PENNSTATE

Implementing Mutex

How do we implement Critical Sections/Mutual Exclusion?

- Disable Interrupts
  - **Effectively stops scheduling other threads.**
- Busy-wait/spinlock Solutions
  - **Pure software solutions**
  - **Integrated hardware-software solutions**
- Blocking Solutions

No Interrupts in period of time will garantee no context switching in period of time.

① Disable interrupts
② Busy wait / spinlock Solution
③ Blocking solution

PENNSTATE

Disabling Interrupts

- Advantages: Simple to implement

- Disadvantages:
  - **Do not want to give such power to users**
  - **Does not work on a multiprocessor**
  - **Disables multiprogramming even if another thread is NOT interested in critical section**

PENNSTATE

S/w solns. with busy-waiting

- Overall philosophy: Keep checking some state (variables) until they indicate other threads are not in critical section.

- However, this is a non-trivial problem.

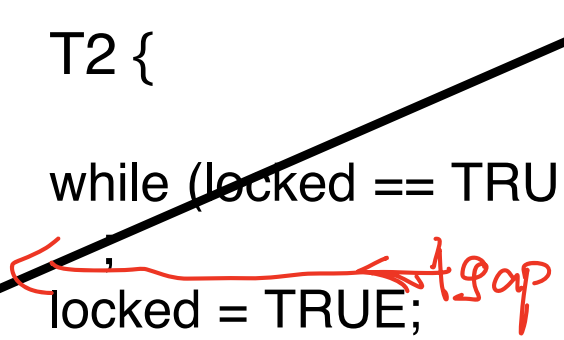
```
locked = FALSE;

T1 {                                    T2 {

while (locked == TRUE)                  while (locked == TRUE)
   ;                                       ;
locked = TRUE;                          locked = TRUE;

/************                           /************
(critical section code)                 (critical section code)
/************                           /************

locked = FALSE;                         locked = FALSE;
}                                       }
```



We have a race condition again since there is a gap between detection locked is FALSE, and setting locked to TRUE.

