CSC 4103 - Operating Systems
Fall 2009

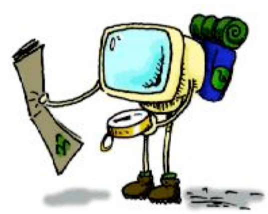
Lecture - IX
Process Synchronization - II

Tevfik Koşar

Louisiana State University
September 22nd, 2009

Roadmap

- Solutions for Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber

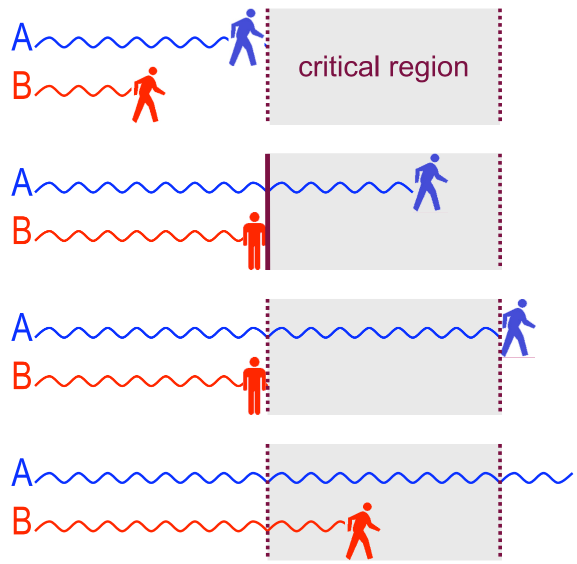


Mutual Exclusion

- <u>Desired effect: mutual exclusion from the critical region</u>

HOW is this achieved??

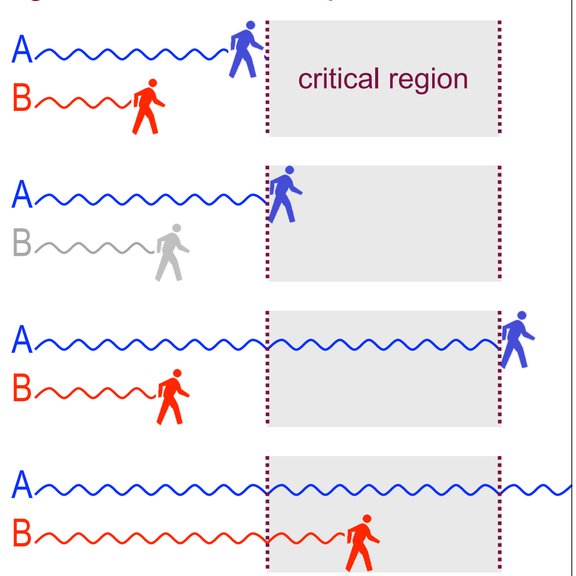
1. thread A reaches the gate to the critical region (CR) before B
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
3. thread A exits CR; thread B can now enter
4. thread B enters CR



Mutual Exclusion

- <u>**Implementation 1** — disabling hardware interrupts</u>

1. thread A reaches the gate to the critical region (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



Mutual Exclusion

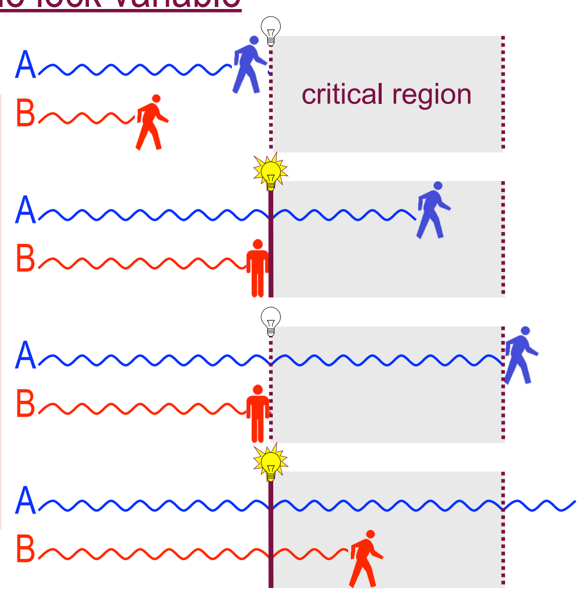
- **Implementation 1 — ~~disabling hardware interrupts~~**
  - ✓ it works, but not reasonable!
  - ✓ what guarantees that the user process is going to ever exit the critical region?
  - ✓ meanwhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - ✓ the critical region 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 (...);
}
```


Mutual Exclusion

- **Implementation 2 — simple lock variable**

1. thread A reaches CR and finds a lock at 0, which means that A can enter
2. thread A sets the lock to 1 and enters CR, which prevents B from entering
3. thread A exits CR and resets lock to 0; thread B can now enter
4. thread B sets the lock to 1 and enters CR



Mutual Exclusion

- **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 {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```


Mutual Exclusion

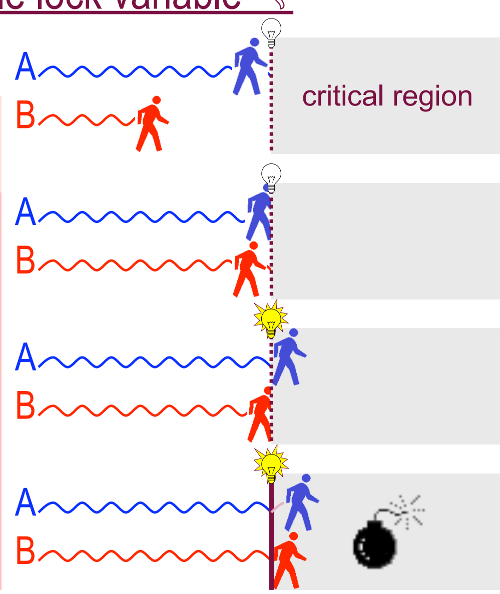
- **Implementation 2 — ~~simple lock variable~~**

1. thread A reaches CR and finds a lock at 0, which means that A can enter

1.1 but before A can set the lock to 1, B reaches CR and finds the lock is 0, too

1.2 A sets the lock to 1 and enters CR but cannot prevent the fact that . . .

1.3 . . . B is going to set the lock to 1 and enter CR, too



Mutual Exclusion

➢ 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 is off and setting the lock

