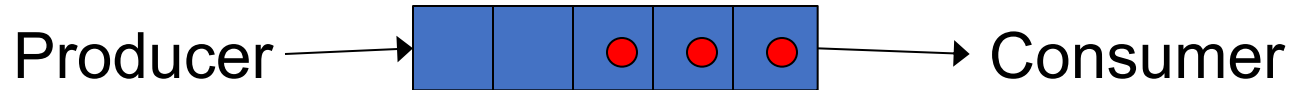


Operating Systems Principles

CPU Management

Process Synchronization

Motivating Example: Producer and Consumer



```
while (true) {  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE); // do nothing  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

```
while (true) {  
    while (count == 0); // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed  
}
```

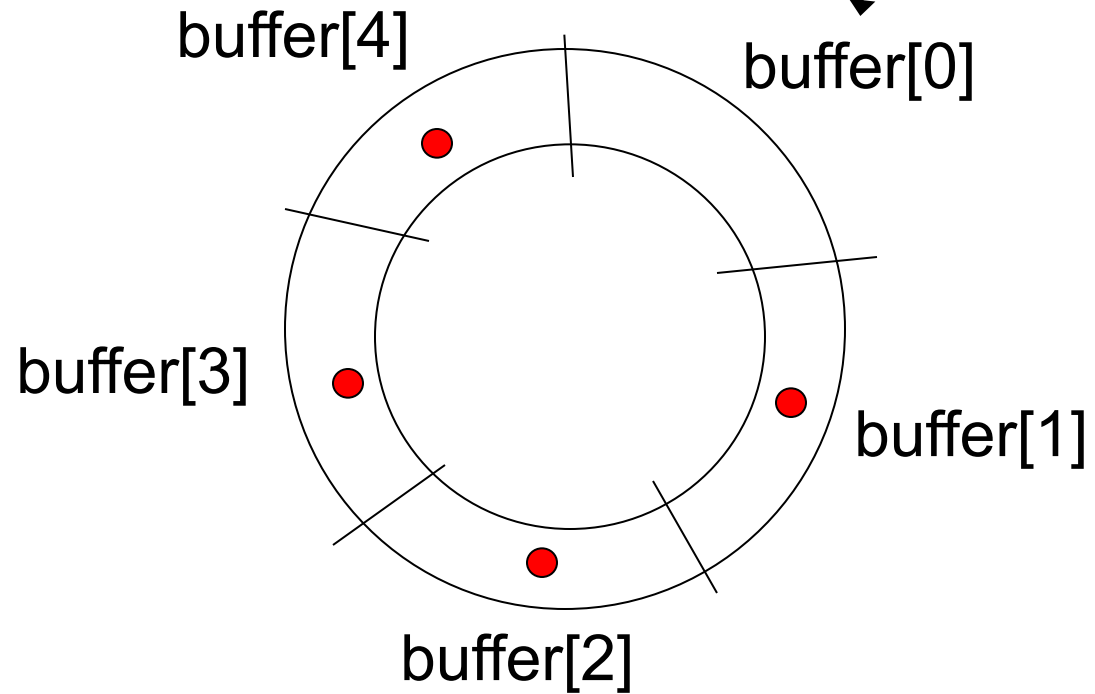
- Shared variables: buffer and count
- Local variables: in, out, nextProduced, nextConsumed

*Non-shared
variables*

out (consumer)

in (producer)

*Shared
variables*



count = 4

A Problem!

`count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

`count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

