Synchronisation

My formulation :
  data struct + synch

Synchronisation problem make OS challenge.

Race condition between updating variable, threads scheduling at any time.

In assemebly :

```
mov c, r1
add r1, 1
mov r1,c
```

in thread 1 and 2:

```
// if c is 5
t1 : mov c, r1
t2: mov c, r1
t1: add r1, 1   // c = 6
t1 : mov r1,c
t2 : add r1,1
t2 : mov r1,c // c = 67

// should have been
```

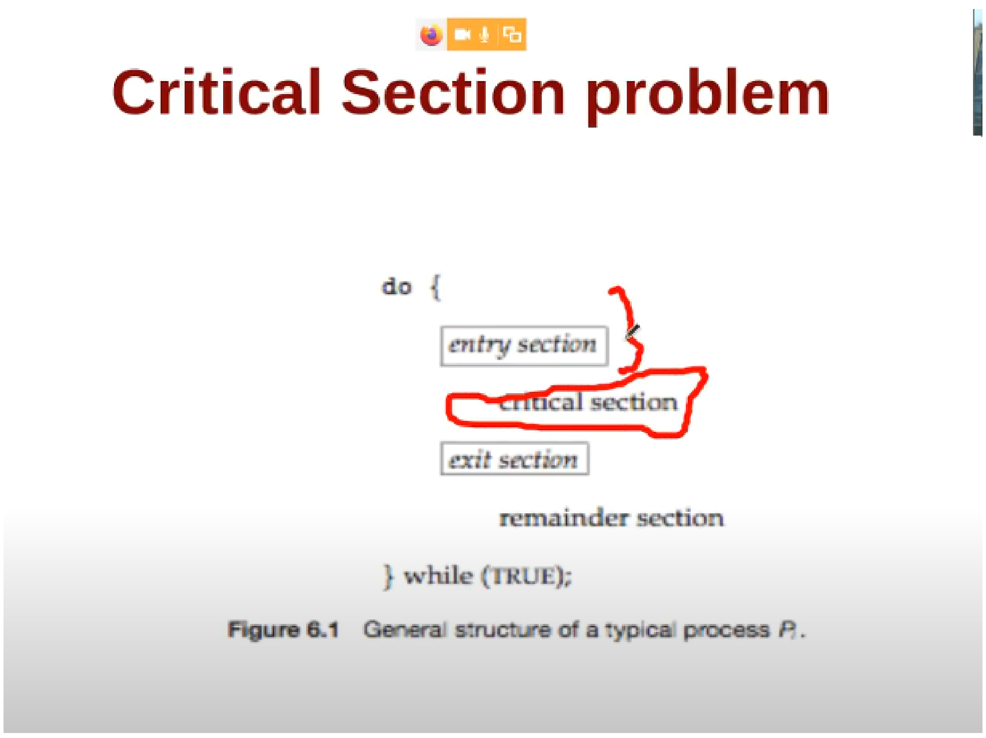
Race reasons:

1. Interruptible kernel:
   1. if interruptible, then modification on any kernel data struct can be left incomplete.
   2. Introduces **concurrency.**
2. Multiprocessor sys. :
   1. On SMP sys: memory is shared, kernel and proc code run on all processor.
   2. Some variables can be updated parellely.
3. Non interruptible kernel on multiprocessor : rarely
4. non interruptible on uni-processor :

Critical Section Problem:

Consider sys of n processes
each process has a critical segment of code
  process may be changing in common variables, updating tables, writing files
  when one process in critical section, no other may be in its critical section

Critical section problem is to design protocol to resolve this
Each processes must ask permission to enter critical section in **entry** section, may follow **critical** section with exit section then **remainder** section.



Critical Section problem

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Figure 6.1 General structure of a typical process P_i.

