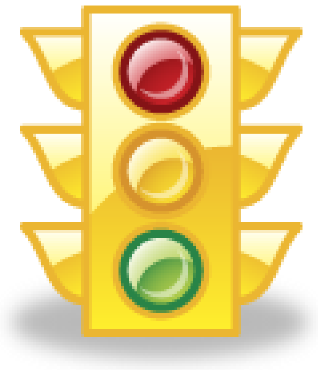
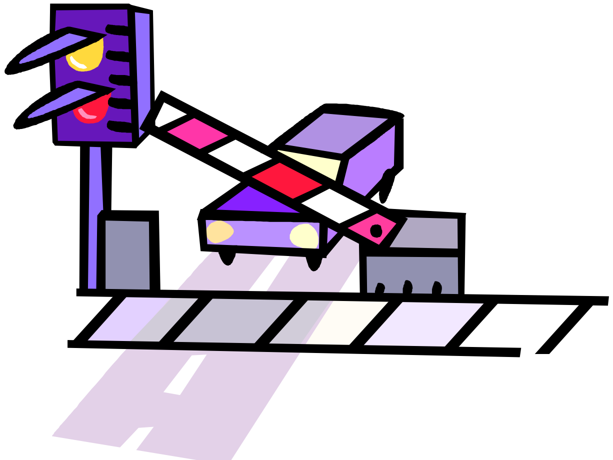
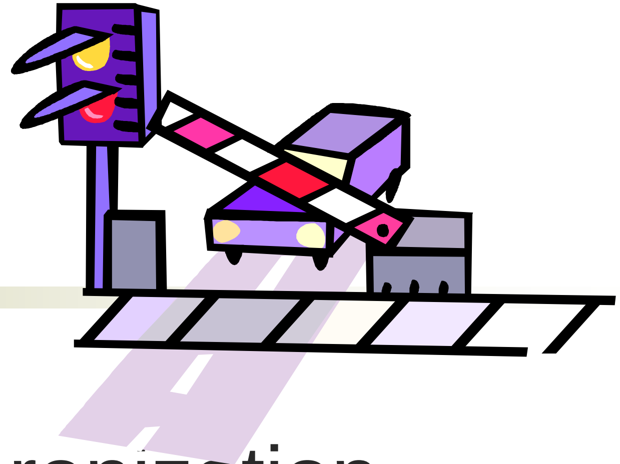
Introduction to Synchronization





Overview



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

Playing together is not easy

- Easy to share data among threads
- But, not always so easy to do it correctly...
- Easy case: one obvious “owner”
  - e.g., main() creates arguments, hands off to child thread
  - child now owns it, no one else will never read or write it
- What if threads need to work together? e.g., in web server
  - multiple threads concurrently access cache of files in memory, occasionally adding or removing
  - multiple threads concurrently update count of total # clients

Do threads conflict in practice?

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>

int cnt = 0;

void * worker( void *ptr ) {
   int i;
   for (i = 0;
        i < 50000; i++)
      cnt++;
}
```

```
#define NUM_THREADS 2
int main(void) {
   pthread_t threads[NUM_THREADS];
   int i, res;

   for (i = 0; i < NUM_THREADS; i++)  {
      res = pthread_create(&threads[i],
                NULL, worker, NULL);
   }
   for (i = 0; i < NUM_THREADS; i++) {
      res = pthread_join(threads[i], NULL);
   }
    /* Print result */
    printf("Final value: %d\n", cnt);
}
```

Do threads conflict in practice?

- If everything worked...

```
$ ./20-counter
Final value: 100000
```

- Q: What are the minimum and maximum final value?
- Q: What value do you expect in practice?

What's yours is mine …

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);
}
```

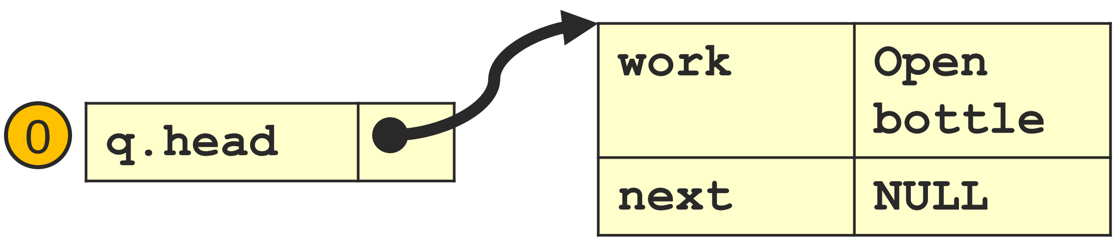
Can We Share?

