

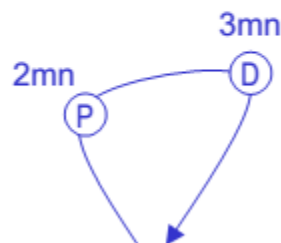
# Mutual Exclusion

# Mutual Exclusion

- ❑ Race conditions
- ❑ Solutions to mutual exclusion
- ❑ Semaphores

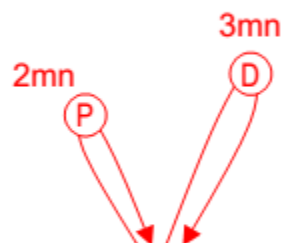
## ➤ Illustration: two shopping scenarios

### ✓ Single-threaded shopping



- you are in the grocery store
- first you go to produce and grab salad and apples, then you go to dairy and grab milk, butter and cheese
- it took you about  $1mn \times 5 \text{ items} = 5mn$

### ✓ Multithreaded shopping



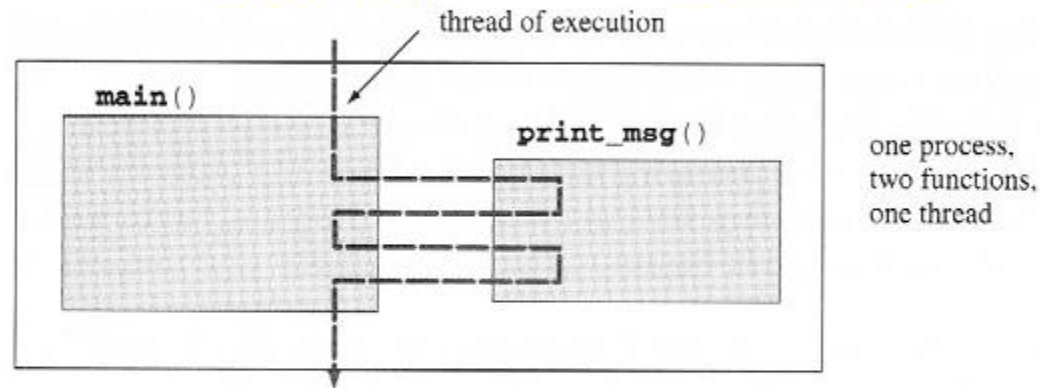
- you take your two kids with you to the grocery store
- you send them off in two directions with two missions, one toward produce, one toward dairy
- you wait for their return (at the slot machines) for a maximum duration of about  $1mn \times 3 \text{ items} = 3mn$

```
void main(...)  
{  
    char *produce[] = { "salad", "apples", NULL };  
    char *dairy[] = { "milk", "butter", "cheese", NULL };  
  
    print_msg(produce);  
    print_msg(dairy);  
}  
  
void print_msg(char **items)  
{  
    int i = 0;  
    while (items[i] != NULL) {  
        printf("grabbing the %s...", items[i++]);  
        fflush(stdout);  
        sleep(1);  
    }  
}
```

Single-threaded shopping code

## ➤ Results of single-threaded shopping

- ✓ total duration  $\approx$  5 seconds; outcome is deterministic



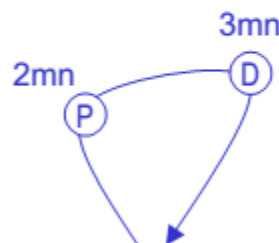
Molay, B. (2002) *Understanding  
Unix/Linux Programming* (1st Edition).

```
> ./single_shopping
grabbing the salad...
grabbing the apples...
grabbing the milk...
grabbing the butter...
grabbing the cheese...
>
```

Single-threaded shopping diagram and output

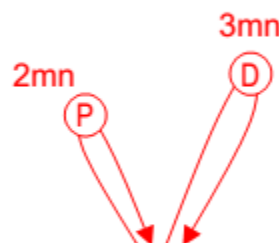
## ➤ Illustration: two shopping scenarios

### ✓ Single-threaded shopping



- you are in the grocery store
- first you go to produce and grab salad and apples, then you go to dairy and grab milk, butter and cheese
- it took you about  $1mn \times 5 \text{ items} = 5mn$

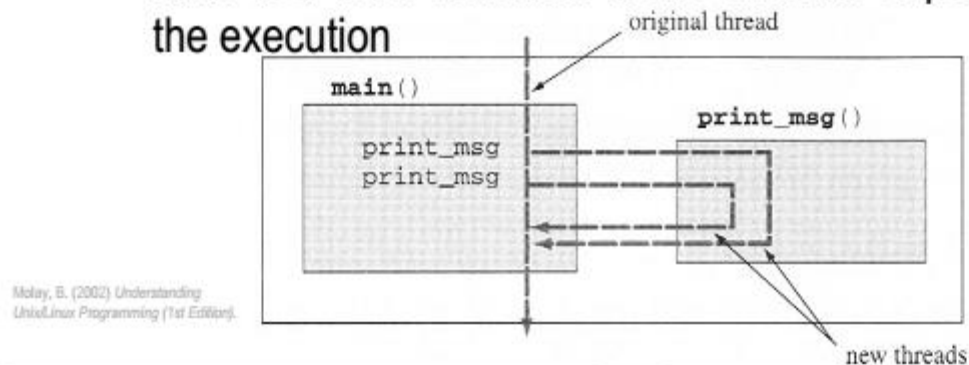
### ✓ Multithreaded shopping



- you take your two kids with you to the grocery store
- you send them off in two directions with two missions, one toward produce, one toward dairy
- you wait for their return (at the slot machines) for a maximum duration of about  $1mn \times 3 \text{ items} = 3mn$

## ➤ Inconsequential race condition in the shopping scenario

- ✓ there is a “race condition” if the outcome depends on the order of the execution



```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

Multithreaded shopping diagram and possible outputs

```

void main(...)
{
    char *produce[] = { "salad", "apples", NULL };
    char *dairy[] = { "milk", "butter", "cheese", NULL };
    void *print_msg(void *);
    pthread_t th1, th2;

    pthread_create(&th1, NULL, print_msg, (void *)produce);
    pthread_create(&th2, NULL, print_msg, (void *)dairy);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
}

void *print_msg(void *items)
{
    int i = 0;
    while (items[i] != NULL) {
        printf("grabbing the %s...", (char *) (items[i++]));
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}

```

*} send the kids off!*

*} wait for their return*

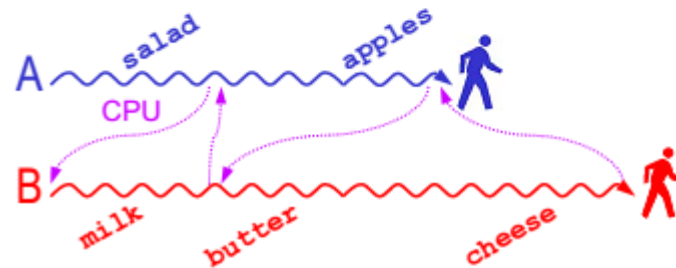
**Multithreaded shopping code**



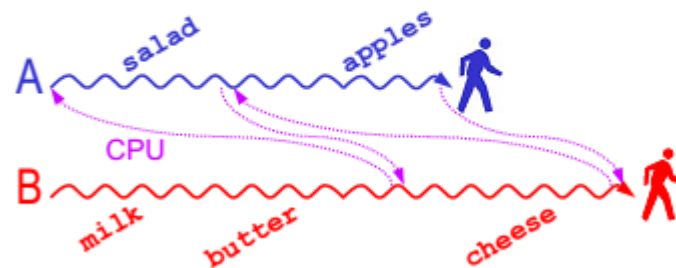
## ➤ Inconsequential race condition in the shopping scenario

- ✓ the outcome depends on the CPU scheduling or “interleaving” of the threads (separately, each thread always does the same thing)

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

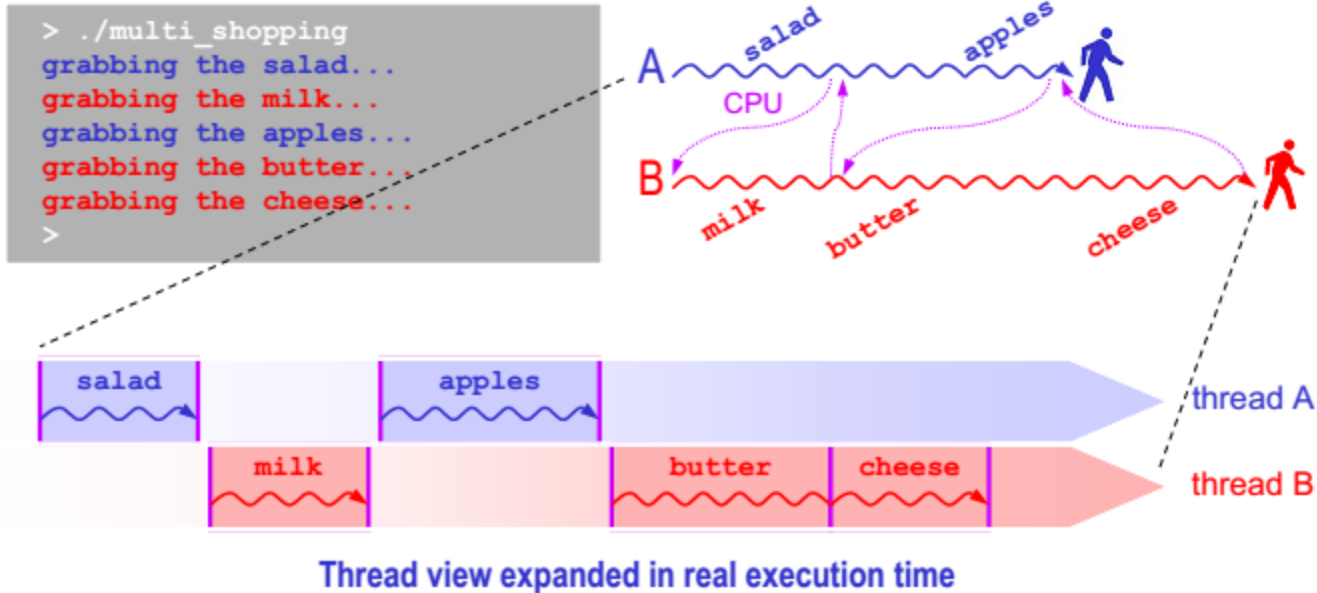