Consider this execution interleaving with “count = 4” initially:

- Producer executes `register1 = count` {register1 = 4}
- Producer executes `register1 = register1 + 1` {register1 = 5}
- Context Switch -----
- Consumer executes `register2 = count` {register2 = 4}
- Consumer executes `register2 = register2 - 1` {register2 = 3}
- Consumer executes `count = register2` {count = 3}
- Context Switch -----
- Producer executes `count = register1` {count = 5}

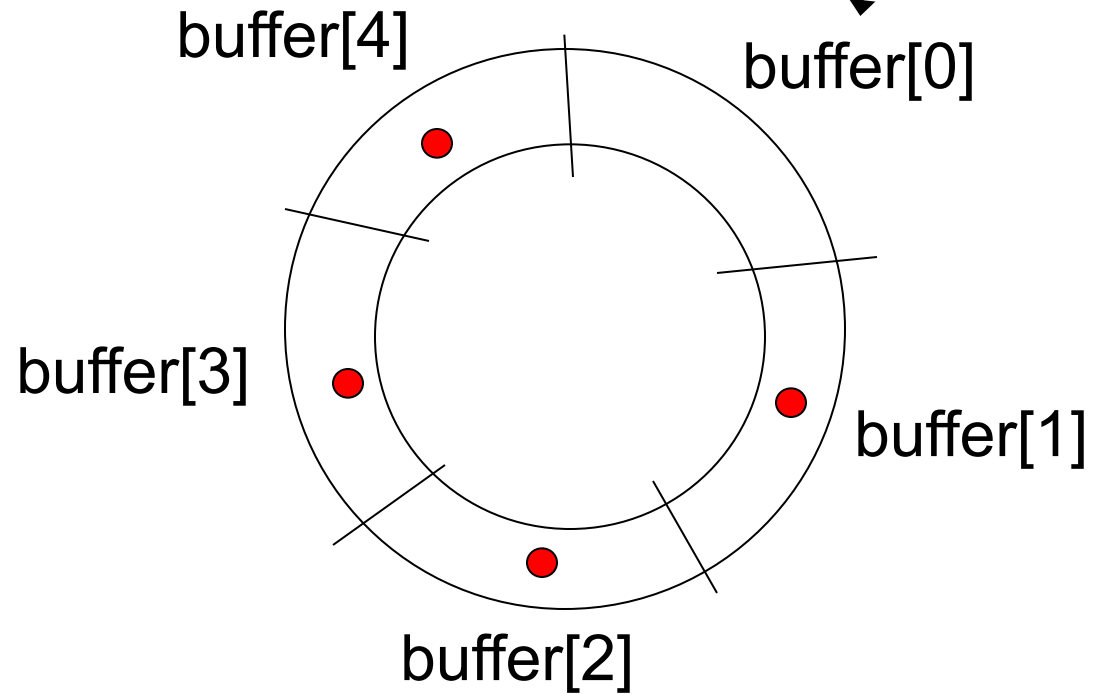
Now producer thinks buffer is full when actually there is one empty slot!

*Non-shared
variables*

out (consumer)

in (producer)

*Shared
variables*

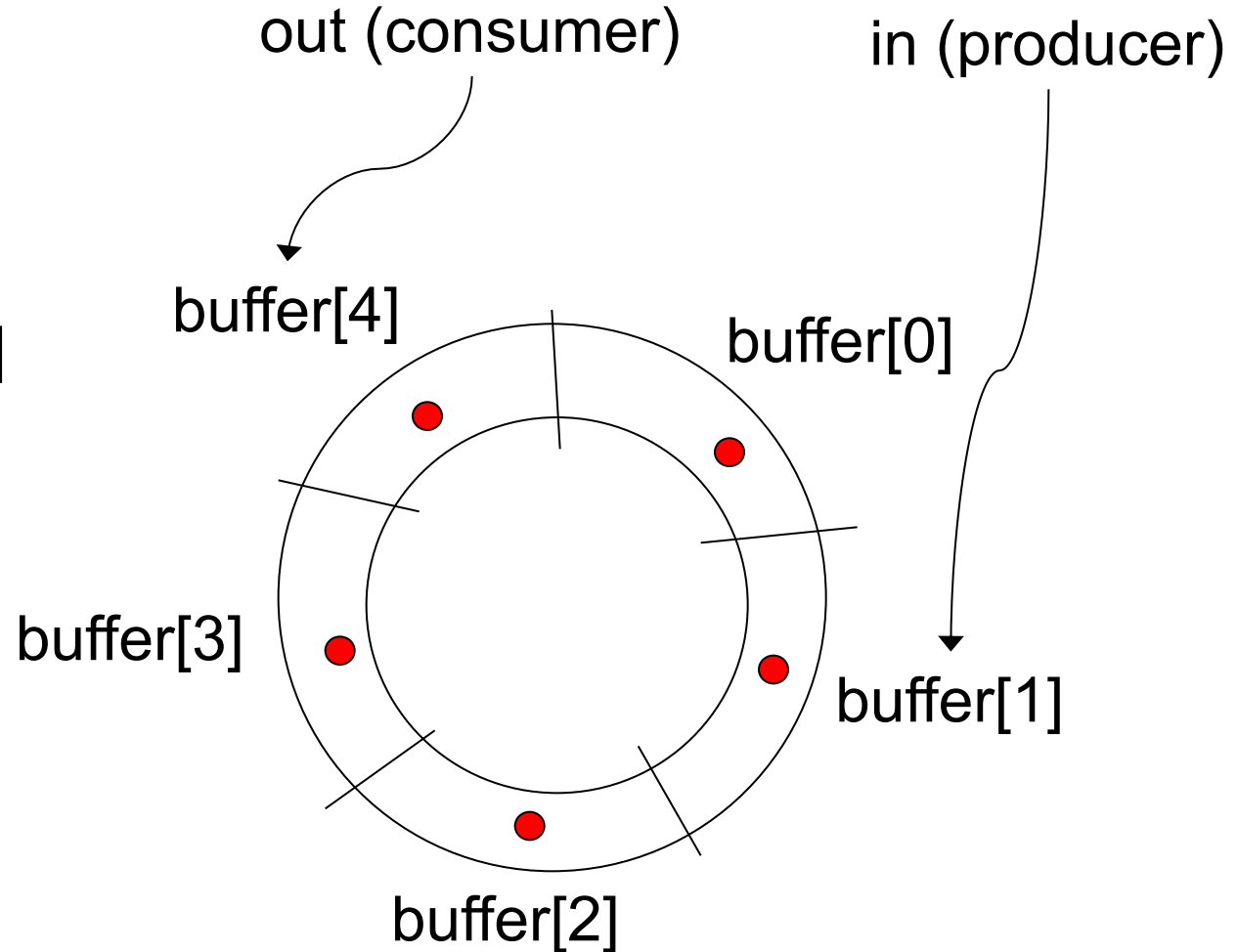


count = 4

Producer adds a
new item at buffer[0]

Makes in \rightarrow buffer[1]

count still 4 since it
is context switched
out