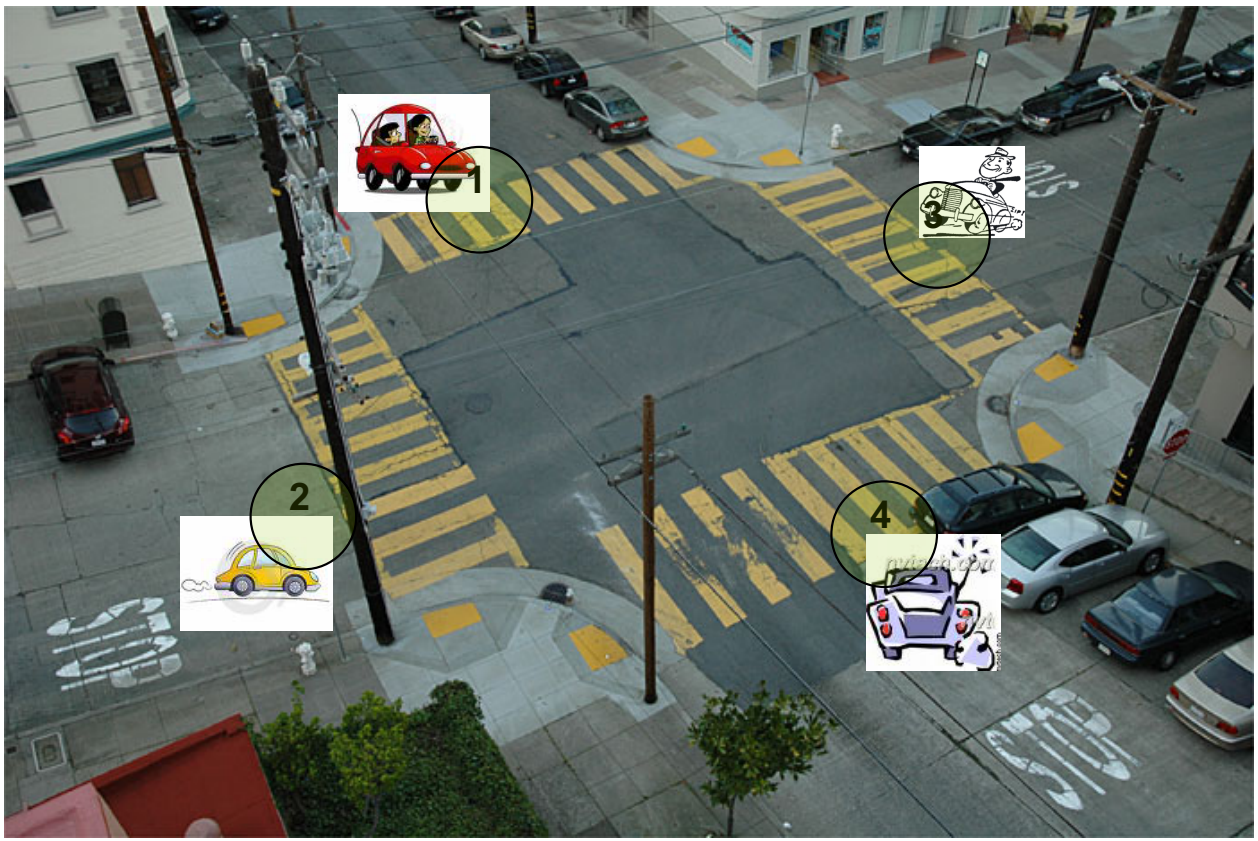
Desirables

- **Should achieve mutual exclusion – Correctness**
- **A process that requests to access a critical section should not be blocked if no one else is using the critical section – Progress**
- **No process should have to wait forever to enter its critical section – Bounded Waiting**

If a critical section is free and one process is interested in using it it should be used immediately

