 2

strings:	$T_1$
	NULL
	NULL
	NULL

←

# Example Race

Consider two threads running the following code:

```
char *strings[4];  
int nstrings;  
  
void setstring(char *str) {  
    int index = nstrings;  
    strings[index] = str;  
    nstrings++;  
}
```

$T_1$  index:

$T_2$  index:

nstrings: 2

strings:	$T_1$	←
	NULL	
	NULL	
	NULL	

This is probably not what was intended!

# Critical Sections

```
1 void setstring(char *str) {  
2     int index = nstrings;  
3     strings[index] = str;  
4     nstrings++;  
5 }
```

Lines 2-4 of `setstring()` form a **critical section**.

A **critical section** is a **region of code** that must be accessed by **at most one control flow at a time**.

# Critical Sections

Critical sections often contain code that accesses shared state.

In most cases, any write to shared state is a critical section.<sup>1</sup>

Reads from shared state may not be critical sections, particularly if the state is immutable or changes infrequently.

It is important to define critical sections carefully and completely.

---

<sup>1</sup>There exist protocols that allow concurrent writes, however.



# Atomic Operations

Atomic operations are the simplest synchronization mechanism.

An atomic operation:

- Cannot be interrupted
- Appears as if no other operations run concurrently
- Always either fully succeeds or fails with no effects

Every atomic operation requires hardware support.

Not all machine instructions are atomic!

(In fact, often very few are.)

# Atomic Operations in C

C provides **no guaranteed atomic operations**.<sup>2</sup>

Atomic operations for synchronization from C require one of:

- Inline assembly code
- Library functions
- Knowledge of the compiler implementation
- Kernel assistance

---

<sup>2</sup>There is `sig_atomic_t`, which is atomic *with respect to signals*.

# Mutual Exclusion

Mutual exclusion is a tool for ensuring that **only one logical control flow** accesses some resource.

It is one of the **most basic** synchronization methods.

Mutual exclusion maps almost directly to critical sections:

- The code of the critical section is the resource

# The Mutex

A software tool for providing mutual exclusion is the **mutex**.

It provides two operations:

- Lock
- Unlock

<i>Operation</i>	<i>Mutex State</i>	<i>Action</i>
Lock	Unlocked	Lock mutex immediately <sup>3</sup>
Lock	Locked	Block until unlocked, then lock
Unlock	Locked	Unlock mutex immediately
Unlock	Unlocked	Implementation dependent

<sup>3</sup>If a flow locks a mutex that the *same flow* has already locked, behavior is implementation-dependent.

# Synchronization with Mutexes

Mutexes can be used to provide **synchronization**.

They can:

- Ensure that two actions do not happen **simultaneously**
- Ensure that one action **follows another**

They can be **used to create** mechanisms to:

- Directly **order events**
- Ensure that two events **begin** at the same time
- ...

Many **other synchronization “primitives”** use mutexes.

# Using Mutexes around Critical Sections

The typical use of a mutex is to **protect a critical section**.

Every concurrent flow will:

1. Lock a mutex
2. Execute the critical section
3. Unlock the mutex

Since **only one flow** can lock the mutex at a time, this **ensures mutual exclusion in the critical section**.

# Semaphores

Semaphores are a generalization of the mutex.

A semaphore is associated with a number.

There are two operations on a semaphore, variously named:

- P and V
- down and up
- wait and post
- ...

We will use P and V (after the original Dijkstra paper).

# Semaphore Operations

P (for *proberen* in Dutch, or “to test”)

V (for *verhogen*, “to increment”)

A semaphore  $s$  is initialized with a nonnegative integer.

P( $s$ ) attempts to decrement the integer:

- If it can be decremented and remain nonnegative (*i.e.*, it is  $\geq 1$ ), P returns immediately
- If decrementing it would make it negative (*i.e.*, it is 0), P blocks until it is  $> 0$

V( $s$ ) increments the integer:

- If the incremented value is 1, it releases one flow blocked on P, if such a flow exists



# Semaphores as Mutexes

A semaphore **initialized with the value 1** behaves like a mutex.

- $P(s)$  succeeds immediately for the first logical control flow to attempt it
- Any further flows **block on  $P(s)$**  because  $s$  is now zero
- $V(s)$  **releases one flow** blocked on  $s$  because  $s$  is now one

<i>Semaphore value</i>	<i>Equivalent mutex state</i>
1	Unlocked
0	Locked

# Lecture Question

Ask a lecture question!

