Lecture: 18

```
/* Initialization */
while (TRUE) {
  /* entry code */
    CS() /* critical section */
  /* exit code */
   Non_CS()/* non-critical section */
}
```

No assumption about how often the critical section is accessed

Wrapper code

Another Try

ME ✓

Progress ✗

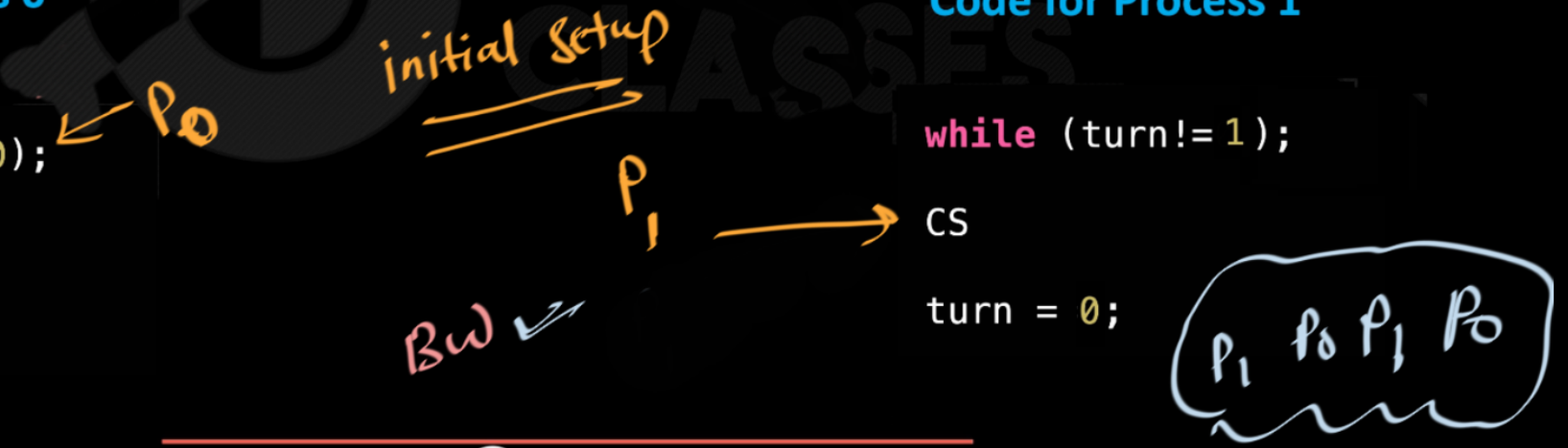
```
int turn = 1  // shared variable
```

Code for Process 0

```
while (turn!=0);
CS
turn = 1;
```

Code for Process 1

```
while (turn!= 1);
CS
turn = 0;
```



Yet Another Try...

```
int want[2] = {False, False}; // shared variable
```

Code for Process 0

```
want[0] = True;
while (want[1]);

CS

want[0] = False;
```

Code for Process 1

```
want[1] = True;
while (want[0]);

CS

want[1] = False;
```

ME ✓

Progress ✗ (because of deadlock)

Progress ✗ deadlock ✗

GO CLASSES

921-Omi to Everyone 7:01 PM

9

Sir if shared variable like turn etc are not initialsized so in exam we have to take all possibilities which he can take it

↳ yes

if in any of the execution, (or initialization) some condition is not satisfied then we say overall it is not satisfied.

Peterson's Solution

Peterson's Solution

```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

Code for Process 0

```
want[0] = True;
favored = 1;
while (want[1] && favored ==1);

CS

want[0] = False;
```

Code for Process 1

```
want[1] = True;
favored = 0;
while (want[0] && favored ==0);

CS

want[1] = False;
```


```
// shared variables
int want[2] = {False, False};
```

Code for Process 0

```
want[0] = True;
while (want[1] );
CS
want[0] = False;
```

Code for Process 1

```
want[1] = True;
while (want[0] );
CS
want[1] = False;
```

Peterson's Solution

it is a polite way of saying that i favour other process

i am interested

```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

Code for Process 0

```
want[0] = True;
favored = 1;
while (want[1] && favored ==1);

CS

want[0] = False;
```