```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

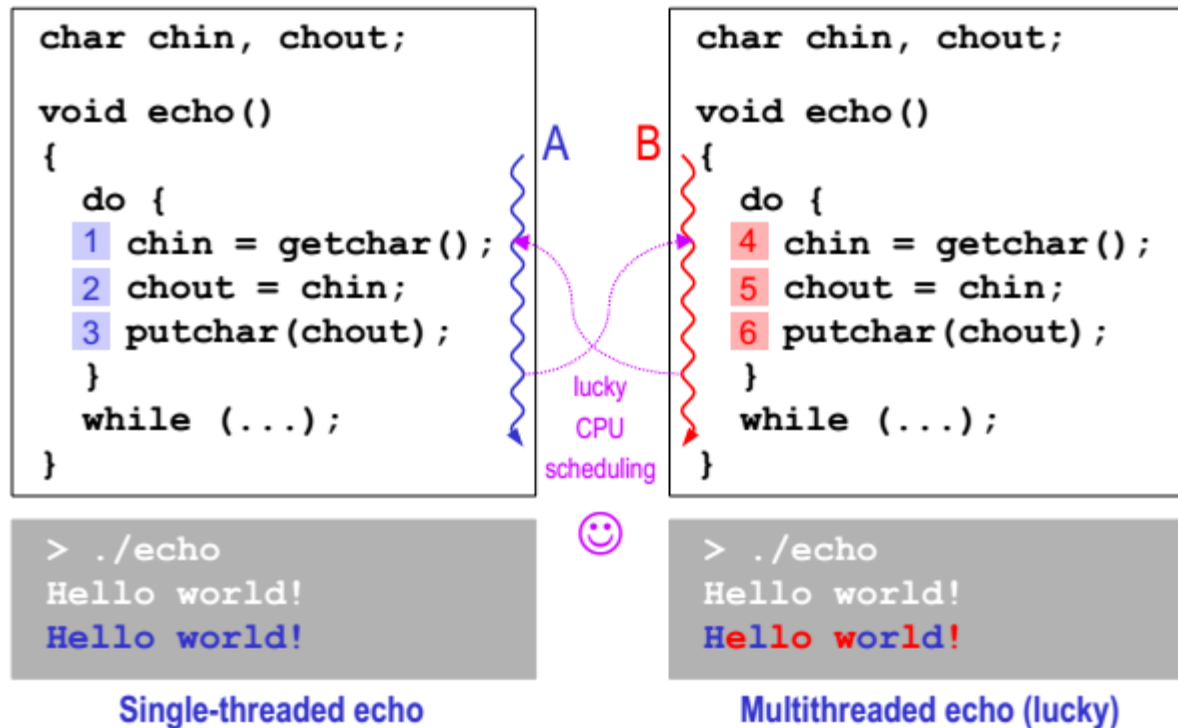


## ➤ Inconsequential race condition in the shopping scenario

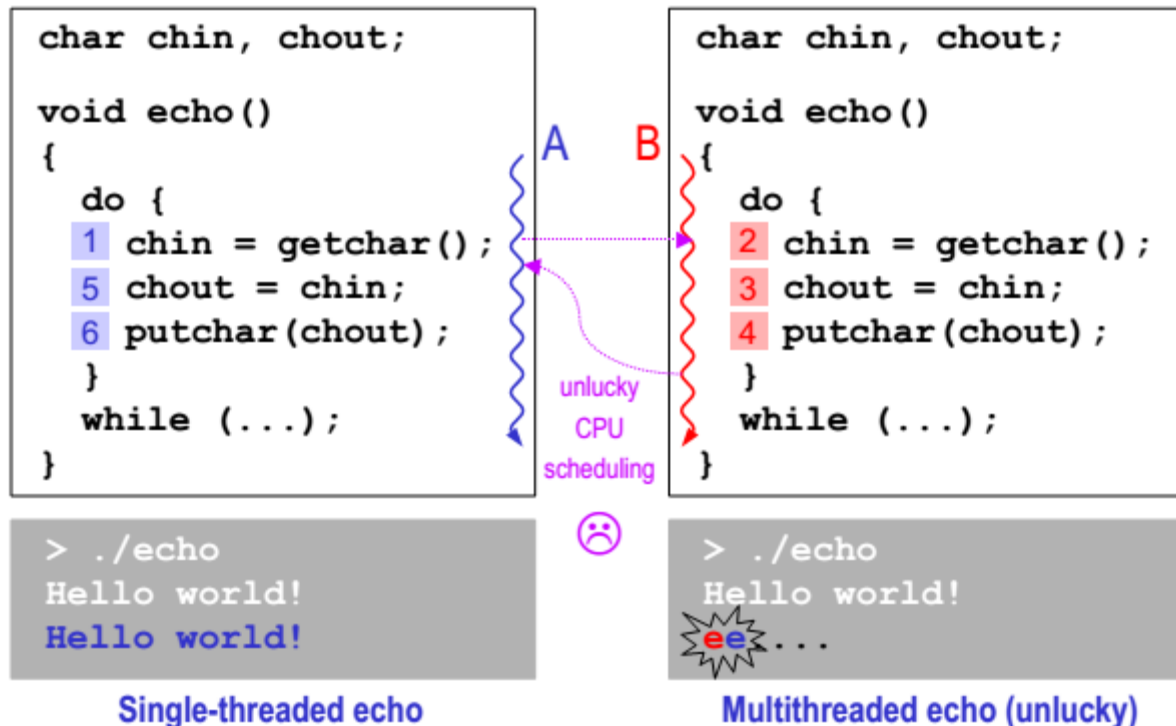
- ✓ the CPU switches from one process/thread to another, possibly on the basis of a preemptive clock mechanism



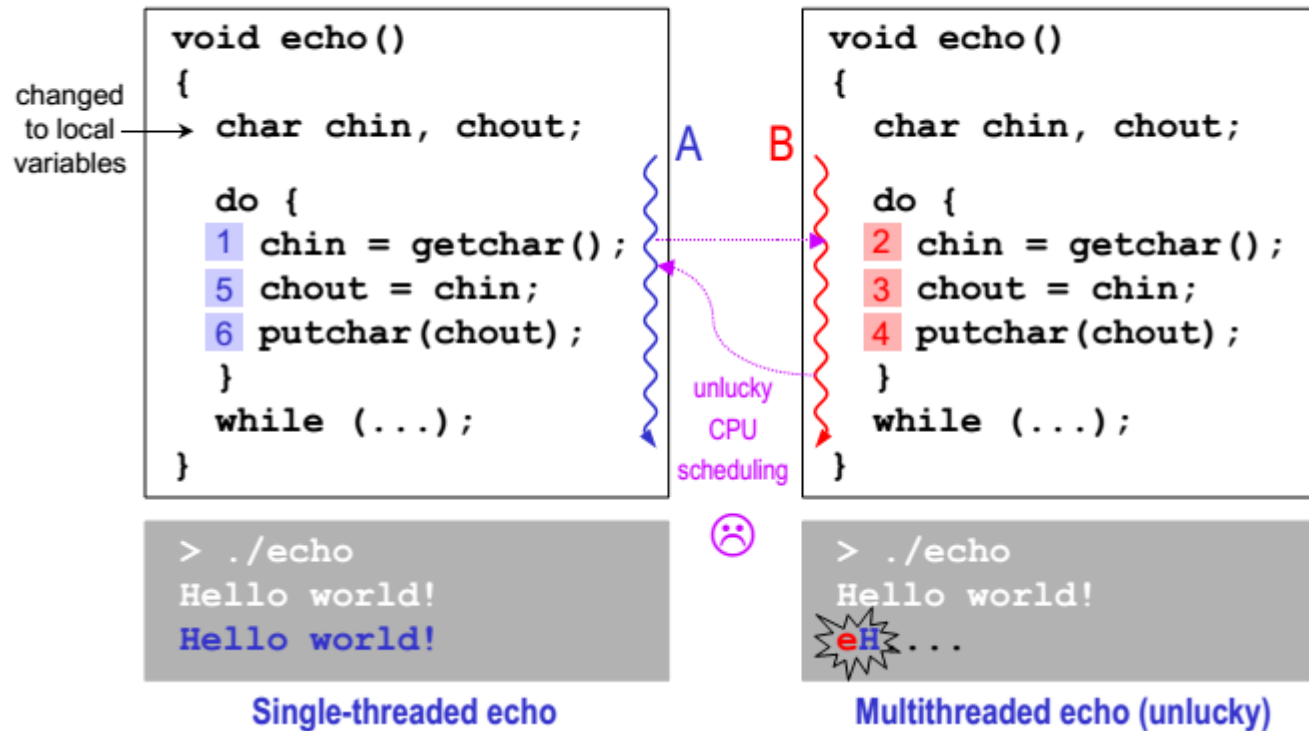
➤ Consequential race conditions in I/O & variable sharing



➤ Consequential race conditions in I/O & variable sharing



## ➤ Consequential race conditions in I/O & variable sharing



## ➤ Consequential race conditions in I/O & variable sharing

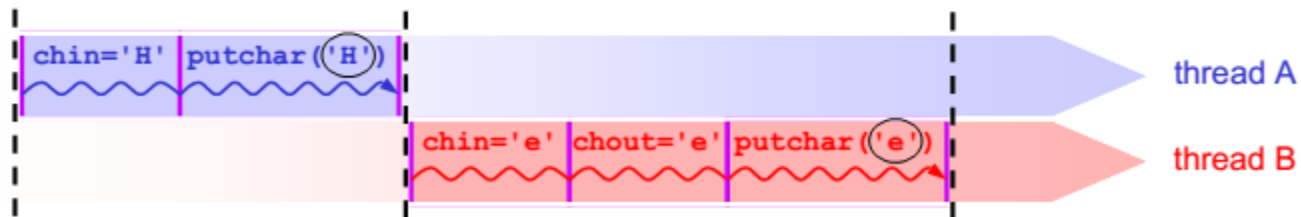
- ✓ note that, in this case, replacing the global variables with local variables did not solve the problem
  - ✓ we actually had two race conditions here:
    - one race condition over assigning values to shared variables
    - another race condition over which thread is going to write to output first; this one 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)*

## ➤ How to avoid race conditions?

- ✓ find a way to keep the instructions together
- ✓ this means actually reverting from too much interleaving and going back to “indivisible” blocks of execution!



(a) too much interleaving may create race conditions



(b) keeping “indivisible” blocks of execution avoids race conditions

➤ 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)*



➤ We need mutual exclusion from critical regions

- ✓ critical regions can be protected from concurrent access by padding them with entrance and exit mechanisms (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

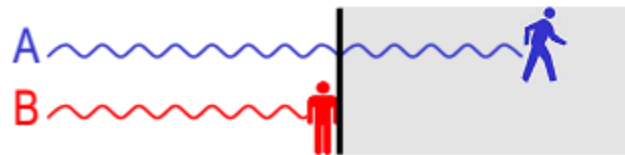
➤ **Desired effect: mutual exclusion from the critical region**

HOW is this  
achieved??

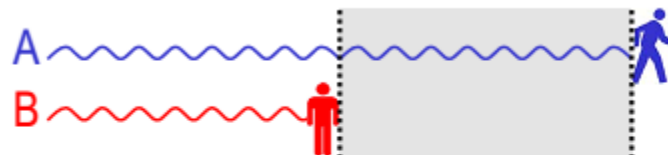
1. thread A reaches the gate to the critical region (CR) before B



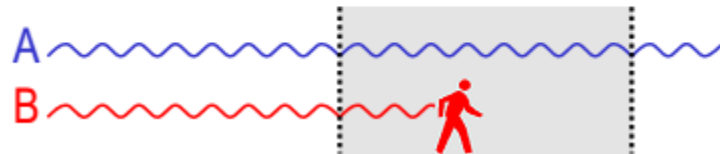
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)



3. thread A exits CR; thread B can now enter



4. thread B enters CR



➤ **Implementation 0 — disabling hardware interrupts**

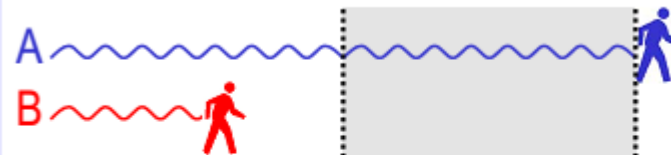
1. thread A reaches the gate to the critical region (CR) before B



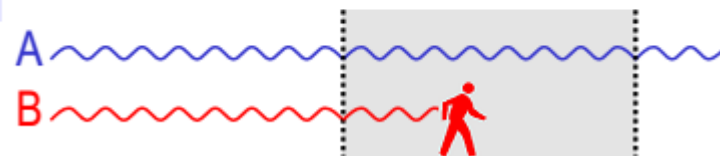
2. as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled



3. as soon as A exits CR, it reenables interrupts; B can be scheduled again

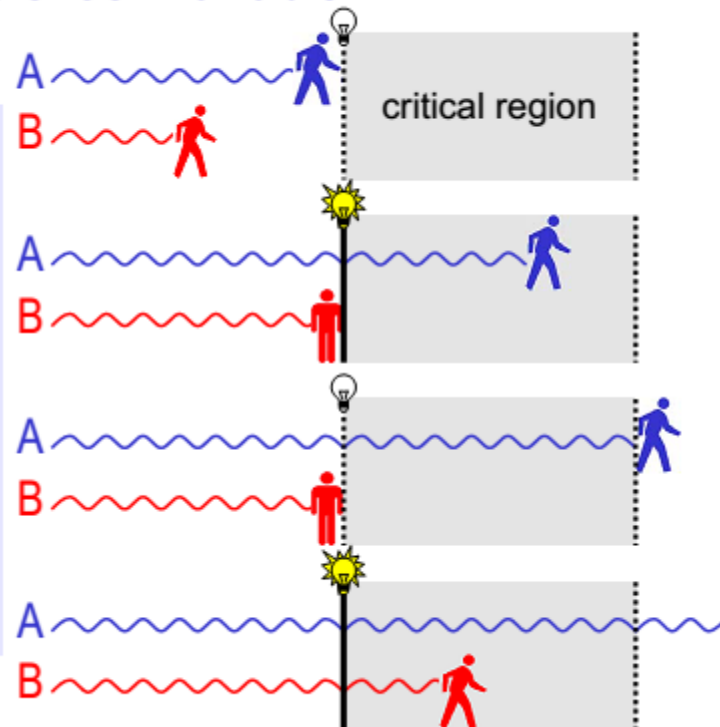


4. thread B enters CR



## ➤ Implementation 1 — 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



## ➤ Implementation 1 — 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 (...);  
}
```

## ➤ Implementation 1 — ~~simple lock variable~~ 🦋

- ✓ suffers from the very fatal 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 inbetween 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 (...);
}
```

