Operating Systems Concepts

Process Synchronization and Locks

CS 4375, Fall *2025*
Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu
October 29, 2025

Summary

- OS structure
  - Kernel space, user space
- Processes
  - What are processes? How do we get execute, separate, etc. between different processes?
- Scheduling
  - Different algorithms and analysis
- Memory
  - Virtual and physical memory

Agenda

- Process Synchronization
  - Multiprogramming: multiple processes or threads may require consensus to work together
    - How do we achieve that? What issues may occur for shared data?
  - Race Condition
  - Critical-Section Problem
    - Different Solutions

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the **orderly execution of cooperating processes**
- Consider consumer-producer problem:
  - Initially `count = 0`
  - It is incremented by the producer after it produces a new buffer.
  - It is decremented by the consumer after it consumes a buffer.

Consumer-Producer Problem

Shared Variables: `count=0, buffer[]`

Consumer

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

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++;
}
```

Race Condition

- `count++` could be implemented as:
  - `register1 = count`
  - `register1 = register1 + 1`
  - `count = register1`
- `count--` could be implemented as:
  - `register2 = count`
  - `register2 = register2 - 1`
  - `count = register2`
- What would happen if they both want to run?

Race Condition

Consider this execution interleaving with "`count = 5`" initially:

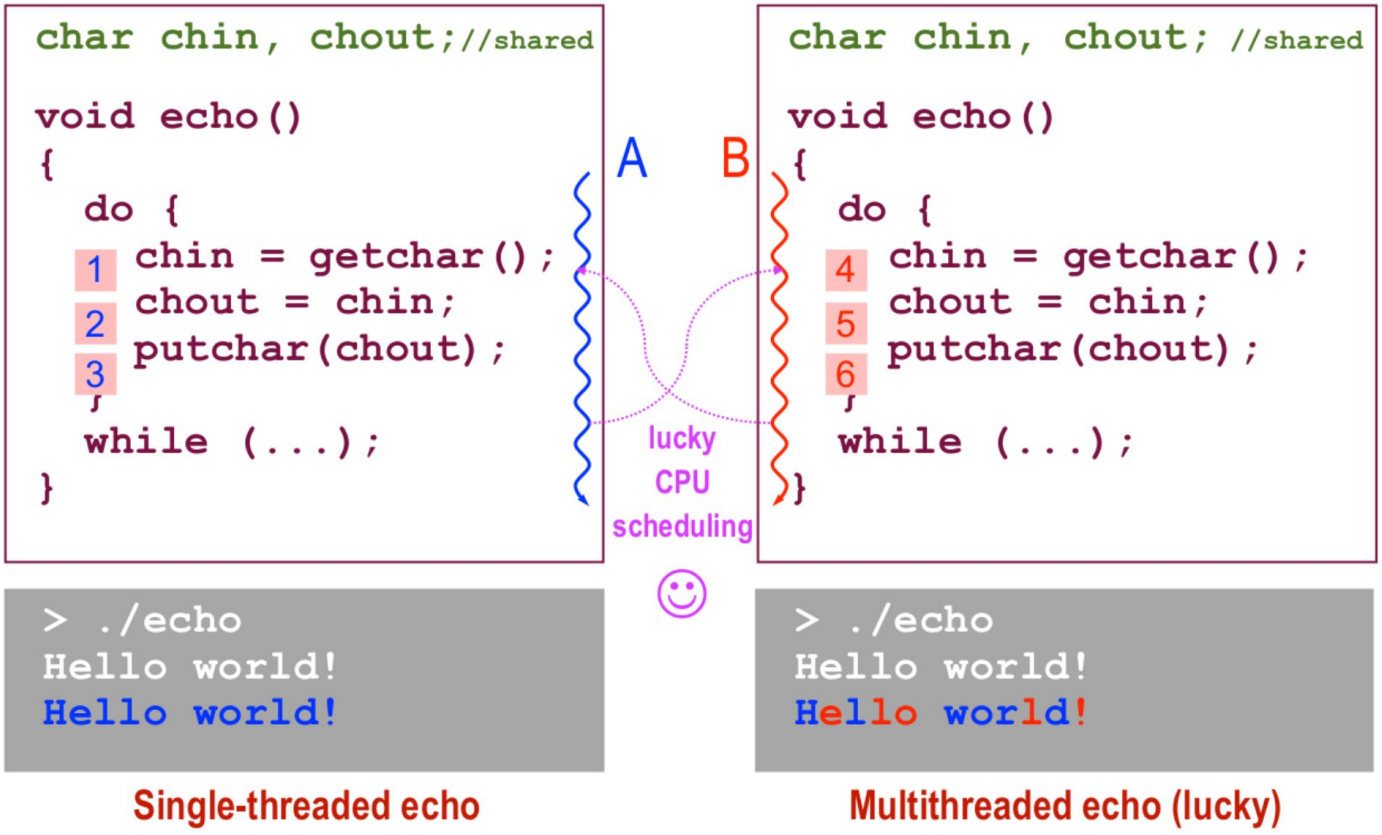
```
1.  producer executes:     register1 = count          {register1 = 5}
2.  producer executes:     register1 = register1 + 1  {register1 = 6}
3.  consumer executes:     register2 = count          {register2 = 5}
4.  consumer executes:     register2 = register2 - 1  {register2 = 4}
5.  producer executes:     count = register1          {count = 6}
6.  consumer executes:     count = register2          {count = 4}
```

Race Condition

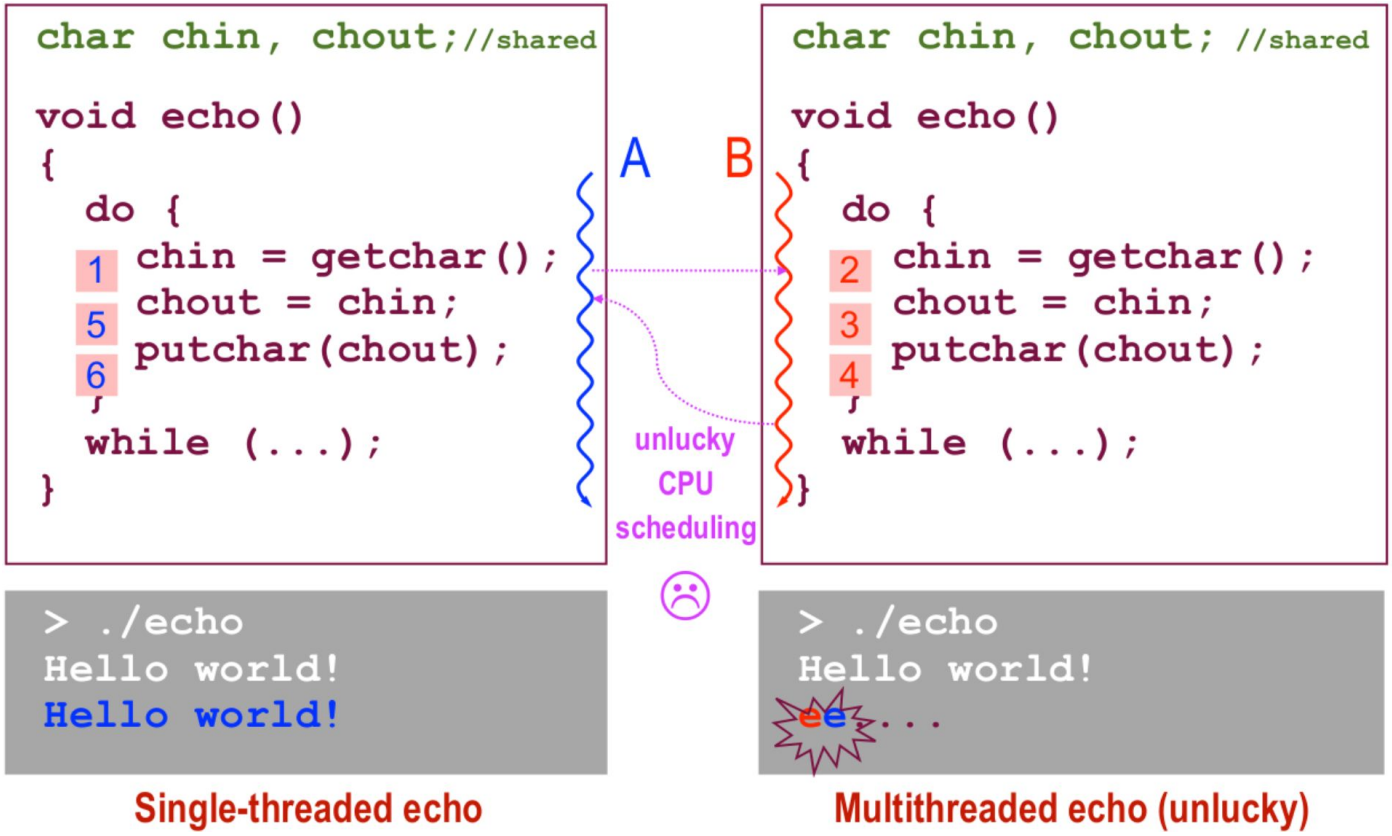
- Race condition: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends on the order of execution.
- To prevent race conditions, concurrent processes must be synchronized
  - Ensure that only one process at a time is manipulating the shared variable.