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);
}
```



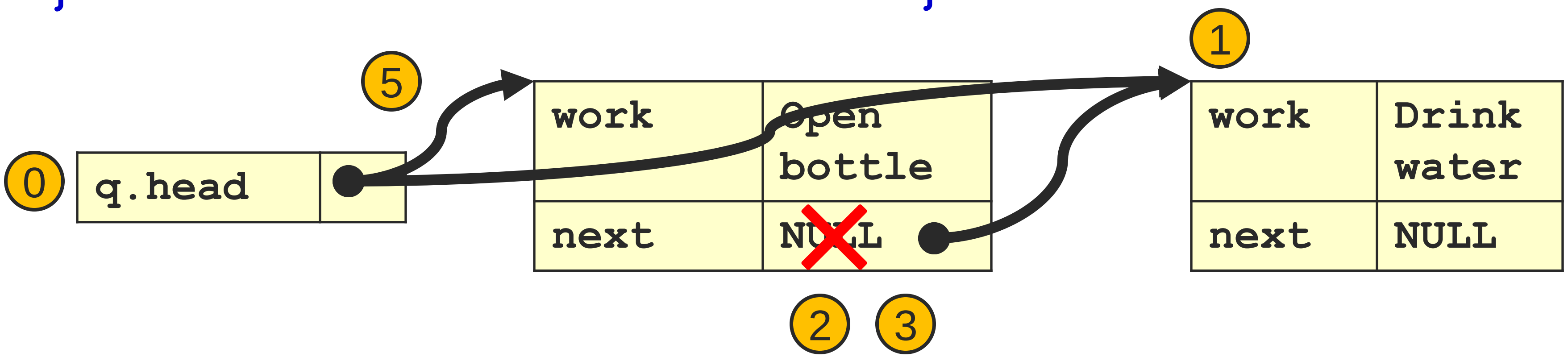
Can We Share?

Producer thread:

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

Consumer thread:

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



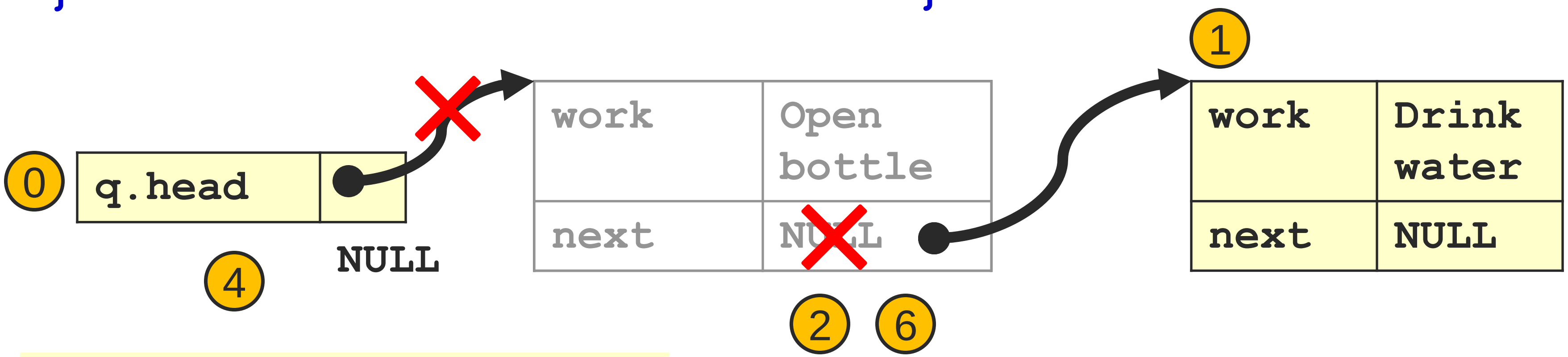
Something went horribly wrong …

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 drink my water!

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?

Incrementing Variables

- How is **`x++`** implemented?

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

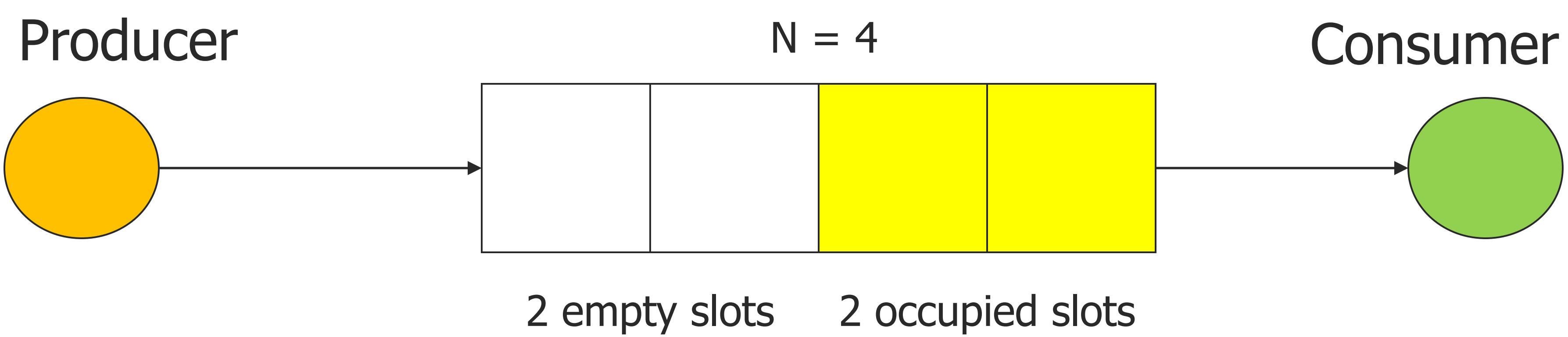
What could happen?