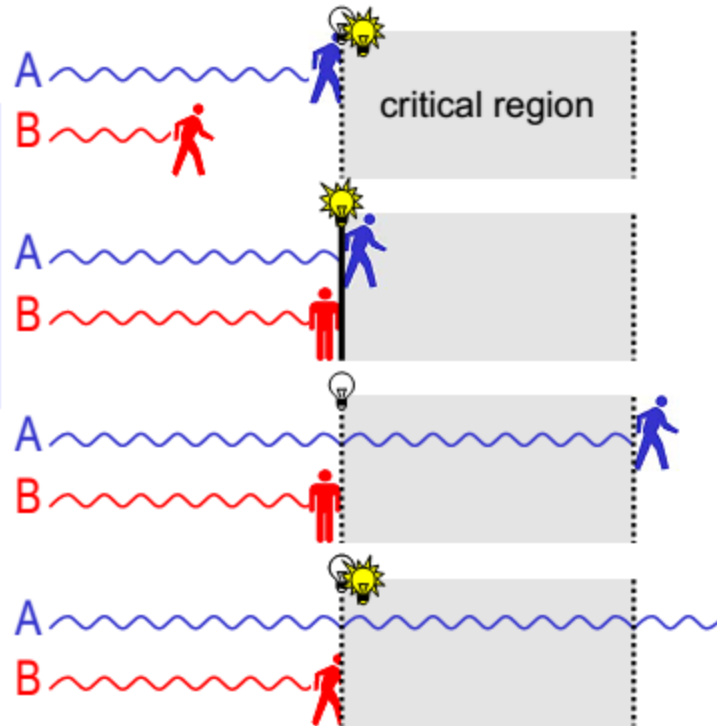
➤ **Implementation 2 — “indivisible” lock variable** 👍

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



## ➤ Implementation 2 — “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 (...);
}
```



➤ **Implementation 2 — “indivisible” lock  $\Leftrightarrow$  one key** 👍

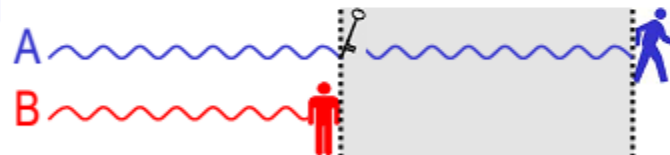
1. thread A reaches CR and finds a key and takes it



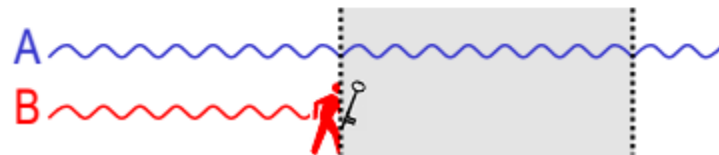
- 1.1' even if B comes right behind A, it will not find a key



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



## ➤ Implementation 2 — “indivisible” lock $\Leftrightarrow$ 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 (...);  
}
```

# Test and Lock Instruction (TSL)

- ❑ Many computers have the following type of instruction: **TSL REGISTER, LOCK**
  - Reads **LOCK** into register **REGISTER**
  - Stores a nonzero value at the memory location **LOCK**
  - The operations of reading the word and storing it are guaranteed to be indivisible
  - The CPU executing the **TSL** instruction locks the memory bus to prohibit other CPUs from accessing memory until the **TSL** instruction is done
  - **LOCK** is a shared variable

# Using the TSL Operation

- ❑ Before entering its critical region, a process calls `enter_region`
- ❑ What if `LOCK` is 1?
  - Busy wait until lock is 0
- ❑ When leaving the critical section, a process calls `leave_region`

# Using the TSL Operation

- ❑ Assume two processes:  $P_0$  and  $P_1$
- ❑ LOCK is initialized to zero
- ❑ Assume that  $P_0$  wants to enter the critical section
- ❑ It executes the TSL instruction.
  - The register value is 0 which reflects the value of LOCK
  - LOCK is set to 1

# Using the TSL Operation

- ❑ Now  $P_1$  wants to enter the critical section; It executes the TSL instruction
  - The register value is 1 which reflects the value of LOCK
  - $P_1$  cannot enter the critical section
  - It repeats the TSL instruction and comparison operation until it can get into the critical section
- ❑  $P_0$  is done with the critical section
  - LOCK becomes 0
- ❑ The next time  $P_1$  executes the TSL instruction and comparison operation it finds that the register value (which reflects LOCK) is zero. It can now enter the critical section.

# Note

- The assembly code for  $x = x + 1$  may look something like this:

ld r1,x

add r1,1

st r1,x

- An interrupt can occur between any of the assembly code language instructions

# Mars Pathfinder

- ❑ Considered flawless in the days after its July 4<sup>th</sup>, 1997 landing on the Martian surface
- ❑ After a few days the system kept resetting causing losses of data
- ❑ Press called it a "software glitch"
- ❑ What was the problem?
  - Answer later on!



# Mars Pathfinder

Report for the  
Seminar Series on Software Failures

Mars Pathfinder: Priority Inversion Problem



Submitted By: Risat Mahmud Pathan

# Mars Pathfinder

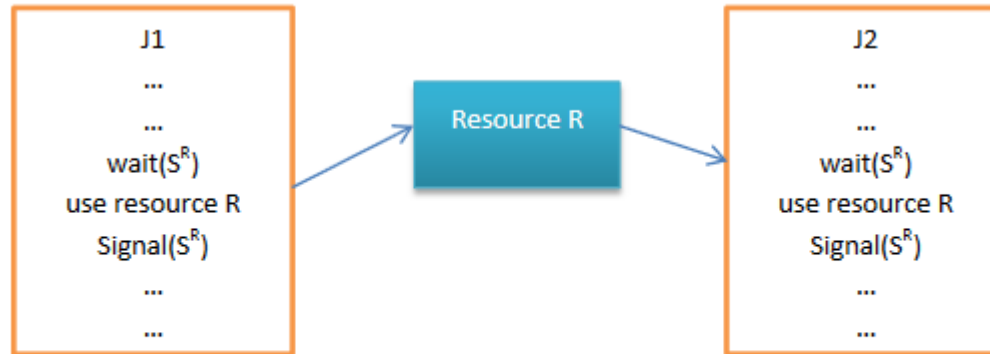
This report presents the software failure known as "priority inversion problem" that occurred in Mars Pathfinder which was designed and built at the Jet Propulsion Laboratory (JPL), California Institute of Technology. The problem due to priority inversion caused the spacecraft to reset itself several times after it was landed on Mars on July 4, 1997. Resetting

This report is based on these two emails cited in [1, 2] and two magazine articles [3,4]. However, to understand the priority inversion problem which is a synchronization problem in real-time systems, I will also formally present the problem and its solution based on [5].

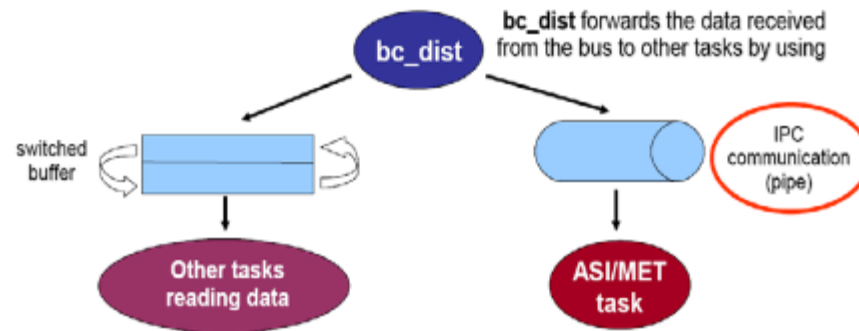
priority active task by preempting, if any, the execution of a lower priority task [7]. In other words, a higher priority task can preempt the execution of a lower priority task and a lower priority task cannot preempt the execution of a higher priority task. A lower-priority task resumes its execution later when there is no higher priority active task awaiting execution.

Tasks generally share resources, for example, data structure, main memories, files, processor registers, I/O units, communication bus, and so on. It is important to enforce effective synchronization mechanisms when multiple tasks share a resource in order to ensure consistent view of the system. One way to ensure such synchronization is using semaphore for each shared resource. A task holding a semaphore has exclusive access to that resource. No task can hold a semaphore if it is currently locked by another task (i.e., at most one task can execute in the critical section).

# Mars Pathfinder



**Figure 1:** The request and release requests of the shared resource R by jobs J1 and J2. Blocking of the higher priority job J1 by the lower priority job J2 can happen depending on when tasks are executed.



**Figure 7:** Data distribution mechanism by bc\_dist task which uses pipe() facility for IPC with the ASI/MET task.

resources protected by mutexes. In other words, the bc\_dist (a higher priority task) and ASI/MET (a lower priority task) both shared the data structure of the pipe(). The following

# Race Condition

- ❑ A **race condition** exists in a program when the result of program execution depends on the precise order of execution of the threads of the process or other processes
- ❑ Since thread/process execution speed varies, data values produced can be inconsistent

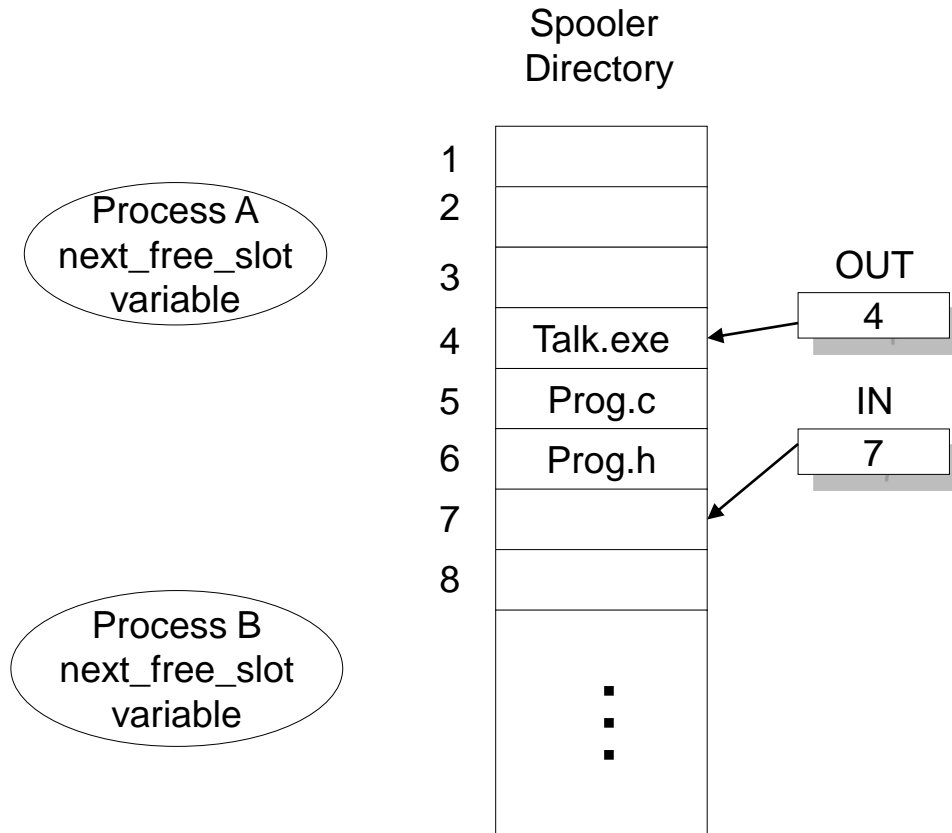
# Race Condition Example (1)

- ❑ Printing a file requires that the file is copied to the spooler directory
  - In Windows the spooler directory is in  
`%SystemRoot%\SYSTEM32\SPOOL\PRINTERS`
- ❑ Print spooler process takes files from a well-defined directory and sends them one-at-a-time to the printer.