- The statements:
  - `count++;`
  - `count--;`
    - must be performed atomically

- Atomic operation means an operation without interruption

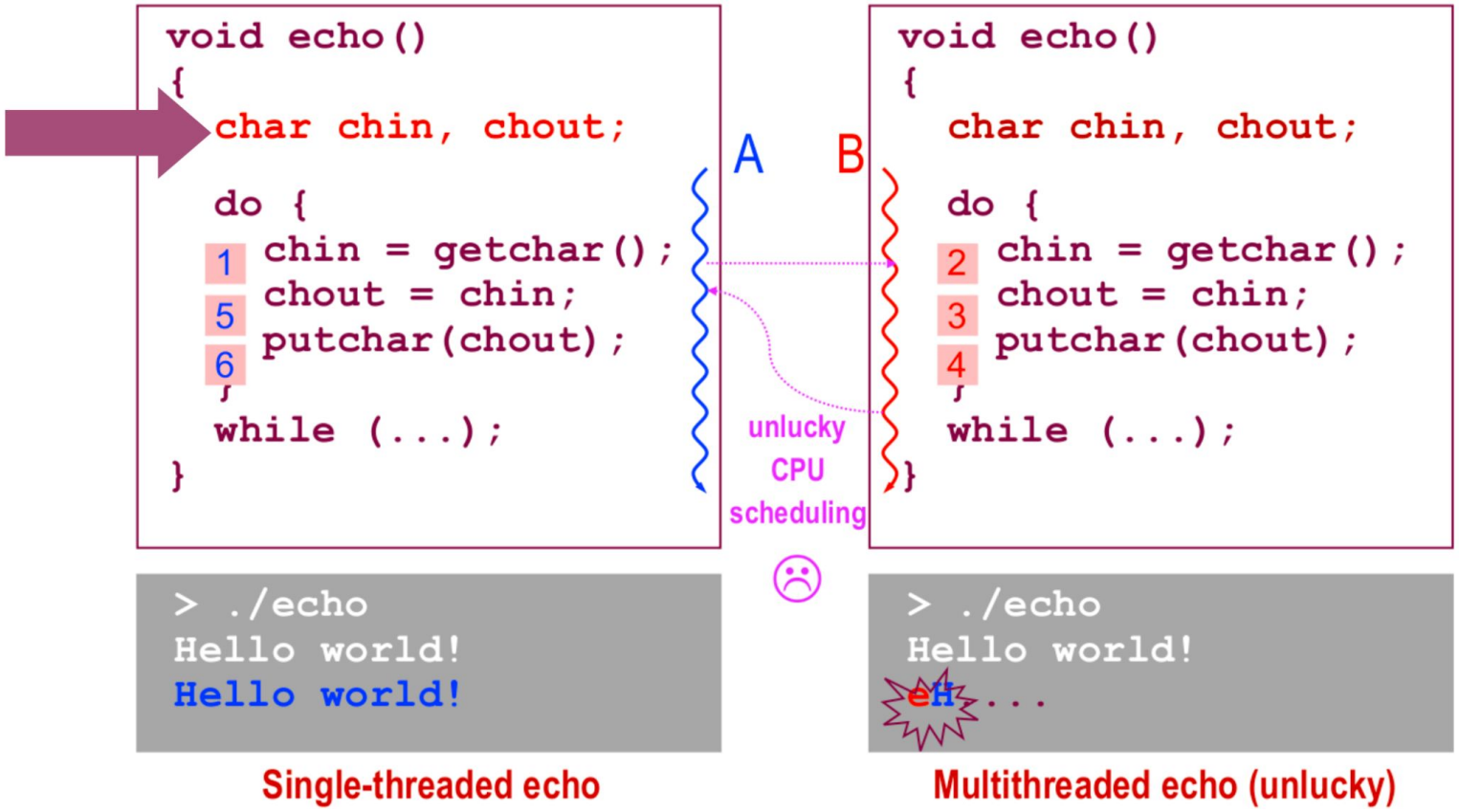
Race Condition



Race Condition



Race Condition



Race Condition

- Significant race conditions in I/O & variable sharing
- In this case, replacing the global variables with local variables did not solve the problem
- We actually had **two** race conditions here:
    - One race condition in the **shared variables** and the order of value assignment
    - Another race condition in the **shared output stream**
        - Which thread is going to write to output first (this race persisted even after making the variables local to each thread)

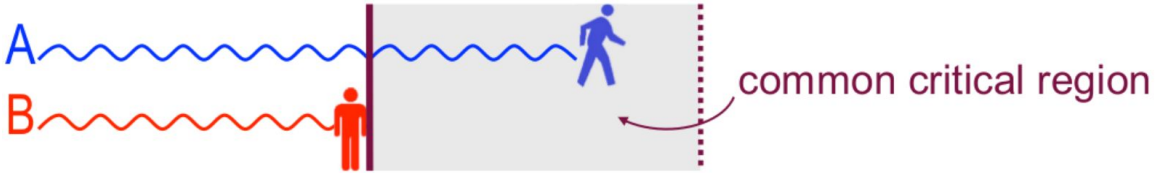
➢ Generally, race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)

Critical Section

- Critical section/region: segment of code in which the process may be changing shared data (e.g. common variables)
- No two processes should be executing in their critical sections at the same time
    - prevents race conditions
