Part II
Process Management
Chapter 6: Process Synchronization

Process Synchronization

- **Why is synchronization needed?**
- **Race Conditions**
- **Critical Sections**
- **Pure Software Solutions**
- **Hardware Support**
- **Semaphores**
- **Monitors**
- **Message Passing**

Why is Synchronization Needed? 1/4

```
int  Count = 10;
```

Process 1 **Process 2**

```
Count++;   Count--;
```

`Count` = ? 9, 10 or 11?

Why is Synchronization Needed? 2/4

```
int  Count = 10;
```

Process 1	Process 2
⋮	⋮
`LOAD  Reg, Count`	`LOAD  Reg, Count`
`ADD   #1`	`SUB   #1`
`STORE Reg, Count`	`STORE Reg, Count`
⋮	⋮

The problem is that the execution flow may be switched in the middle!

Why is Synchronization Needed? 3/4

Process 1 Inst	Process 1 Reg	Process 1 Memory	Process 2 Inst	Process 2 Reg	Process 2 Memory
LOAD	10	10			
			LOAD	10	10
			SUB	9	10
ADD	11	10			
STORE	11	11	*erases the previous value 11*		
			STORE	9	9

Why is Synchronization Needed? 4/4

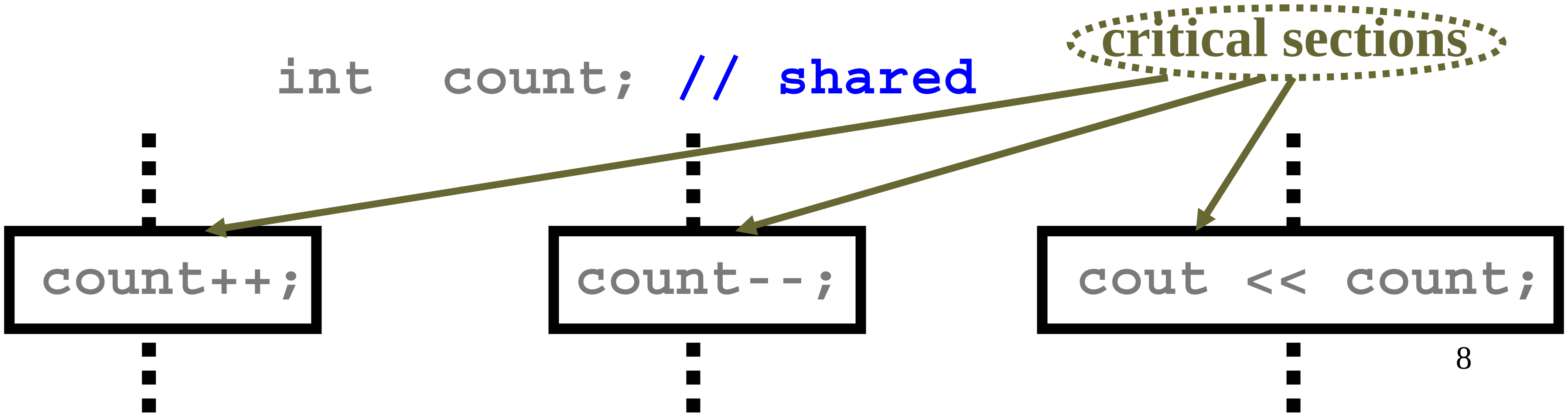
Process 1 Inst	Process 1 Reg	Process 1 Memory	Process 2 Inst	Process 2 Reg	Process 2 Memory
LOAD	10	10			
ADD	11	10			
			LOAD	10	10
			SUB	9	10
			STORE	9	9
STORE	11	11	*erases the previous value 9*		