```
x++:    r1 = x
        r1 = r1 + 1
        x = r1
```

Thread 1: **`x++;`**	Thread 2: **`x++;`**	**r1**	**r2**	**x**

Producer/Consumer Problem

- Producer process "produces" information
- Consumer process "consumes" produced information
- Challenge: Bounded Buffer
  - Buffer has max capacity N
  - Producer can only add if buffer has room (i.e., count < N)
  - Consumer can only remove if buffer has item (i.e., count > 0)



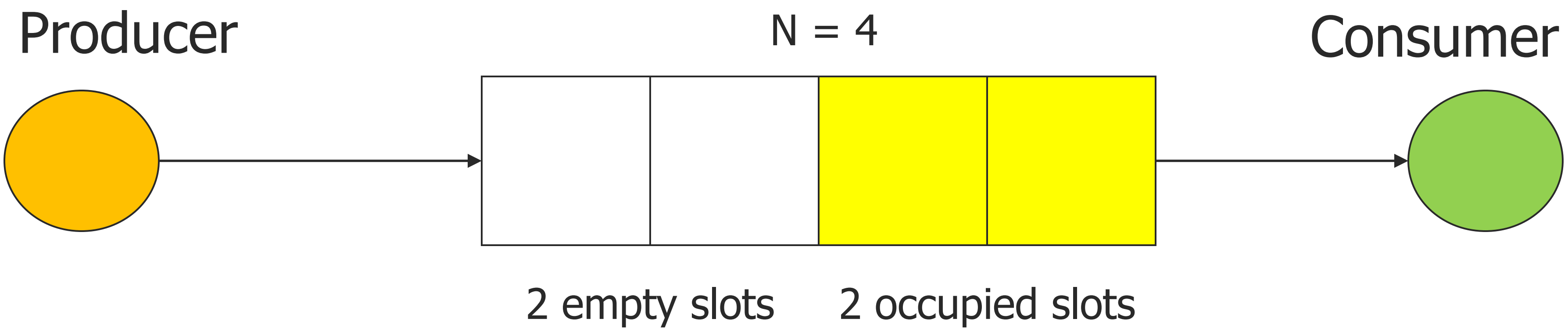
Producer/Consumer Problem

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);
}
```