# Race Condition Example (1)

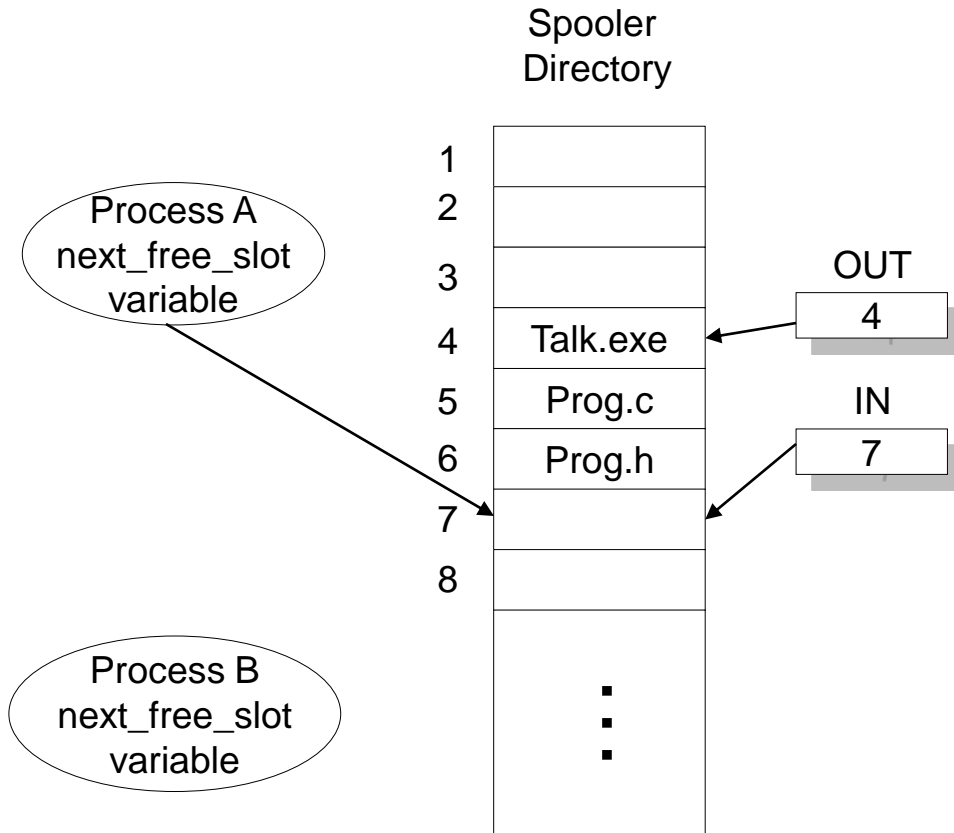
- ❑ Assume a spooler directory array (in shared memory) has a large number of slots
  - Numbered 0, 1, 2
  - Each slot has a file name
- ❑ Two other variables:
  - **In** points to the first empty slot where a new filename can be entered.
  - **Out** points to the first non-empty slot, from where the spooler will read a filename and print the corresponding file.

# Race Condition Example (1)



- ❑ Slots 1,2,3 are empty indicating the files in those slots have printed
- ❑ Each process has a local variable `next_free_slot` representing an empty slot
- ❑ Assume both process A and B want to print files.
  - Each wants to enter the filename into the first empty slot in spooler directory

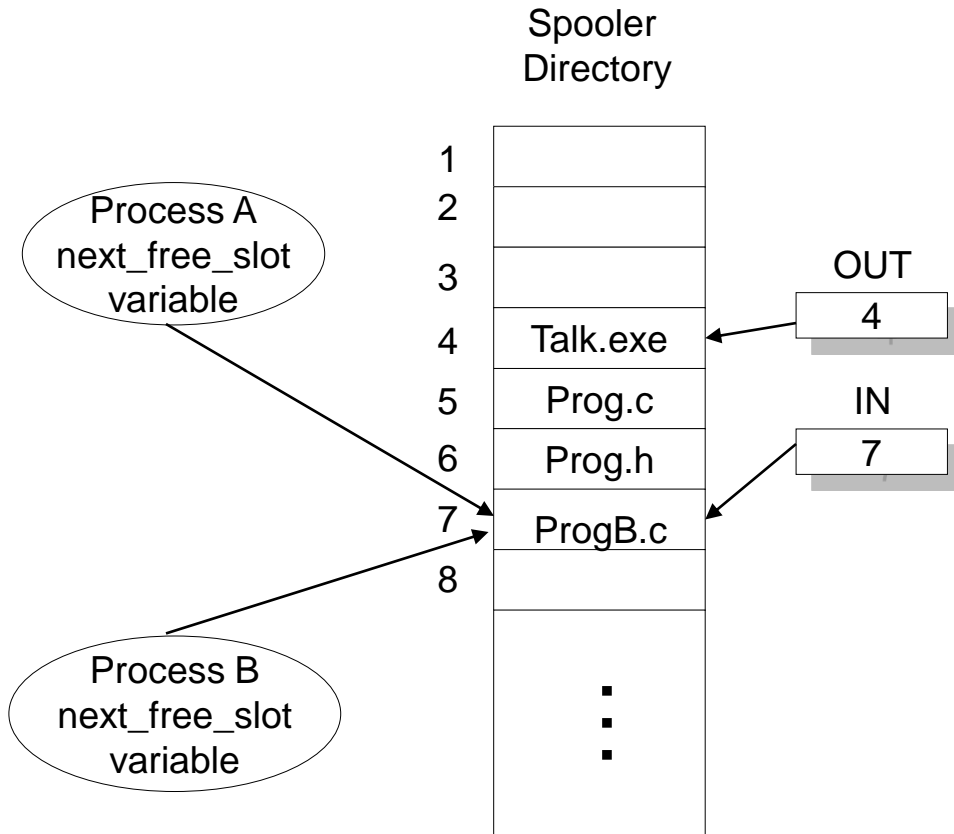
# Race Condition Example (1)



- ❑ Process A reads in and stores the value 7 in its local variable, `next_free_slot`
- ❑ Process A's time quanta expires
- ❑ Process B is picked as the next process to run

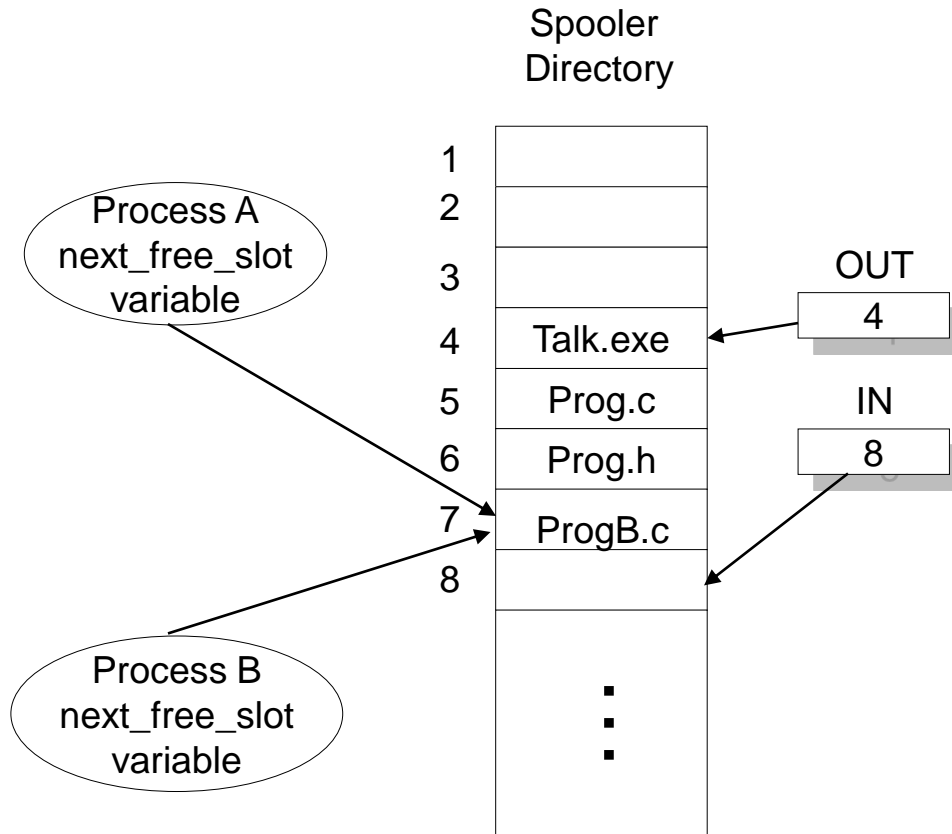


# Race Condition Example (1)



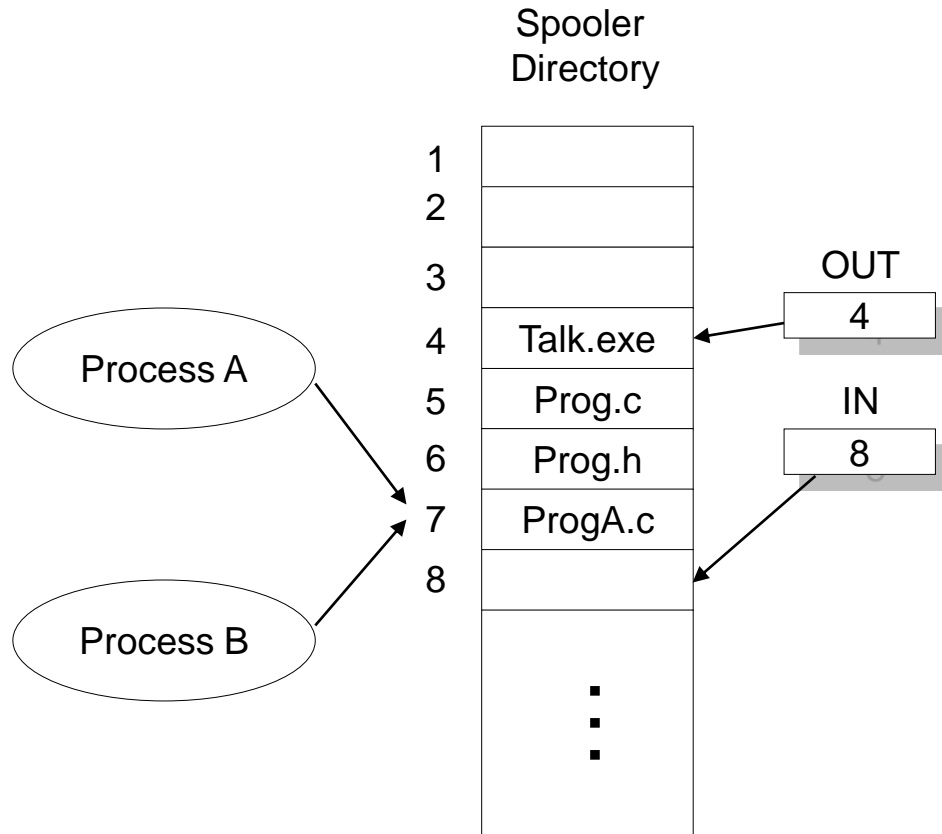
- ❑ Process B reads in and stores the value 7 in its local variable, `next_free_slot`
- ❑ Process B writes a filename to slot 7

# Race Condition Example (1)



- Process B then updates the **In** variable to 8
- Process A now gets control of the CPU

# Race Condition Example (1)



- ❑ Process A writes its filename to slot 7 which erases process B's filename.
- ❑ Process B does not get its file printed.

# Race Condition (2)

- ❑ It is not just the OS that has to deal with race conditions
- ❑ So do application programmers

# Race Condition (2)

- ❑ Application: Withdraw money from a bank account
- ❑ Two requests for withdrawal from the same account comes to a bank from two different ATM machines
- ❑ A thread for each request is created
- ❑ Assume a balance of \$1000

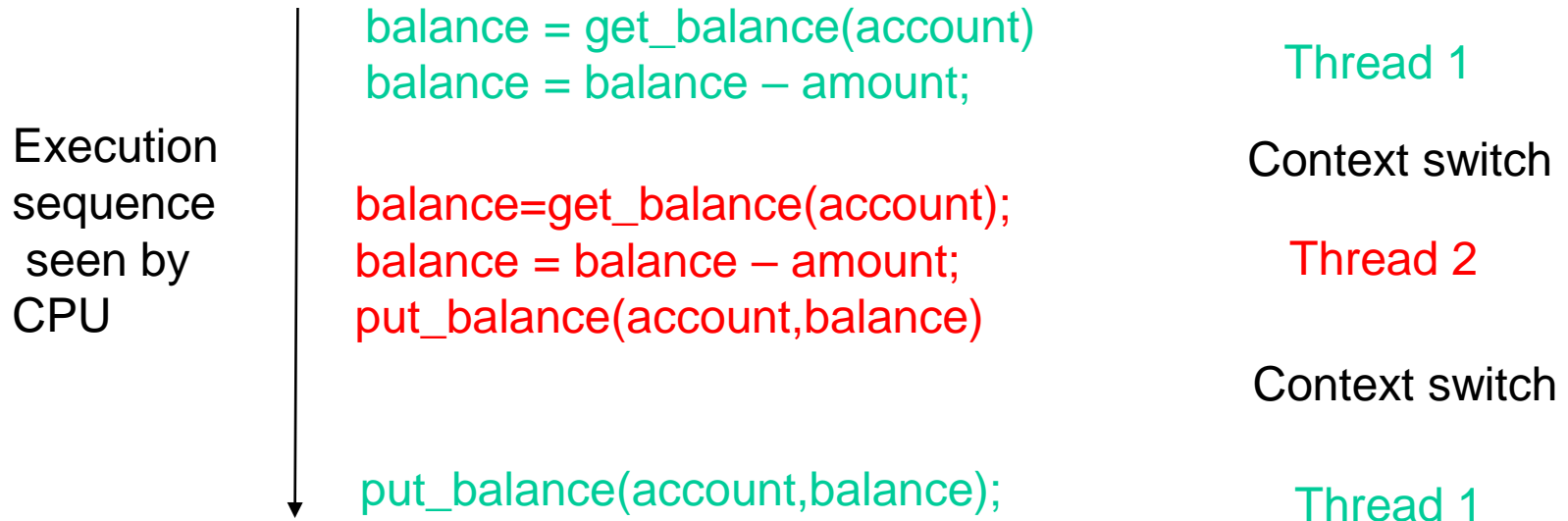
# Race Condition (3)

```
int withdraw(account, amount)
{
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance)
    return balance
}
```

What happens if both requests request that \$1000 be withdrawn?