```
while (lock);   lock = TRUE;
```

- ✓ it may happen that the other thread gets scheduled exactly in between these two actions (falls in the gap)
- ✓ so they both find the lock off and then they both set it and enter

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```


Mutual Exclusion

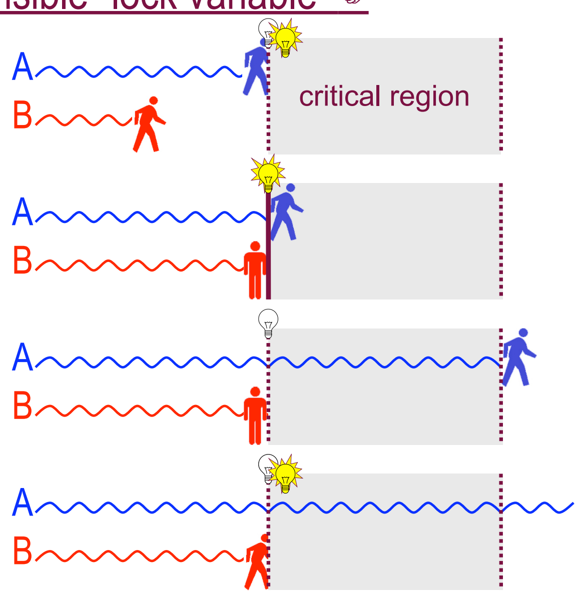
➢ Implementation 3 — "indivisible" lock variable

1. thread A reaches CR and finds the lock at 0 *and* sets it in one shot, then enters

1.1' even if B comes right behind A, it will find that the lock is already at 1

2. thread A exits CR, then resets lock to 0

3. thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR



Mutual Exclusion

➢ **Implementation 3** — “indivisible” lock variable ☝

✓ the indivisibility of the “test-lock-and-set-lock” operation can be implemented with the hardware instruction **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:
    MOVE LOCK,#0          | store a 0 in lock
    RET                   | return to caller
```

```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Mutual Exclusion

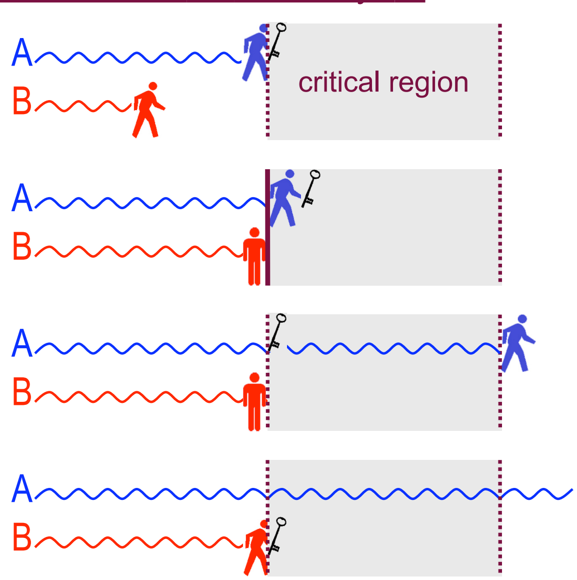
➢ **Implementation 3** — “indivisible” lock ⇔ one key ☝

1. thread A reaches CR and finds a key *and* takes it

1.1’ even if B comes right behind A, it will not find a key

2. thread A exits CR and puts the key back in place

3. thread B finds the key and takes it, just before entering CR



Mutual Exclusion

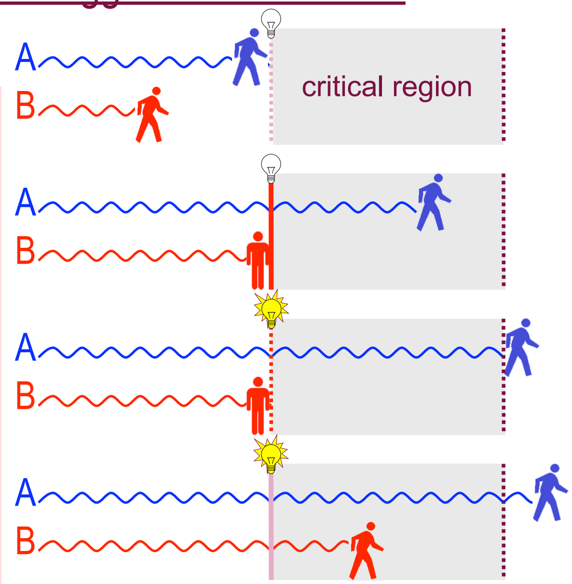
- **Implementation 3 — “indivisible” lock ⇔ one key** ☝
  - ✓ “holding” a unique object, like a key, is an equivalent metaphor for “test-and-set”
  - ✓ this is similar to the “speaker’s baton” in some assemblies: only one person can hold it at a time
  - ✓ holding is an indivisible action: you see it and grab it in one shot
  - ✓ after you are done, you release the object, so another process can hold on to it