Producer/Consumer Problem

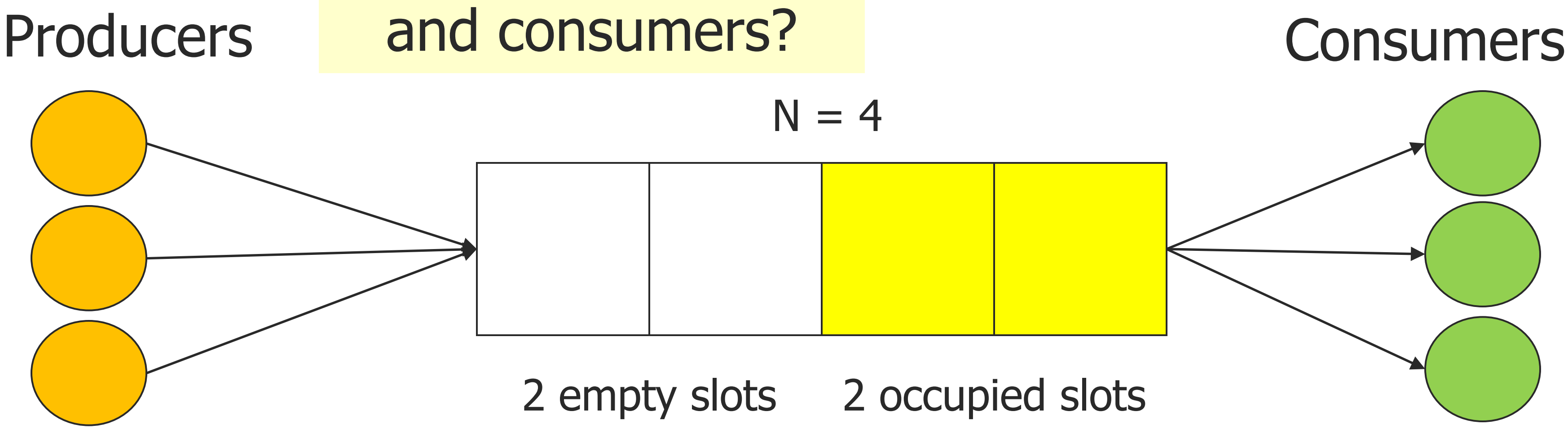
Producer threads:

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

Consumer threads:

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

What happens with multiple producers and consumers?



Multiple Producers: Shared Queue

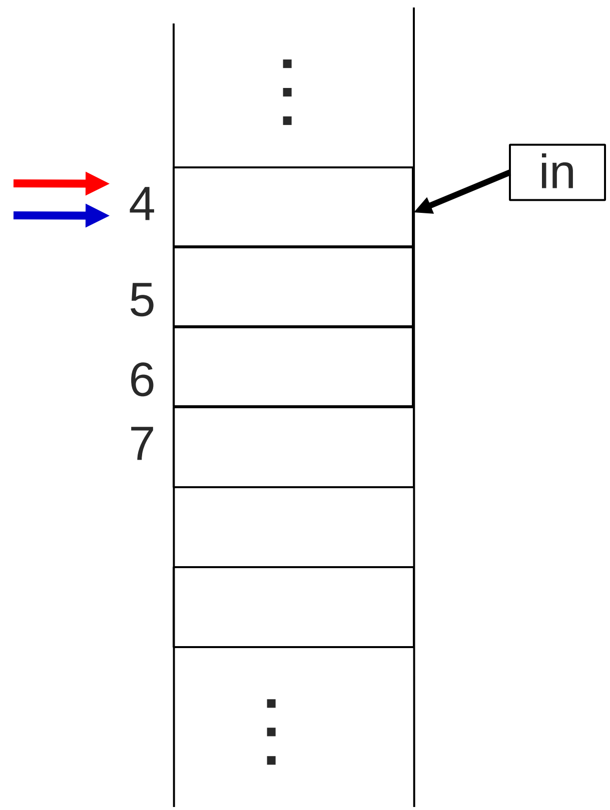
Process 1

```
int my_next_free;
---------------------
my_next_free = in;
```