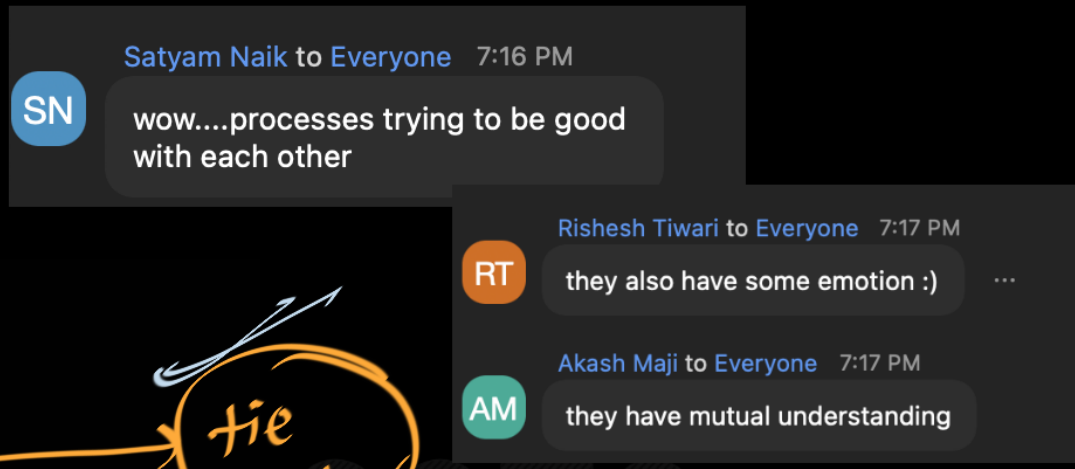
Code for Process 1

```
want[1] = True;
favored = 0;
while (want[0] && favored ==0);

CS

want[1] = False;
```

GO CLASSES




```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

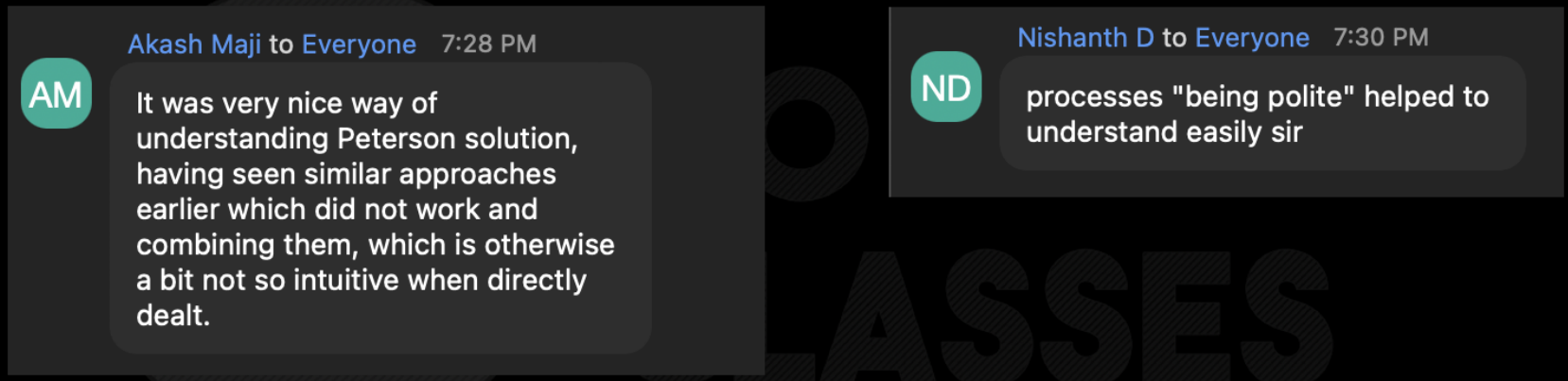
tie breaker

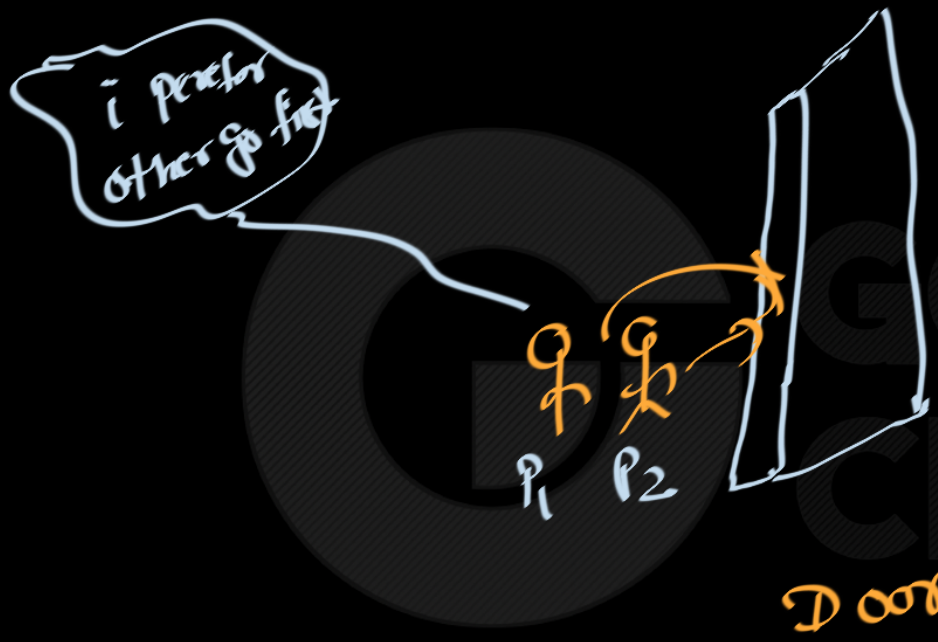
Purpose of want[] is to tell that, "i am interested"

Purpose of favour is to be polite and say i favour other process.

"favoured" will break the tie if both Processes are interested in CS.

it will break the tie in the by agreeing to the process who executed it last.


Akash Maji to Everyone 7:28 PM
AM
It was very nice way of understanding Peterson solution, having seen similar approaches earlier which did not work and combining them, which is otherwise a bit not so intuitive when directly dealt.
Nishanth D to Everyone 7:30 PM
ND
processes "being polite" helped to understand easily sir



Both the persons interested to go inside.

ME

Process

