CS343: Operating System

Synchronization

Lect19 : 12th Sept 2023

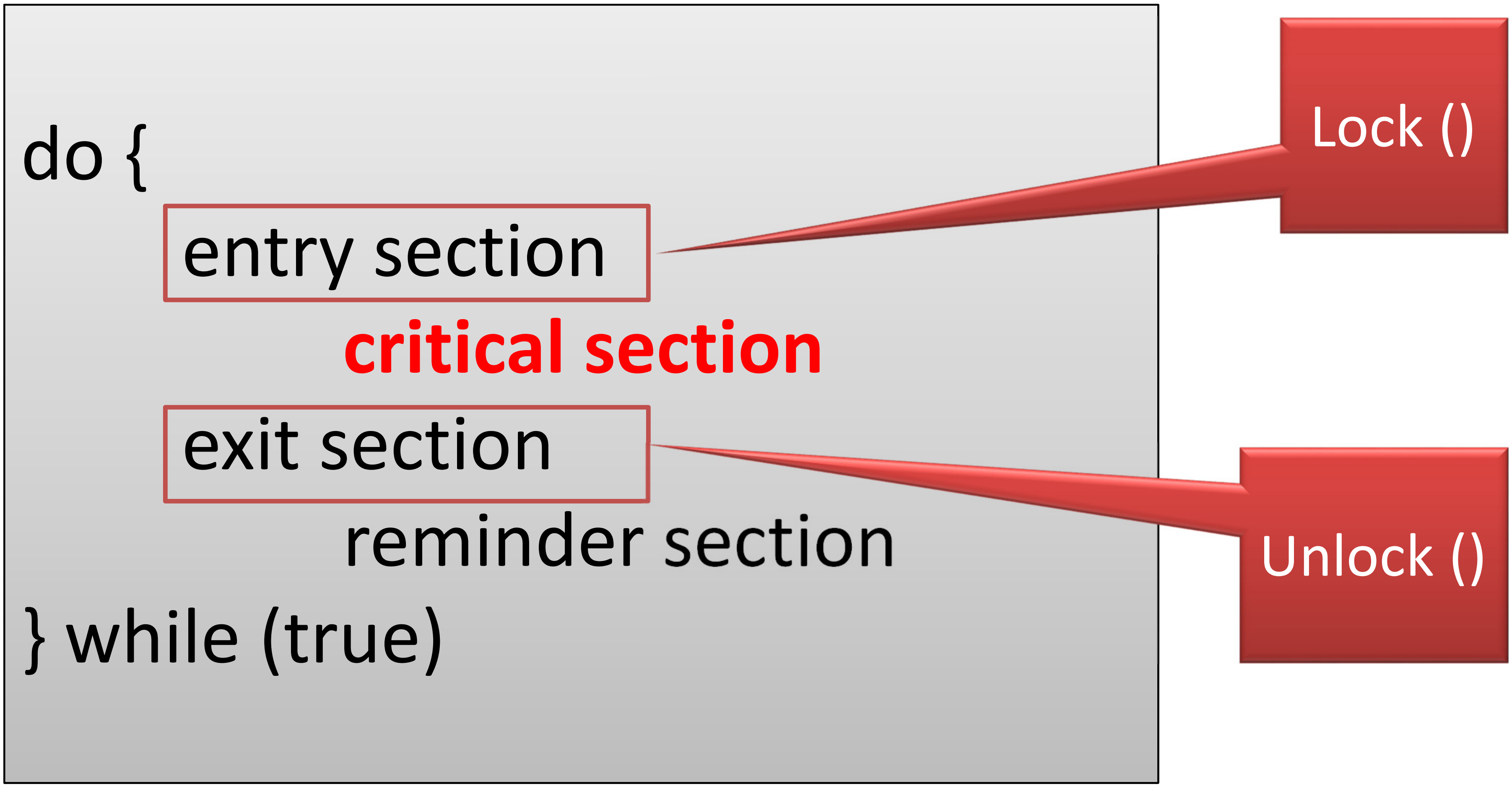
Dr. A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Synchronization
  - Critical Section Problem
- Solution to CS Problems
  - Two Threads Solutions: Peterson's Solution
  - N Thread Solutions: Filter and Bakery Algorithms
- Sync Hardware
  - CAS, TAS, LL-LC, BackupLock

Critical Section

- General structure of process $\boldsymbol{P_i}$



Peterson's Algorithm: Combine LockOne and LockTwo

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while(flag[j]&&victim==i) {};
  }
public void unlock() {
  flag[i] = false;
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

No longer interested

Peterson's Lock: Lock 3

- Satisfy Mutual Exclusion
- Satisfy Deadlock Free
- Satisfy Starvation Free

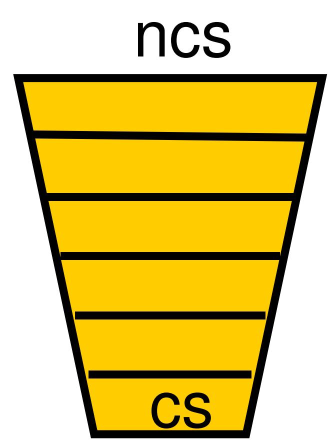
  – Proof …..