```
register1 = count {register1 = 4}  
register1 = register1 + 1 {register1 = 5}
```

out (consumer)

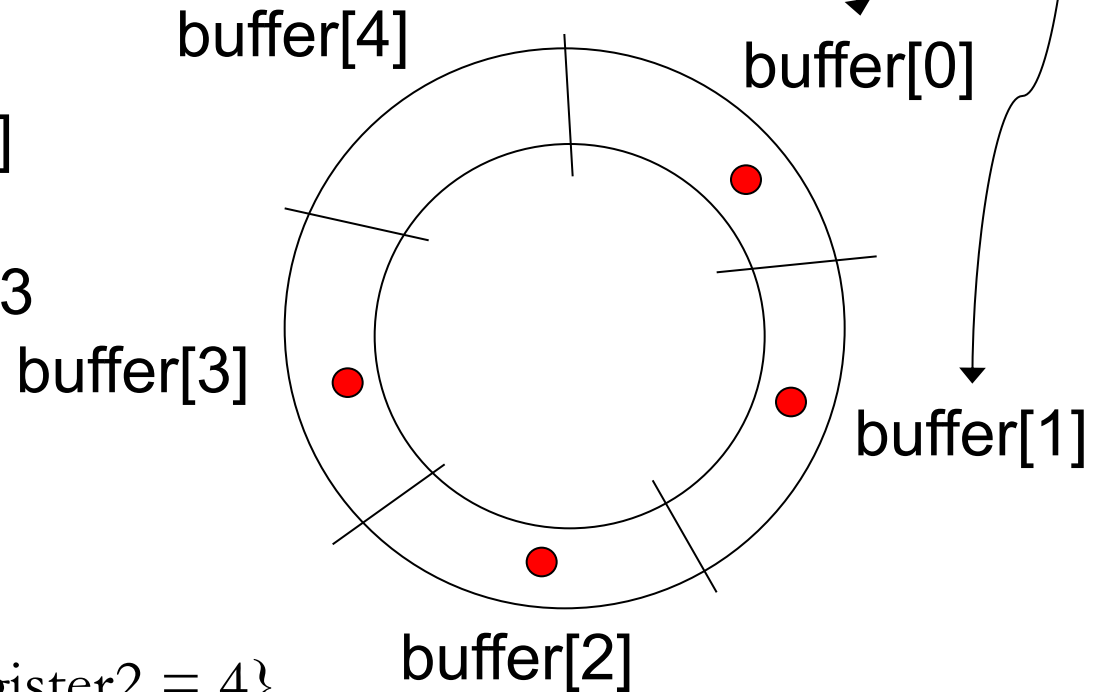
in (producer)

Consumer removes
item at buffer[4]

Makes out \rightarrow buffer[0]

Decrements count to 3

Context switched out

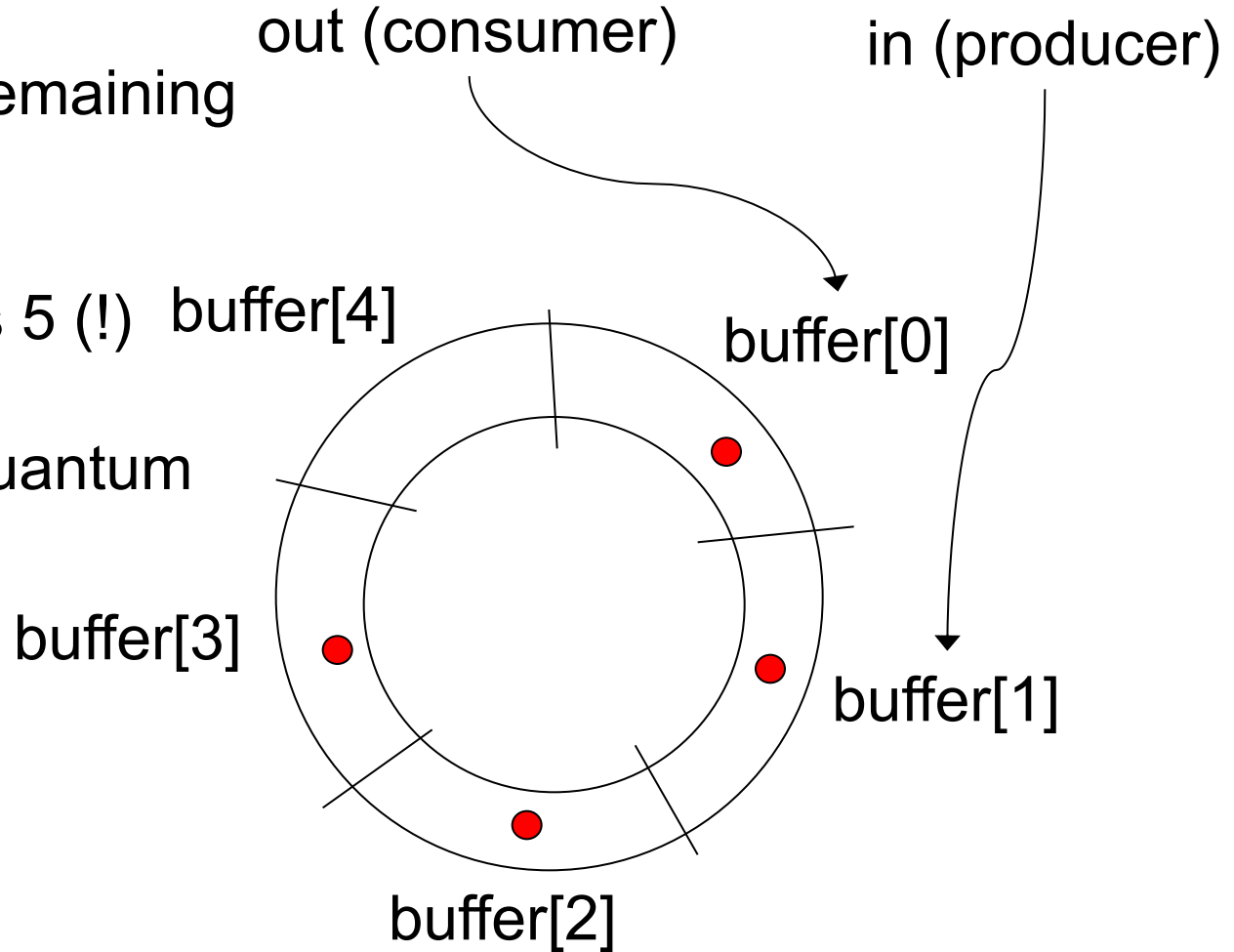