Store **NEW** into `my_next_free`;

```
in=my_next_free+1
```

Shared memory



Process 2

```
int my_next_free;
---------------------
my_next_free = in
```

Store **NEW** into `my_next_free`;

```
in=my_next_free+1
```

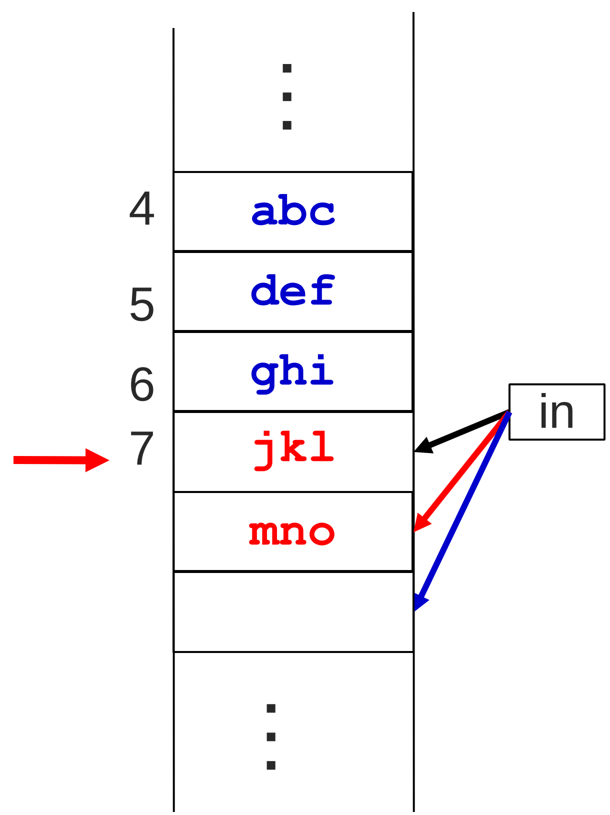
Multiple Producers: Shared Queue: Correct

Process 1

```
int my_next_free;
```

1. `my_next_free = in;`
2. Store `jkl` into `my_next_free`;
3. `in=my_next_free+1`

Shared memory



Process 2

```
int my_next_free;
```

4. `my_next_free = in`
5. Store `mno` into `my_next_free`;
6. `in=my_next_free+1`

Multiple Producers: Example: Problem

Process 1

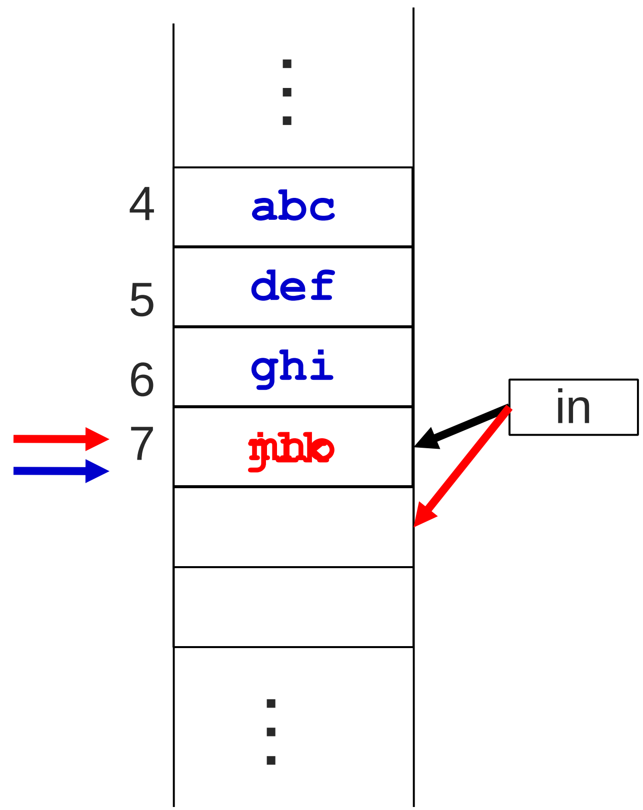
```
int my_next_free;
```

1 `my_next_free = in;`

3 Store `jkl` into `my_next_free`;

4 `in=my_next_free+1`

Shared memory



Process 2

```
int my_next_free;
```

2 `my_next_free = in`

5 Store `mno` into `my_next_free`;

6 `in=my_next_free+1`

Introducing: Critical Region (Critical Section)

```
Process {
  while (true) {

    Access shared variables;

    Do other work
  }
}
```