Nthread Synchronization

Filter Algorithm for *n* Threads

There are n-1 “waiting rooms” called levels

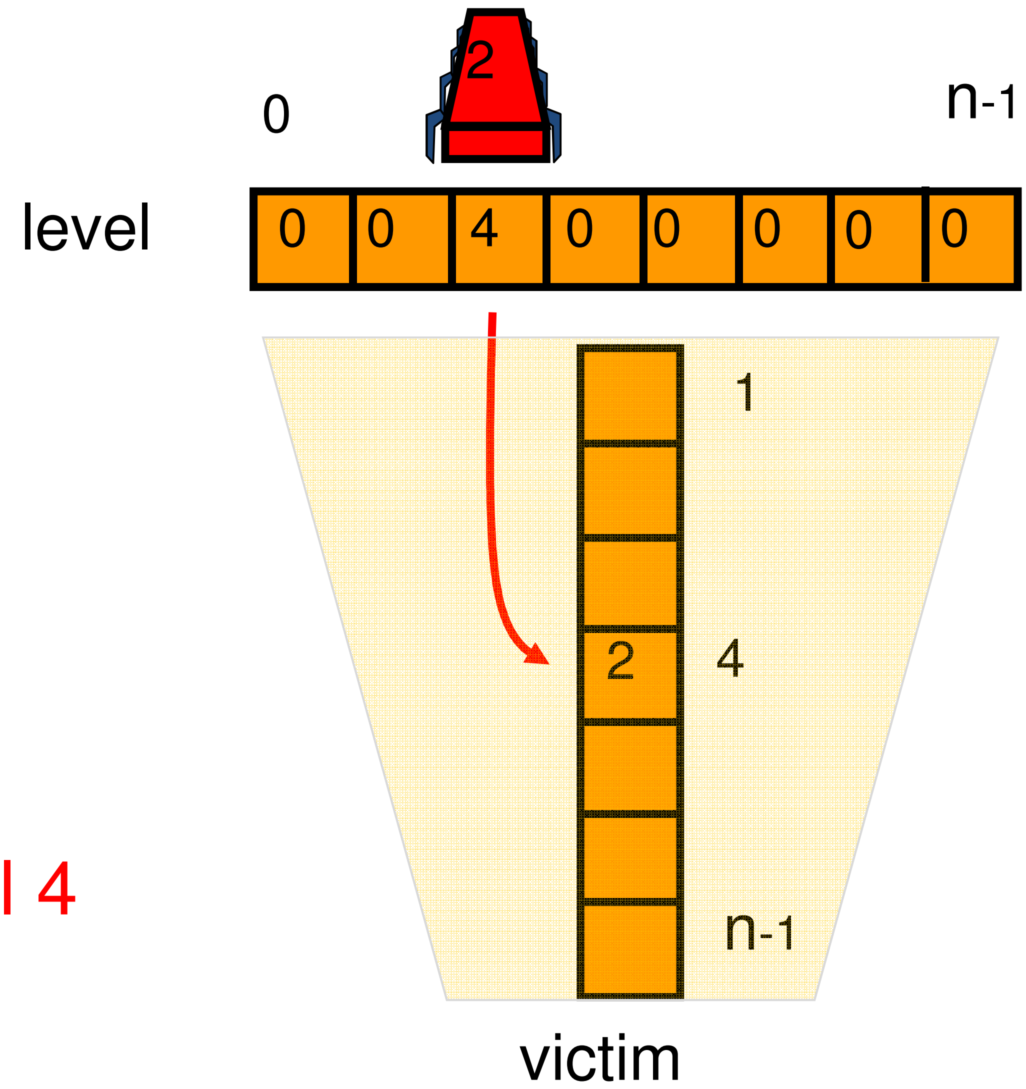
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



Filter

```
class FilterLock {
   int level[n];// level[i] for thread i
   int victim[n];// victim[L] for level L
  public FilterInit(int n) {
     for(int i=1;i<n;i++)
        level[i]=0;
     }}
   ….
}
```

