CSC 4103 - Operating Systems
Fall 2009
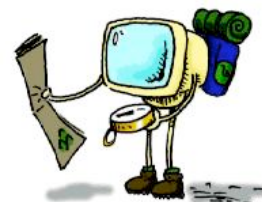
LECTURE - VIII
PROCESS SYNCHRONIZATION - I

Tevfik Koşar

Louisiana State University
September 17th, 2009

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.

3

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
```

4

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
  • **counter++;**
  • **counter--;**
    must be performed atomically.

✦ Atomic operation means an operation without interruption.

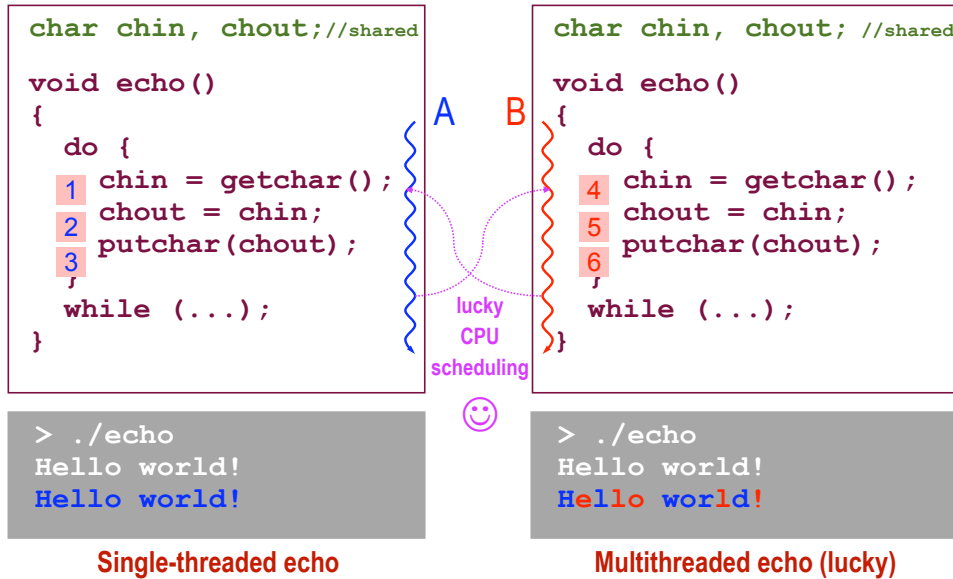
Race Condition

• 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 }
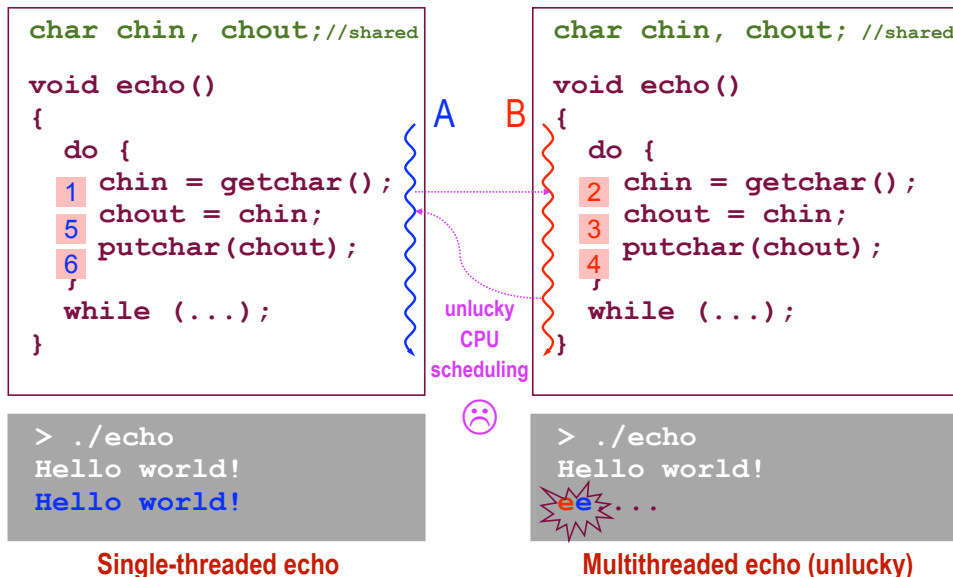   S5: consumer execute count = register2   {count = 4}

Race Condition

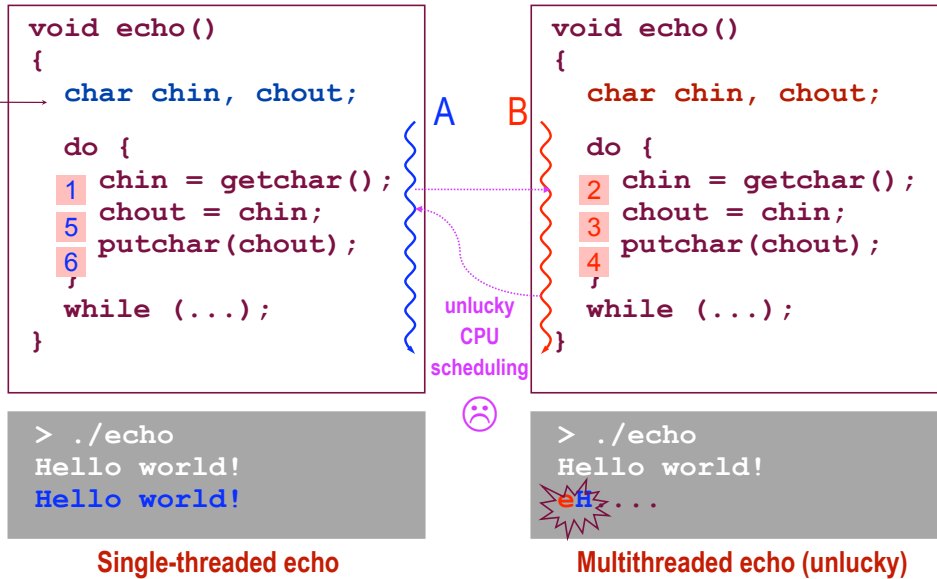➢ <u>Significant race conditions in I/O & variable sharing</u>