- Critical section problem: design a protocol that the processes use to cooperate

Critical Section

- The “indivisible” execution blocks are critical sections/regions
  - A critical section/region is a section of code that should be executed by only one process or thread at a time



  - Although it is not necessarily the same region of memory or section of program in both processes

➢ Physically different or not, what matters is that these regions cannot be interleaved or executed in parallel (pseudo or real)

Solution to Critical-Section Problem

A solution to the critical-section problem **must satisfy** the following requirements:

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

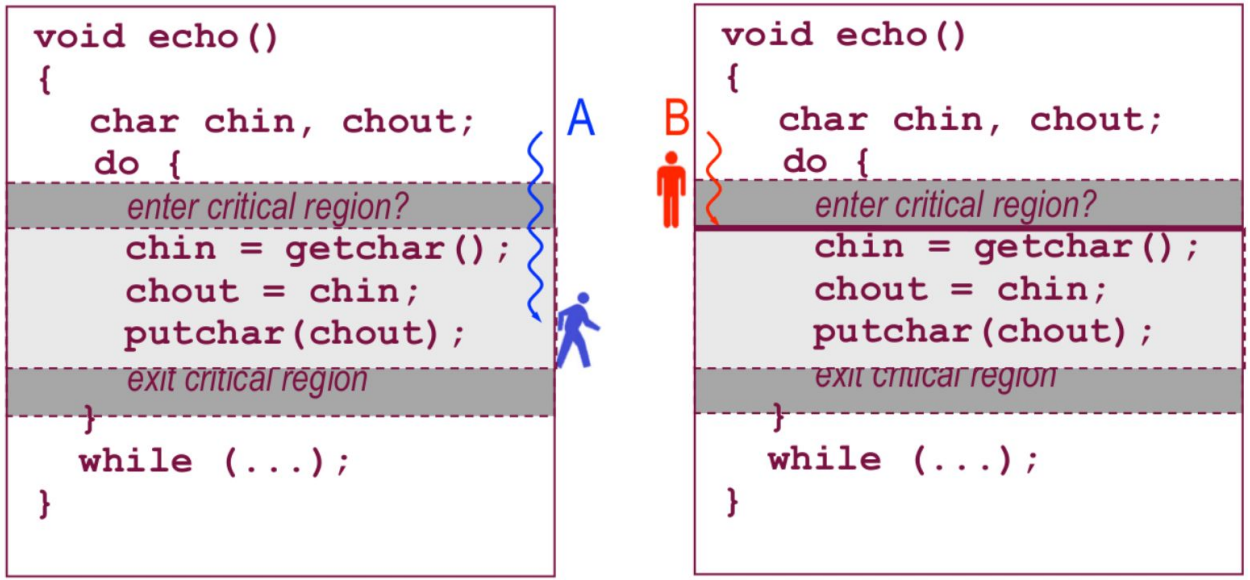
b. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

c. Bounded Waiting - 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

Solution to Critical-Section Problem

- We need mutual exclusion from critical sections
  - Critical sections can be protected from concurrent access by padding them with entrance and exit gates (we'll see how)
    - A thread must try to check in, then it must check out