Filter



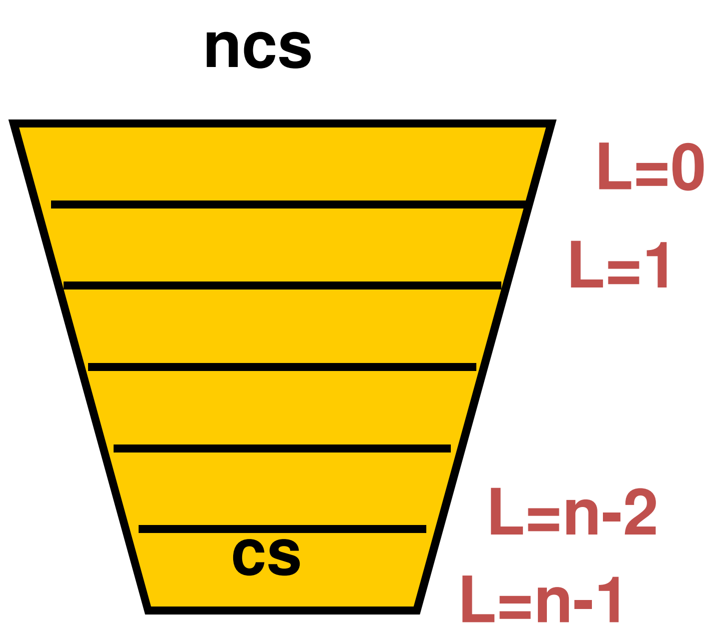
Thread 2 at level 4

Filter

```
class FilterLock {
  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃ k!=i level[k]>=L) &&
             victim[L]==i) {};
      }
    }
  }
  public void unlock(){ level[i]=0;}
}
```

Claim: Mutex

- Start at level L=0
- At most n-L threads enter level L
- Mutual exclusion at level L=n-1



No Starvation

- Filter Lock satisfies properties:
  - Just like Peterson Alg at any level
  - So no one starves
- But what about fairness?
  - **Threads can be overtaken by others**

Bounded Waiting

- Want stronger fairness guarantees
- Thread not “overtaken” too much
- If A starts before B, then A enters before B?
- But what does “start” mean?
- Need to adjust definitions ....

Bakery Algorithm

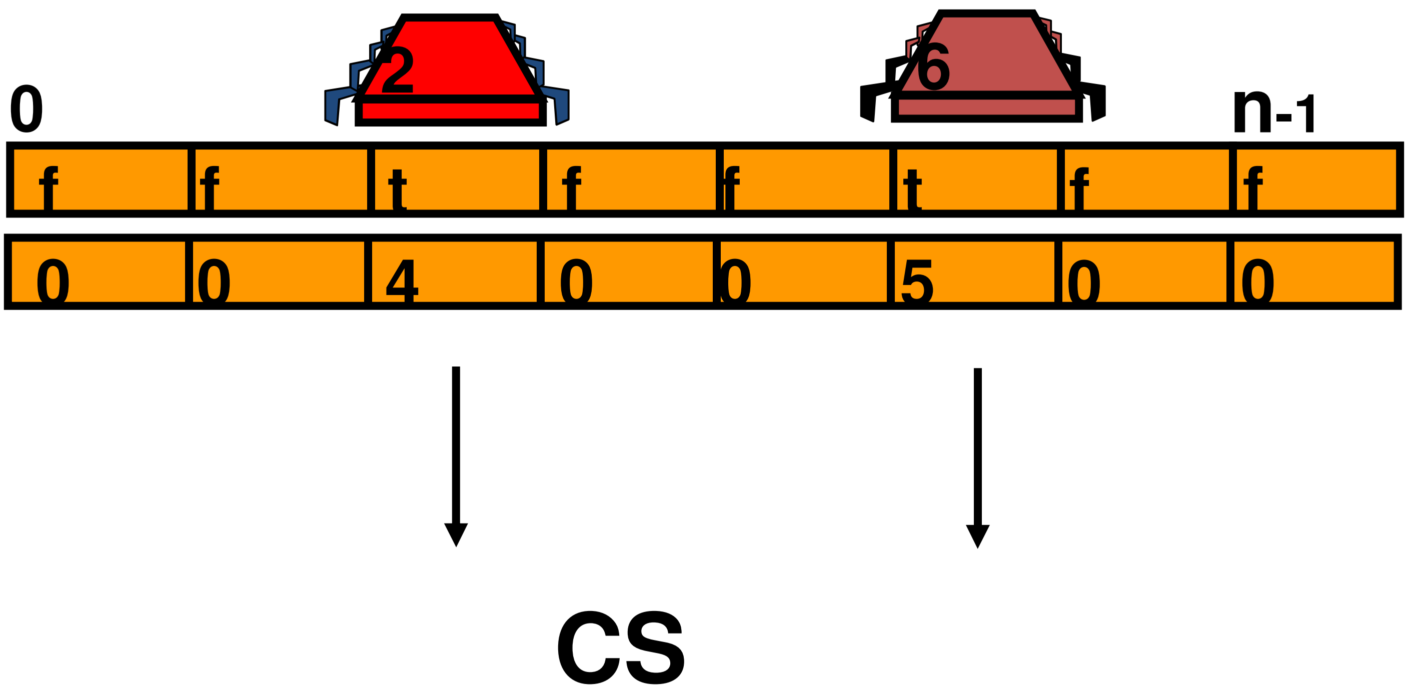
- Similar to Bakery Shop
- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$
    - If $a > b$, or $a = b$ and $i > j$