```
void echo()
{
   char chin, chout;
   do {
       take key and run
       chin = getchar();
       chout = chin;
       putchar(chout);
       return key
   }
   while (...);
}
```


Mutual Exclusion

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

1. thread A reaches CR, finds a lock at 0, and enters <u>without changing the lock</u>
2. however, the lock has an <u>opposite meaning for B</u>: “off” means do not enter
3. only when A exits CR does it change the lock to 1; thread B can now enter
4. thread B sets the lock to 1 and enters CR: it will reset it to 0 for A after exiting



Mutual Exclusion

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

- ✓ the "toggle lock" is a shared variable used for strict alternation
- ✓ here, entering the critical region means only testing the toggle: it must be at 0 for A, and 1 for B
- ✓ exiting means switching the toggle: A sets it to 1, and B to 0

A's code

```
while (toggle);
   /* loop */
```

```
toggle = TRUE;
```

B's code

```
while (!toggle);
   /* loop */
```

```
toggle = FALSE;
```

```
bool toggle = FALSE;

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


Mutual Exclusion

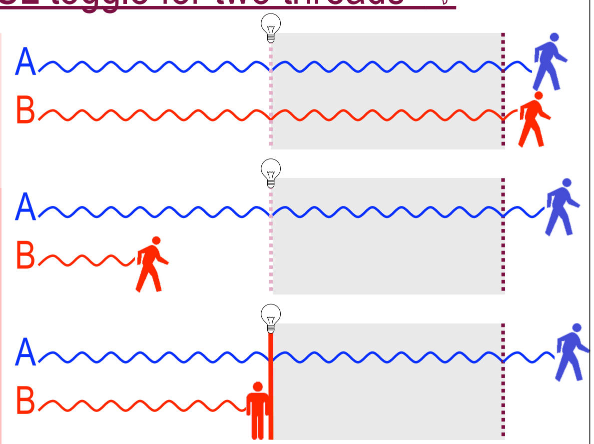
➢ **Implementation 4** — ~~no-TSL toggle for two threads~~

5. thread B exits CR and switches the lock back to 0 to allow A to enter next

5.1 but scheduling happens to make B faster than A and come back to the gate first

5.2 as long as A is still busy or interrupted in its noncritical region, B is barred access to its CR

® *this violates item 2. of the chart of mutual exclusion*

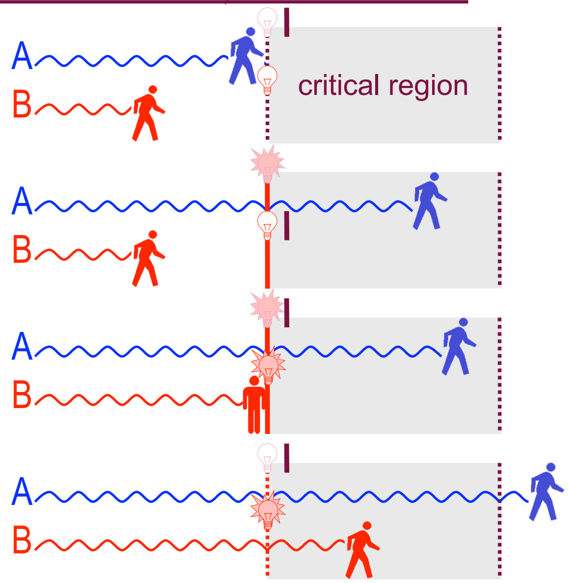


® *this implementation avoids TSL by splitting test & set and putting them in enter & exit; nice try... but flawed!*