```
register2 = count {register2 = 4}  
register2 = register2 - 1 {register2 = 3}  
count = register2 {count = 3}
```

Producer resumes remaining
portion of count++

Count now becomes 5 (!) buffer[4]

Wastes remaining quantum
thinking buffer is full!



count = register1 {count = 5}

Race Condition

- Arose because shared variables were manipulated using **non-atomic** operations
 - Consisting of instructions that could interleave in arbitrary ways depending on the scheduling
- Solution: Make these operations **atomic**
- What if the underlying CPU doesn't provide atomic versions of these operations?
 - **Emulate atomicity**

Example: Producer and Consumer

Producer

```
while (true) {  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE); // do nothing  
    buffer[in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

Consumer

```
while (true) {  
    while (count == 0); // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

We would like these operations to behave as if they were atomic

- Shared variables: buffer and count
- Local variables: in, out, nextProduced, nextConsumed

How to emulate atomicity?

- **Key idea:** If we ensure the instructions comprising these operations do not interleave, it would be as if they are atomic
 - Convince yourself of this
- That is, make the execution of these operations **mutually exclusive**

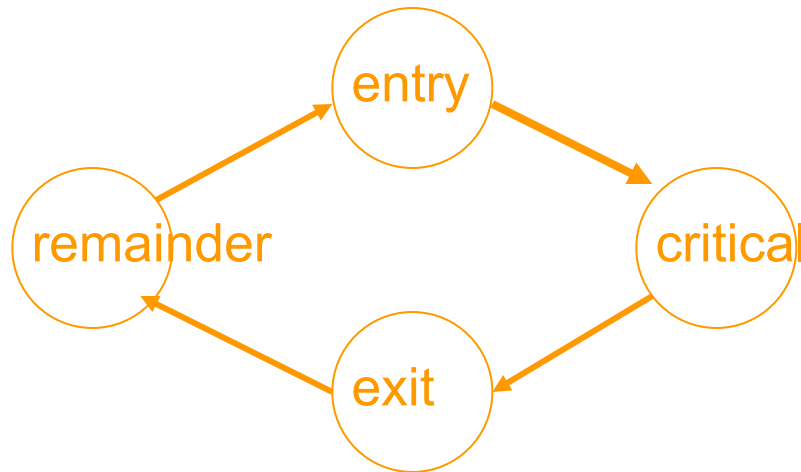
Mutual Exclusion

- Only one thread should be able to access a shared resource at a time
- Code that uses mutual exclusion to synchronize its execution is called a **critical section**
 - Only one thread at a time can execute its critical section
 - All other threads are forced to wait on entry
 - When one thread leaves the critical section, another can enter