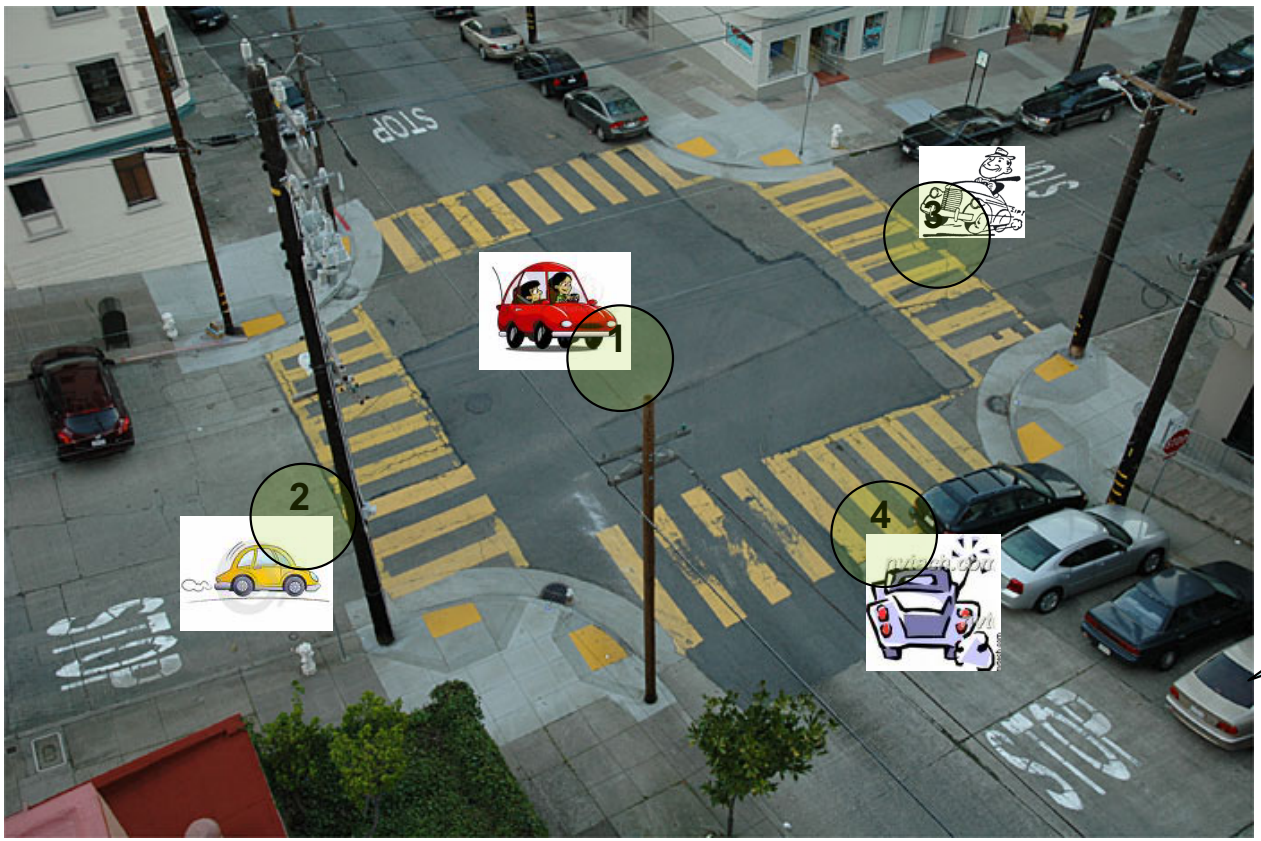
Mutual Exclusion: Traffic Analogy

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Mutual Exclusion: Traffic Analogy

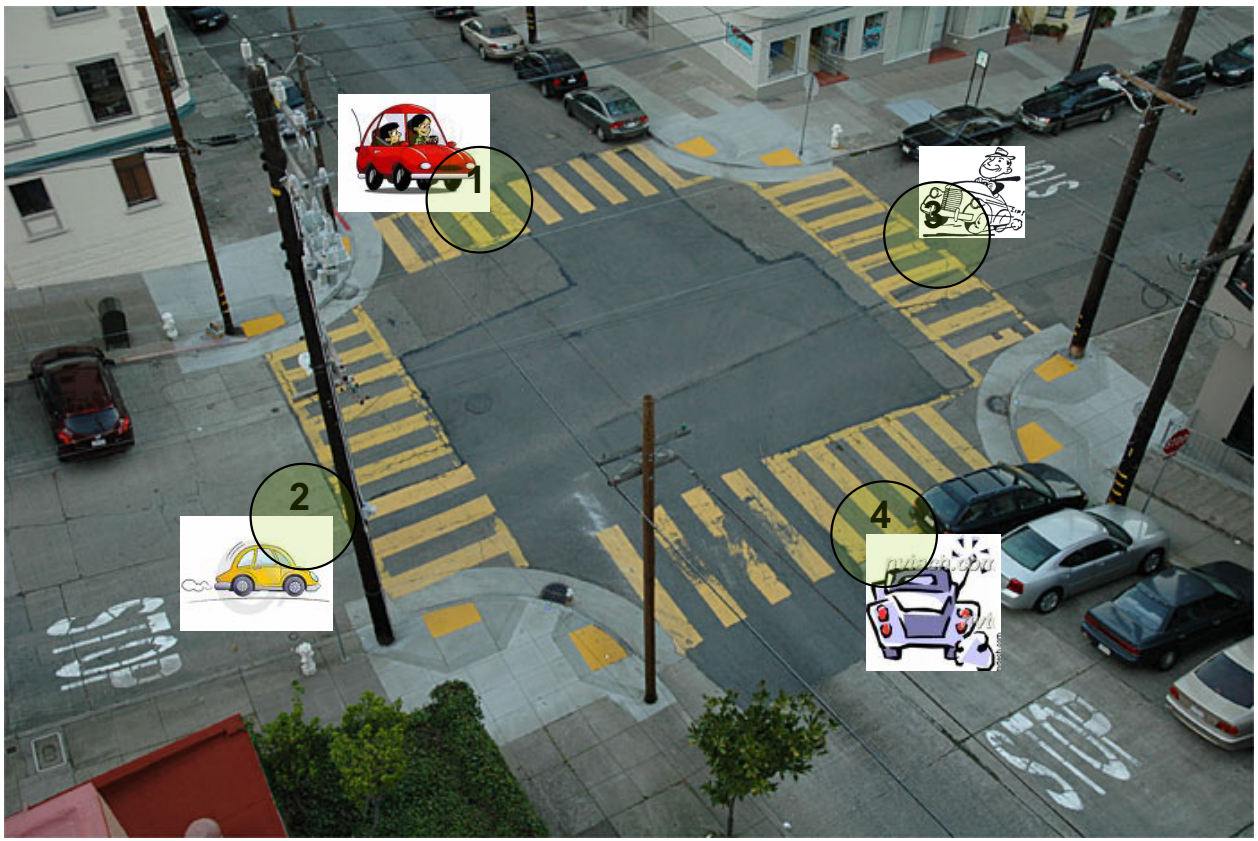
If process P_i is executing in its critical section, then no other processes can be executing in their critical sections