```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

Code for Process 0

```
want[0] = True;
favored = 1;
while (want[1] && favored ==1);

CS

want[0] = False;
```

Code for Process 1

```
want[1] = True;
favored = 0;
while (want[0] && favored ==0);

CS

want[1] = False;
```

favor is breaking the tie (in case of deadlock)

Bus

```
// shared variables
int want[2] = {False, False};
int favored = 0;
```

Code for Process 0

Can not re enter

```
want[0] = True;
favored = 1;
while (want[1] && favored ==1);

CS

want[0] = False;
```

Process 0

Code for Process 1

Process 1

```
want[1] = True;
favored = 0;
while (want[0] && favored ==0);

CS

want[1] = False;
```

Stuck

T F F T

wants F F

favored = 0

```
int turn;
boolean flag[2];
```

```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Source : Galvin

Figure 5.2 The structure of process P_i in Peterson's solution.


```
widget x; //protected variable
bool she_wants(false);
bool he_wants(false);
enum theirs {hers, his} turn;
```

Her Thread

```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

His Thread

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

if she wants and it is her turn then i will wait

Consider the following correct solution to the critical section problem that was discussed in class. The shared variables are the integer `turn` and the integer array `flag[2]`. There are two processes, P0 and P1. These processes execute the following code in a loop running on a single CPU with a time sharing operating system.

```
1: P0 code:                                  |  P1 code:
2: while(1) {                                |  while(1) {
3:   flag[0] = 1;                            |   flag[1] = 1;
4:   turn = 1;                               |        turn = 0;
5:   while((flag[1]==1)&&(turn==1));         |   while((flag[0]==1)&&(turn==0));
6:        <critical section>                 |        <critical section>
7:   flag[0] = 0;                            |   flag[1] = 0;
8:        <remainder section>                |        <remainder section>
9:}                                          |  }
```

a. [4pt] What is meant by **bounded waiting**? Does this solution satisfy bounded waiting? If so, what is the bound?

b. [1pt] What can you say about the values of the shared variables if you know that P1 is in its critical section?

c. [1pt] What can you say about the values of the shared variables if you know that P1 is in its remainder section?

d. [2pt] What can you say about the values of the shared variables if you know that P0 is in its critical section and P1 is in its remainder section?