```
char chin, chout;//shared          char chin, chout; //shared

void echo()                        void echo()
{                    A    B        {
  do {                               do {
   1  chin = getchar();             4  chin = getchar();
   2  chout = chin;                 5  chout = chin;
   3  putchar(chout);              6  putchar(chout);
  }                      lucky        }
  while (...);           CPU          while (...);
}                      scheduling   }
```

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

Single-threaded echo   **Multithreaded echo (lucky)**

Race Condition

➢ <u>Significant race conditions in I/O & variable sharing</u>

```
char chin, chout;//shared          char chin, chout; //shared

void echo()                        void echo()
{                    A    B        {
  do {                               do {
   1  chin = getchar();             2  chin = getchar();
   5  chout = chin;                 3  chout = chin;
   6  putchar(chout);              4  putchar(chout);
  }                     unlucky       }
  while (...);           CPU          while (...);
}                      scheduling   }
```

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

Single-threaded echo   **Multithreaded echo (unlucky)**

Race Condition

➤ <u>Significant race conditions in I/O & variable sharing</u>

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

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

A   B

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

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

unlucky
CPU
scheduling

☹

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

Single-threaded echo

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

Multithreaded echo (unlucky)

9

Race Condition

➤ <u>Significant race conditions in I/O & variable sharing</u>

  ✓ in this case, replacing the global variables with local variables did not solve the problem

  ✓ we actually had <u>two</u> race conditions here:

   ▪ one race condition in the <u>shared variables</u> and the order of value assignment

   ▪ another race condition in the <u>shared output stream</u>: 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)

10

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

➢ <u>The "indivisible" execution blocks are critical regions</u>

   ✓ a critical region is a section of code that may be executed by only one process or thread at a time

common critical region
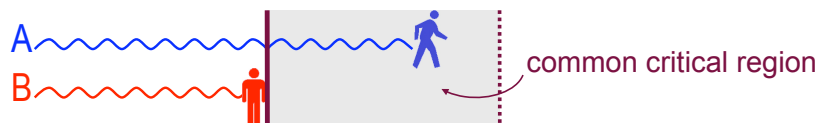
   ✓ although it is not necessarily the same region of memory or section of program in both processes

A's critical region
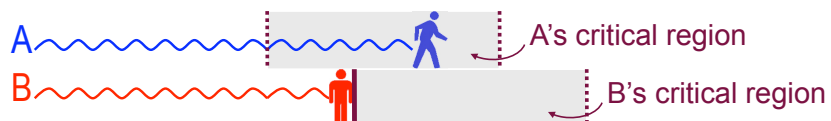
B's critical region

==> *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
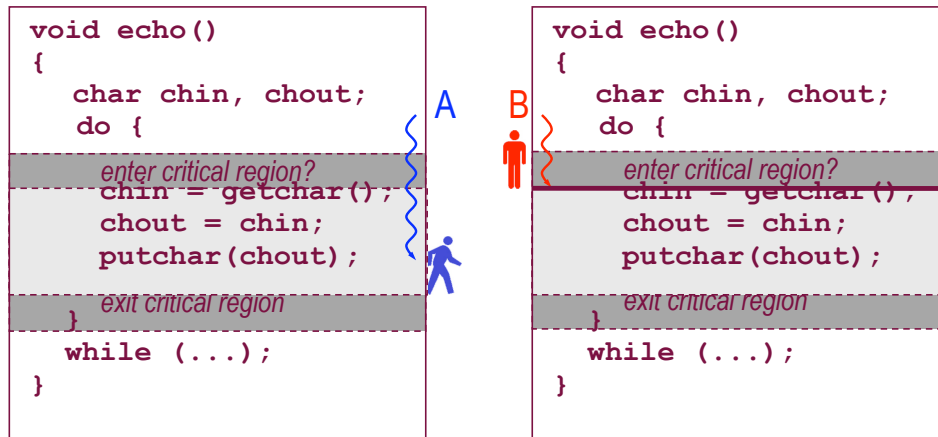
13

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
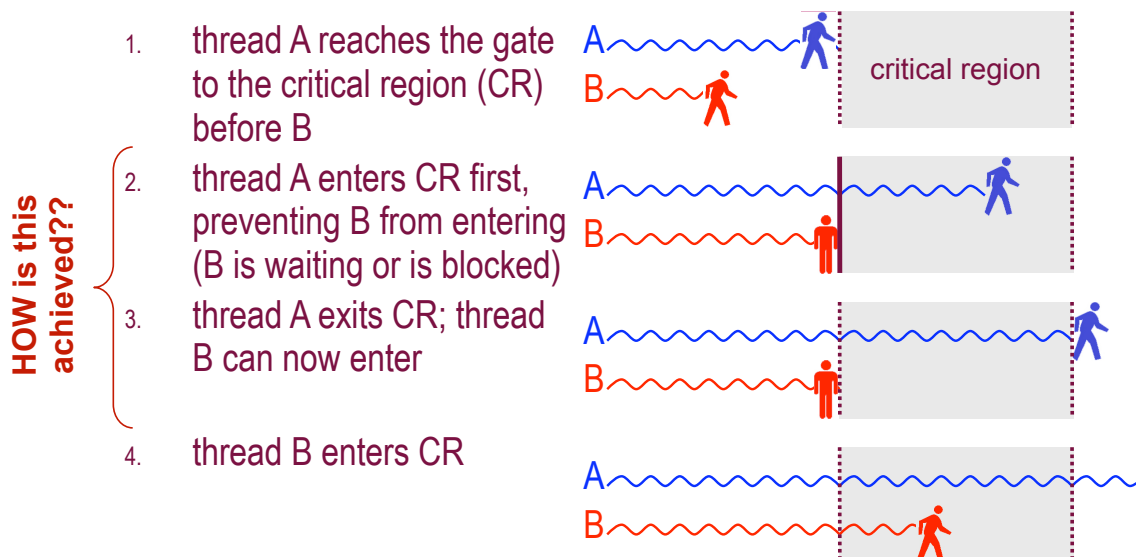   - No assumption concerning relative speed of the N processes

14

Critical Section

➢ <u>We need **mutual exclusion** from critical regions</u>
  ✓ 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

B
```
void echo()
{
   char chin, chout;
   do {
      enter critical region?
      chin = getchar();
      chout = chin;
      putchar(chout);

