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 Problem

- Consider system of ***n*** processes

$$\{p_0, p_1, \dots p_{n-1}\}$$

- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section

Critical Section Problem

- ***Critical section problem*** is to design protocol to solve this
  - Each process must ask permission to enter critical section in **entry section**,
  - May follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P****ᵢ*

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (true)
```

Lock ()

Unlock ()

Solution to Critical-Section Problem

- **Mutual Exclusion**
- **Progress**
- **Bounded Waiting**

Solution to Critical-Section Problem

- **Mutual Exclusion**
  - If process P_i is executing in its CS
  - Then no other processes can be executing in their CS
- **Progress : Deadlock free**
  - If no process is running in its CS and there exist some processes that wish to enter their CS,
  - Then the selection of the processes that will enter the CS next cannot be postponed indefinitely

Solution to Critical-Section Problem

- **Bounded Waiting : Starvation**
  - A bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non- preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Locking Algorithms

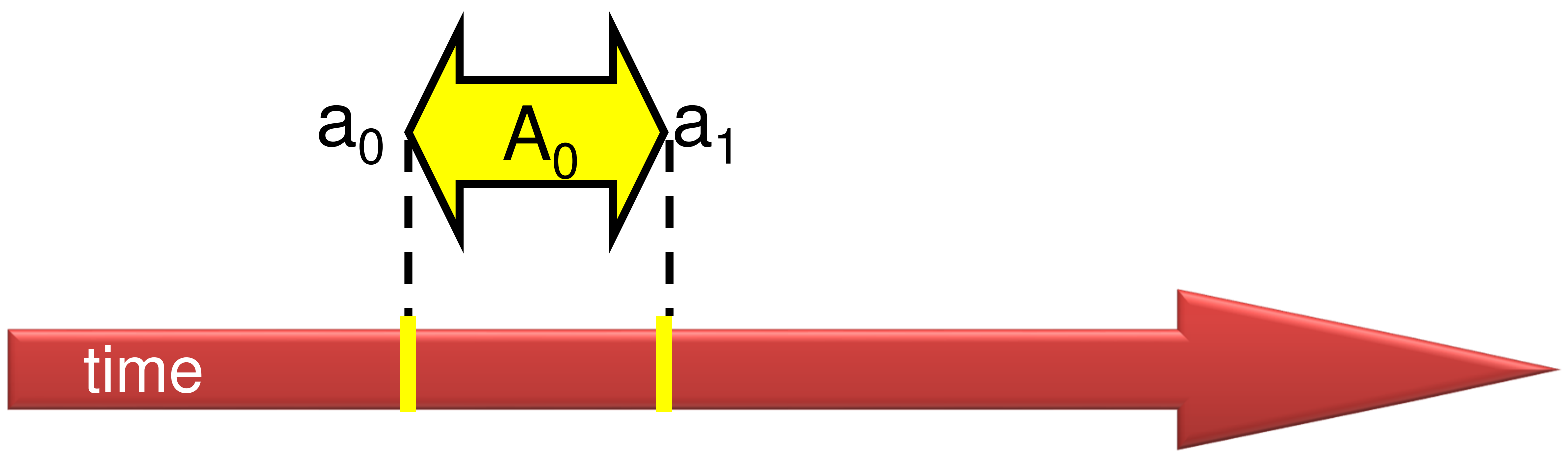
Ref: Galvin Book and "*Art of Multiprocessor Programming*" by Herlihy and Shavit

Events

- An *event* a_0 of thread A is
  - Instantaneous
  - No simultaneous events (break ties)

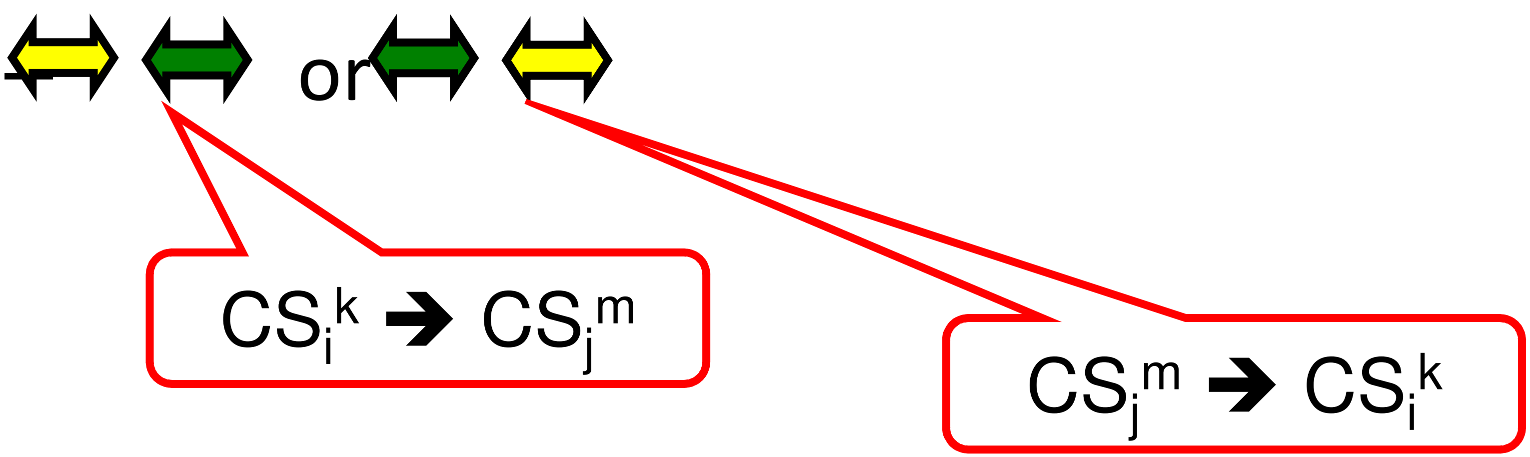
Intervals

- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Mutual Exclusion

- Let CS_i^k be thread i's k-th critical section execution
- And CS_j^m be j's m-th execution
- Then either



Two-Thread vs *n*-Thread Solutions

- 2-thread solutions first
  - Illustrate most basic ideas
  - Fits on one slide
- Then *n*-thread solutions

LockOne : Algorithm 1