Race Conditions

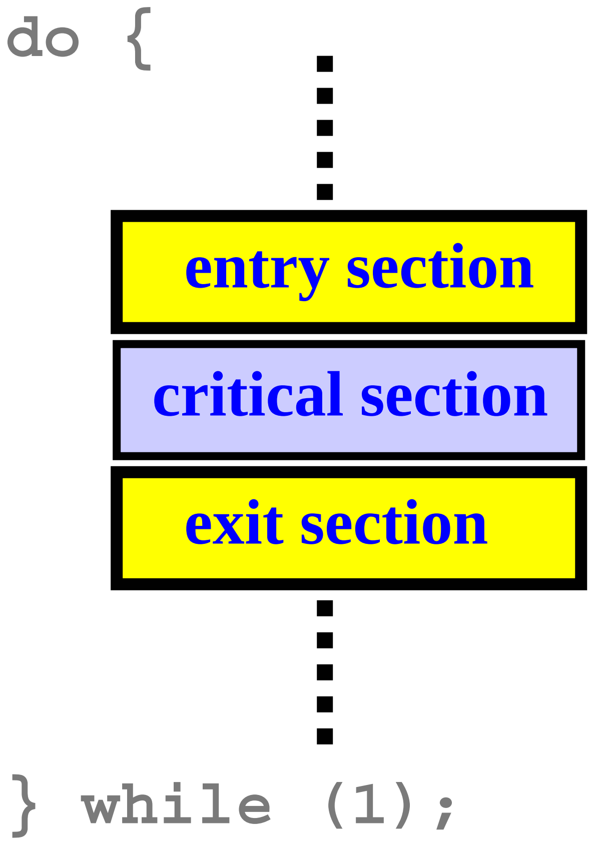
- **A *Race Condition* occurs, if**
  - **two or more processes/threads access and manipulate the same data concurrently, and**
  - **the outcome of the execution depends on the particular order in which the access takes place.**
- ***Synchronization* is needed to prevent race conditions from happening.**
- **Synchronization is a difficult topic. Don’t miss a class; otherwise, you will miss a lot of things.**

Critical Section and Mutual Exclusion

- A *critical section* is a section of code in which a process accesses shared resources.
- Thus, the execution of critical sections must be *mutually exclusive* (*e.g.*, at most one process can be in its critical section at any time).
- The *critical-section problem* is to design a protocol that processes can use to cooperate.



The Critical Section Protocol



- A **critical section protocol** consists of **two** parts: an ***entry section*** and an ***exit section***.
- **Between them is the critical section that must run in a mutually exclusive way.**

Solutions to the Critical Section Problem

- **Any solution to the critical section problem must satisfy the following three conditions:**
  - **Mutual Exclusion**
  - **Progress**
  - **Bounded Waiting**
- **Moreover, the solution cannot depend on CPU's relative speed and scheduling policy.**

Mutual Exclusion

- **If a process P is executing in its critical section, then *no* other processes can be executing in their critical sections.**
- **The entry protocol should be capable of blocking processes that wish to enter but cannot.**
- **Moreover, when the process that is executing in its critical section exits, the entry protocol must be able to know this fact and allows a waiting process to enter.**

Progress

- **If no process is executing in its critical section and some processes wish to enter their critical sections, then**
  - **Only those processes that are waiting to enter can participate in the competition (to enter their critical sections).**
  - **No other process can influence this decision.**
  - **This decision cannot be postponed indefinitely.**

