CSE 421/521 - Operating Systems
Fall 2012

LECTURE - VIII
PROCESS SYNCHRONIZATION - I

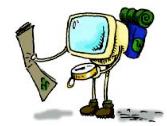
Tevfik Koşar

University at Buffalo

September 25th, 2012

Roadmap

- Process Synchronization
- Race Conditions
- Critical-Section Problem
  - Solutions to Critical Section
  - Different Implementations



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 is set to 0
  - It is incremented by the producer after it produces a new buffer
  - and is decremented by the consumer after it consumes a buffer.

Shared Variables: count=0, buffer[]

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


Race Condition

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

  must be performed atomically.
- Atomic operation means an operation without interruption.

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
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count   {register1 = 5}
  S1: producer execute register1 = register1 + 1   {register1 = 6}
  S2: consumer execute register2 = count   {register2 = 5}
  S3: consumer execute register2 = register2 - 1   {register2 = 4}
  S4: producer execute count = register1   {count = 6 }
  S5: consumer execute count = register2   {count = 4}

Race Condition

➢ Significant race conditions in I/O & variable sharing

```
char chin, chout;//shared

void echo()
{
  do {
1   chin = getchar();
2   chout = chin;
3   putchar(chout);
  }
  while (...);
}
```

A B — lucky CPU scheduling ☺

```
char chin, chout; //shared

void echo()
{
  do {
4   chin = getchar();
5   chout = chin;
6   putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
Hello world!
```

Multithreaded echo (lucky)

Race Condition

➢ Significant race conditions in I/O & variable sharing

```
char chin, chout;//shared

void echo()
{
  do {
1   chin = getchar();
5   chout = chin;
6   putchar(chout);
  }
  while (...);
}
```

A B — unlucky CPU scheduling ☹

```
char chin, chout; //shared

void echo()
{
  do {
2   chin = getchar();
3   chout = chin;
4   putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
ee...
```

Multithreaded echo (unlucky)

Race Condition

➢ Significant race conditions in I/O & variable sharing

```
void echo()
{
  char chin, chout;

  do {
1   chin = getchar();
5   chout = chin;
6   putchar(chout);
  }
  while (...);
}
```

A B — unlucky CPU scheduling ☹

```
void echo()
{
  char chin, chout;

  do {
2   chin = getchar();
3   chout = chin;
4   putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
eH...
```

Multithreaded echo (unlucky)

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)

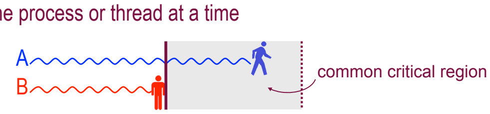
==> *generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)*

Critical Section/Region

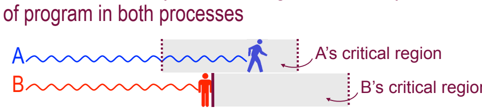
- Critical section/region: segment of code in which the process may be changing shared data (eg. common variables)
- No two processes should be executing in their critical sections at the same time --> prevents race conditions
- Critical section problem: design a protocol that the processes use to cooperate

Critical Section

➢ The "indivisible" execution blocks are critical regions
  - ✓ a critical region is a section of code that may 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



==> *but 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:

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. 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

Solution to Critical-Section Problem

3. 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

Mutual Exclusion

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

```
void echo()
{
   char chin, chout;
   do {
     enter critical region?
     chin = getchar();
     chout = chin;
     putchar(chout);
     exit critical region
   }
   while (...);
}
```

A

```
void echo()
{
   char chin, chout;
   do {
     enter critical region?
     chin = getchar();
     chout = chin;
     putchar(chout);
     exit critical region
   }
   while (...);
}
```

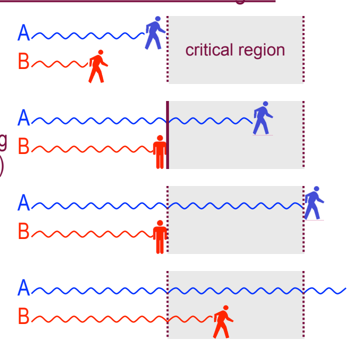
B

Mutual Exclusion

- Desired effect: mutual exclusion from the critical region

HOW is this achieved??

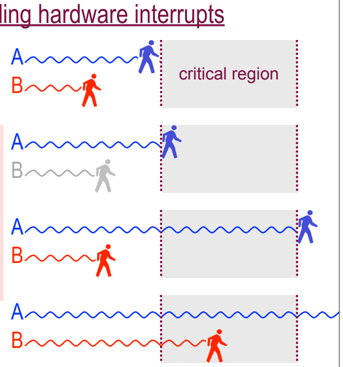
1. thread A reaches the gate to the critical region (CR) before B
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
3. thread A exits CR; thread B can now enter
4. thread B enters CR