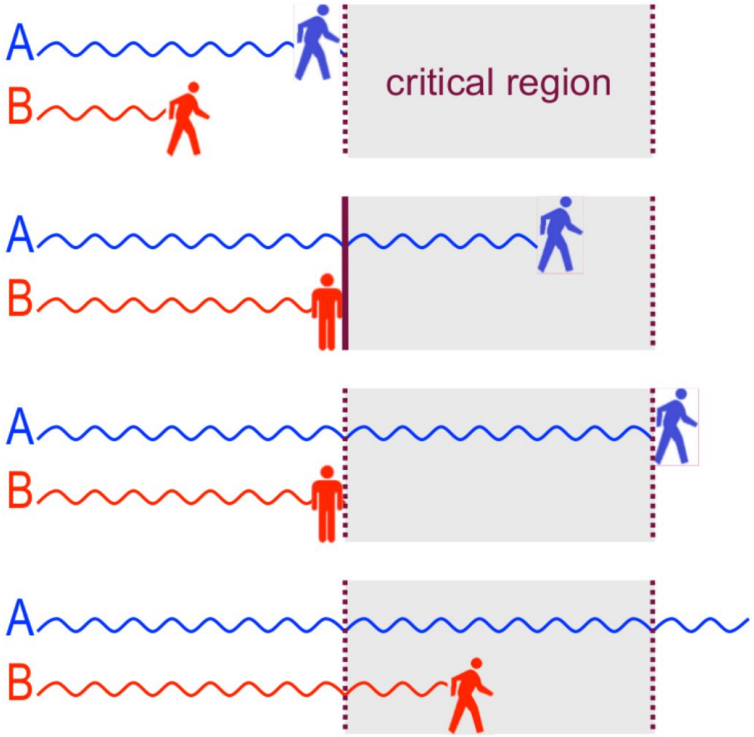


Solution to Critical-Section Problem

- Desired effect: mutual exclusion from the critical region (CR)

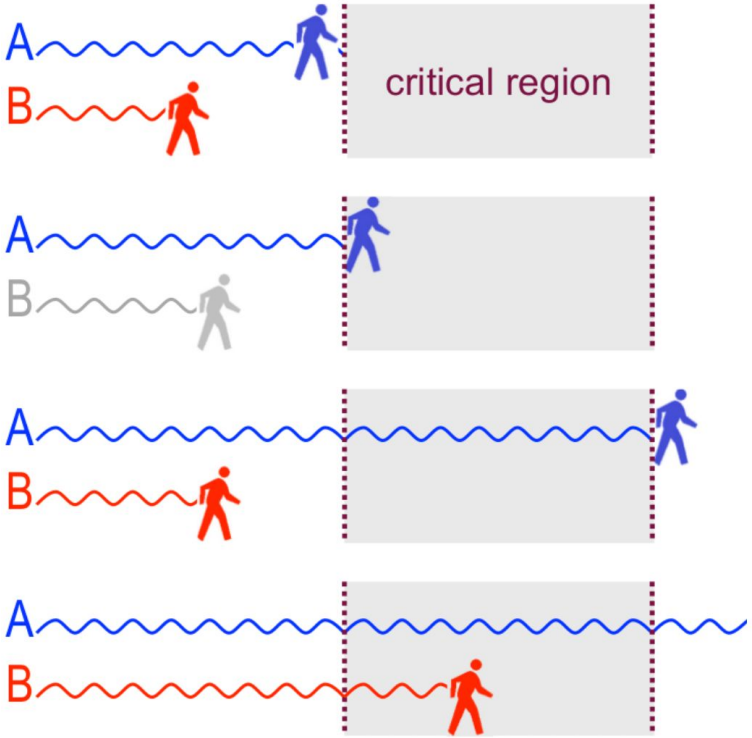
1. Thread A reaches the CR before B
2. Thread A enters CR first preventing B from entering (B is waiting or is blocked)
3. Thread A exits CR. thread B can now enter
4. Thread B enters CR

HOW?



Solution to Critical-Section Problem - Solution 1

- **Implementation 1** - disabling hardware interrupts
  1. Thread A reaches the CR before B
  2. As soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
  3. As soon as A exits CR, it enables interrupts. B can be scheduled again
  4. Thread B enters CR



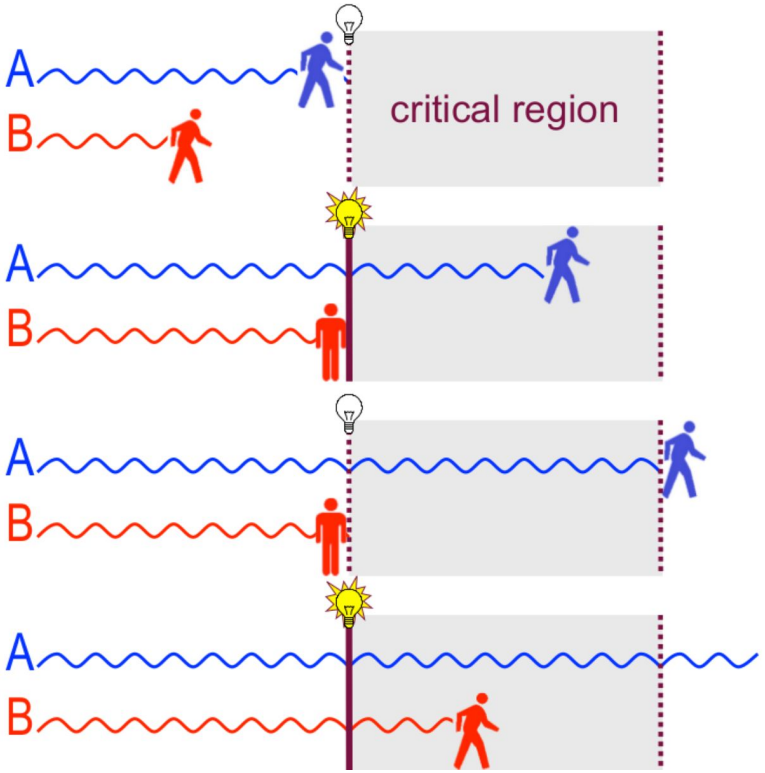
Solution to Critical-Section Problem - Solution 1

- **Implementation 1** - disabling hardware interrupts
  - It works, but it is not reasonable!
  - What guarantees that the user process is going to ever exit the critical region?
  - Meanwhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - The critical region becomes one physically indivisible block, not logically.
  - Also this is not working in multi-processors