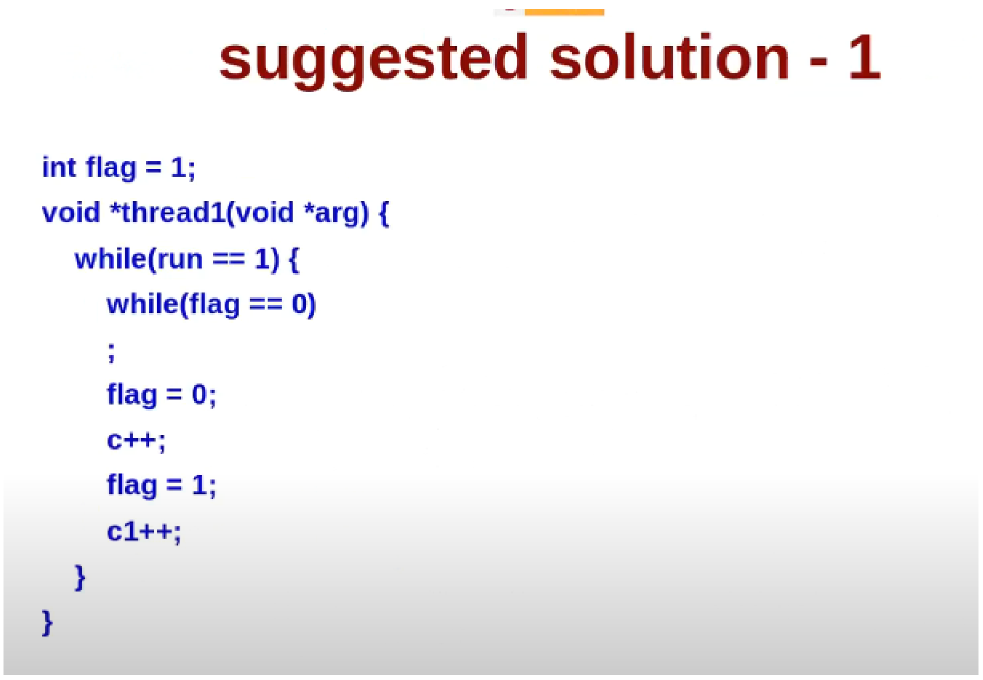
Expected Solution Characteristics:

1. Mutual exclusion :
    if process Pi is executing in its critical section then no other processes can be executing in their critical section

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 processe that will enter the critical section next cannot be postponed indefinitely.

3. bounded waiting:
    a bound must exist on the number of times that other proceses are allowed to enter their critial sections after a process has made a request to enter its critical section and before that request is granted.

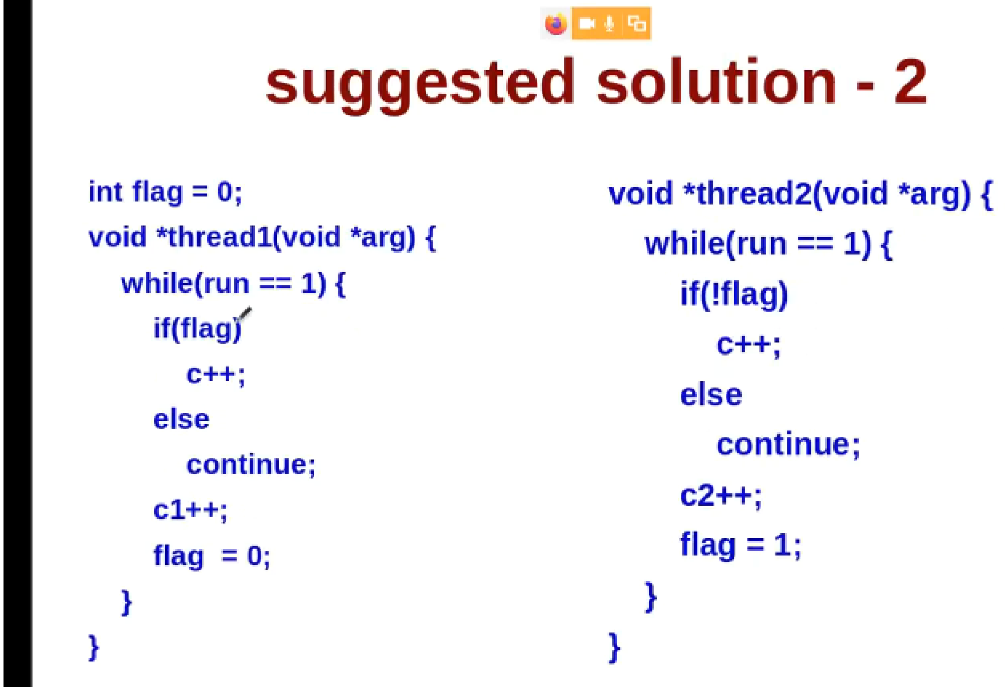
Assume that wach process executes at a non zero speed
No assumption concerning relative speed of the n processes



suggested solution - 1

```
int flag = 1;
void *thread1(void *arg) {
   while(run == 1) {
      while(flag == 0)
      ;
      flag = 0;
      c++;
      flag = 1;
      c1++;
   }
}
```

This doesn’t work because it assumes that it is **atomic** while (flag ==)