Only one car can be in the square area between the stop signs at a given time

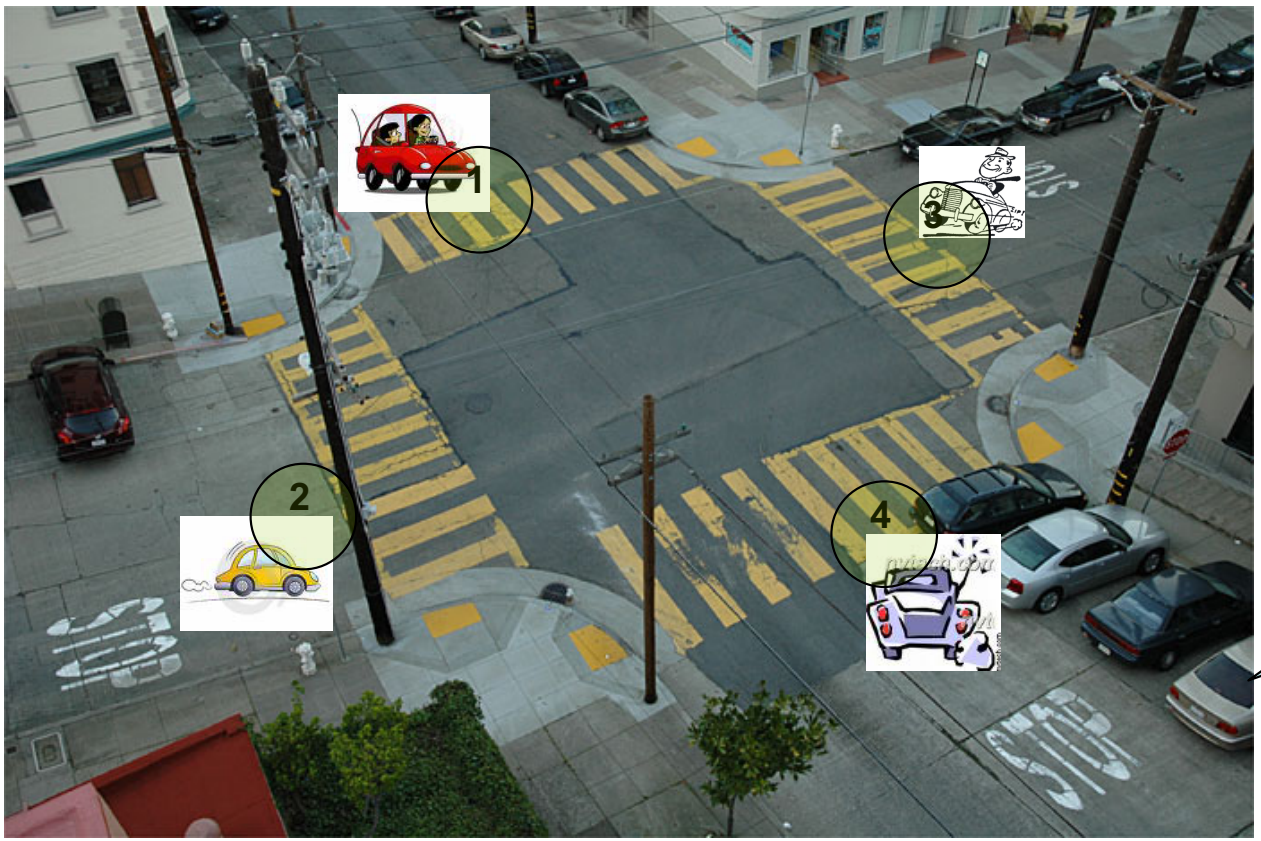
Progress: Traffic Analogy

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



Progress: Traffic Analogy