```
void echo()
{
   char chin, chout;
   do {
      disable hardware interrupts
      chin = getchar();
      chout = chin;
      putchar(chout);
      enable hardware interrupts
   }
   while (...);
}
```

Solution to Critical-Section Problem - Solution 2

- **Implementation 2** - simple lock variable
  - Thread A reaches the CR and finds a lock at 0 (which means A can enter)
  - Thread A sets the lock to 1 and enters CR, which prevents B from entering
  - Thread A exits CR and resets lock to 0, thread B can enter
  - Thread B sets the lock to 1 and enters CR



Solution to Critical-Section Problem - Solution 2

- **Implementation 2** - simple lock variable
  - The "lock" is a shared variable
  - Entering the critical region means testing and then setting the lock
  - Exiting means resetting the lock

```
while (lock);
     /* do nothing: loop */
lock = TRUE;
```

```
lock = FALSE;
```

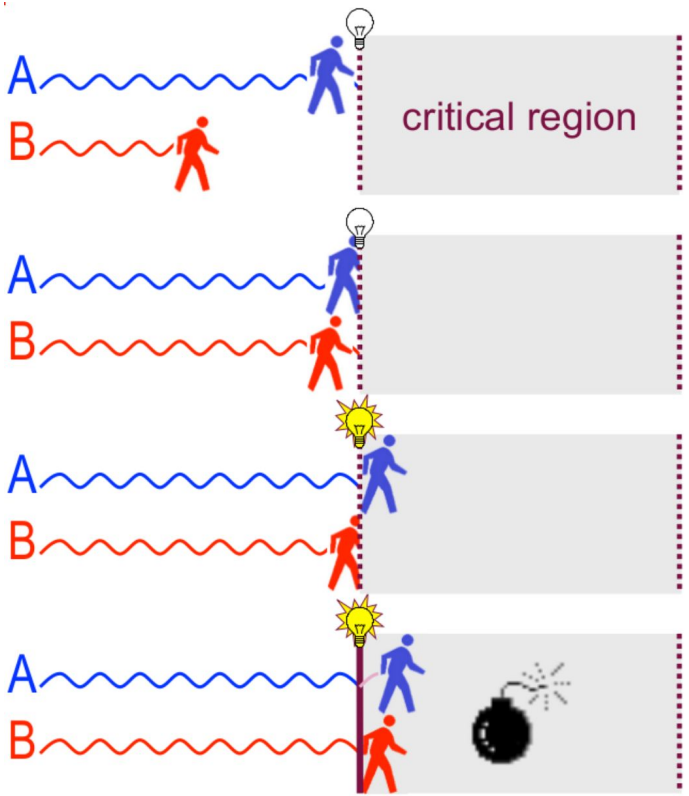
```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      chin = getchar();
      chout = chin;
      putchar(chout);
   }
   while (...);
}
```

test lock, then set lock

reset lock

Solution to Critical-Section Problem - Solution 2

- **Implementation 2** - simple lock variable
  1. Thread A reaches the CR and finds a lock at 0 (which means A can enter)
  2. Before A can set the lock to 1, Thread B reaches CR and also finds the the lock is 0
  3. A sets the lock to 1 and enters CR but cannot prevent the fact that…
  4. … B is going to set the lock to 1 and enter CR too!



Solution to Critical-Section Problem - Solution 2

- **Implementation 2** - simple lock variable
  - Suffers from the very flaw we want to avoid: a race condition
  - The problem comes from the small gap between testing that the lock off and setting it

```
while (lock);
      /* do nothing: loop */
lock = TRUE;
```

  - The other thread might get scheduled right in between the gap
  - So they both find the lock off and they both set it and enter

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      chin = getchar();
      chout = chin;
      putchar(chout);
   }
   while (...);
}
```

test lock, then set lock

reset lock

Solution to Critical-Section Problem - Solution 3

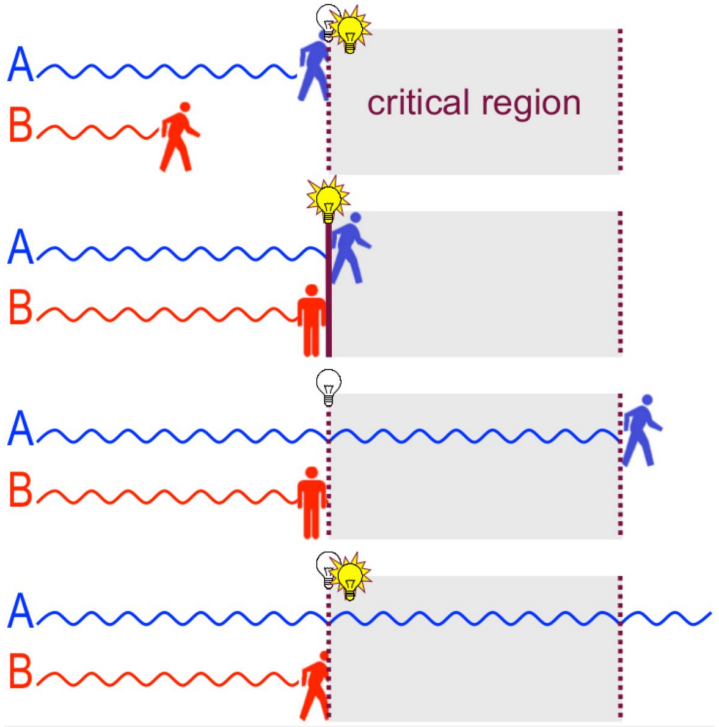
- **Implementation 3 -** “indivisible” lock variable
  - The “indivisibility” of the “test-lock-and-set-lock” operation can be implemented with the hardware TSL

```
enter_region:
     TSL REGISTER, LOCK          /* copy lock to register AND set lock to 1 */
     CMP REGISTER, #0            /* was lock zero? */
     JNE enter_region            /* if it was non-zero, lock was set. so loop */
     RET                         /* return to caller, critical region entered */

leave_region:
     MOV LOCK, #0                /* store a 0 in lock */
     RET                         /* return to caller */
```