```
do {  
    entry section  
        CRITICAL SECTION  
    exit section  
        REMAINDER SECTION  
} while (TRUE);
```

Mutual Exclusion/Critical Section Problem (Mutex)

- Each process's code is divided into four sections:

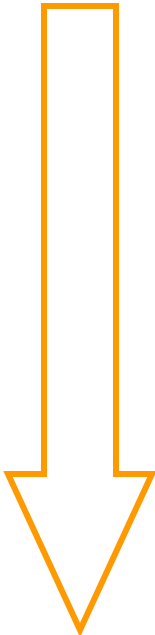


- **entry:** synchronize with others to ensure mutually exclusive access to the ...
- **critical:** use some resource; when done, enter the...
- **exit:** clean up; when done, enter the...
- **remainder:** not interested in using the resource

Mutual Exclusion Algorithms

- A *mutual exclusion algorithm* specifies code for entry and exit sections to ensure:
 - **mutual exclusion:** at most one process is in its critical section at any time, and
 - some kind of "liveness" condition. There are three commonly considered ones...
- Mutex = Mutual Exclusion + Liveness

Mutex Liveness Conditions

- 
- **1. no deadlock (not considered in your text):** if a process is in its entry section at some time, then later some process is in its critical section
 - **2. progress/no lockout/no starvation:** if a process is in its entry section at some time, then later the *same* process is in its critical section
 - **3. bounded waiting:** no lockout + while a process is in its entry section, other processes enter the critical section no more than a certain number of times.