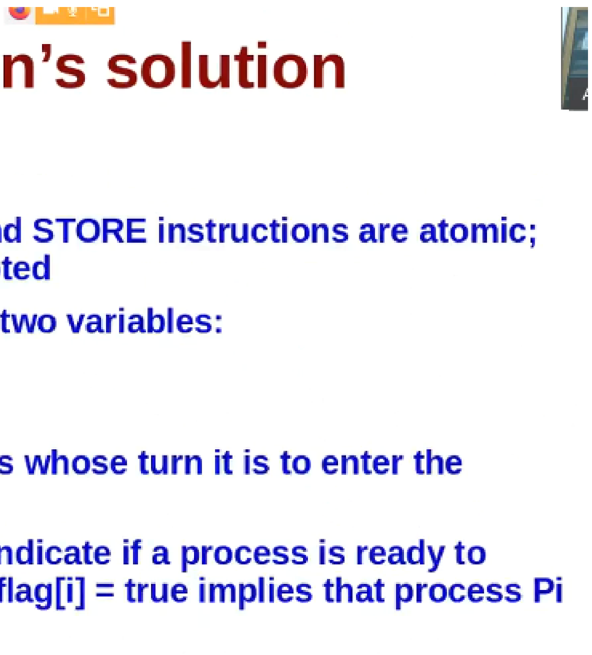


suggested solution - 2

```
int flag = 0;
void *thread1(void *arg) {
   while(run == 1) {
      if(flag)
         c++;
      else
         continue;
      c1++;
      flag  = 0;
   }
}
```

```
void *thread2(void *arg) {
   while(run == 1) {
      if(!flag)
         c++;
      else
         continue;
      c2++;
      flag = 1;
   }
}
```

Peterson’ solution :



Peterson’s solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

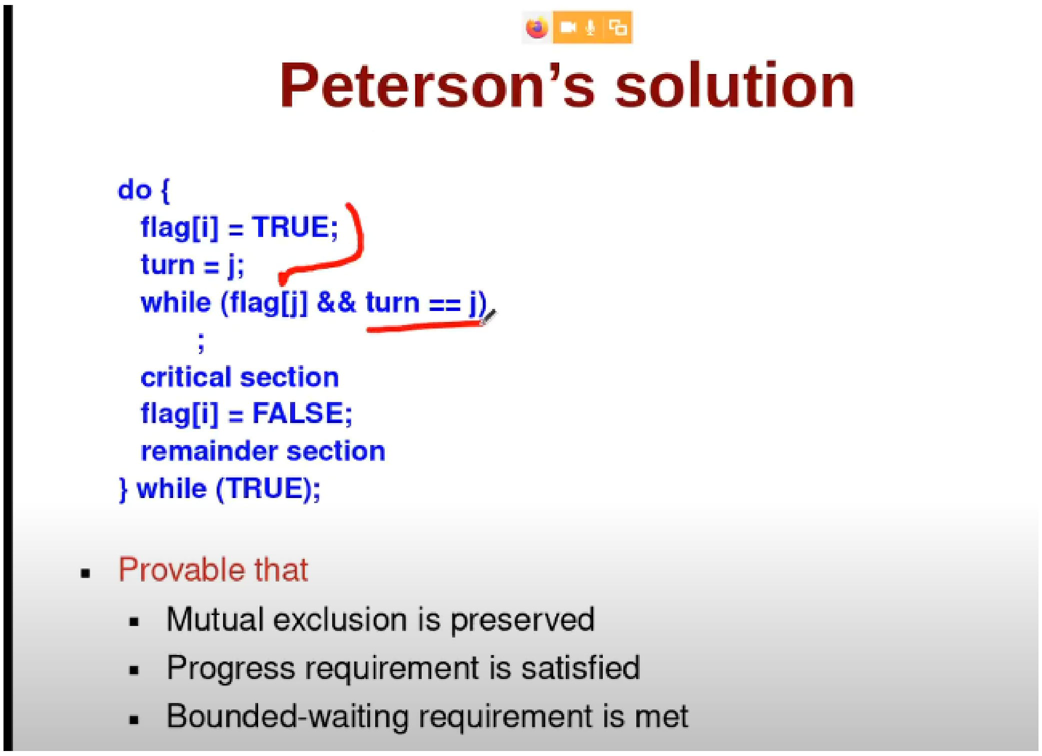
Peterson’s solution: code and provable that

Enter critical section when it is interested i.e. when it is true

then it should give other process a chance i.e. turn = j

while you are interested and you are given a chance i am waiting (flag[j] and turn == j)

if then i will enter critical sections
then i am not interested i.e. will be false



Peterson's solution

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

- Provable that
  - Mutual exclusion is preserved
  - Progress requirement is satisfied
  - Bounded-waiting requirement is met