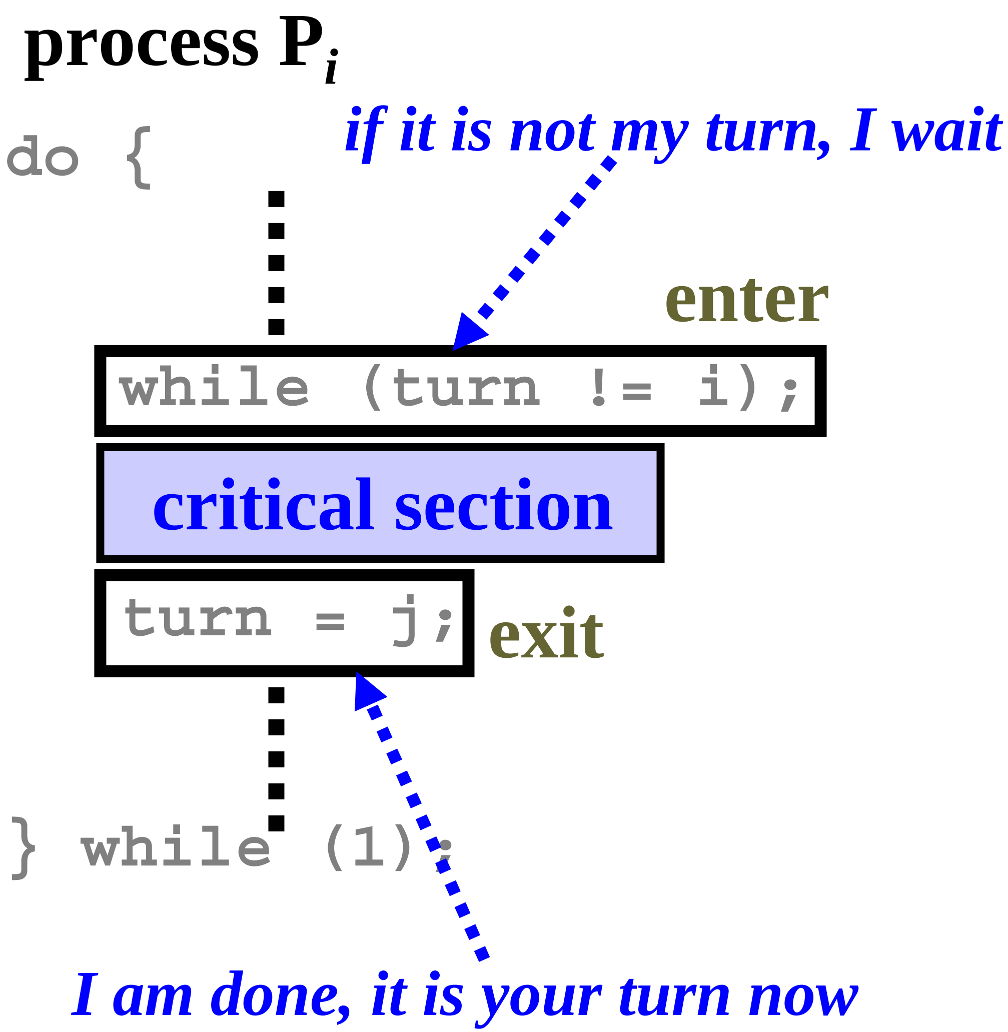
Bounded Waiting

- **After a process made a request to enter its critical section and before it is granted the permission to enter, there exists a *bound* on the number of times that other processes are allowed to enter.**
- **Hence, even though a process may be blocked by other waiting processes, it will not be waiting forever.**