 - These conditions are increasingly strong (each implies the previous ones)
 - Exercise: Prove this
 - **Note: Slightly different terminology than your text**

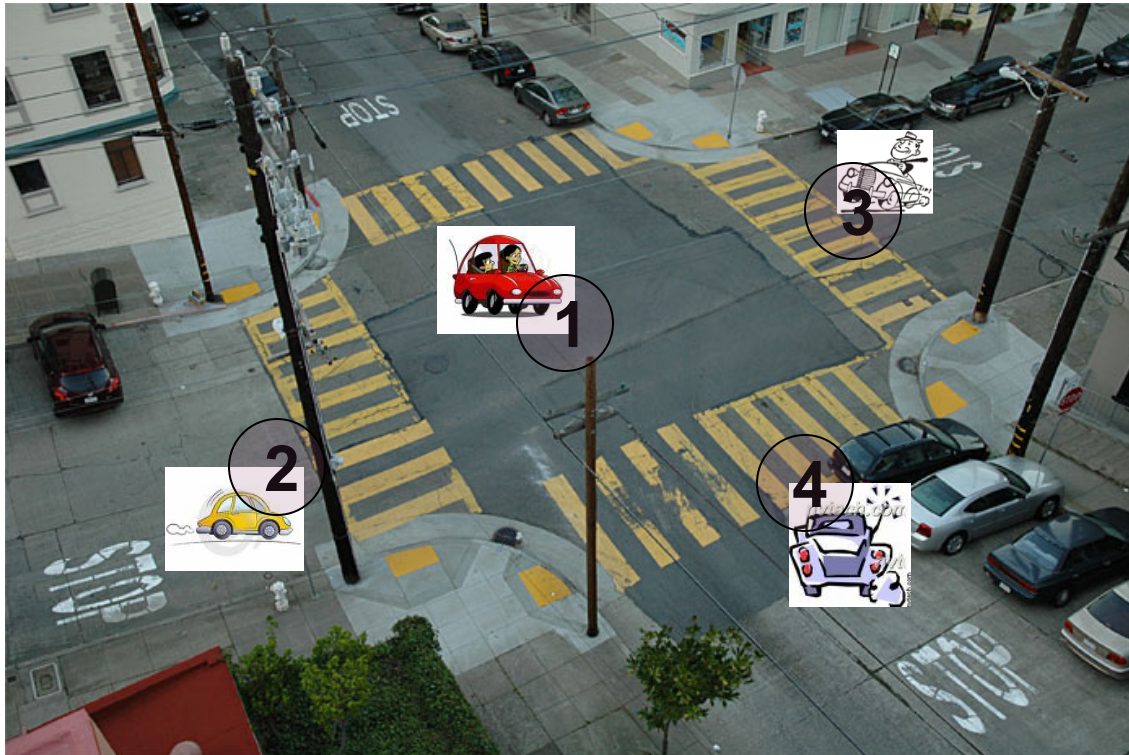
Mutual Exclusion: Traffic Analogy

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Mutual Exclusion: Traffic Analogy

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Only one car can be in the square area between the stop signs at a given time

2. Progress: Traffic Analogy

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



2. Progress: Traffic Analogy

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



If the square is empty and there is some car, it will get to cross (no starvation)

3. Bounded Waiting: Traffic Analogy

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



Any car has to wait at most for 3 others to pass after it stops (fairness)

Mutex Algorithms: Entry/Exit Sections

- The code for the entry and exit sections is allowed to assume that
 - no process stays in its critical section forever
 - shared variables used in the entry and exit sections are not accessed during the critical and remainder sections

Complexity Measure for Mutex

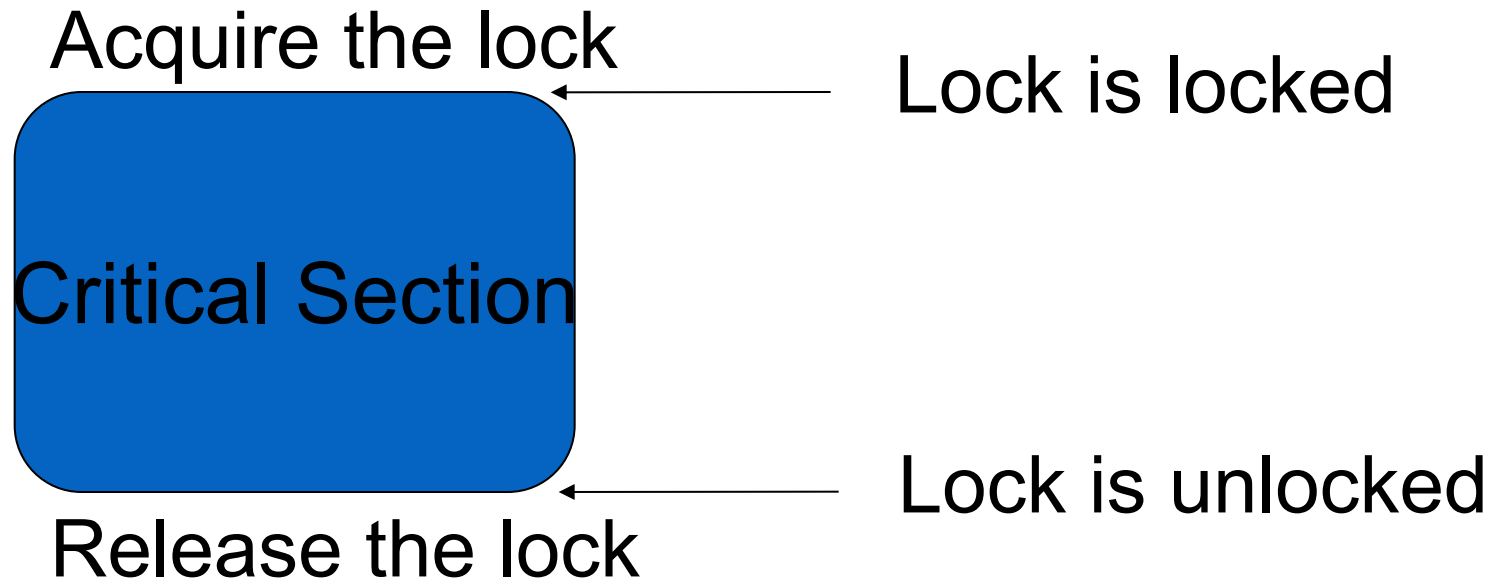
- Main complexity measure of interest for shared memory mutex algorithms is amount of shared space needed
 - By the entry and exit sections