```
class LockOne {
bool flag[2]={0,0};
public void lock() {
    int j,i=ThreadID.get();j=1-i;
    flag[i] = true;
    while (flag[j]) {}
 }
public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
}
```

LockOne : Algorithm 1

```
class LockOne {
bool flag[2]={0,0};
public void lock() {
    int j,i=ThreadID.get();j=1-i;
    flag[i] = true;
    while (flag[j]) {}
 }
public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
}
```

Each thread has flag

LockOne : Algorithm 1

```
class LockOne {
bool flag[2]={0,0};
public void lock() {
    int j,i=ThreadID.get();j=1-i;
    flag[i] = true;
    while (flag[j]) {}
 }
public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
}
```

Set my flag

LockOne : Algorithm 1

```
class LockOne {
bool flag[2]={0,0};
public void lock() {
    int j,i=ThreadID.get();j=1-i;
    flag[i] = true;
    while (flag[j]) {}
 }
public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
}
```

Wait for other flag to become false

LockOne : Algorithm 1

```
class LockOne {
bool flag[2]={0,0};
public void lock() {
    int j,i=ThreadID.get();j=1-i;
    flag[i] = true;
    while (flag[j]) {}
 }
public void unlock() {
    int i = ThreadID.get();
    flag[i] = false;
}
```

I have released

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

- $\text{write}_A(\text{flag[A]=true}) \rightarrow \text{read}_A(\text{flag[B]==false}) \rightarrow \text{CS}_A$

- $\text{write}_B(\text{flag[B]=true}) \rightarrow \text{read}_B(\text{flag[A]==false}) \rightarrow \text{CS}_B$

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

From the Assumption

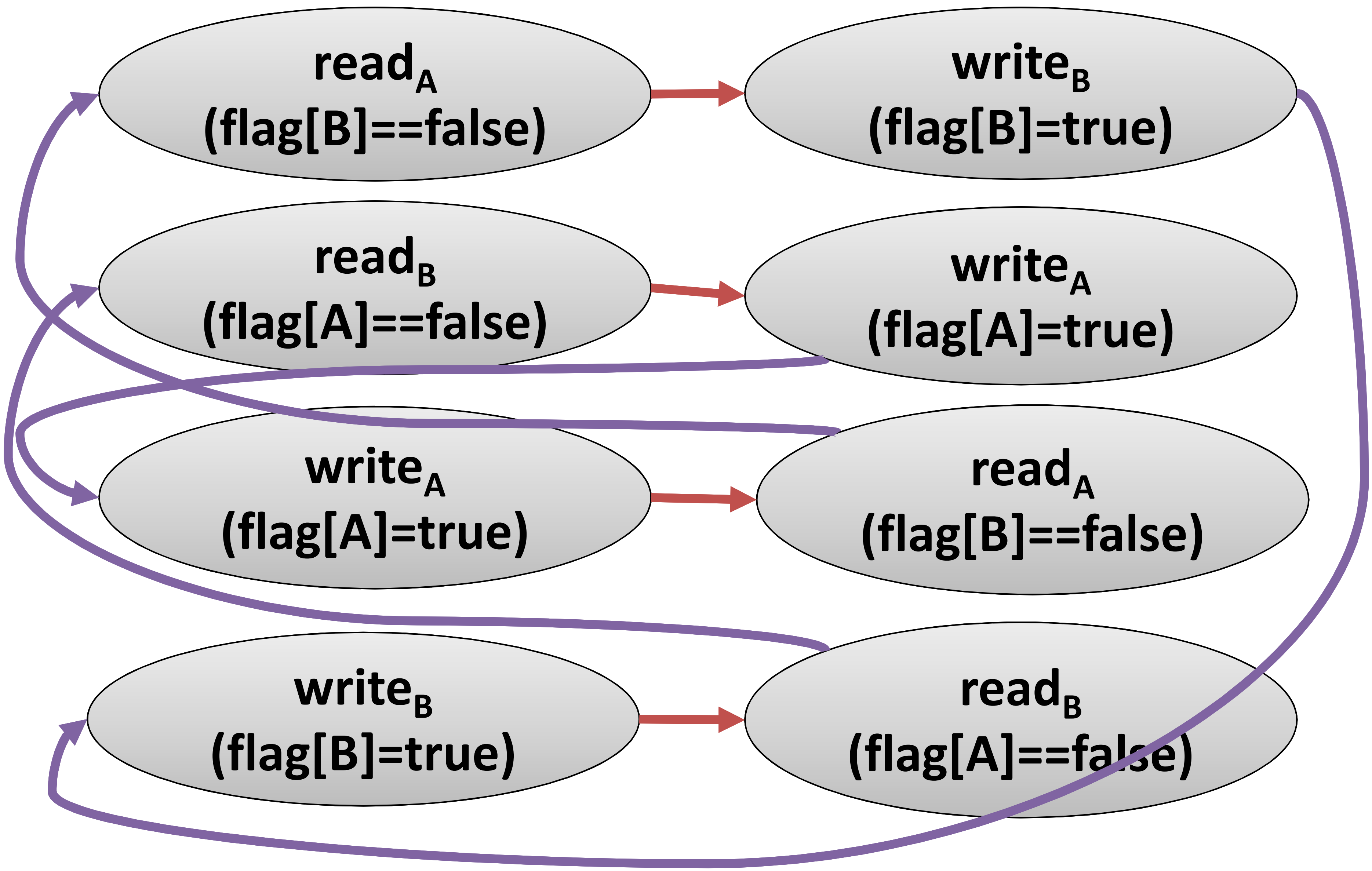