Software Solutions for Two Processes

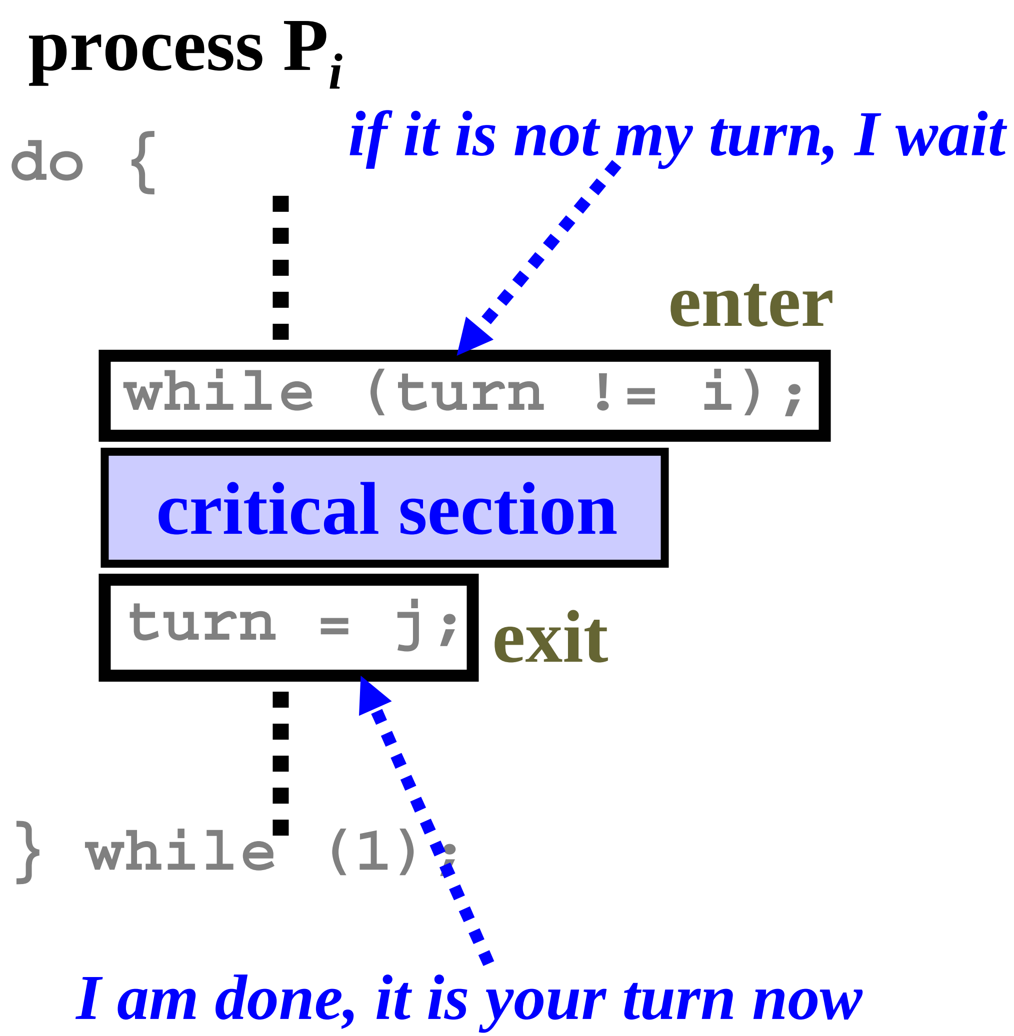
- **Suppose we have two processes, P_0 and P_1.**
- **Let one process be P_i and the other be P_j, where $j = 1 - i$. Thus, if $i = 0$ (*resp*., $i = 1$), then $j = 1$ (*resp*., $j = 0$).**
- **We want to design the enter-exit protocol for a critical section so that mutual exclusion is guaranteed.**

Algorithm I: 1/2



- **Global variable `turn` controls who can enter the critical section.**
- **Since `turn` is either 0 or 1, only one can enter.**
- **However, processes are forced to run in an alternating way.**

Algorithm I: 2/2



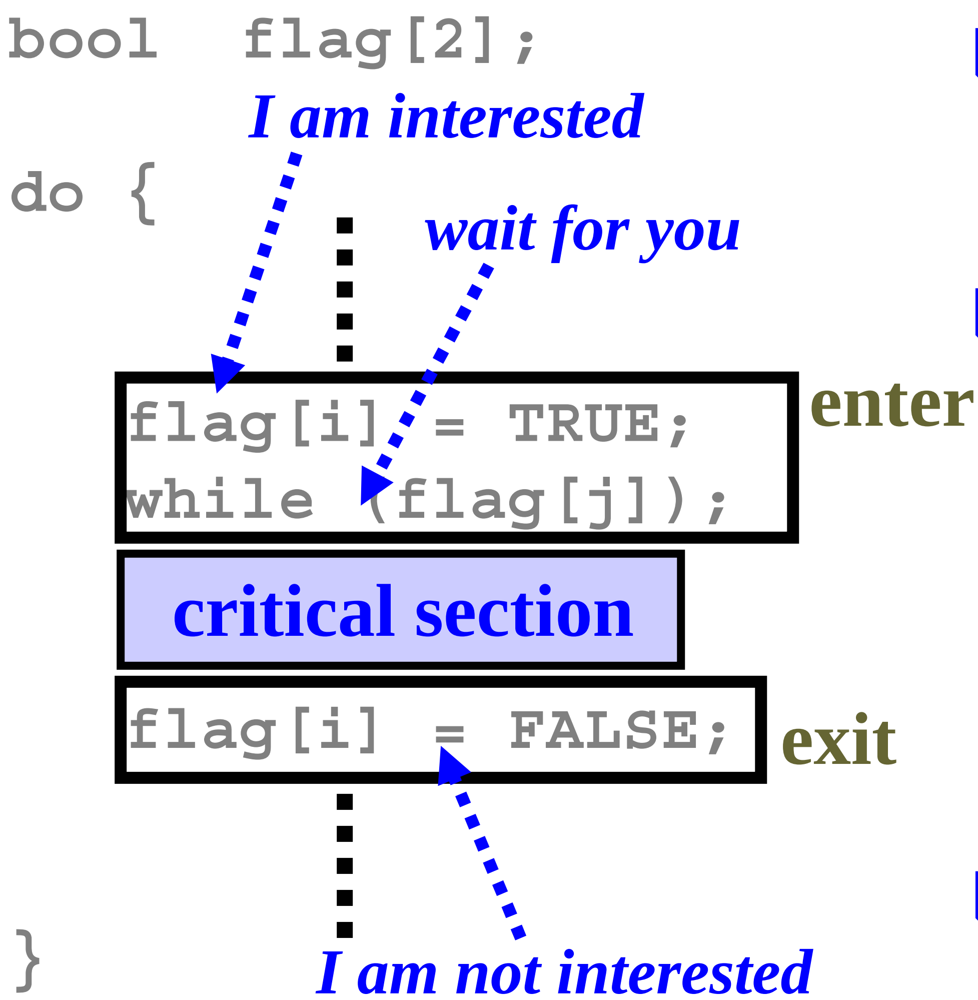
- This solution does not fulfill the *progress* condition.
- If P_j exits by setting `turn` to *i* and terminates, P_i can enter but cannot enter again.
- Thus, an irrelevant process can block other processes from entering a critical section.