Introducing: Critical Region (Critical Section)

```
Process {
  while (true) {
    ENTER CRITICAL REGION
    Access shared variables;
    LEAVE CRITICAL REGION
    Do other work
  }
}
```

Critical Region Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

Mutual Exclusion

Critical Region Requirements

- Mutual Exclusion
  - At most one process in critical region
  - No other process may execute within the critical region while a process is in it
  - Safety
- Progress
- Bounded Wait

Mutual Exclusion Progress

Critical Region Requirements

- Mutual Exclusion
- Progress
  - If no process is waiting in its critical region and several processes are trying to get into their critical section, then one of the waiting processes should be able to enter the critical region
  - Liveness – no deadlocks
- Bounded Wait

Mutual Exclusion

Progress

Bounded Wait

Critical Region Requirements

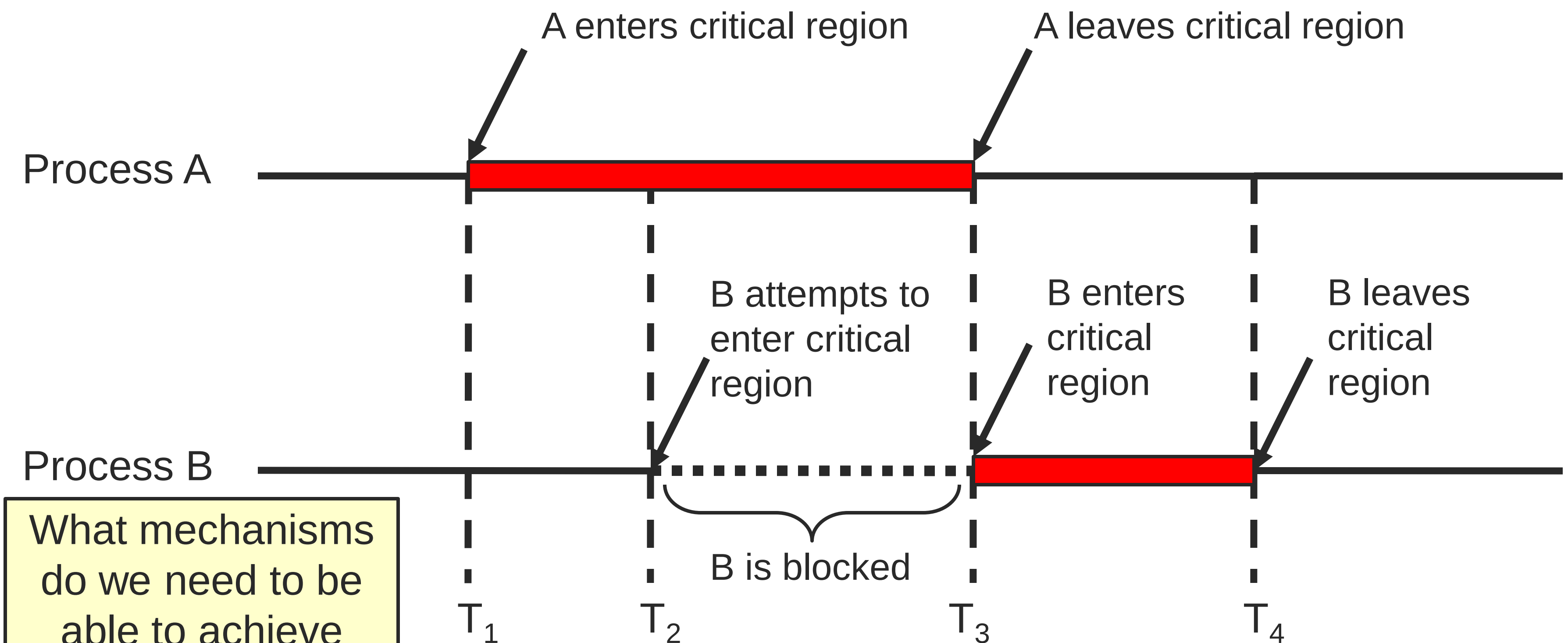
- Mutual Exclusion
- Progress
- Bounded Wait
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical region
  - Liveness – no starvation