Solution to Critical-Section Problem - Solution 3

- **Implementation 3 -** "indivisible" lock variable

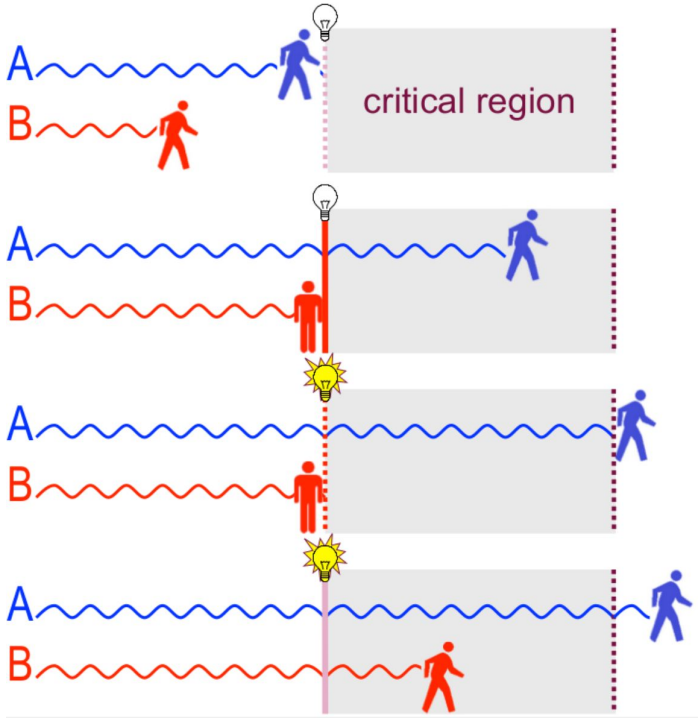
1. Thread A reaches the CR and finds a lock at 0 and sets it in one shot, then enters
2. Even if thread B comes right behind A it will find that the lock is already at 1
3. Thread A exits CR and resets lock to 0, thread B can enter
4. Thread B finds the lock is 0 and sets it in one shot, then enters



Solution to Critical-Section Problem - Solution 4

- **Implementation 4 -** no-TSL toggle for two threads

1. Thread A reaches the CR Finds a lock at 0, and enters **without changing the lock**
2. However, the lock has an **opposite meaning for B**: “off” means do not enter
3. Only when A exits CR, it changes the lock to 1 and thread B can now enter
4. Thread B enters CR, it will reset it to 0 for A after exiting



Solution to Critical-Section Problem - Solution 4

- **Implementation 4 -** no-TSL toggle for two threads
  - The "toggle lock" is a shared variable used for strict alternation
  - Here, entering CR means **only testing** the toggle: it must be 0 for A and 1 for B
  - Exiting means switching the toggle: A sets it to 1, B sets it to 0

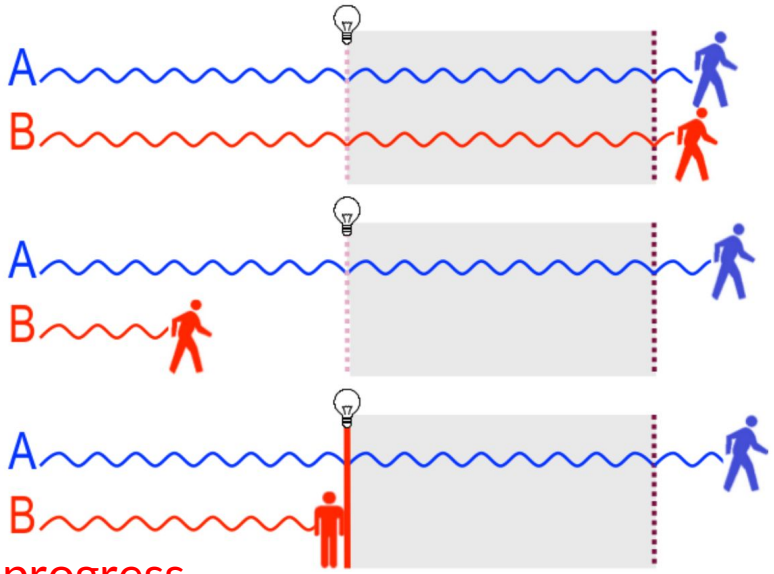
A's code	B's code
`while (toggle); /* loop */`	`while (!toggle); /* loop */`
`toggle = TRUE;`	`toggle = FALSE;`

```
bool toggle = FALSE;

void echo()
{
   char chin, chout;
   do {
      test toggle
      chin = getchar();
      chout = chin;
      putchar(chout);
      switch toggle
   }
   while (...);
}
```

Solution to Critical-Section Problem - Solution 4

- **Implementation 4 -** no-TSL toggle for two threads

1. Thread B exits CR and switches the lock back to 0 to allow A to enter next
2. But scheduling happens to make B faster than A and come back to the gate first
3. As long as A is still busy or interrupted in its noncritical Region, B is barred access to its CR



This violates item *b* from solution requirements: progress

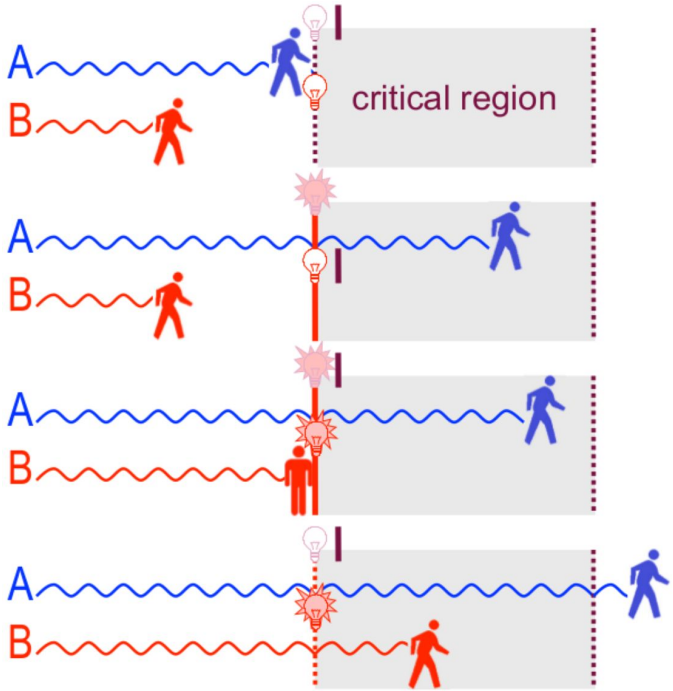
This implementation avoids TSL by splitting test & set and putting them in enter & exit

Nice try... but flawed!