# Condition Variables

Condition variables allow a logical control flow to block until some condition is met.

They work with mutexes to provide efficient blocking.

The holder of a mutex that is locked can block for a certain condition by:

- Waiting on a condition variable in a loop
- Testing the condition when awakened
- Breaking the loop if the condition is met

A concurrent flow that may have satisfied the condition can:

- Broadcast to all waiting flows
- Signal one waiting flow

# Mutex Interactions

The waiting flow **must hold a mutex** to wait.

The mutex **must protect data** used in the condition check.

Upon waiting, **the mutex will be unlocked**.

Upon awaking, **the waiting flow will re-lock the mutex**.

This means that the waiting flow **cannot assume the protected data remained unchanged** while it waited.

This complicated mutex interaction **allows the signaling flow to modify the protected data**.

# Wake and Check

The **wake and check** procedure allows:

- A thread to safely signal the condition even if it is **not sure** it has been met
- A condition to be signaled **in the presence of newcomers** that may falsify it before the waiting thread is scheduled
- Threads to be **spuriously woken** for other reasons (e.g., asynchronous notifications)

# Example Condition Control Flow

Mutex m

ConditionVariable cv

Data d

```
waiter() {  
    lock m  
    while condition on d {  
        wait on cv  
    }  
    take action  
    unlock m  
}
```

```
signaler() {  
    lock m  
    modify d  
    signal cv  
    unlock m  
}
```

# Deadlock

**Deadlock** is a condition in concurrent programming where two or more concurrent flows are **waiting for each other** and thus can never make progress.

Consider:

flow A:

```
lock mutex m0  
lock mutex m1  
do something
```

flow B:

```
lock mutex m1  
lock mutex m0  
do something
```

If flow A is interrupted by flow B **after locking m0 and before locking m1**, deadlock occurs.

Neither flow can **proceed**, and neither can **release the other**.

# Necessary Conditions

For deadlock to occur, **all of the following** must be true [3]:

- At least one resource is **mutually exclusive**.
- Flows **hold locks** while waiting for other locks to become available
- Locks cannot be **preempted**: once a flow holds a lock, it holds it until it voluntarily releases it.
- A circular chain of flows exists, such that each flow holds some lock required by the next flow



# Avoiding Deadlock

Deadlock is **caused by synchronization**.

There are **various techniques** to avoid deadlock.

For deadlock caused by **mutual exclusion** on **multiple locks**, there is a simple solution:

- All mutexes in a system are **ordered** (perhaps artificially)
- All flows lock mutexes **in order**
- All flows unlock mutexes **in reverse order**

# Summary

- A **race** is a situation where program correctness depends on the **order of operations in concurrent flows**.
- **Data races** are races involving **modification of data**.
- **Synchronization** is the **deliberate ordering of events**.
- A **critical section** is a **region of code** that must be accessed by **at most one concurrent flow at a time**.
- Synchronization primitives:
  - **Atomic operations**
  - **Mutexes**
  - **Semaphores**
  - **Condition variables**
- **Deadlock** is a program error **caused by synchronization**.

# Next Time ...

- POSIX threads
- POSIX mutexes
- POSIX semaphores
- POSIX condition variables
- Basically POSIX

# References I

## Optional Readings

- [1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Chapter 26: 26.2–26.6. Arpaci-Dusseau Books. URL: <https://pages.cs.wisc.edu/~remzi/OSTEP/>.
- [2] Randal E. Bryant and David R. O'Hallaron. *Computer Science: A Programmer's Perspective*. Third Edition. Chapter 12: 12.4-12.7. Pearson, 2016.
- [3] E. G. Coffman Jr., M. J. Elphick, and A. Shoshani. "System Deadlocks". In: *Computing Surveys* 3.2 (June 1971).

# License

Copyright 2018–2025 Ethan Blanton, All Rights Reserved.

Copyright 2024 Eric Mikida, All Rights Reserved.

Copyright 2022–2025 Carl Alphonse, All Rights Reserved.

Copyright 2019 Karthik Dantu, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see <https://www.cse.buffalo.edu/~eblanton/>.