      exit critical region
   }
   while (...);
}
```

Mutual Exclusion

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

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

HOW is this achieved??

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

A

B
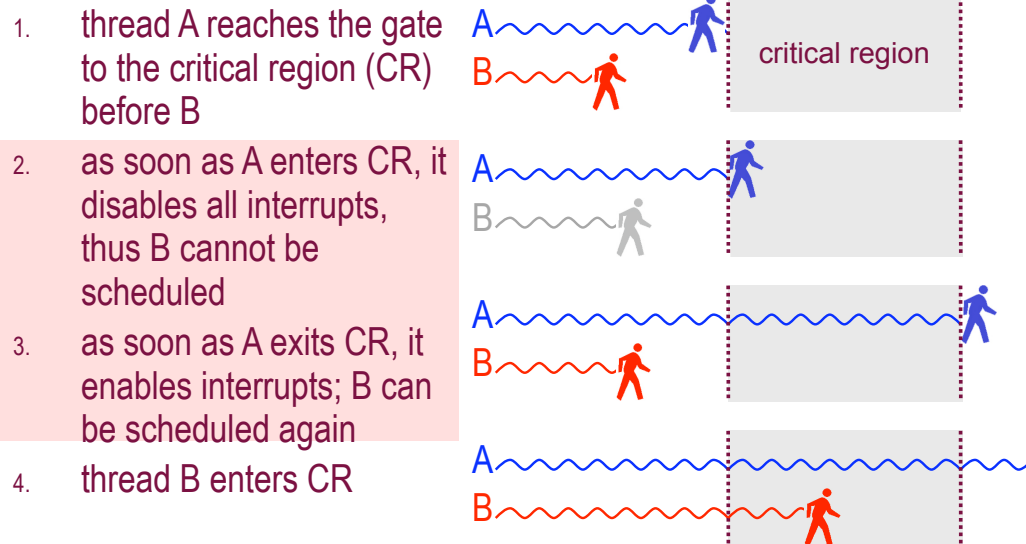
critical region

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 (...);
}
```

Summary

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

Hmm.
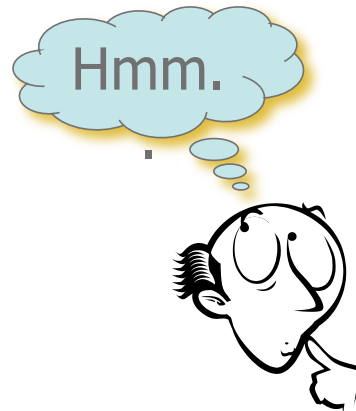
- Next Lecture: Synchronization - II
- Reading Assignment: Chapter 6 from Silberschatz.

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