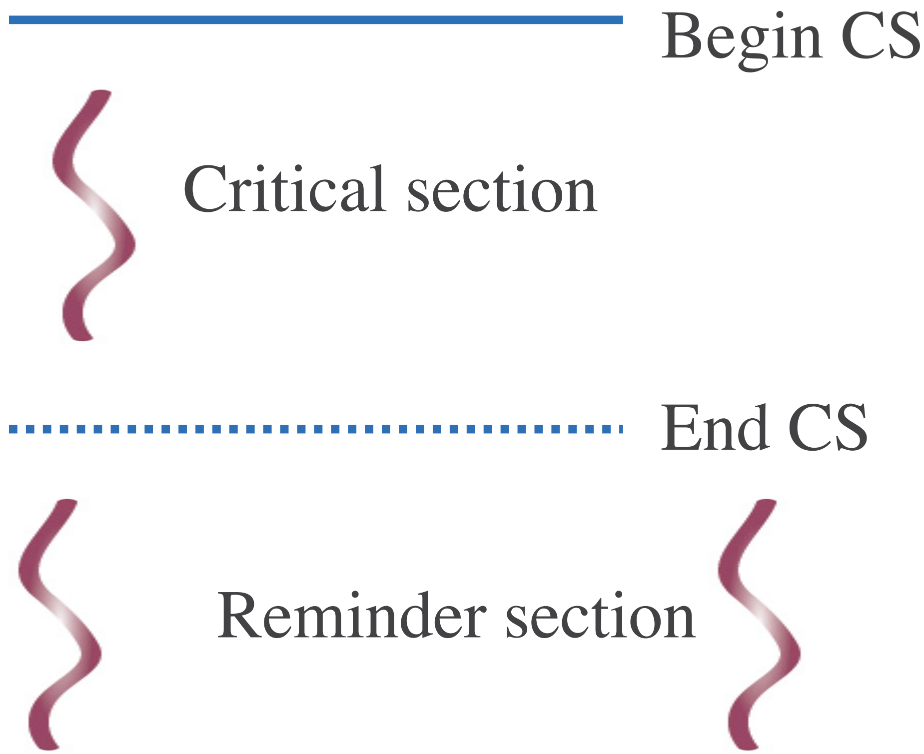
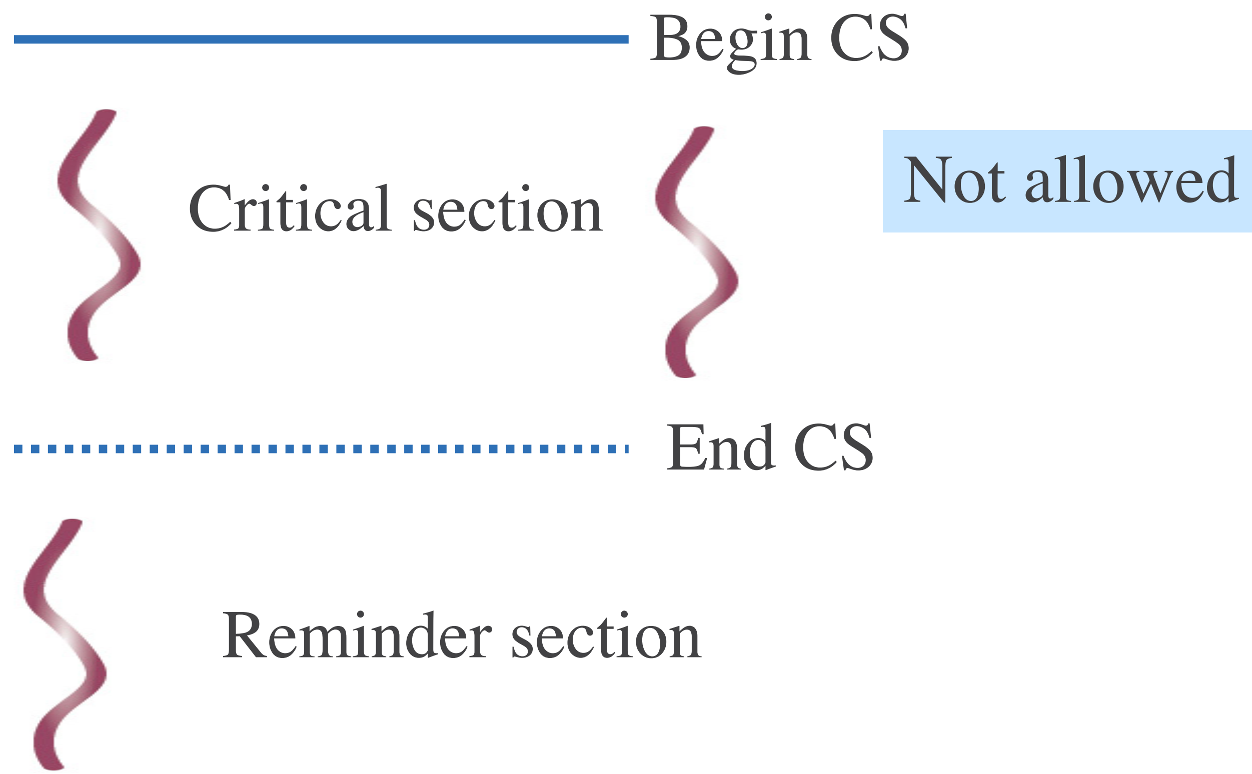
Critical section problem