Algorithm II: 1/2

```
bool  flag[2];

do {
      ...
   flag[i] = TRUE;        // enter   (I am interested)
   while (flag[j]);       //         (wait for you)

   critical section

   flag[i] = FALSE;       // exit    (I am not interested)
      ...
}
```

- Variable `flag[i]` is the "state" of process P_i: *interested* or *not-interested*.
- P_i expresses its intention to enter, waits for P_j to exit, enters its section, and finally changes to "I am out" upon exit.

Algorithm II: 2/2



- **The correctness of this algorithm is *timing dependent*!**
- **If both processes set `flag[i]` and `flag[j]` to `TRUE` at the same time, then both will be looping at the `while` forever and no one can enter.**
- **Bounded waiting does not hold.**

Algorithm III: a Combination 1/4

```
bool flag[2];    // process Pi
int  turn;

do {
      ⋮
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j);
      critical section
   flag[i] = FALSE;
      ⋮
}
```

I am interested (flag[i] = TRUE;)

yield to you first (turn = j;)

enter

wait while you are interested and it is your turn. (while (flag[j] && turn == j);)

critical section

I am done (flag[i] = FALSE;)

exit

Algorithm III: Mutual Exclusion 2/4

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process P_i

- If both processes are in their critical sections, then
  - `flag[j] && turn == j` (P_i) and `flag[i] && turn == i` (P_j) are both `FALSE`.
  - `flag[i]` and `flag[j]` are both `TRUE`
  - Thus, `turn == i` and `turn == j` are `FALSE`.
  - Since `turn` can hold one value, only one of `turn == i` or `turn == j` is `FALSE`, but not both.
  - We have a contradiction and P_i and P_j cannot be in their critical sections at the same time.

Algorithm III: Progress 3/4

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

process P_i

- If P_i is waiting to enter, it must be executing its `while` loop.
- Suppose P_j is not in its critical section:
  - If P_j is not interested in entering, `flag[j]` was set to `FALSE` when P_j exits. Thus, P_i may enter.
  - If P_j wishes to enter and sets `flag[j]` to `TRUE`, it will set `turn` to `i` and P_i may enter.
- In both cases, processes that are not waiting do not block the waiting processes from entering.