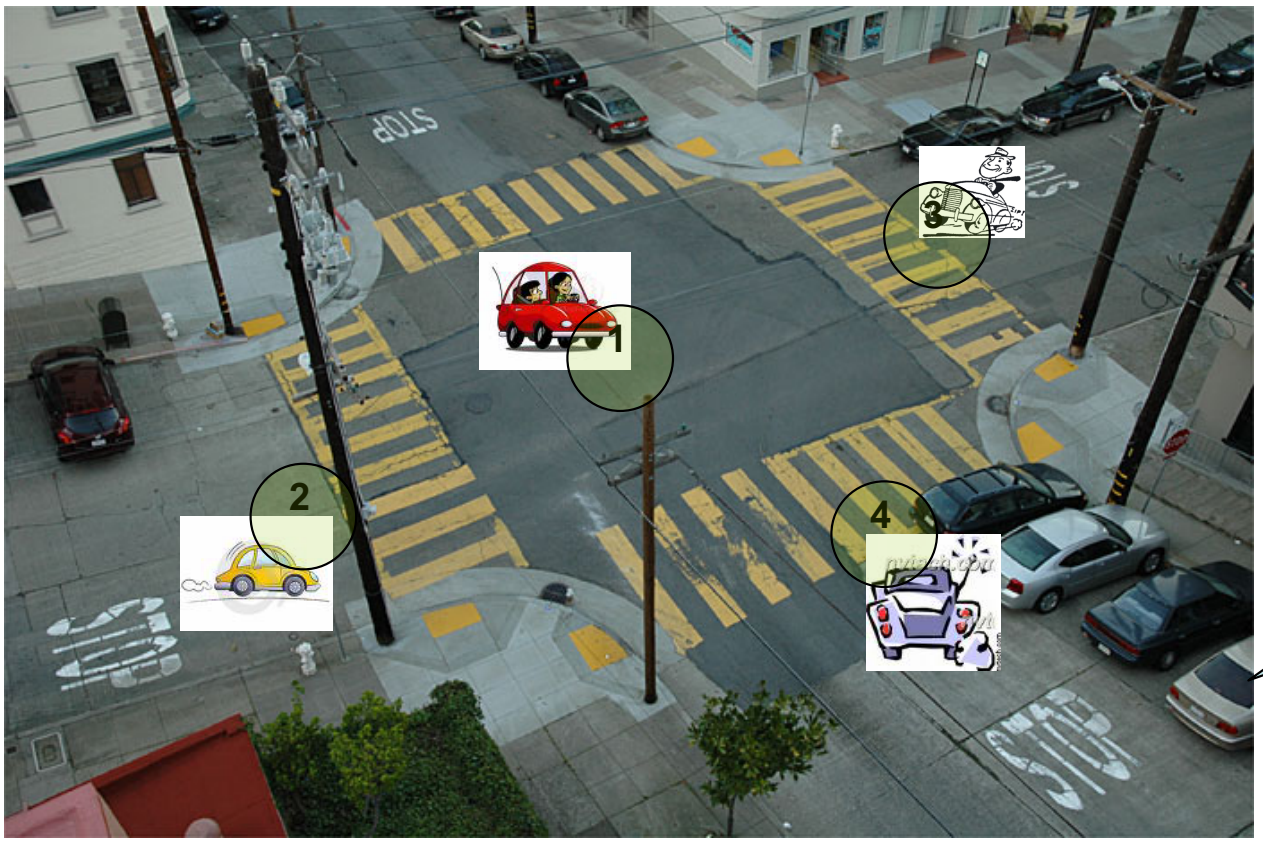
If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely



The first vehicle to the intersection has the right of way. In the event of a tie, the vehicle to the right has the right of way.