# Race Condition 2

- ❑ Both threads will read a balance of \$1000
- ❑ Both threads will allow for \$1000 to be withdrawn



# Critical Sections and Mutual Exclusion

- ❑ A **critical section** is any piece of code that accesses shared data
- ❑ **Mutual exclusion** ensures that only one thread/process accesses the shared data



# Conditions for Mutual Exclusion

- ❑ **Mutual Exclusion:** No two threads simultaneously in critical section
- ❑ **Progress:** No thread running outside its critical section may block another thread
- ❑ **Bounded Waiting:** No thread must wait forever to enter its critical section
- ❑ No assumptions made about speeds or number of CPUs

# Mutual Exclusion - solutions

- ❑ Supervisory server.
- ❑ CSMA/CD Binary Exponential Back-off.
- ❑ Peterson's solution.
- ❑ Strict Alternation.
- ❑ Disabling Interrupts.
- ❑ Spinning.
- ❑ TSL.
- ❑ Semaphores.

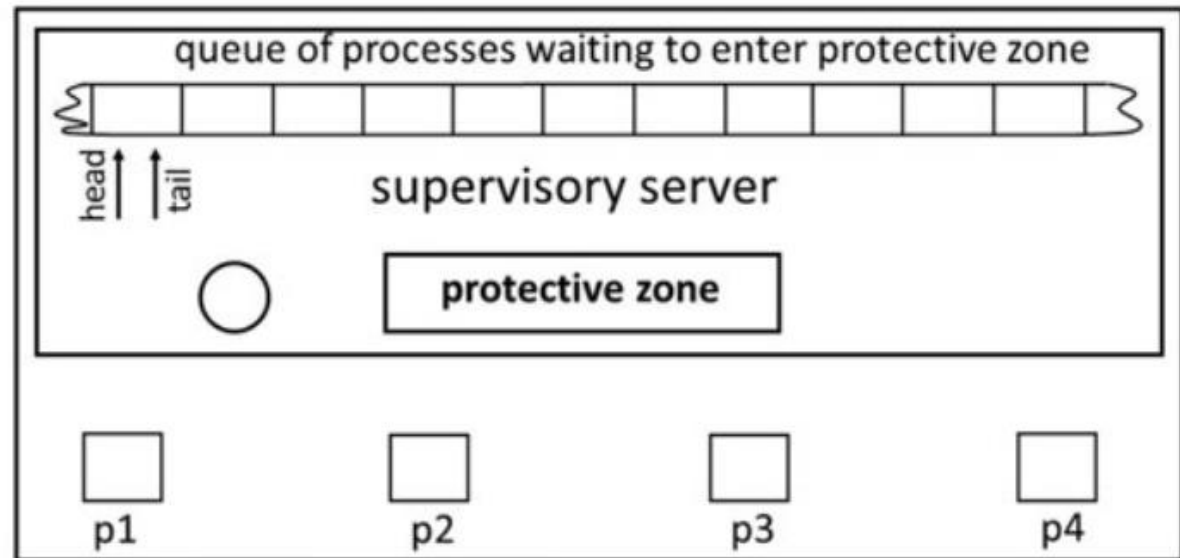
# Mutual Exclusion by Supervisory Server

- ❑ **Requests:** Supervisory server receives requests for protected resources from user processes and make decision if recourses can be accessed.
- ❑ **Permission:** If yes, it sends a permission message to the requesting process, otherwise sends a **refusal** message and puts the process in the queue of waiting processes.
- ❑ **Release:** On completion of executing of protective region, the process sends a release message to the server. The server removes a waiting process from the head of the queue and reactivates it.

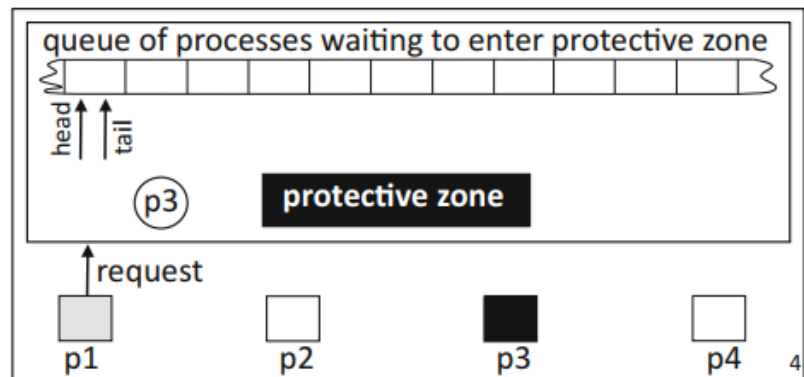
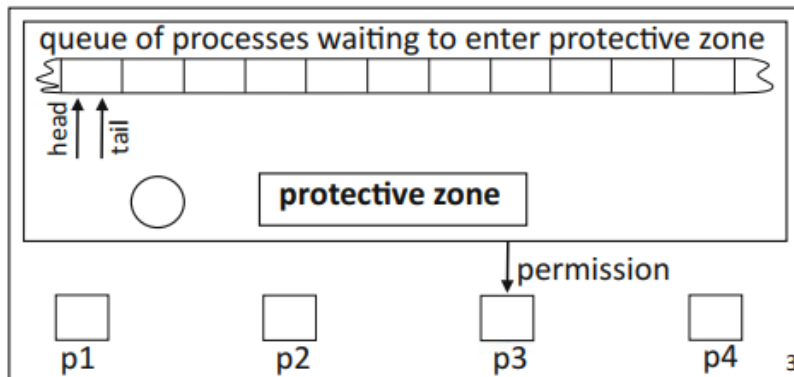
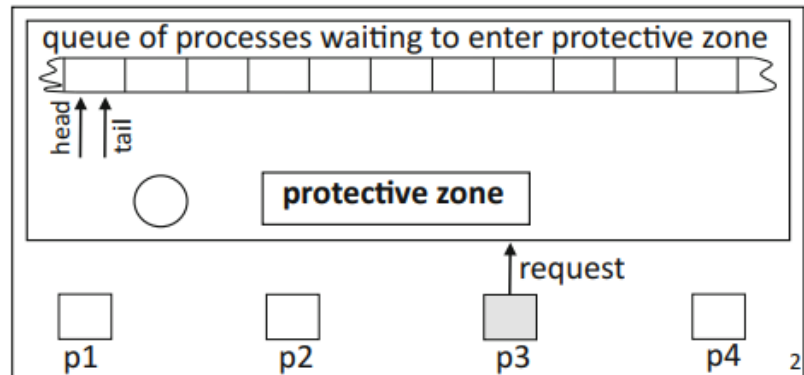
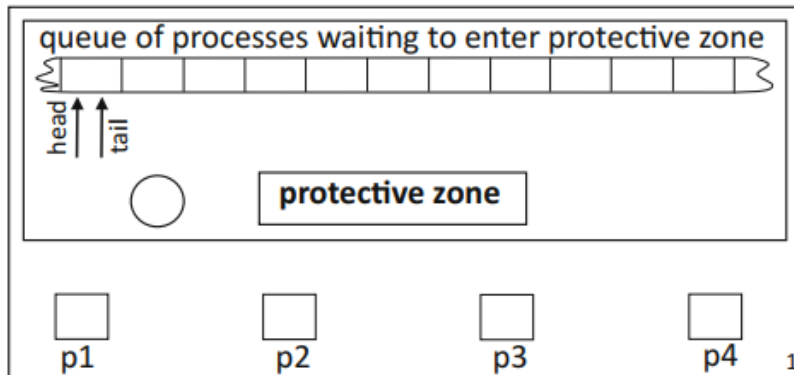
# Mutual Exclusion by Supervisory Server

state of process:

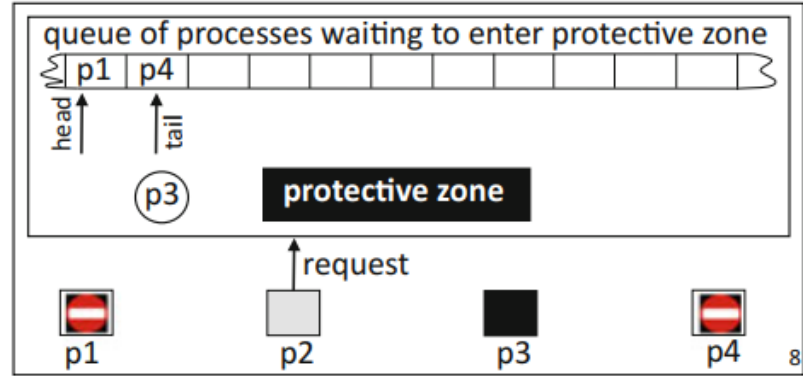
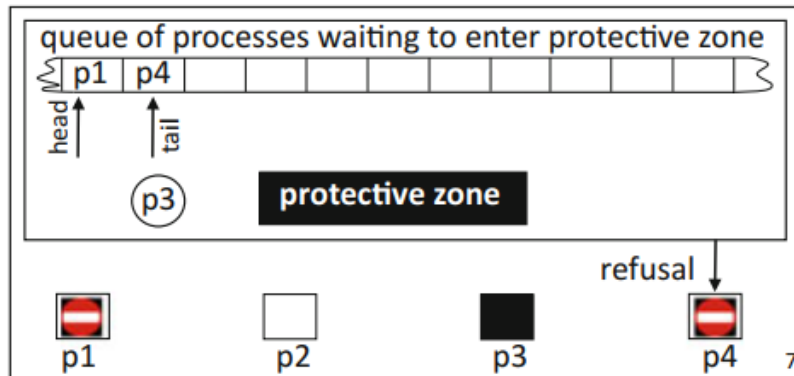
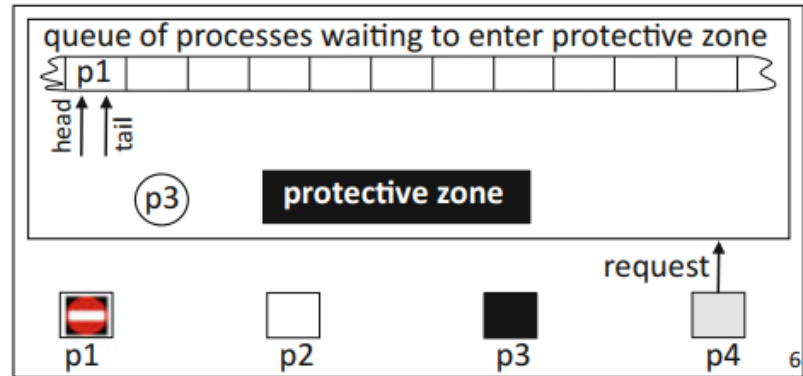
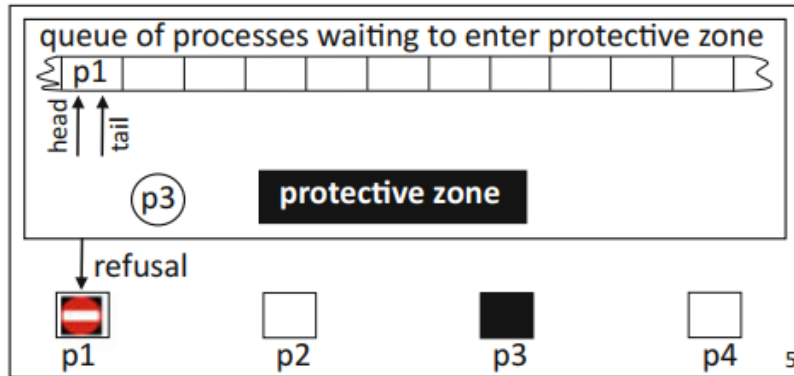
-  execution of local section
-  request for executing protective zone
-  permission
-  refusal
-  execution of protective zone
-  release of protective zone