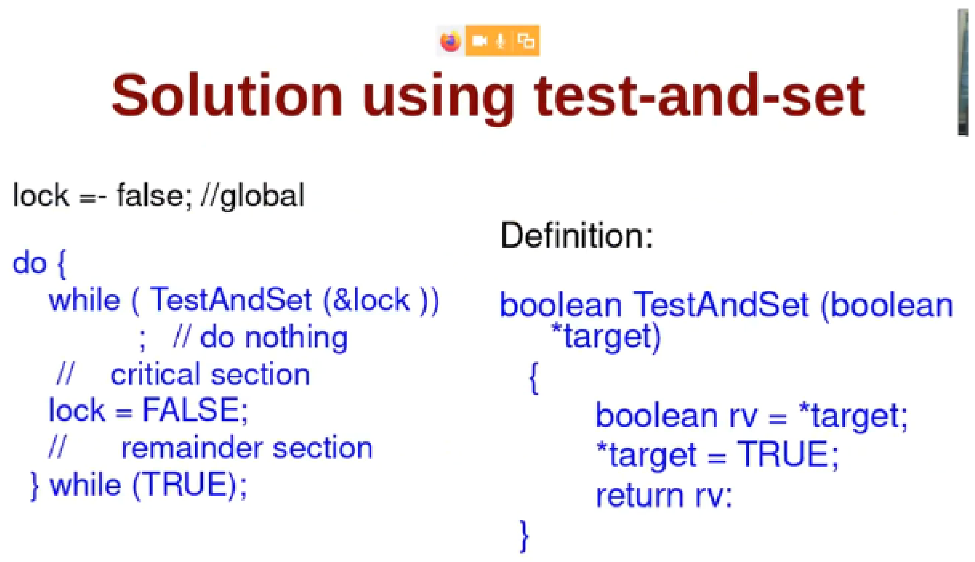
Hardware Solution:

ACTUALLY IMPLEMENTED:

many sys provide h/w support for crit. sect. code
in uniprocessor : could disable interrupts
curr. Running code would exec w/p premption (preemption : proc or kernel which is running have interrupted whose is not willing to give up its cpu)

to inefficient in multi-processing
OS using this is not broadly scalable
Modern machine provide special atomic h/w instr.

atomic : non-interruptible
either test memory word and set value
or swap contents of memory words
2 operations rd/wr done automatically by h/w



Solution using test-and-set

```
lock =- false; //global

do {
   while ( TestAndSet (&lock ))
           ;   // do nothing
    //    critical section
   lock = FALSE;
   //      remainder section
 } while (TRUE);
```

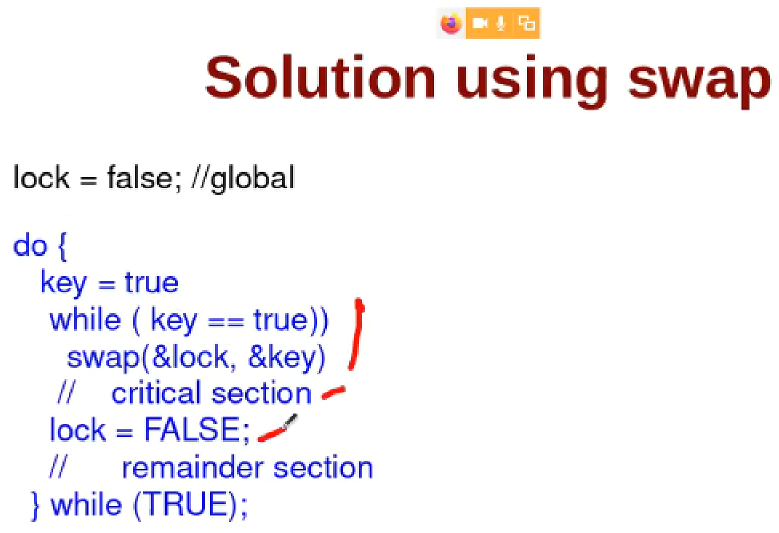
Definition:

```
boolean TestAndSet (boolean
     *target)
  {
        boolean rv = *target;
        *target = TRUE;
        return rv:
  }
```

here at first rv return FALSE, but it has switched to TRUE. Then in the next is waited in the while loop.
After the process execution in the critical section is over it again set the to FALSE.

Solution using swap

swap : a machine instruction to swap the values of key and lock. At first when key is true, proc enters and the is swapped with lock i.e. made false and lock as true,
then it executes the critical section w/o any interruption after exiting it makes LOCk false again



Solution using swap

```
lock = false; //global

do {
   key = true
    while ( key == true))
      swap(&lock, &key)
    //    critical section
    lock = FALSE;
    //      remainder section
  } while (TRUE);
```