- Space complexity is affected by:
 - how powerful is the type of the shared variables
 - how strong is the progress property to be satisfied (no deadlock vs. no lockout vs. bounded waiting)

Mutex solutions

- We will study the following in this course
 - Mutex lock
 - Condition variable
 - Semaphore

Mutex lock

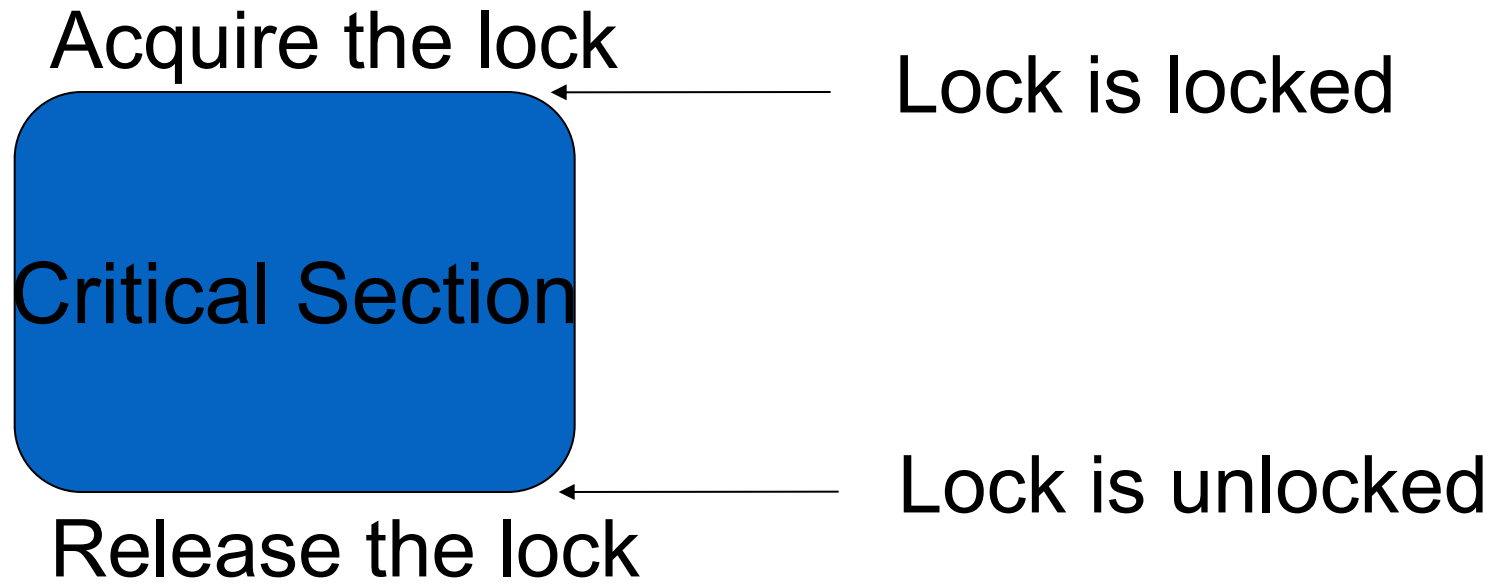
- Usage:



- E.g., pthreads functions:
 - `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;` or
`pthread_mutex_init (&mymutex, attr);`
 - `pthread_mutex_lock (&mutex)`
 - `pthread_mutex_unlock (&mutex)`

Mutex lock

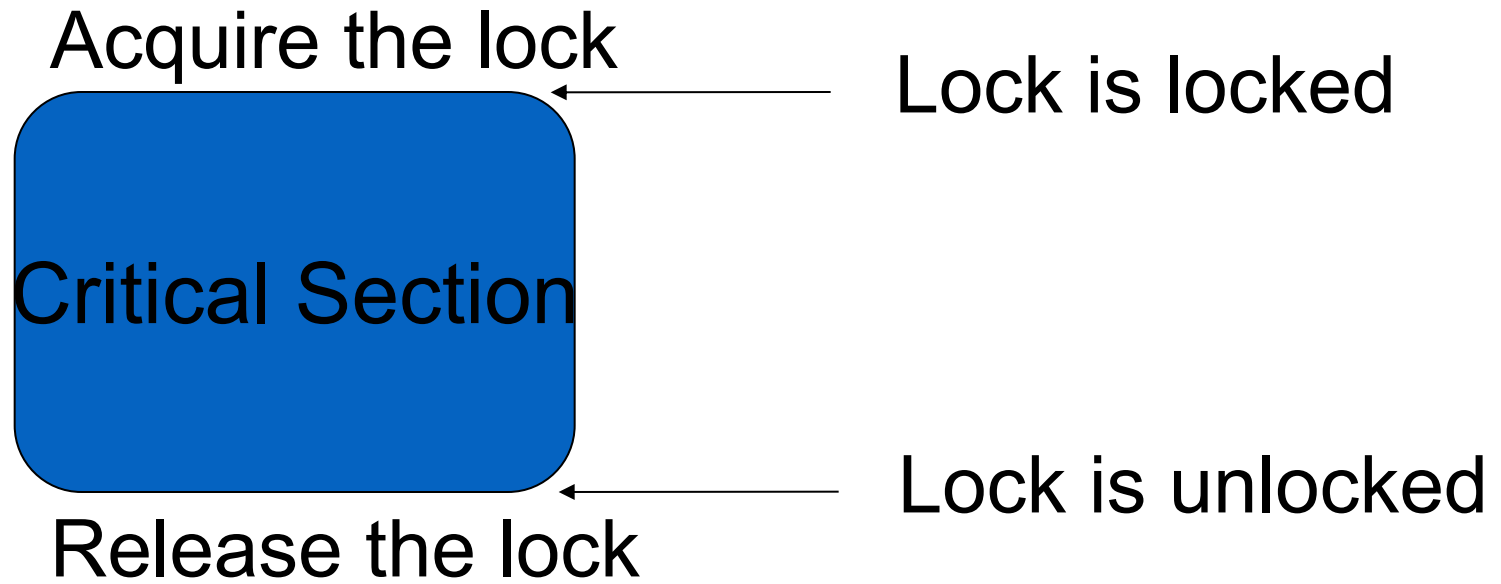
- Usage:



- Care to take when using locks (common mistakes to avoid)
 - Do not attempt to release a lock that you haven't acquired
 - Do not attempt to acquire an already acquired lock
 - Both can lead to race conditions
 - A good implementation of locks would simply not allow these

Mutex lock

- Usage:



- The “trivial” way of implementing such a lock doesn’t work
 - Used a boolean variable
- How to implement a mutex lock?
 - We started with Peterson’s algorithm for two processes

Example to Show Mut. Excl. is Non-trivial

- How about constructing a “lock” s.t. before getting into critical section, we acquire it and after leaving critical section, we release it?
- Is there any trouble if we use a boolean flag as a lock? Consider P1 and P2