- Mutual exclusion:



Critical section problem

- Mutual exclusion:



Critical section problem

- Progress

They decide on whom to enter

Begin CS

Critical section

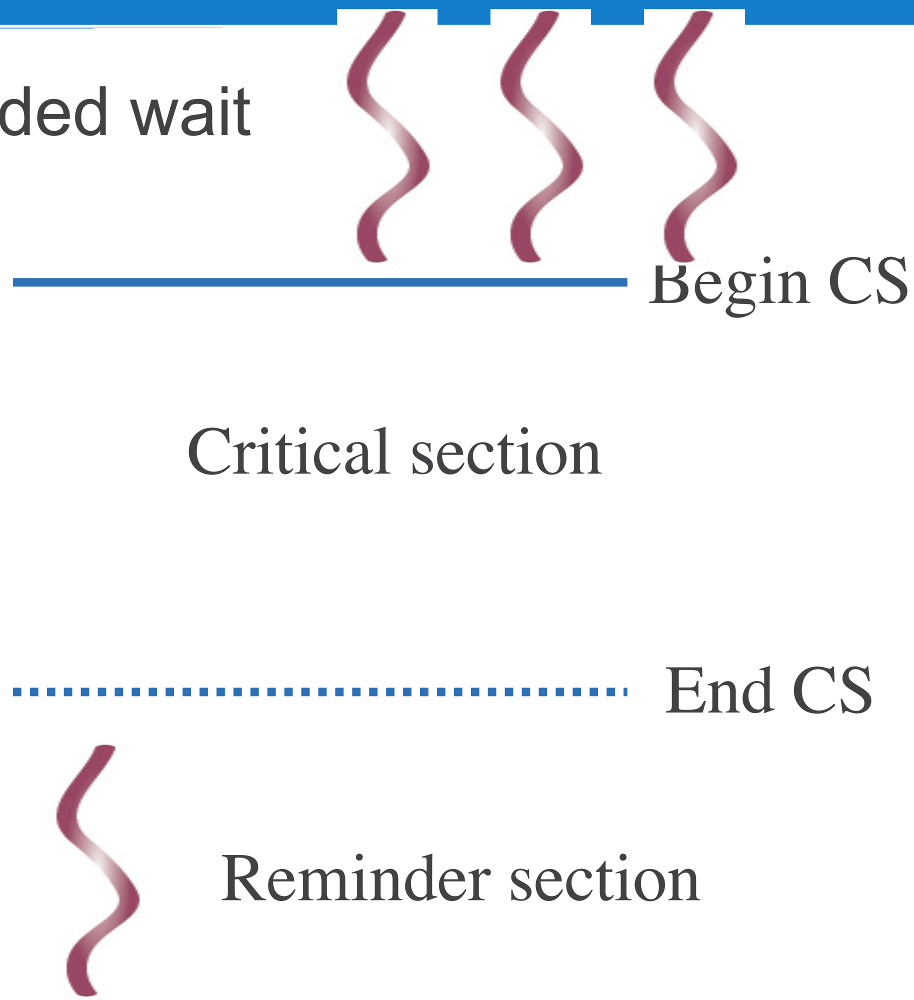
End CS

Reminder section

Not allowed to decide

Critical section problem

- Bounded wait



Classic s/w soln: Peterson's Solution

- Restricted to two processes
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (not true for modern processors)
- The two threads 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 P_i is ready!

Algorithm for Process P_i

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

1) Mutual exclusion because only way thread enter critical section when flag[j] == FALSE or turn == TRUE
2) Only way to enter section is by flipping flag[] inside loop
3) turn = j allows the other thread to make progress

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Have to wait for disable to propagate to all processors
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution using TestAndSet

- Definition of TestAndSet:

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

- Shared boolean variable *lock*., initialized to false.
- Solution:

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

Solution using Swap

- Definition of Swap:

```
void Swap (boolean *a, boolean *b) {
        boolean temp = *a;
        *a = *b;
        *b = temp:
  }
```

- Shared Boolean variable *lock* initialized to FALSE; Each process has a local Boolean variable *key*.
- Solution:

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

Solution with TestAndSet and bounded wait

- boolean waiting[n]; boolean lock; initialized to false

Pi can enter critical section iff waiting[i] == false or key == false

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet (&lock);
        waiting[i] = FALSE;
        // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

Classic synchronization problems

- Bounded buffer problem
- Readers-writer problem
- Dining-philosophers problem
- **The Sleeping Barber problem**