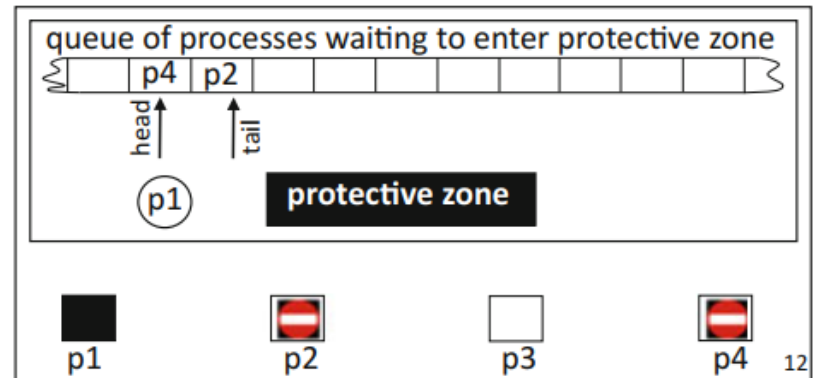
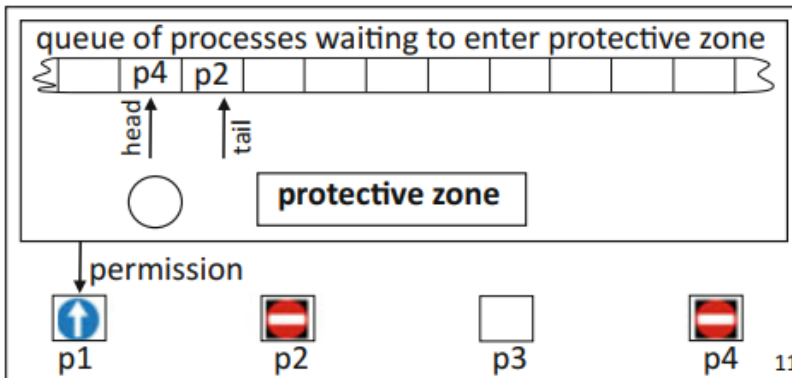
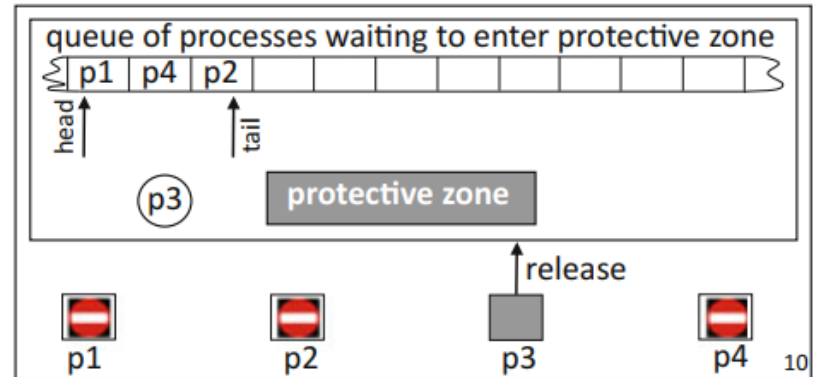
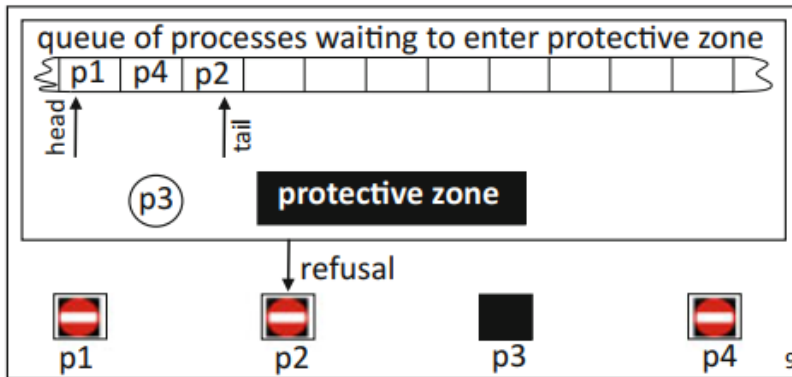
# Mutual Exclusion by Supervisory Server



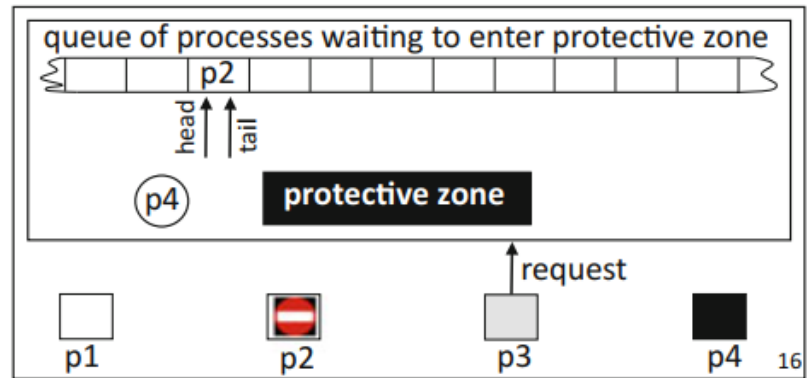
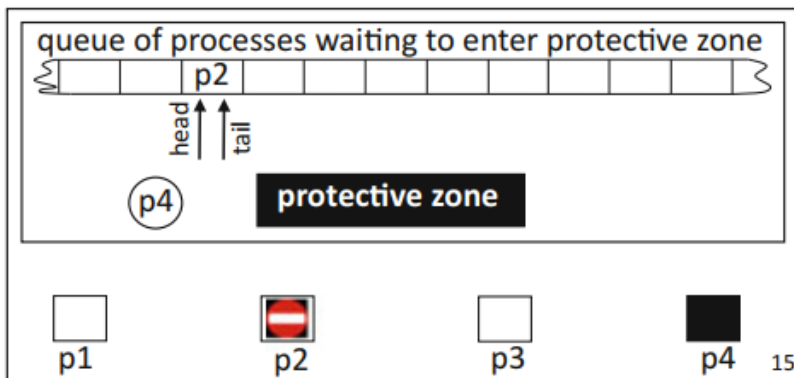
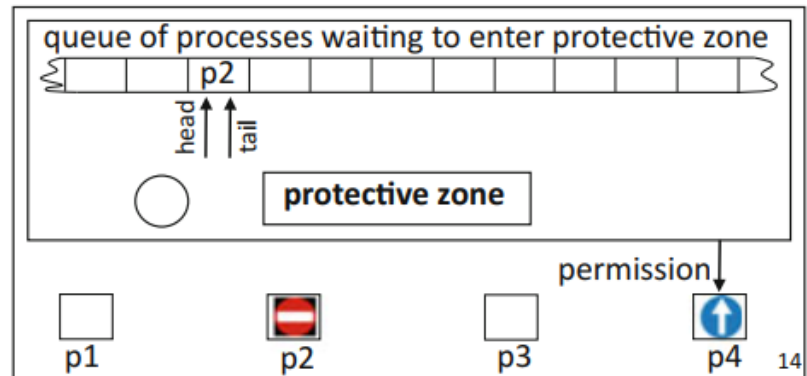
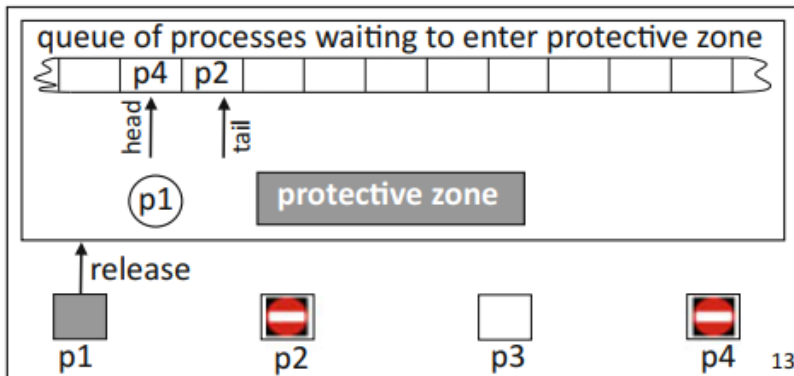
# Mutual Exclusion by Supervisory Server



# Mutual Exclusion by Supervisory Server

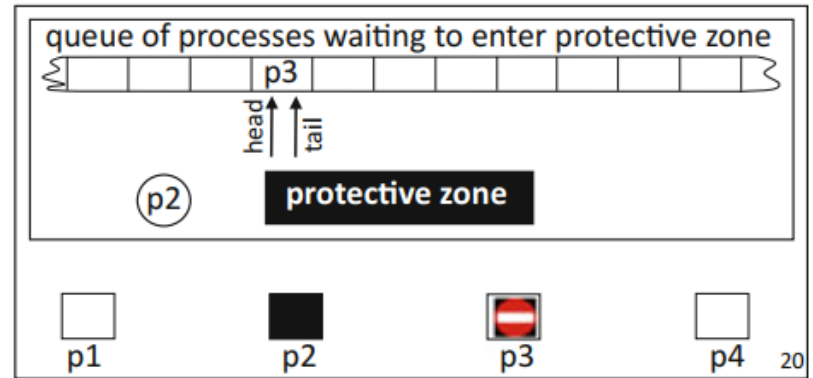
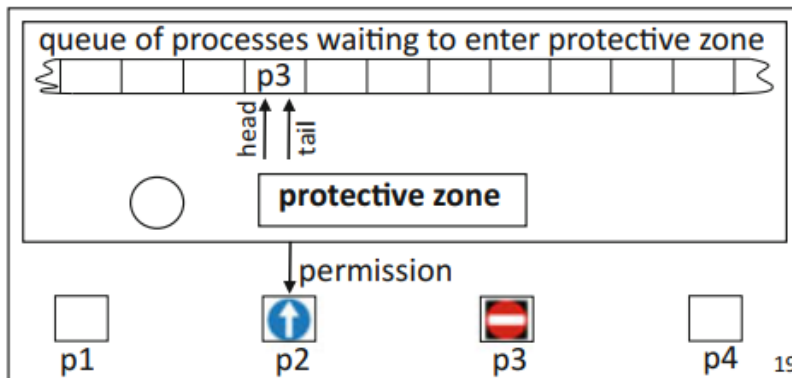
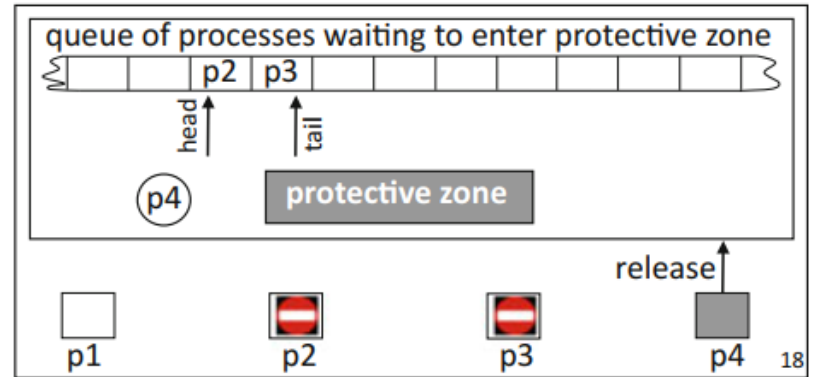
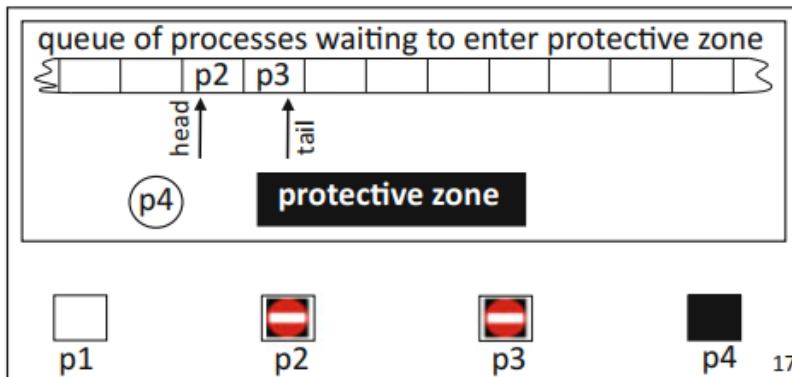