Progress: Traffic Analogy

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

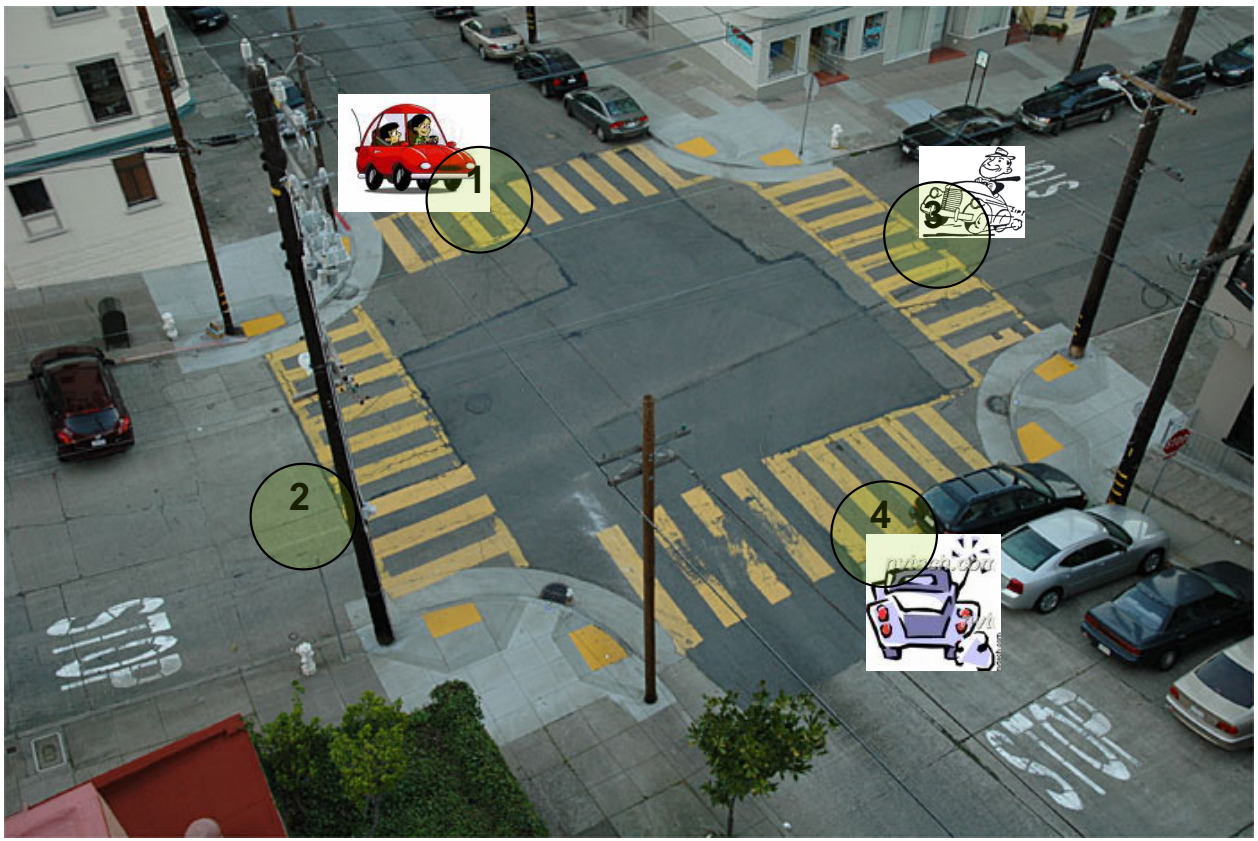


Still need to break a tie if all reach the intersection at the same time.
Coin flip?

Bounded Waiting: Traffic Analogy

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



Any car has to wait at most for 3 others to pass after it stops (fairness)

1. Strict Alternation

```
turn = 0;

T0 {
  while (turn != 0);
   /*********/
  critical section
   /*********/
  turn = 1;
}

T1 {
  while (turn != 1);
   /*********/
  critical section
   /*********/
  turn = 0;
}
```

It works!

Problems:

- requires threads to alternate getting into CS
- does NOT meet Progress requirement.

Fixing the "progress" requirement

```
bool flag[2]; // initialized to FALSE

T0 {
  flag[0] = TRUE;
  while (flag[1] == TRUE)
     ;
  /* critical section */
  flag[0] = FALSE;
}

T1 {
  flag[1] = TRUE;
  while (flag[0] == TRUE)
     ;
  /* critical section */
  flag[1] = FALSE;
}
```