Mutual Exclusion

➢ **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 & 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



Mutual Exclusion

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

- ✓ the mask & two locks are shared
- ✓ entering means: setting one's lock, pushing the mask and tetsing the other's combination
- ✓ exiting means resetting the lock

A's code

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

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

B's code

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

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

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

Mutual Exclusion

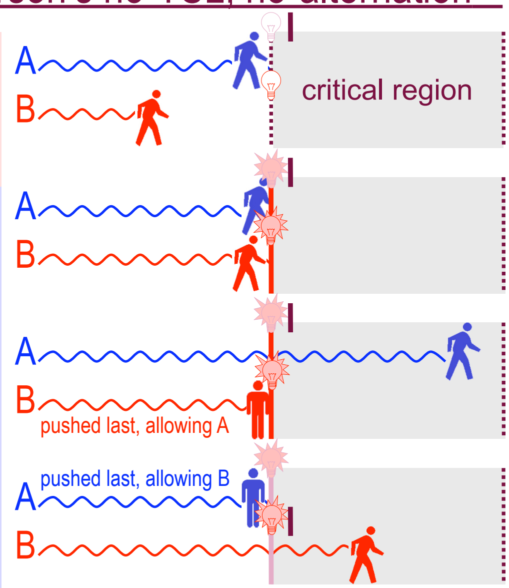
➢ **Implementation 5** — Peterson's no-TSL, no-alternation

1. A and B each have their own lock; an extra toggle is also masking either lock

2.1 A is interrupted between setting the lock & pushing the mask; B sets its lock

2.2 now, both A and B race to push the mask: whoever does it last will allow the other one inside CR

® *mutual exclusion holds!! (no bad race condition)*



Mutual Exclusion

➢ Summary of these implementations of mutual exclusion

- ✓ **Impl. 1 — disabling hardware interrupts**
  - NO: race condition avoided, but can crash the system!
- ✓ **Impl. 2 — simple lock variable (unprotected)**
  - NO: still suffers from race condition
- ✓ **Impl. 3 — indivisible lock variable (TSL)**
  - YES: works, but requires hardware

  this will be the basis for "mutexes"
- ✓ **Impl. 4 — no-TSL toggle for two threads**
  - NO: race condition avoided inside, but lockup outside
- ✓ **Impl. 5 — Peterson's no-TSL, no-alternation**
  - YES: works in software, but processing overhead

Mutual Exclusion

- 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 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 not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify wait() and signal()
  - Originally called P() and V()

  - ```
    wait (S) {
         while S <= 0
                   ; // no-op
           S--;
      }
    ```

  - ```
    signal (S) {
        S++;
      }
    ```

- Less complicated
- Can only be accessed via two indivisible (atomic) operations

Semaphores as Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks

- Provides mutual exclusion
  - Semaphore S;   // initialized to 1
  - wait (S);

    ```
          Critical Section
    signal (S);
    ```

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
.	.
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping Barber Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex for access to the buffer, initialized to 1
- Semaphore full (number of full buffers) initialized to 0
- Semaphore empty (number of empty buffers) initialized to N

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {

        //   produce an item

    wait (empty);
    wait (mutex);

        //  add the item to the  buffer

    signal (mutex);
    signal (full);
```

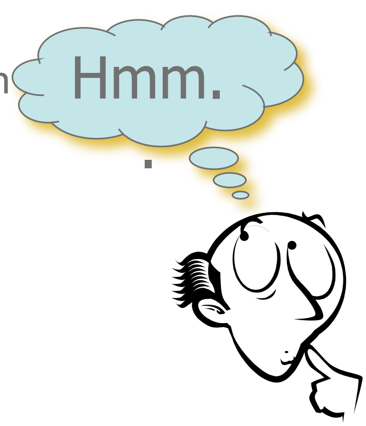
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

      //  remove an item from  buffer

     signal (mutex);
     signal (empty);

       //  consume the removed item
```


Summary

- Solutions for Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
  - Bounded Buffer
  - Readers-Writers
  - Dining Philosophers
  - Sleeping Barber



- Next Lecture: Deadlocks - I
- Reading Assignment: Chapter 6 from Silberschatz.

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 UNR