```
while (lock == 1);    // Do nothing, just wait
                      //position 1

lock = 1;

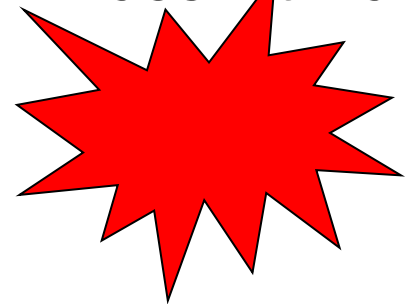
.....

.....                // Critical Section

.....

lock = 0;
```

Doesn't work!



- If P1 interrupted at position 1, race condition might occur
- P1 was waiting until lock == 0. Before it announces its turn (i.e., sets lock = 1), suppose P2 gets scheduled and sees that lock is 0
- Before P2 leaves its critical section (i.e., sets lock = 0), suppose it gets descheduled
- Since P1 was at position 1, it will set lock = 1 and also get into its critical section
- Now, two processes are in the critical section at the same time!

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are **atomic**; that is, cannot be interrupted.
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process **P_i** is ready

Algorithm for Process P_i

```
while (true) {  
    flag[i] = TRUE;  
    turn = j;  
    while ( flag[j] && turn == j);
```

CRITICAL SECTION

```
    flag[i] = FALSE;
```

REMAINDER SECTION

```
}
```

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are **atomic**; that is, cannot be interrupted.
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = true** implies that process P_i is ready

```
while (true) {  
    flag[0] = TRUE;  
    turn = 1;  
    while(flag[1] && turn == 1);
```

CRITICAL SECTION

```
flag[0] = FALSE;
```

REMAINDER

```
}
```

```
while (true) {  
    flag[1] = TRUE;  
    turn = 0;  
    while(flag[0] && turn == 0);
```

CRITICAL SECTION

```
flag[1] = FALSE;
```

REMAINDER

```
}
```