Algorithm III: Bounded Waiting 4/4

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

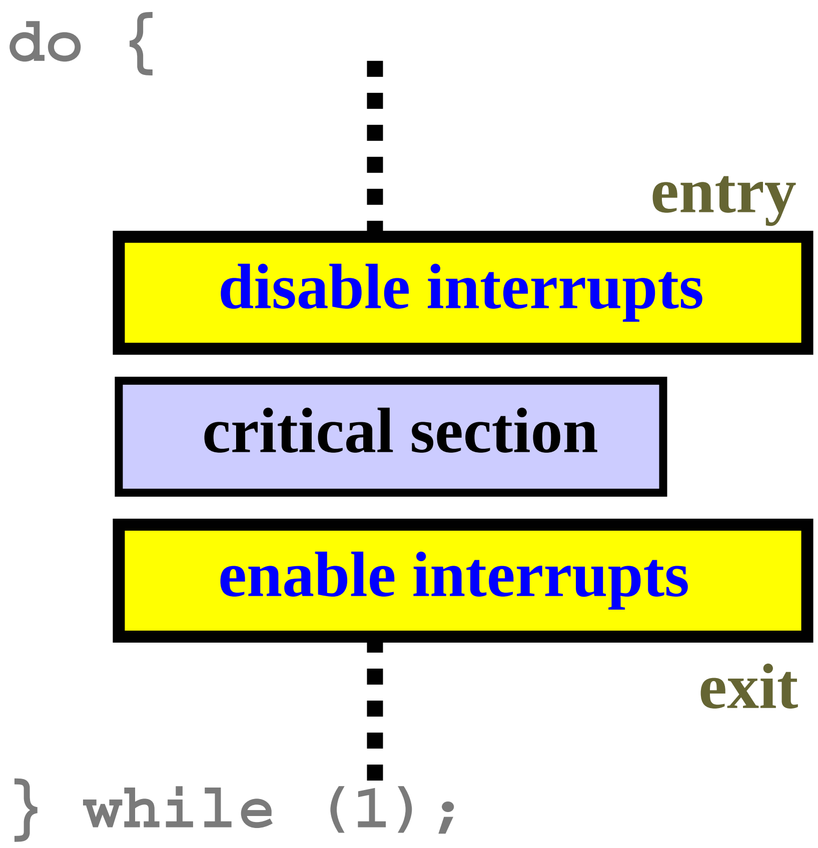
process P_i

- When P_i wishes to enter:
  - If P_j is *outside* of its critical section, then `flag[j]` is `FALSE` and P_i may enter.
  - If P_j is *in* its critical section, eventually it will set `flag[j]` to `FALSE` and P_i may enter.
  - If P_j is *in the entry section*, P_i may enter if it reaches `while` first. Otherwise, P_j enters and P_i may enter *after* P_j sets `flag[j]` to `FALSE` and exits.
- Thus, P_i waits at most one round!

Hardware Support

- **There are two types of hardware synchronization supports:**
  - **Disabling/Enabling interrupts: This is slow and difficult to implement on multiprocessor systems.**
  - **Special *privileged* machine instructions:**
    - **Test and set (`TS`)**
    - **Swap**
    - **Compare and Swap (`CS`)**

Interrupt Disabling



- Because interrupts are disabled, no **context switch** will occur in a critical section.
- Infeasible in a **multiprocessor** system because all CPUs must be informed.
- Some features that depend on interrupts (*e.g.*, clock) may not work properly.

Special Machine Instructions

- ***Atomic*: These instructions execute as one uninterruptible unit. More precisely, when such an instruction runs, all other instructions being executed in various stages by the CPUs will be stopped (and perhaps re-issued later) until this instruction finishes.**
- **Thus, if two such instructions are issued at the same time, even though on different CPUs, they will be executed sequentially.**
- ***Privileged*: These instructions are, in general, privileged, meaning they can only execute in supervisor or kernel mode.**

The Test-and-Set Instruction

```
bool TS(bool *key)
{
    bool save = *key;
    *key = TRUE;
    return save;
}
```

```
bool  lock = FALSE;

do {
    ...
    while (TS(&lock));      // entry
    critical section
    lock = FALSE;           // exit
    ...
} while (1);
```

- **Mutual exclusion is obvious as only one `TS` instruction can run at a time.**
- **However, progress and bounded waiting may not be satisfied.**

Problems with Software and Hardware Solutions

- **All of these solutions use *busy waiting*.**
- ***Busy waiting* means a process waits by executing a tight loop to check the status/value of a variable.**
- **Busy waiting may be needed on a multiprocessor system; however, it wastes CPU cycles that some other processes may use productively.**
- **Even though some personal/lower-end systems may allow users to use some atomic instructions, unless the system is lightly loaded, CPU and system performance can be low, although a programmer may "think" his/her program looks more efficient.**
- **So, we need a better solution.**