Bakery Algorithm

```
class BakeryLock {
   bool flag[n];
   int label[n];
  public BakeryLockInit(int n) {
   for(int i = 0; i < n; i++) {
       flag[i]= false; label[i] = 0;
    }
  }
 …
```

Bakery Algorithm

```
class Bakery Lock {
   bool flag[n];
   int label[n];
```



Bakery Algorithm

```
class BakeryLock {
  public void lock() {
  flag[i]=true;
  label[i]=max(label[0],…,label[n-1])+1;
  while ( ∃k flag[k] &&
           label[i] > label[k]){}
 }
```

Mutual Exclusion

- Suppose A and B in CS together
- Suppose A has earlier label
- When B entered, it must have seen
  - flag[A] is *false*, or
  - label[A] > label[B]

No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)

```
label[i]=max(label[0],…,label[n-1])+1;
```

Synchronization Hardware

Synchronization Hardware

- Many systems provide hardware support for Sync.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

Synchronization Hardware

- Multiprocessor–disable interrupts
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Atomic Sync Instructions

- Test and Set (**TAS**)
- Compare and Swap (**CAS**)
- Exchange (**XCHG**)
- Fetch and Increment (**FAI**)
- **How to provide these in MP/MC?**
  - Load Locked & Store Conditional (**LL-SC**)

test_and_set Instruction : TAS

```
//Definition:
boolean test_and_set (boolean *target) {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
      while (test_and_set(&lock))
      ; /* do nothing */

            /* critical section */

      lock = false;
             /* remainder section */
   } while (true);
```

CAS Instruction

```
int CAS(int *value,
      int expected, int new_value){

         int temp = *value;
         if (*value == expected)
            *value = new_value;
          return temp;
}
```

- Executed atomically
- Returns the original value of passed parameter “value”
- Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Sync Solution using CAS

- Shared integer “lock” initialized to 0;

```
do {
  while (CAS(&lock, 0, 1) != 0)
  ; /* do nothing */

   /* critical section */
        lock = 0;
     /* remainder section */

      } while (true);
```

Synchronization Instruction

- Hardware primitive for atomic read+write is required e.g.
  - Exchange, (XCHG)
  - Test & Set (TAS)
    - // test for unlock (0) then set the lock (1)
  - Fetch & Increment (FAI)

Spin Lock with Exchange Instr.

Lock: 0 indicates free and 1 indicates locked

Code to lock X :

SPINing

Try to test and acquire the lock in a tight loop

r2 = 1

lockit: r2 $\leftrightarrow$ X **//atomic exchange**

if(r2$\neq$0) **goto** lockit //already locked

locks are cached for efficiency, coherence is used in shared multiprocessor

Spin Lock with Exchange Instr.

Better code to lock X :

```
lockit:     r2 = X;              // read lock
            if(r2≠0) goto lockit ; //not available
            r2 = 1;
            r2 ↔ X   ;          //atomic exchange
            if(r2≠0) goto lockit ; //already locked
```

LD Locked & ST conditional

Time

LL r1 X // Reading from a location X

//do some operation

SC r2 X // Storing r2 to location X

if unsuccessful r2 ==1

Store will be unsuccessful if values of X is altered/changed by others processor between time of LL and time of SC you have done

Load linked/Store Conditional

Spin Lock with LL & SC

```
lockit:
          LL r2, X                       //load locked
          if(r2≠0) goto lockit ;         //not available
           r2 = 1 ;
          SC r2, X                        //store conditional
           if(r2==1) goto lockit ;       // store fails redo
```

Spin lock with exponential back-off reduces contention

Atomic XCHG with LL & SC

Simpler to implement

- **Atomic exchange using LL and SC**

```
try:    r3 = r2;                  //move exchange value
        LL r1, X                  // load locked
        SC r3, X                  //store conditional
        if(r3==1) goto try ;      //store fails redo…
        r2 =r1;                   //put loaded value in r2
```

Atomic FAI with LL & SC

Simpler to implement

- Fetch & increment using LL and SC

```
try:    LL r1, X                    //load locked
        r3 = r1 + 1;                //increment
        SC r3, X                    //store conditional
        if(r3==1) goto try ;        //store fails redo…
```