Critical Region Requirements

- Mutual Exclusion
- Progress
- Bounded Wait

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

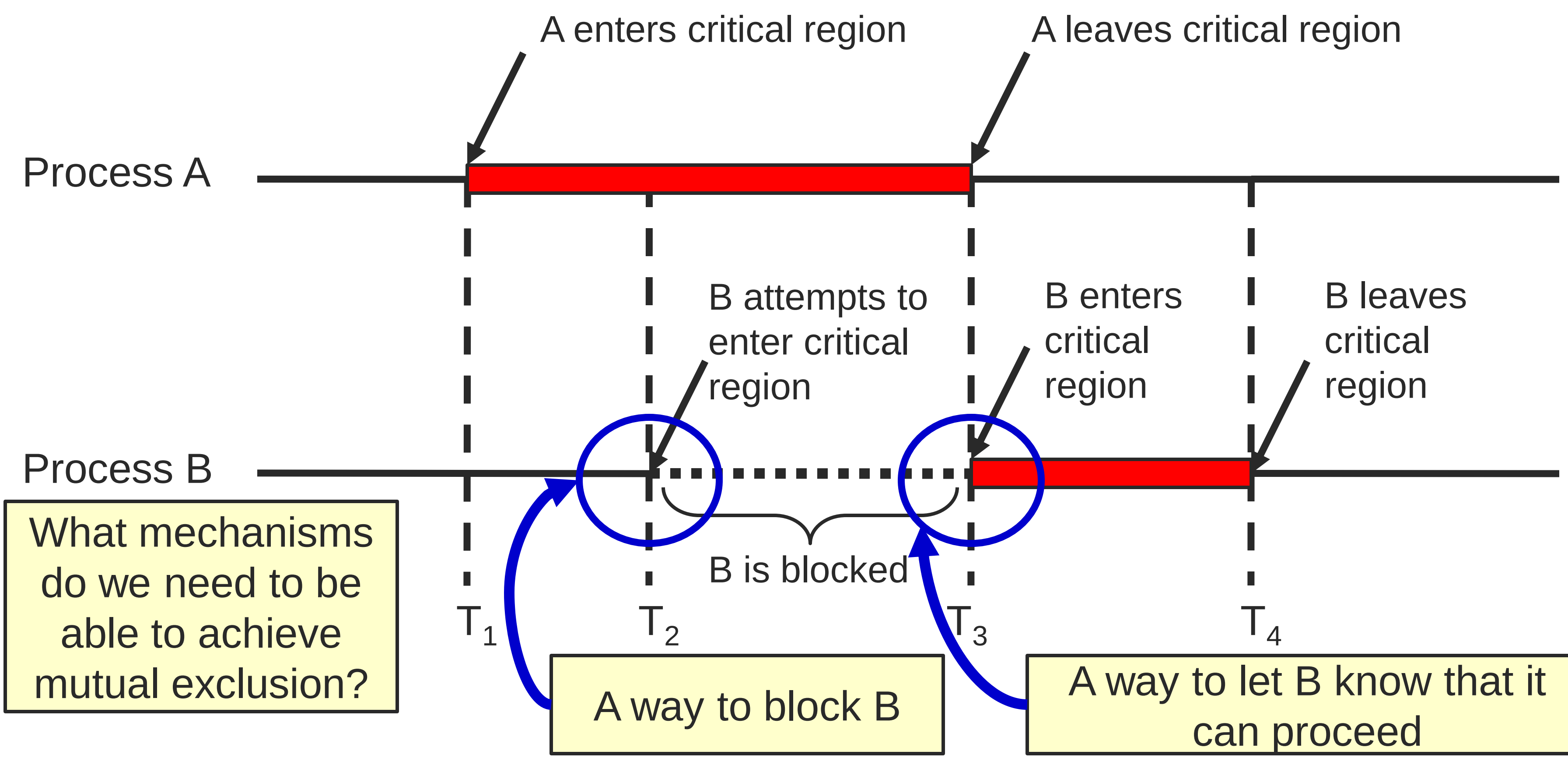
Critical Regions



What mechanisms do we need to be able to achieve mutual exclusion?

Mutual exclusion using critical regions

Critical Regions



Mutual exclusion using critical regions

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