Mutual Exclusion

- **Implementation 1** — disabling hardware interrupts

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

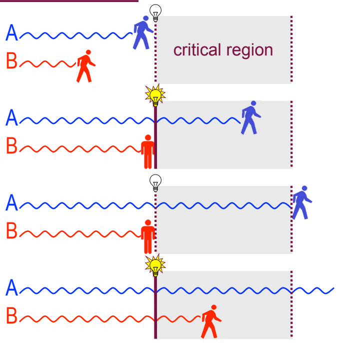
- **Implementation 1** — disabling hardware interrupts
  - ✓ it works, but not reasonable!
  - ✓ what guarantees that the user process is going to ever exit the critical region?
  - ✓ meanwhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - ✓ the critical region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

test lock, then set lock

reset lock

Mutual Exclusion

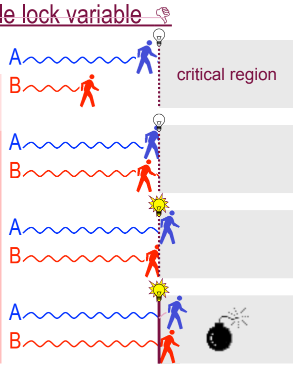
- **Implementation 2** — simple lock variable

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 {
      chin = getchar();
      chout = chin;
      putchar(chout);
   }
   while (...);
}
```

test lock, then set lock

reset lock

Mutual Exclusion

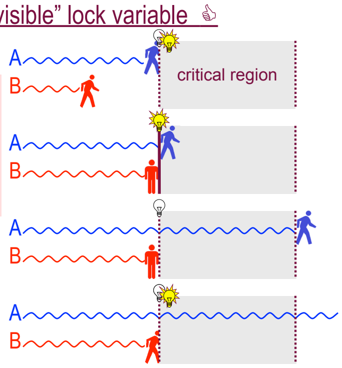
- **Implementation 3** — "indivisible" lock variable

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 {
      chin = getchar();
      chout = chin;
      putchar(chout);
   }
   while (...);
}
```

test-and-set-lock

set lock off

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

Mutual Exclusion

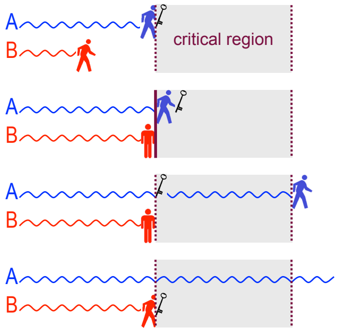
➢ **Implementation 3** — "indivisible" lock ⇔ one key

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

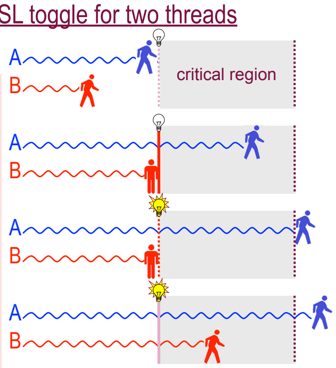
- ✓ "holding" a unique object, like a key, is an equivalent metaphor for "test-and-set"
- ✓ this is similar to the "speaker's baton" in some assemblies: only one person can hold it at a time
- ✓ holding is an indivisible action: you see it and grab it in one shot
- ✓ after you are done, you release the 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 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 does it change the lock to 1; thread B can now enter
4. thread B 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	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 (...);
}
```


Mutual Exclusion

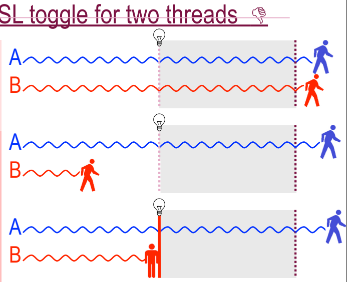
➢ **Implementation 4** — no-TSL toggle for two threads

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*

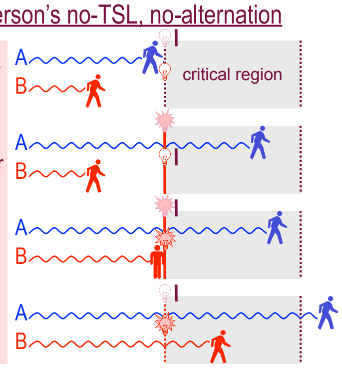


=> this implementation avoids TSL by splitting test & set and putting them in enter & exit; nice try... but flawed!

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

```
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 (...);
}
```

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;
```


Mutual Exclusion

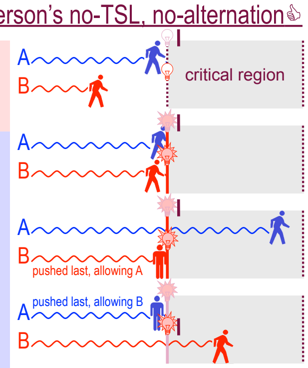
- **Implementation 5** — Peterson's no-TSL, no-alternation

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

Summary

- Process Synchronization
- Race Conditions
- Critical-Section Problem
  - Solutions to Critical Section
  - Different Implementations

- Next Lecture: Synchronization - II
- 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