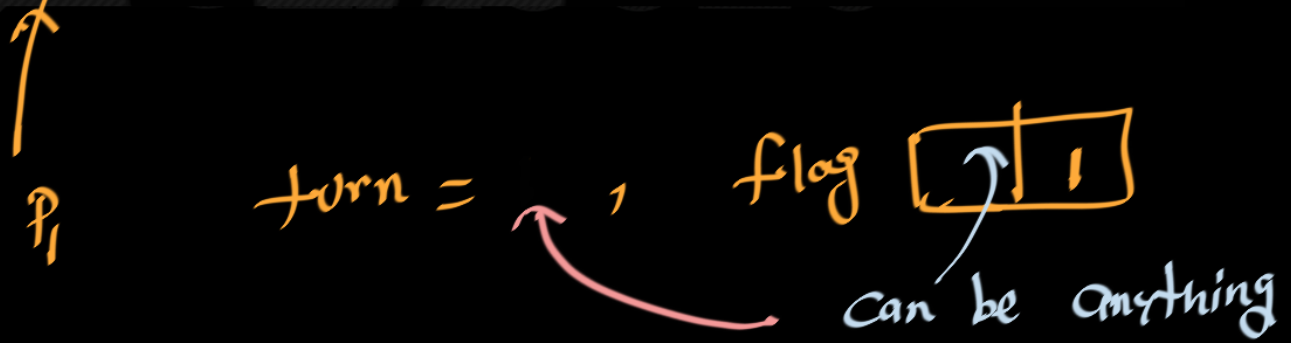
e. [2pt] What is meant by **busy waiting**? Does this solution use busy waiting?

http://www.cs.utsa.edu/~korkmaz/teaching/resources-os-ug/tk-samples/solutions/2017c-Fall-cs3733-Final-solution.pdf

GO CLASSES

```
1: P0 code:
2: while(1) {
3:  flag[0] = 1;
4:  turn = 1;
5:  while((flag[1]==1)&&(turn==1));
6:      <critical section>
7:  flag[0] = 0;
8:      <remainder section>
9:}
```

```
P1 code:
while(1) {
 flag[1] = 1;
      turn = 0;
 while((flag[0]==1)&&(turn==0));
      <critical section>
 flag[1] = 0;
      <remainder section>
}
```



a. [4pt] What is meant by **bounded waiting**? Does this solution satisfy bounded waiting? If so, what is the bound?

b. [1pt] What can you say about the values of the shared variables if you know that P1 is in its critical section?

c. [1pt] What can you say about the values of the shared variables if you know that P1 is in its remainder section?

d. [2pt] What can you say about the values of the shared variables if you know that P0 is in its critical section and P1 is in its remainder section?

e. [2pt] What is meant by **busy waiting**? Does this solution use busy waiting?

c. [1pt] What can you say about the values of the shared variables if you know that P1 is in its remainder section?

```
1: P0 code:                                  P1 code:
2: while(1) {                                while(1) {
3:  flag[0] = 1;                              flag[1] = 1;
4:  turn = 1;                                       turn = 0;
5:  while((flag[1]==1)&&(turn==1));           while((flag[0]==1)&&(turn==0));
6:      <critical section>                          <critical section>
7:  flag[0] = 0;                              flag[1] = 0;
8:      <remainder section>                         <remainder section>
9:}                                          }
```

GO CLASSES

d. [2pt] What can you say about the values of the shared variables if you know that P0 is in its critical section and P1 is in its remainder section?

```
1: P0 code:
2: while(1) {
3:  flag[0] = 1;
4:  turn = 1;
5:  while((flag[1]==1)&&(turn==1));
6:      <critical section>
7:  flag[0] = 0;
8:      <remainder section>
9:}
```

```
P1 code:
while(1) {
 flag[1] = 1;
      turn = 0;
 while((flag[0]==1)&&(turn==0));
      <critical section>
 flag[1] = 0;
      <remainder section>
}
```

we want P_1 to go to CS first

as per question

P_1

P_0

P_1

turn = 0 (P_0 must have executed $\longrightarrow$ P_1 has executed)

turn = 1 (P_1 execute $\longrightarrow$ P_0 execute)

turn = 1

a. [4pt] What is meant by **bounded waiting**? Does this solution satisfy bounded waiting? If so, what is the bound?

2pt- Bounded waiting means that once a given process attempts to enter its critical section, there is a bound on the number of times other processes can enter their critical sections before that given process will enter its critical section.

1pt- This solution satisfies bounded waiting

1pt- with a bound of 1.

b. [1pt] What can you say about the values of the shared variables if you know that P1 is in its critical section?

You only know that `flag[1]` is 1.

Each of the other variables may be either 0 or 1.

c. [1pt] What can you say about the values of the shared variables if you know that P1 is in its remainder section?

