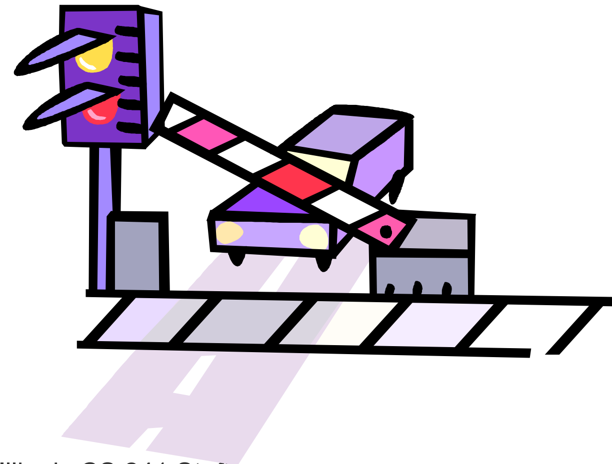
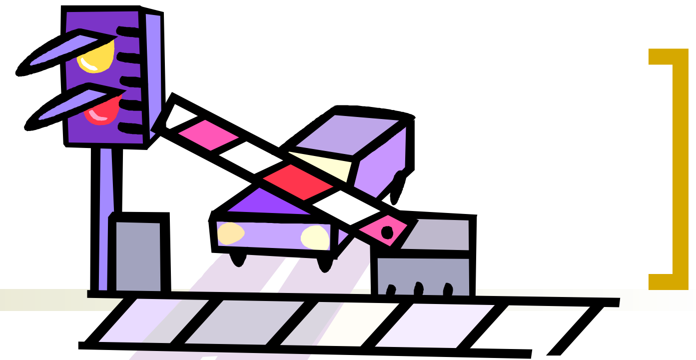
Introduction to Synchronization



Copyright ©: University of Illinois CS 241 Staff

Overview



- Introduction to synchronization
  - Why do we need synchronization?
  - Solution: Critical Regions
  - How to implement a Critical Region inconveniently

Copyright ©: University of Illinois CS 241 Staff

What could go horribly wrong?

Shared state:

```
queue_t q; /* to do list */
```

Producer thread:

```
while (true) {
  Create new work W;
  Find tail of q;
  tail = W;
}
```

Consumer thread:

```
while (true) {
  work = head of q;
  remove head from q;
  do_work(work);
}
```

Copyright ©: University of Illinois CS 241 Staff

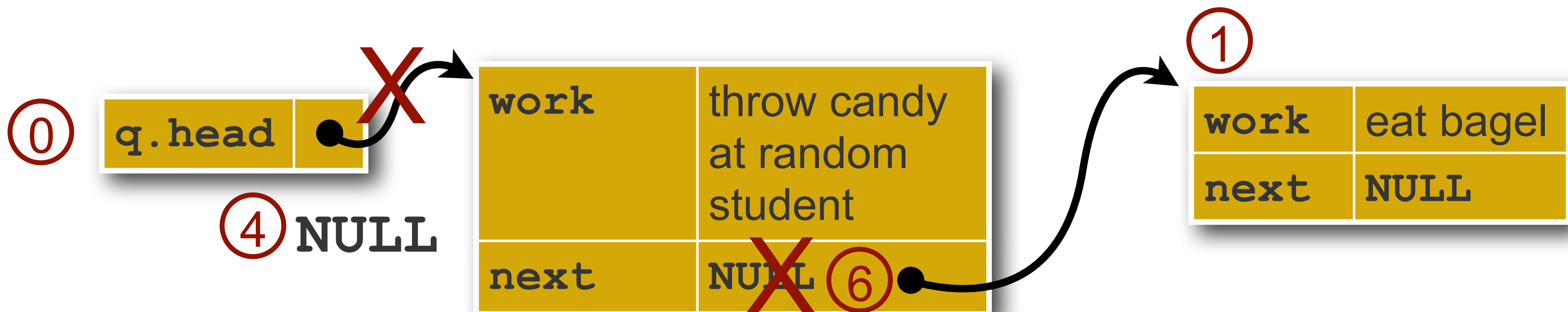
Things going Horribly Wrong

Producer thread:

```
while (true) {
① Create new work W;
② Find tail of q;
⑥ tail = W;
}
```

Consumer thread:

```
while (true) {
③ work = head of q;
④ remove head from q;
⑤ do_work(work);
}
```



I'll never get to eat my bagel. :-(
(Could something worse happen?)

Copyright ©: University of Illinois CS 241 Staff

A simpler example

- We just saw that processes / threads can be preempted at arbitrary times.
  - The previous example might work, or not.
- What if we just use simple operations?

Shared state:	Thread 1:	Thread 2:
`int x=0;`	`x++;`	`x++;`

Are we safe now?

How is `x++` implemented?

```
register1 = x
register1 = register1 + 1
x = register1
```

Copyright ©: University of Illinois CS 241 Staff

This could happen...

Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
`x  = r1`		1		1
	`r2 = x`		1	1
	`r2 = r2+1`		2	1
	`x  = r2`		2	2

Copyright ©: University of Illinois CS 241 Staff

But this could happen too!

Thread 1	Thread 2	r1	r2	x
`r1 = x`		0		0
`r1 = r1+1`		1		0
	`r2 = x`	1	0	0
	`r2 = r2+1`	1	1	0
`x = r1`		1	1	1
	`x = r2`	1	1	1

Copyright ©: University of Illinois CS 241 Staff

Introducing: Critical Region (Critical Section)

```
Process {
   ...

   Access shared variables;

   ...
}
```

Copyright ©: University of Illinois CS 241 Staff

Introducing: Critical Region (Critical Section)

```
Process {
   ...
   ENTER CRITICAL REGION
   Access shared variables;
   LEAVE CRITICAL REGION
   ...
}
```

Copyright ©: University of Illinois CS 241 Staff

Critical Region Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

Copyright ©: University of Illinois CS 241 Staff

Critical Region Requirements

- Mutual Exclusion:
  - No other process must execute within the critical section while a process is in it
- Progress:
  - If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely

Copyright ©: University of Illinois CS 241 Staff

Critical Region Requirements

- Bounded Wait:
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section

Must ensure these requirements without assumptions about number of CPUs, speeds of the threads, or scheduling!

Copyright ©: University of Illinois CS 241 Staff

Summary

- Synchronization is important for correct multi-threading programs
  - Race conditions
- Critical regions
- What's next: protecting critical regions
  - Software-only approaches
  - Semaphores
  - Other hardware solutions

Copyright ©: University of Illinois CS 241 Staff