Bounded buffer problem

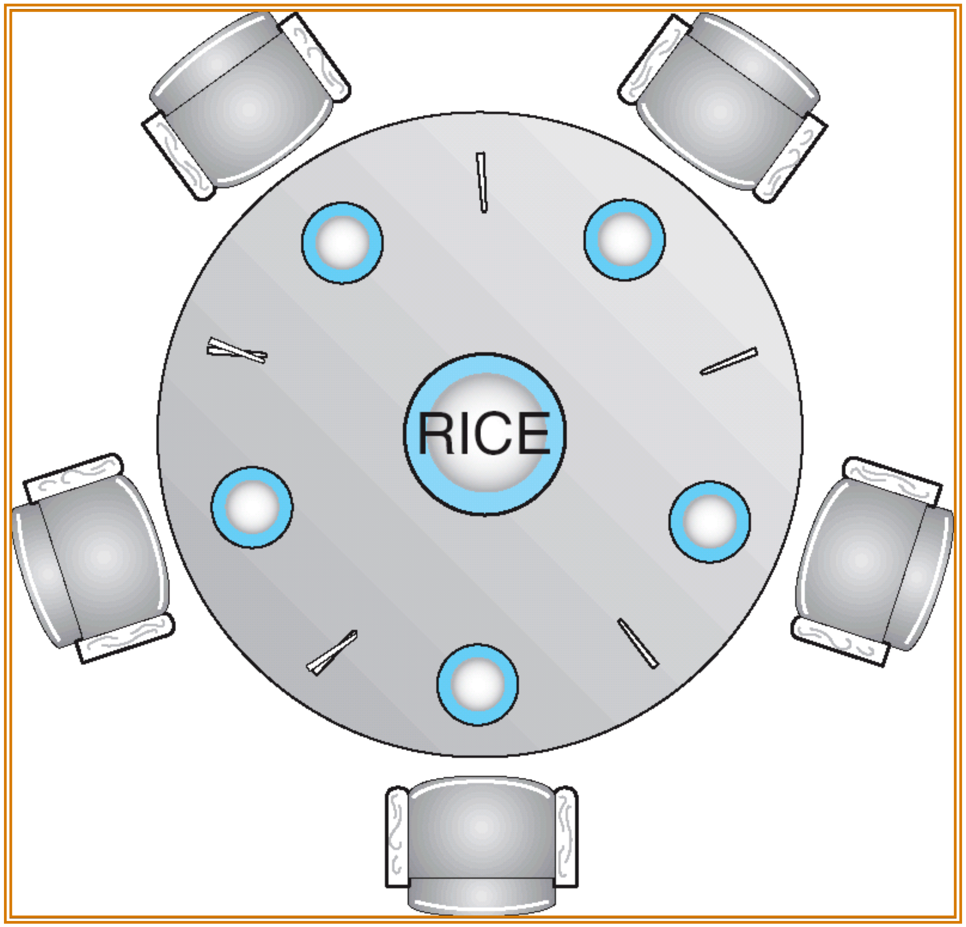
- N element buffer, producer and consumers work with this buffer
- Consumers cannot proceed till producer produced something
- Producer cannot proceed if buffer == N

Reader-writer problem

- Shared database, any number of readers can concurrently read content. Only one writer can write at any one time (with exclusive access)
- Variations:
  - No reader will be kept waiting unless a writer has already received exclusive write permissions
  - Once a writer is ready, it gets exclusive permission as soon as possible. Once a writer is waiting, no further reads are allowed

Dining philosopher's problem

- five philosophers think for some time and then eat
  - Philosophers can only eat if they have both their left and right chopsticks/forks/ at the same time



The Sleeping Barber Problem

- A barbershop consists of a waiting room with `N` chairs, and the barber room containing the barber chair. If there are no customers to be served the barber goes to sleep. If a customer enters the barbershop and all chairs are busy, then the customer leaves the shop. If the barber is busy, then the customer sits in one of the available free chairs. If the barber is asleep, the customer wakes the barber up.

Semaphore synchronization primitive

- TestAndSet are hard to program for end users
- Introduce a simple function called semaphore:
  - Semaphore is an integer, S
  - Only legal operations on S are:
    - Wait() [atomic] - if S > 0, decrement S else loop
    - Signal() [atomic] - increment S
  - wait (S) {

```
wait (S) {
        while S <= 0
            ; // no-op
          S--;
 }
```

  - signal (S) {

```
signal (S) {
   S++;
}
```

  - Counting (S: is unrestricted), binary (mutex lock) (S: 0, 1)

Semaphore usage example

- Assume synch is initialized to 0
  - P2:

    Wait(synch);

    Statements2;
  - P1:

    Statements1;

    signal(synch);

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
                 …
    procedure Pn (…) {……}
     Initialization code ( ….) { … }
                 …
    }
}
```

- In Java, declaring a method *synchronized* to get monitor like behavior
  - What happens to shared variables which are not protected by this monitor?

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()