You only know that `flag[1]` is 0.

Each of the other variables may be either 0 or 1.

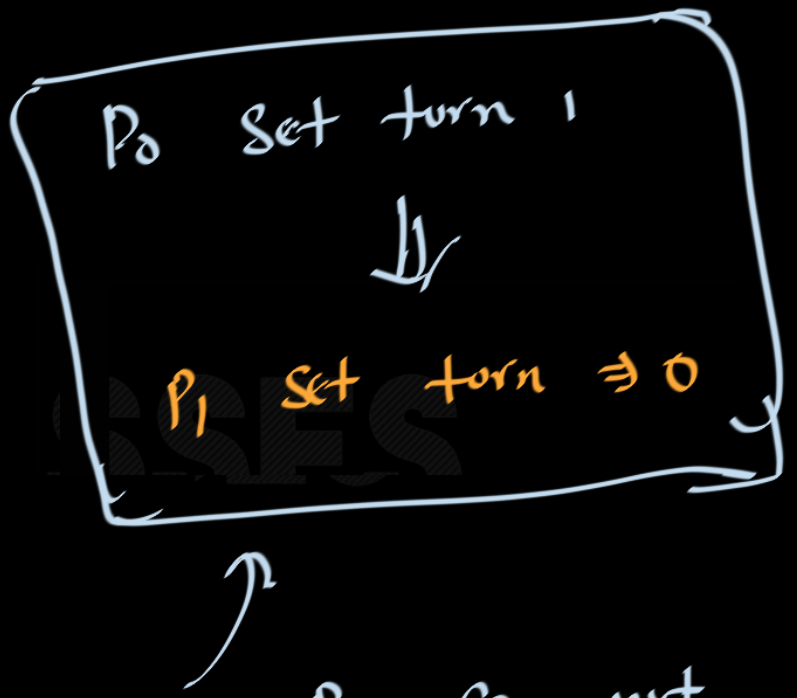
d. [2pt] What can you say about the values of the shared variables if you know that P0 is in its critical section and P1 is in its remainder section?

You know that `flag[0]` is 1 and `flag[1]` is 0.
`turn` must be 1,

otherwise, for `turn` to be `0`

P0 must set `turn` to 0 after P1 sets `turn` to 1.
At this time `flag[0]` is 1 and `turn` is 0,

so P1 cannot get past its while loop until P0 finishes its C.S.,
so P1 is not in its remainder section.



e. [2pt] What is meant by **busy waiting**? Does this solution use busy waiting?

Busy waiting means using the CPU to continually check for an event,

as this solution does in the **while** loop.

GO CLASSES

T_1

Busy waiting or Spinning

while (flag[1]); ← Spinning

↗

loop till the point flag[1] = 0 (Busy in waiting for flag[1] to be true)

⇒ it is scheduled ⇒ taking CPU time and doing nothing

Question :

3. Consider the following solution for the two-process synchronization, where below is the code for P_i (recall that, as usual, P_j is the *other* process where $i = 1 - j$).

```
do {
    flag[i] = true
    turn = j
    while (flag[j] and turn=j);

    CRITICAL SECTION

    flag[i] = false

    REMAINDER SECTION

} while(1)
```

Show that the following three hold:

(a) Mutual exclusion is preserved

(b) Progress requirement is satisfied

(c) Bounded waiting requirement is met

(Use next blank page if needed.)

http://prof.msoltys.com/wp-content/oldpage/se3bb4-w13/midterm1.pdf

Solution: Mutex: suppose P_0, P_1 are both in the critical section. That would mean that `flag[0]=flag[1]=true` and so they both entered their respective critical sections by breaking out of the while-loop with `turn=0` (for P_0) and `turn=1` (for P_1). But `turn` is a shared variable, and hence this is not possible.

Progress: suppose that P_0 is ready for another round of CS, and that P_1 is in its RS. That means that P_1 just set `flag[1]=false` which breaks P_0 out of the while loop. The dual argument shows the same for P_1 ready for CS and P_0 in its RS.

Bounded waiting: If P_0 is ready to enter its CS, then P_1 will be able to enter its CS at most once; as soon as P_1 exits its CS, it turns `flag[1]=false` allowing P_0 entry. Again, the dual argument shows the same for P_1 and P_0.

We have seen Software based solutions,

which means all the soln's were just

piece of code.

Do you think any disadvatage of Peterson's soln.