Problem:
Both can set their flags to true and wait indefinitely for the other

violate bounded waiting

int otherid;
otherid = 1-myid;
interested[myid] = true

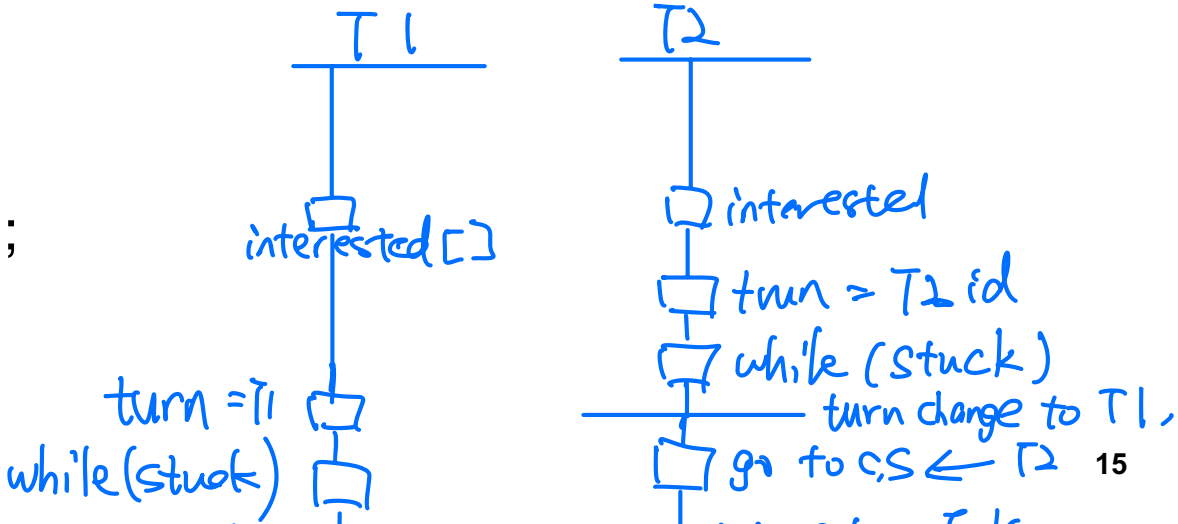
2. Peterson's Algorithm

```
int turn;
int interested[N];  /* all set to FALSE initially */

enter_CS(int myid) {   /* param. is 0 or 1 based on T0 or T1 */
   int other;

   otherid = 1 – myid; /* id of the other thread*/
   interested[myid] = TRUE;
   turn = myid;
   while (turn == myid && interested[otherid] == TRUE)
                ;
}

leave_CS(int myid) {
   interested[myid] = FALSE;
}
```



PENNSTATE

interest[1]=false
T1, run

interest = false

Intuitively …

- This works because a thread can enter CS, either because
  - **Other thread is not even interested in critical section**
  - **Or even if the other thread is interested, this process did the "turn = myid" first.**

first

PENNSTATE

Prove that

- It is correct (achieves mutex)
  - **If both are in critical section, then 1 condition at least must have been false for both threads.**
  - **This has to be the "turn == myid" which cannot be false for both threads.**

PENNSTATE

Prove that

- There is progress
  - **If a thread is waiting in the loop, the other person has to be interested.**
  - **One of the two will definitely get in during such scenarios.**

PENNSTATE

Prove that

- There is bounded waiting
  - **When there is only one thread interested, it gets through**
  - **When there are two threads interested, the first one which did the "turn = myid" statement goes through.**
  - **When the current thread is done with CS, the next time it requests the CS, it will get it only after any other thread waiting at the loop.**

PENNSTATE

- We have looked at only 2 thread solutions.

- How do we extend for multiple threads?

PENNSTATE

More than 2 thread solution

- Analogy to serving different customers in some serial fashion.
  - **Make them pick a number/ticket on arrival.**
  - **Service them in increasing tickets**
  - **Need to use some tie-breaker in case the same ticket number is picked (e.g. larger thread id wins).**

3. Bakery Algorithm

Notation: (a,b) < (c,d) if a<c or a=c and b<d

Every thread has a unique id (integer) Pi

```
bool choosing[0..n-1];
int number[0..n-1];

enter_CS(myid) {
   choosing[myid] = TRUE;
   number[myid] = max(number[0],number[1], .... number[n-1]) + 1;
   choosing[myid] = FALSE;
   for (j=0 to n-1) {
         while (choosing[j])
               ;
         while (number[j] != 0) && ((number[j],Pj)<(number[myid],myid))
               ;
   }
}
leave_CS(myid) {
   number[myid] = 0;
}
```

PENNSTATE

Exercise

- Show that it meets
  - **Mutex**
  - **Progress**
  - **Bounded waiting requirements**