- Load Store are atomic
  - It will behaves the Partial Order

- $\text{read}_A(\text{flag[B]==false}) \rightarrow \text{write}_B(\text{flag[B]=true})$
- $\text{read}_B(\text{flag[A]==false}) \rightarrow \text{write}_A(\text{flag[A]=true})$

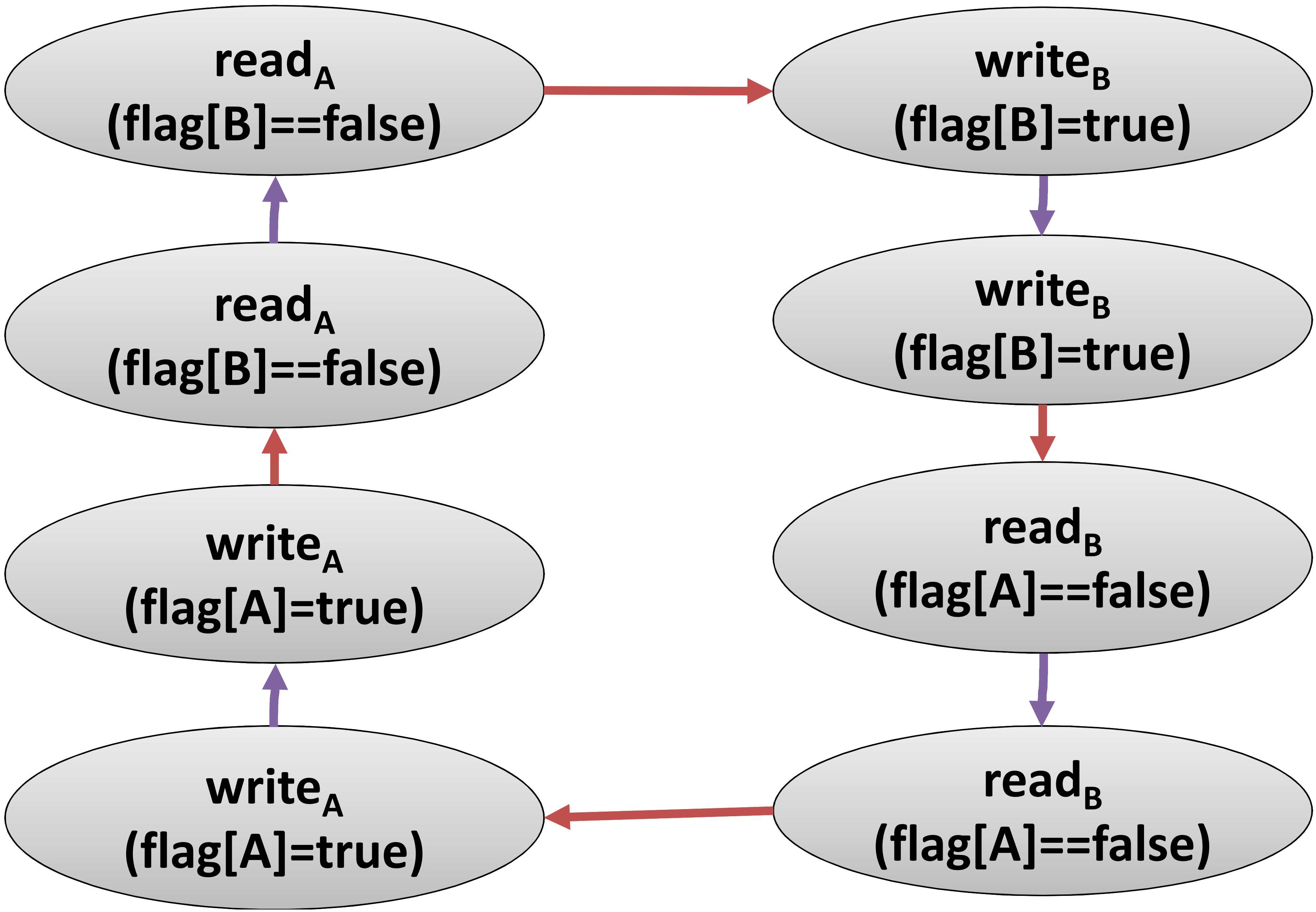
Combining

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

Combining

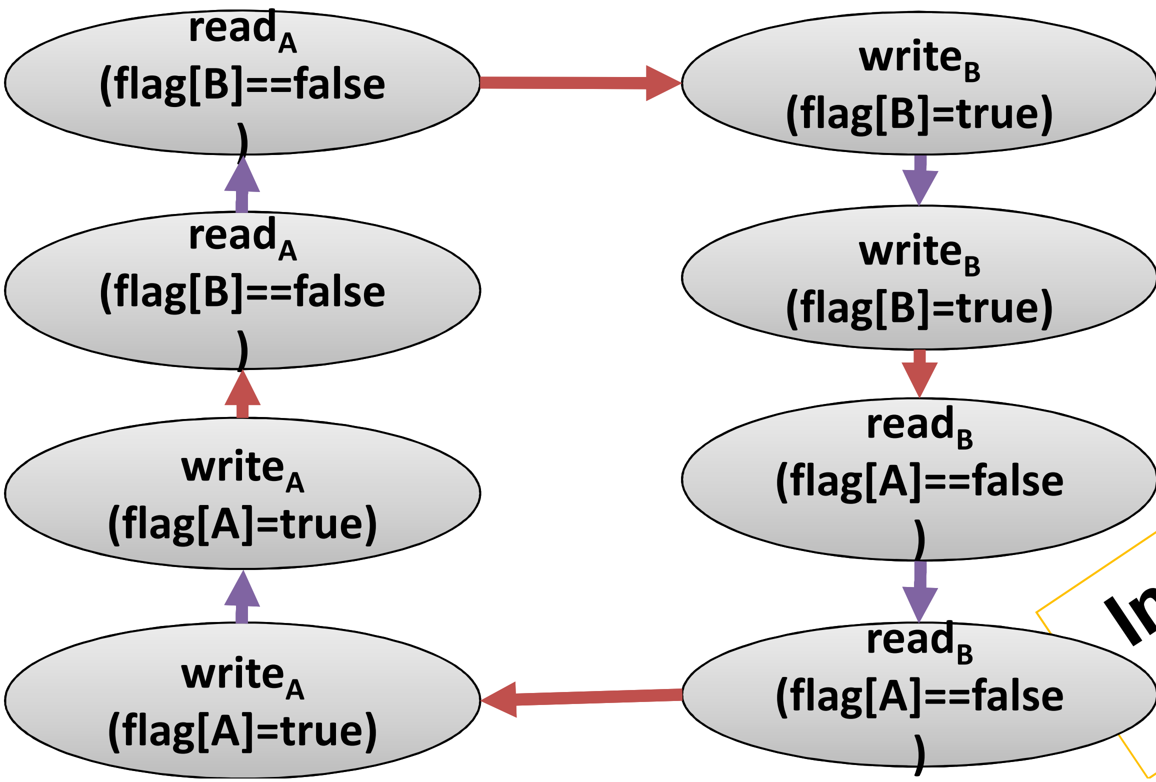

read$_A$ (flag[B]==false)
write$_B$ (flag[B]=true)
read$_B$ (flag[A]==false)
write$_A$ (flag[A]=true)
write$_A$ (flag[A]=true)
read$_A$ (flag[B]==false)
write$_B$ (flag[B]=true)
read$_B$ (flag[A]==false)

Combining: Rearranging



Cycle!

Satisfy
Mutual Execution
Proved



Impossible in a partial order

Deadlock Freedom: Progress

- LockOne Fails deadlock-freedom
  - Sequential executions OK
  - Concurrent execution