# Mutual Exclusion by Supervisory Server

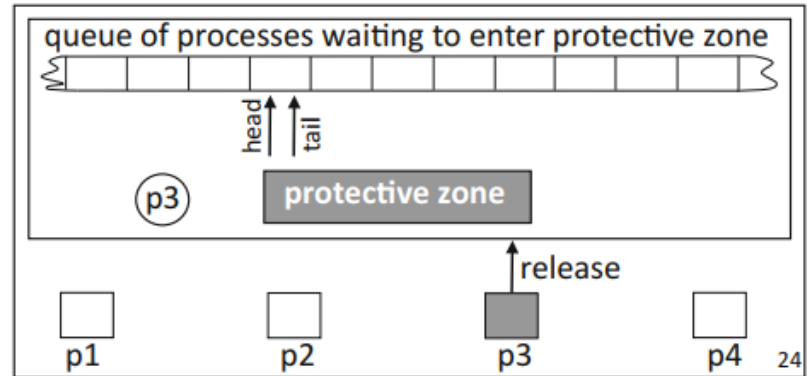
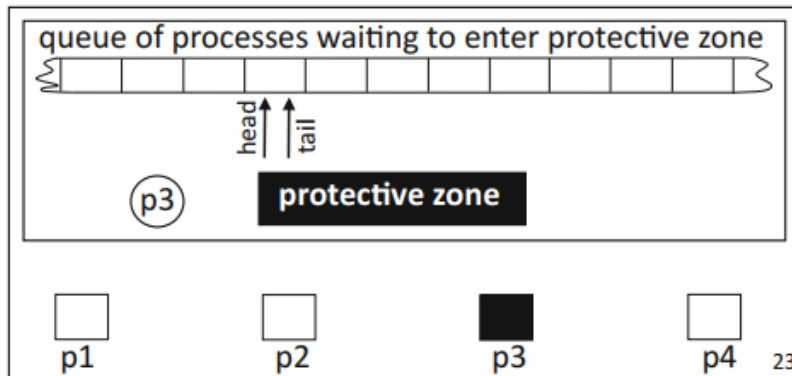
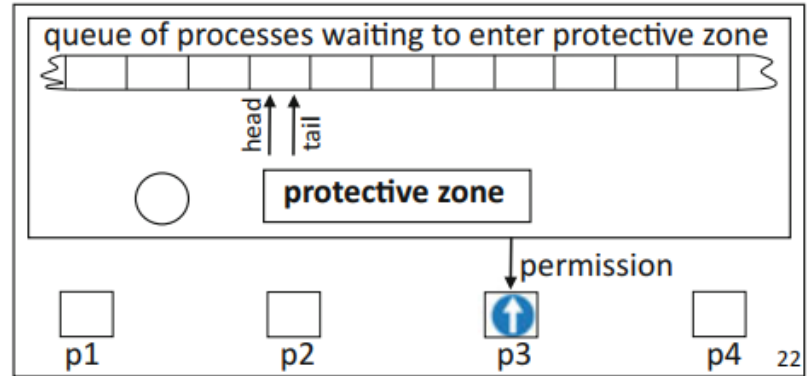
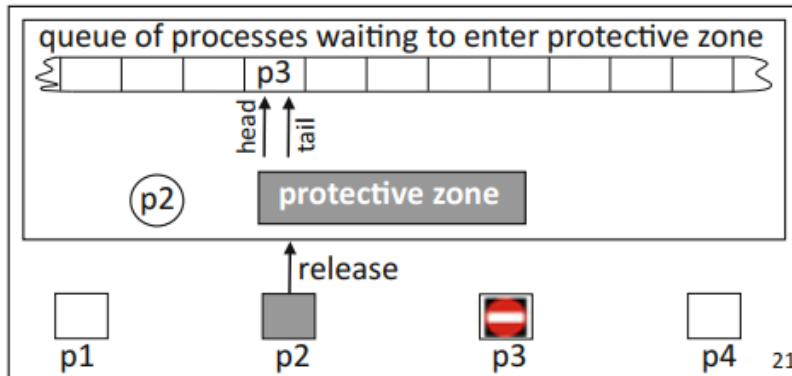




# Mutual Exclusion by Supervisory Server



# Mutual Exclusion by Supervisory Server



# Multiple Access Protocols-MACs

- ❑ Single shared broadcast channel
- ❑ Two or more simultaneous transmission by nodes
  - **Collision** if node receives two or more signals at the same time. Data is lost.
- ❑ Channel Partitioning: Divide channel in small pieces (time slots, frequency, code)
- ❑ **Random Access**: Channel not divided, allow collisions. Ex.: Aloha, CSMA, CSMA/CD.
- ❑ Taking Turns: Polling, token passing.

# CSMA/CD

## Carrier Sense Multiple Access

### ❑ Carrier sense:

- Listen before speaking, and don't interrupt.
- Checking if someone else is already sending data and wait until other nodes are done.

### ❑ Collision Detection (CD):

- If someone else starts talking at the same time, stop.
- Determines when two nodes are transmitting at the same time by detecting that received data is electronically unreadable.

### ❑ Randomness:

- Don't start talking again right away.
- Waiting for a random time before trying again.

# CSMA/CD

## Carrier Sense Multiple Access

- ❑ CSMA: Listen before transmit.
  - If channel sensed idle: transmit entire frame.
  - If channel sensed busy: defer transmission.
- ❑ Human analogy:
  - Don't interrupt others!.
- ❑ Does this eliminate all collisions?
  - No. because of nonzero propagation delay.
  - Two nodes may not hear each other's before sending.

# CSMA/CD

## Carrier Sense Multiple Access

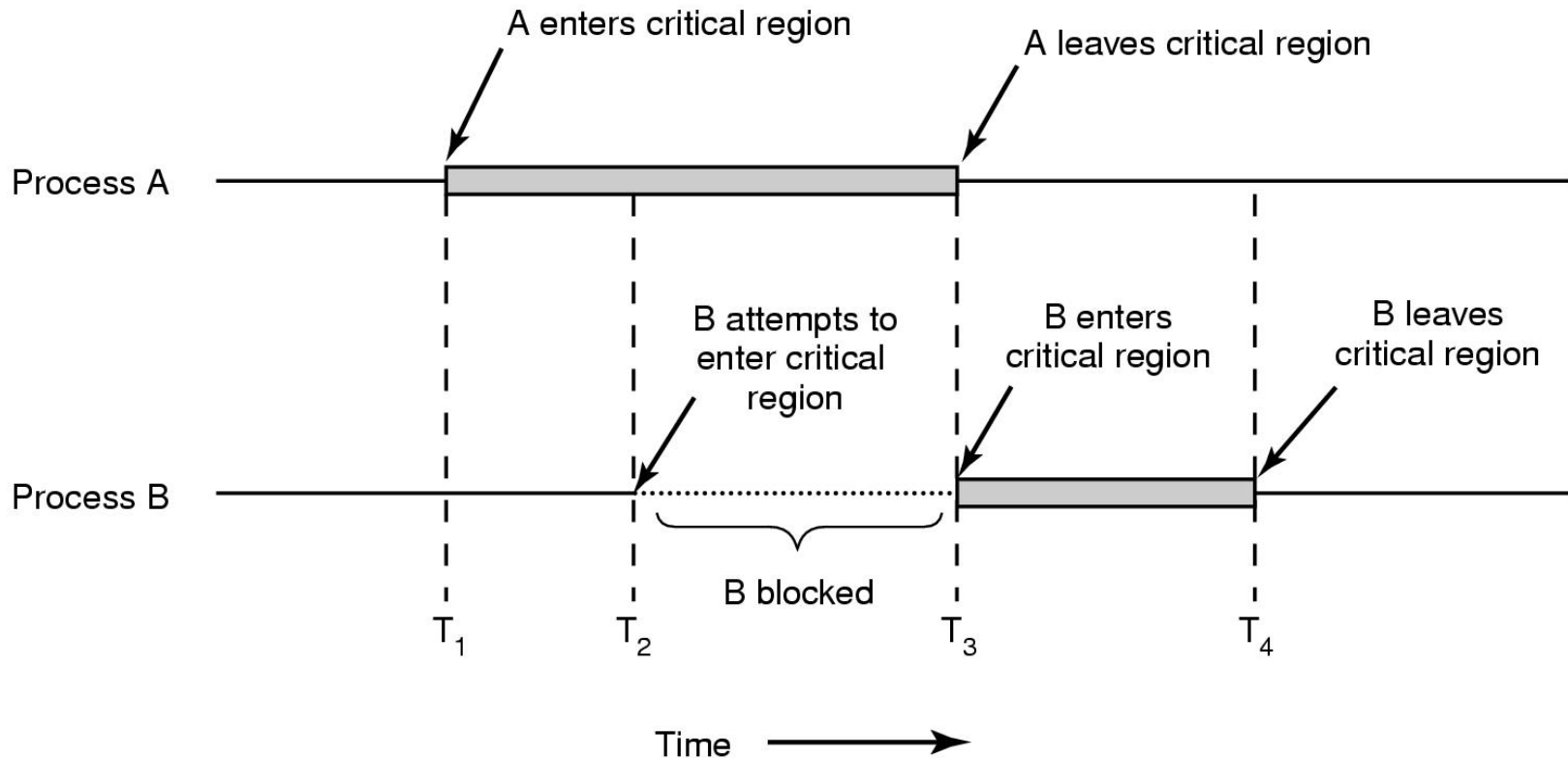
- ❑ CSMA: Listen before transmit.
  - If channel sensed idle: transmit entire frame.
  - If channel sensed busy: defer transmission.
- ❑ Human analogy:
  - Don't interrupt others!.
- ❑ Does this eliminate all collisions?
  - No. because of nonzero propagation delay.
  - Two nodes may not hear each other's before sending.

# CSMA/CD

## Carrier Sense Multiple Access

- ❑ Carrier sense: Wait for the link to be idle.
- ❑ Collision detection: Listen while transmitting.
  - No collision: transmission is complete.
  - Collision: abort transmission and send jam signal.
- ❑ Randomness: **Binary exponential back-off**.
  - After the  $m^{\text{th}}$  collision, chooses a  $K$  at random from  $\{0, \dots, 2^m - 1\}$ ,  $m$  is collision number. Don't start talking again right away.
  - Waiting for a random time before trying again.
    - Waiting =  $K \times \text{RTT}$  (round trip time)

# Mutual Exclusion in Critical Sections





# Peterson's Solution

- ❑ Solution developed in 1981
- ❑ Considered revolutionary at the time
- ❑ Restricted to two processes
- ❑ Assumes that the LOAD and STORE instructions are atomic i.e., cannot be interrupted (although this is no longer true for modern processors)

# Peterson's Solution

- ❑ Each process has its own copy of variables
- ❑ Two variables are shared
  - `int turn;`
  - `int flag[2]`
- ❑ The variable `turn` indicates whose turn it is to enter the critical section.

# Peterson's Solution

- The **flag** array is used to indicate if a process is ready to enter the critical section.
  - **flag[0] = true** implies that process  $P_0$  is ready!
  - **flag[0] = false** implies that process  $P_0$  is not ready!
  - **flag[1] = true** implies that process  $P_1$  is ready!
  - **flag[1] = false** implies that process  $P_1$  is not ready!
  - All values initialized to false (0)

# Peterson's Solution

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
    CRITICAL SECTION  
    flag[0] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

**Code for  $P_0$**

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
    CRITICAL SECTION  
    flag[1] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

**Code for  $P_1$**

- ❑ Only way  $P_0$  enters critical section is when  $\text{flag}[1] == \text{FALSE}$  or  $\text{turn} == 0$
- ❑ What if  $P_0$  and  $P_1$  are in their critical sections at the same time
  - ❑ This implies that  $\text{flag}[0] == \text{flag}[1] == \text{TRUE}$
  - ❑ If  $\text{turn} = 0$  then  $P_1$  cannot break out of its while loop
  - ❑ If  $\text{turn} = 1$  then  $P_0$  cannot break out of its while loop
  - ❑ This implies that  $P_0$  and  $P_1$  could not be in their critical sections at the same time

# Peterson's Solution

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
    CRITICAL SECTION  
    flag[0] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

**Code for  $P_0$**

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
    CRITICAL SECTION  
    flag[1] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

**Code for  $P_1$**

Can  $P_0$  block  $P_1$  even if  $P_0$  is not in its critical section?