Solution to Critical-Section Problem - Solution 5

- **Implementation 5 -** Peterson's no-TSL, no-alternation

1. A and B each have their own lock, an extra toggle is also masking either lock
2. A arrives first, sets its lock, pushes the mask to the other lock and may enter
3. Then B also sets its lock and pushes the mask, but must wait until A's lock is reset
4. A exits the CR and resets its lock B may now enter



Solution to Critical-Section Problem - Solution 5

- **Implementation 5 -** Peterson's no-TSL, no-alternation
  - The mask(turn) and two locks are shared
  - Entering means: setting one's lock, Pushing the mask and testing **other's** lock.
  - Exit means resetting one's lock

A's code	B's code

`lock[A] = TRUE;`
`turn = B;`
`while (lock[B] &&`
`turn == B);`
`/* loop */`
	`lock[B] = TRUE;`
	`turn = A;`
	`while (lock[A] &&`
	`turn == A);`
	`/* loop */`

`lock[A] = FALSE;`	`lock[B] = FALSE;`

```
bool lock[2];
int mask;
int A = 0, B = 1;
void echo()
{
   char chin, chout;
   do {
      chin = getchar();
      chout = chin;
      putchar(chout);
   }
   while (...);
}
```

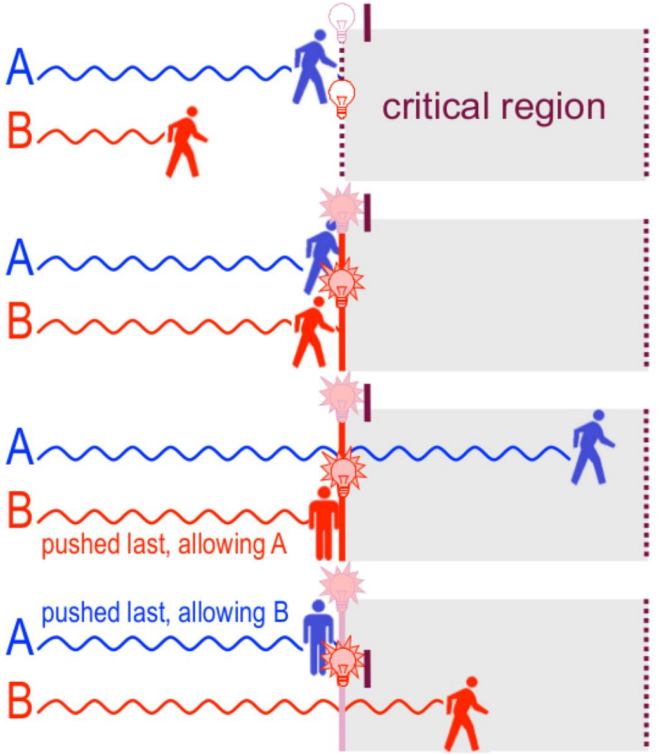
set lock, push mask, and test

reset lock

Solution to Critical-Section Problem - Solution 5

- **Implementation 5 -** Peterson's no-TSL, no-alternation

1. A and B each have their own lock, an extra toggle is also masking either lock
2. A is interrupted between setting the lock and pushing the mask. B sets its lock
3. Now both A and B race to push the mask. Whoever does it last will allow the other one enter



Mutual exclusion holds!!
No bad race condition

Solution to Critical-Section Problem

- Summary of implementations of mutual exclusion
  - Implementation 1 - disabling hardware interrupts
    - NO: race condition avoided, but can crash the system!
  - Implementation 2 - simple lock variable (unprotected)
    - NO: still suffers from race condition
  - Implementation 3 - indivisible lock variable (TSL)
    - YES: works, but requires hardware support

This will be the basis for "mutexes"

  - Implementation 4 - no-TSL toggle for two threads
    - NO: race condition avoided inside, but lockup outside
  - Implementation 5 - Peterson's no-TSL, no-alternation
    - YES: works in software, but has processing overhead

Solution to Critical-Section Problem

- Problem: All implementations (2-5) rely on busy waiting
  - “Busy waiting” means that the process/thread continuously executes a tight loop until some condition changes
  - Busy waiting is bad ☹️
    - **Waste of CPU time**- the busy process is not doing anything useful, yet remains “READY” instead of “BLOCKED”
    - **Paradox of inversed priority**- by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and liberating B! (B is working against its own interest)

- We need for the waiting process to block, not keep idling!

Synchronization in Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions

  (Atomic = non-interruptible)

  - Either test memory word and set value
  - Swap contents of two memory words

Acknowledgements

- “Operating Systems Concepts” book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- “Operating Systems: Internals and Design Principles” book and supplementary material by W. Stallings
- “Modern Operating Systems” book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo

Announcement

- Homework 4 released
  - Due on November 10th at 11.59PM
- Next Class
  - Discussion about homework 4
  - Please read the PDF before the next class for better understanding