GO CLASSES

Disadvantage of Peterson's soln (or any other working soln)

→ Code are slow (compared to hardware)

→ Proving the correctness is also an effort

→ Peterson's soln is only for 2 processes

→ we can generalise to n process but n must be known in prior. (out of syllabus)

→ Busy waiting (wasting CPU cycles)

Hardware Supported Solutions

Hardware Supported Solutions

- Disabling Interrupt

- Special Machine Instructions
  - test-and-set, compare-and-swap, ...

1. Hardware Solutions: Disabling Interrupt

- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

if you disable the interrupts before entering into CS, it will disallow any kind of interrupt including context switch

in CS, CPU will just be executing CS without preemption

Working of Disabling Interrupt

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Source: Galvin

Problems with Disabling Interrupt

Unfortunately, this solution is not as feasible in a multiprocessor environment.

→ for every processor, we have dedicated "interrupt lines"

Source: Galvin

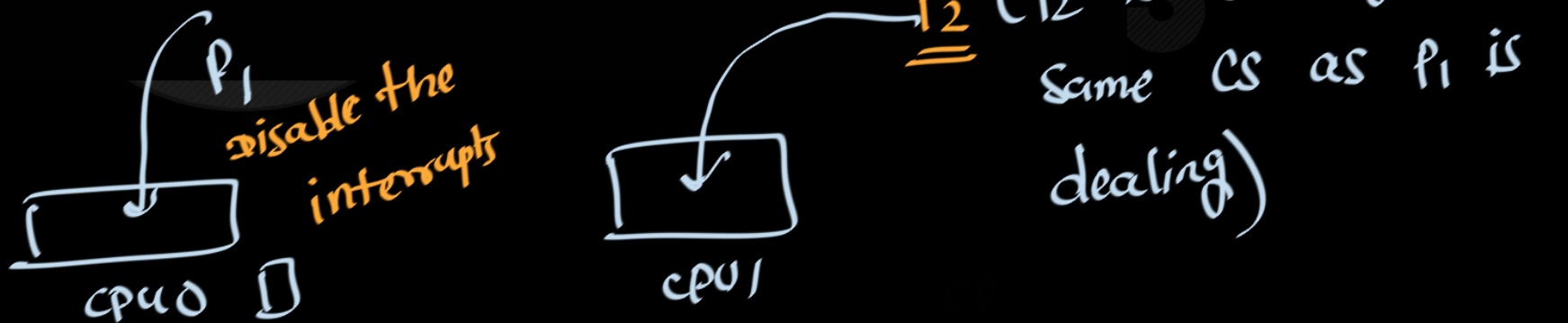
Problems with Disabling Interrupt

Unfortunately, this solution is not as feasible in a multiprocessor environment.

→ for every processor, we have dedicated "interrupt lines"

Source: Galvin

→



Problems with Disabling Interrupt

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Source: Galvin

Problems with Disabling Interrupt (cont..)

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor (with two or more CPUs) disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Source: Tanenbaum

Optional read from William Stalling

Interrupt Disabling

In a uniprocessor system, concurrent processes cannot have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts. A process can then enforce mutual exclusion in the following way (compare to Figure 5.4):

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed. The price of this approach, however, is high. The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes. Another problem is that this approach will not work in a multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

GO CLASSES

Other Soln that we seek from hardware : H/w itself Provide ATOMIC "Read - Modify - write" instructions (RMW)

mov , load


Rishesh Tiwari to Everyone 8:27 PM
RT
You done post - mortam of everything of Peterson solution today :)
1

2. Hardware Solutions: Atomic operations

Atomic operations

Processors provide means to execute read-modify-write operations atomically on a memory location

- Typically applies to at most 8-bytes-long variables

Common atomic operations

- test_and_set(type *ptr): sets **ptr* to 1 and returns its previous value
- fetch_and_add(type *ptr, type val): adds *val* to **ptr* and returns its previous value
- compare_and_swap(type *ptr, type oldval, type newval): if $*ptr == oldval$, set **ptr* to *newval* and returns `true`; returns `false` otherwise

- **Examples:**
  - **Test&Set:** (most architectures) read a value, write '1' back to memory.
  - **Exchange:** (x86) swaps value between register and memory.
  - **Compare&Swap:** (68000) read value, if value matches register value r1, exchange register r2 and value.

https://lass.cs.umass.edu/~shenoy/courses/spring10/lectures/Lec08.pdf