- ❑ No.
- ❑ When  $P_0$  finishes with its critical section it sets `flag[0]` to `FALSE`
- ❑ `flag[0]` being `FALSE` allows  $P_1$  to break out of its while loop

# Peterson's Solution

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
    CRITICAL SECTION  
    flag[0] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

Code for  $P_0$

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
    CRITICAL SECTION  
    flag[1] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);
```

Code for  $P_1$

- ❑ Initially  $\text{flag}[0] = \text{flag}[1] = \text{FALSE}$
- ❑ Let's say that  $P_0$  wants to enter the critical section (CS)
  - Assignments:  $\text{flag}[0] = \text{TRUE}$ ,  $\text{turn} = 1$
  - The condition  $\text{flag}[1] \&\& \text{turn} == 1$  is evaluated
    - $\text{turn}$  is 1 but  $\text{flag}[1]$  is FALSE; Thus, the condition evaluates to false which allows  $P_0$  to enter the critical section

# Peterson's Solution

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
    CRITICAL SECTION  
    flag[0] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);  
Code for P0
```

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
    CRITICAL SECTION  
    flag[1] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);  
Code for P1
```

- Let's say that P<sub>1</sub> wants to enter the critical section while P<sub>0</sub> is in the critical section.
  - Assignments: `flag[1] = TRUE, turn = 0`
  - The condition `flag[0] && turn == 0` is evaluated
    - `turn` is 0 but `flag[0]` is TRUE; Thus, the condition evaluates to true; P<sub>1</sub> enters the while loop. It is not allowed in the critical section

# Peterson's Solution

```
do {  
    flag[0] = TRUE;  
    turn = 1;  
    while ( flag[1] && turn == 1);  
    CRITICAL SECTION  
    flag[0] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);  
Code for P0
```

```
do {  
    flag[1] = TRUE;  
    turn = 0;  
    while ( flag[0] && turn == 0);  
    CRITICAL SECTION  
    flag[1] = FALSE;  
    REMAINDER SECTION  
} while (TRUE);  
Code for P1
```

- ❑ Let's say that P<sub>0</sub> leaves the critical section
  - Sets `flag[0] = FALSE`
- ❑ The next time P<sub>1</sub> evaluates `flag[0] && turn == 0`
  - It finds that `flag[0]` is FALSE which means that the while loop's condition is false
  - Now P<sub>1</sub> can enter the critical section




# Peterson's Solution

- ❑ Peterson's solution is not guaranteed to work on modern computer architectures
- ❑ Modern CPUs reorder accesses to improve execution efficiency

# Peterson's Solution - code reorder

- Example:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```




A red curved arrow points from the `lw $t2, 4($t1)` instruction to the `sw $t2, 0($t1)` instruction, indicating a data hazard where the second instruction writes to a register used by the third instruction before it has been updated.

Data hazard

- Reordered code:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

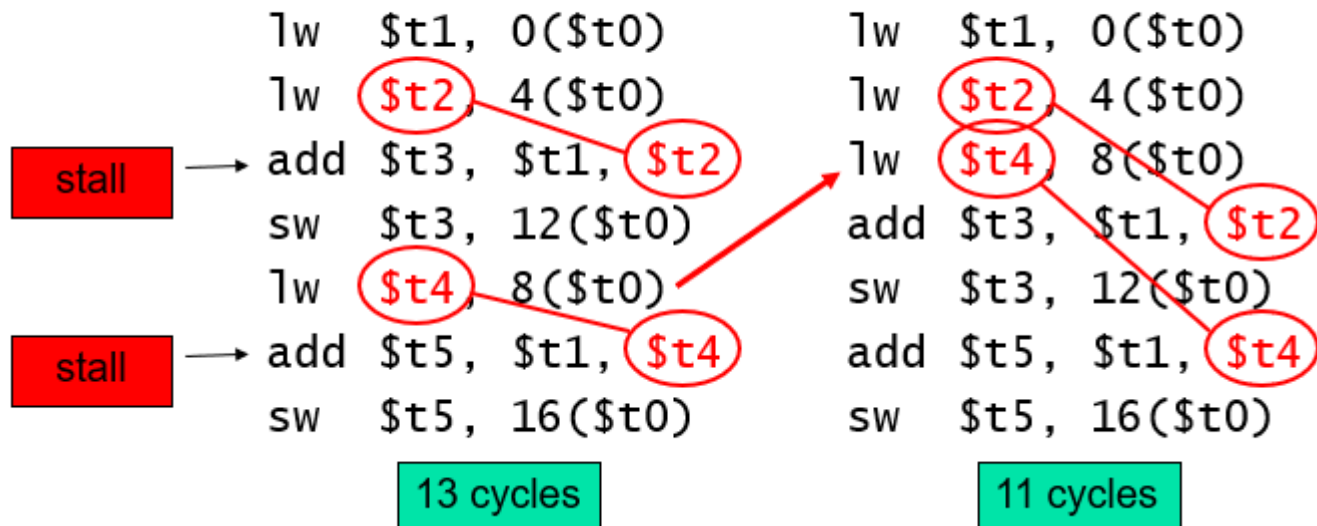


A red curved arrow points from the `sw $t0, 4($t1)` instruction to the `sw $t2, 0($t1)` instruction, indicating that these two instructions have been reordered to eliminate the data hazard.

Interchanged

# Peterson's Solution -code reorder

- C code for  $A = B + E$ ;  $C = B + F$ ;



# Problems

- ❑ Peterson's algorithm and TSL may seem to observe the four conditions for mutual exclusion.
- ❑ Unfortunately, they have other problems.
  - **Busy waiting**: Much CPU time is wasted as processes are looping awaiting admission to their critical regions.
  - **Priority inversion**:
    1. Two processes: H and L; H with high priority, L with low priority.
    2. H should run whenever it is ready.
    3. L is in its critical region, when H becomes ready.
    4. H is scheduled but begins busy waiting.
    5. L can never leave its critical region since it is not allowed to run.
    6. H busy waits forever and L stays ready forever.
    7. This violates condition mutual exclusion.

# Priority Inversion

- ❑ Assume three tasks with priorities  $L < M < H$
- ❑ Assume also that both  $L$  and  $H$  share the same Critical Section  $CS$ .  $M$  does not share  $CS$ .
- ❑ Assume the processing sequence:
  - $L$  is running in  $CS$
  - $H$  also needs to run in  $CS$ ;  $H$  waits for  $L$  to come out of  $CS$
  - $M$  interrupts  $L$  and starts running
    - $M$  runs until completion and relinquishes CPU
  - $L$  resumes and starts running until the end of  $CS$
  - $H$  enters the  $CS$  and starts executing

# Strict Alternation Solution

```
while( true ) {  
    while ( turn != 0 );  
    CRITICAL SECTION  
    turn = 1;  
    NONCRITICAL SECTION  
}
```

Code for  $P_0$

```
While( true ) {  
    while ( turn != 1 );  
    CRITICAL SECTION  
    turn = 0;  
    NONCRITICAL SECTION  
}
```

Code for  $P_1$

- ❑ Assume  $\text{turn} = 0$  initially.
- ❑ Are there any problems with this solution?

# Synchronization Hardware

- ❑ Any solution to the critical section problem requires a **lock**
- ❑ Race conditions are prevented by requiring that critical sections be protected by locks
  - Process must acquire a lock before entering a critical section
  - Process must release the lock when it exists the critical section.
- ❑ We will present several hardware solutions

# Mutual Exclusion via Disabling Interrupts

- ❑ Process disables all interrupts before entering its critical region
- ❑ Enables all interrupts just before leaving its critical region
- ❑ CPU is switched from one process to another only via clock or other interrupts
- ❑ So, disabling interrupts guarantees that there will be no process switch



# Mutual Exclusion via Disabling Interrupts

## ❑ Disadvantage:

- Gives the power to control interrupts to user (what if a user turns off the interrupts and never turns them on again?)
- Does not work in the case of multiple CPUs. Only the CPU that executes the disable instruction is affected.

## ❑ Not suitable approach for the general case but can be used by the kernel when needed

# Test and Lock Instruction (TSL)

- ❑ Many computers have the following type of instruction: **TSL REGISTER, LOCK**
  - Reads **LOCK** into register **REGISTER**
  - Stores a nonzero value at the memory location **LOCK**
  - The operations of reading the word and storing it are guaranteed to be indivisible
  - The CPU executing the **TSL** instruction locks the memory bus to prohibit other CPUs from accessing memory until the **TSL** instruction is done

# Using the TSL Instruction

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

# Using the TSL Operation

- ❑ Before entering its critical region, a process calls `enter_region`
- ❑ What if `LOCK` is 1?
  - Busy wait until lock is 0
- ❑ When leaving the critical section, a process calls `leave_region`

# Using the TSL Operation

- ❑ Assume two processes:  $P_0$  and  $P_1$
- ❑ LOCK is initialized to zero
- ❑ Assume that  $P_0$  wants to enter the critical section
- ❑ It executes the TSL instruction.
  - The register value is 0 which reflects the value of LOCK
  - LOCK is set to 1

# Using the TSL Operation

- ❑ Now  $P_1$  wants to enter the critical section; It executes the TSL instruction
  - The register value is 1 which reflects the value of LOCK
  - $P_1$  cannot enter the critical section
  - It repeats the TSL instruction and comparison operation until it can get into the critical section
- ❑  $P_0$  is done with the critical section
  - LOCK becomes 0
- ❑ The next time  $P_1$  executes the TSL instruction and comparison operation it finds that the register value (which reflects LOCK) is zero. It can now enter the critical section.

# Semaphores

Copyright © 1990-9, Eric A. Schmelz



# Semaphores

- Consider a system that can only support ten users ( $S=10$ ). Whenever a user logs in,  $P$  is called, decrementing the semaphore  $S$  by 1. Whenever a user logs out,  $V$  is called, incrementing  $S$  by 1 representing a login slot that has become available. When  $S$  is 0, any users wishing to log in must wait until  $S > 0$  and the login request is enqueued onto a FIFO queue; mutual exclusion is used to ensure that requests are enqueued in order. Whenever  $S$  becomes greater than 0 (login slots available), a login request is dequeued, and the user owning the request is allowed to log in.



# Semaphores

- ❑ A physical semaphore is a signaling system that uses visual communication.
- ❑ A software semaphore serves the same purpose, albeit using non-visual means.
- ❑ It is used to:
  - ensure mutual exclusion
  - send signals from one process to another.

# What is a semaphore?

- A semaphore is an integer variable with the following three operations.
  1. **Initialize:** You can initialize the semaphore to any non-negative value.
  2. **Decrement:** A process can decrement the semaphore, if its value is positive. If value is 0, the process blocks (gets put into queue). It is said to be sleeping on the semaphore. This is the **down** operation.
  3. **Increment:** If value is 0 and some processes are sleeping on the semaphore, one is unblocked. Otherwise, value is incremented. This is the **up** operation.
- All semaphore operations are implemented as primitive actions (possibly using TSL).
- In Unix based system, unblocking is done using the **signal** system call.

# Using Semaphores

- ❑ Assume two processes:  $P_0$  and  $P_1$
- ❑ Initialize the semaphore variable,  $s$ , to 1
  - Why not zero?
- ❑ Now what would happen if  $P_0$  executes the down operation?
  - The semaphore  $s$  is currently 1.
  - It becomes 0 and  $P_0$  enters the critical section
- ❑ Now what would happen if  $P_1$  executes the down operation?
  - The semaphore  $s$  is currently 0
  - $P_1$  blocks

# Using Semaphores

- ❑ Assume now that  $P_0$  is done with the critical section
- ❑ It calls the **up** function
  - $P_1$  is unblocked
  - If there was no process waiting to enter the critical section, the value of **s** would become one.

# Using Semaphores

- ❑ What happens if there are three processes:  
 $P_0, P_1, P_2$
- ❑ Assume  $P_0$  enters its critical section
- ❑ If  $P_1$  and  $P_2$  execute the **down** operation, they will block
- ❑ When  $P_0$  leaves the critical section then  $P_1$  is unblocked allowing  $P_1$  to enter its critical section
  - $P_2$  is still blocked
- ❑ What if  $P_0$  wants to enter its critical section again when  $P_1$  is in it?

# Using semaphore.h

- ❑ `sem_init(sem_t *sem, int pshared, unsigned int count);`
  - Pshared = 0 sem is shared between threads.
- ❑ `sem_wait(sem_t *sem);`
- ❑ `sem_post(sem_t *sem);`
- ❑ `sem_destroy(sem_t *sem);`

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
```

```
sem_t mutex;
```

```
void* thread (void* arg)
{
    sem_wait(&mutex);
    printf("\nEntered thread\n");
    sleep(4);
    printf("\nExit thread\n");
    sem_post(&mutex);
}
```

```
int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread, NULL);
    sleep(2);
    pthread_create(&t2, NULL, thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    sem_destroy(&mutex);
    return( 0 );
}
```

# Binary Semaphore

- A special case of a semaphore in which  $S$  is either 0 or 1
- Sometimes called a **mutex**

# Summary

- ❑ Defined race condition
- ❑ Examined different solutions