```
//Thread A

flag[i] = true;
while (flag[j]){}
```

```
//Thread B

flag[j] = true;
while (flag[i]){}
```

Deadlock Freedom: Progress

- LockOne Fails deadlock-freedom
  - Sequential executions OK
  - Concurrent execution can deadlock

```
//Thread A

flag[i] = true;
while (flag[j]){}
```

```
//Thread B

flag[j] = true;
while (flag[i]){}
```

Both will wait for each other to finish

Lock Two: Algorithm 2

```
class LockTwo  {
int victim;
 public void lock() {
  int i = ThreadID.get();
  victim = i;
  while (victim == i) {};
 }
 public void unlock() {}
}
```

Lock Two: Algorithm 2

```
class LockTwo {
int victim;
 public void lock() {
   int i = ThreadID.get();
   victim = i;
   while (victim == i) {};
 }
 public void unlock() {}
}
```

Let other go first

Lock Two: Algorithm 2

```
class LockTwo {
int victim;
 public void lock() {
  int i = ThreadID.get();
  victim = i;
  while (victim == i) {};
 }
 public void unlock() {}
}
```

Wait for permission

Lock Two: Algorithm 2

```
class LockTwo {
 private int victim;
 public void lock() {
  int i = ThreadID.get();
  victim = i;
  while (victim == i) {};
 }
 public void unlock() {}
}
```

Nothing to do

LockTwo Claims

- Satisfies mutual exclusion
  - If thread **`i`** in CS, then **`victim == j`**
  - Cannot be both 0 and 1
- Not deadlock free
  - **Concurrent execution does not :(**
  - **Sequential execution deadlocks**

```
public void LockTwo() {
  victim = i;
  while (victim == 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;
}
```

Peterson's Algorithm: Combine LockOne and LockT

Announce I'm interested

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

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

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

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 .....

Mutual Exclusion

(1) $\text{write}_B(\text{Flag}[B]=\text{true}) \rightarrow \text{write}_B(\text{victim}=B)$

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

From the Code

Also from the Code

(2) $\text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B])$
$\rightarrow \text{read}_A(\text{victim})$

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

Assumption

(3)

$write_B(victim=B) \rightarrow write_A(victim=A)$

Without Loss of Generality (WLOG) assume A is the last thread to write victim

Combining Observations

(1) $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{write}_B(\text{victim}=B)$

(3) $\text{write}_B(\text{victim}=B) \rightarrow \text{write}_A(\text{victim}=A)$

(2) $\text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B])$
$\rightarrow \text{read}_A(\text{victim})$

Combining Observations

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

(2) $\text{write}_A(\text{victim=A}) \rightarrow \text{read}_A(\text{flag[B]})$
$\rightarrow \text{read}_A(\text{victim})$

Combining Observations

(1) $\text{write}_B(\text{flag[B]=true}) \rightarrow$

(3) $\text{write}_B(\text{victim=B}) \rightarrow$

(2) $\text{write}_A(\text{victim=A}) \rightarrow \text{read}_A(\text{flag[B]})$
$\rightarrow \text{read}_A(\text{victim})$

Thread A read flag[B] == true and victim == A, so Thread A could not have entered the CS

Deadlock Free

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

- Thread blocked
  - only at `while` loop
  - only if other's flag is true
  - only if it is the **victim**
- IF (other's flag is false) then

  one or the other not the victim

Starvation Free

- Thread *i* blocked only if *j* repeatedly re-enters so that **flag[*j*]==true** and **victim==*i***
- When *j* re-enters
  – it sets **victim** to *j*.
  – So *i* gets in

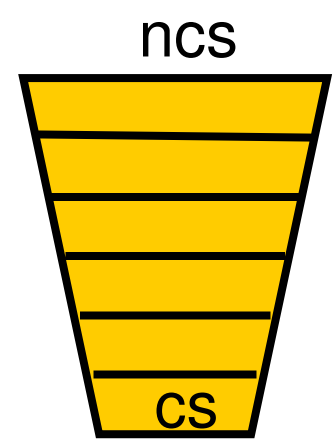
```
public void lock() {
  flag[i] = true; victim = i;
  while(flag[j] &&  victim==i){};
}
public void unlock() {
  flag[i] = false;
}
```

Nthread Synchronization

Filter Algorithm for *n* Threads

There are n-1 “waiting rooms” called levels

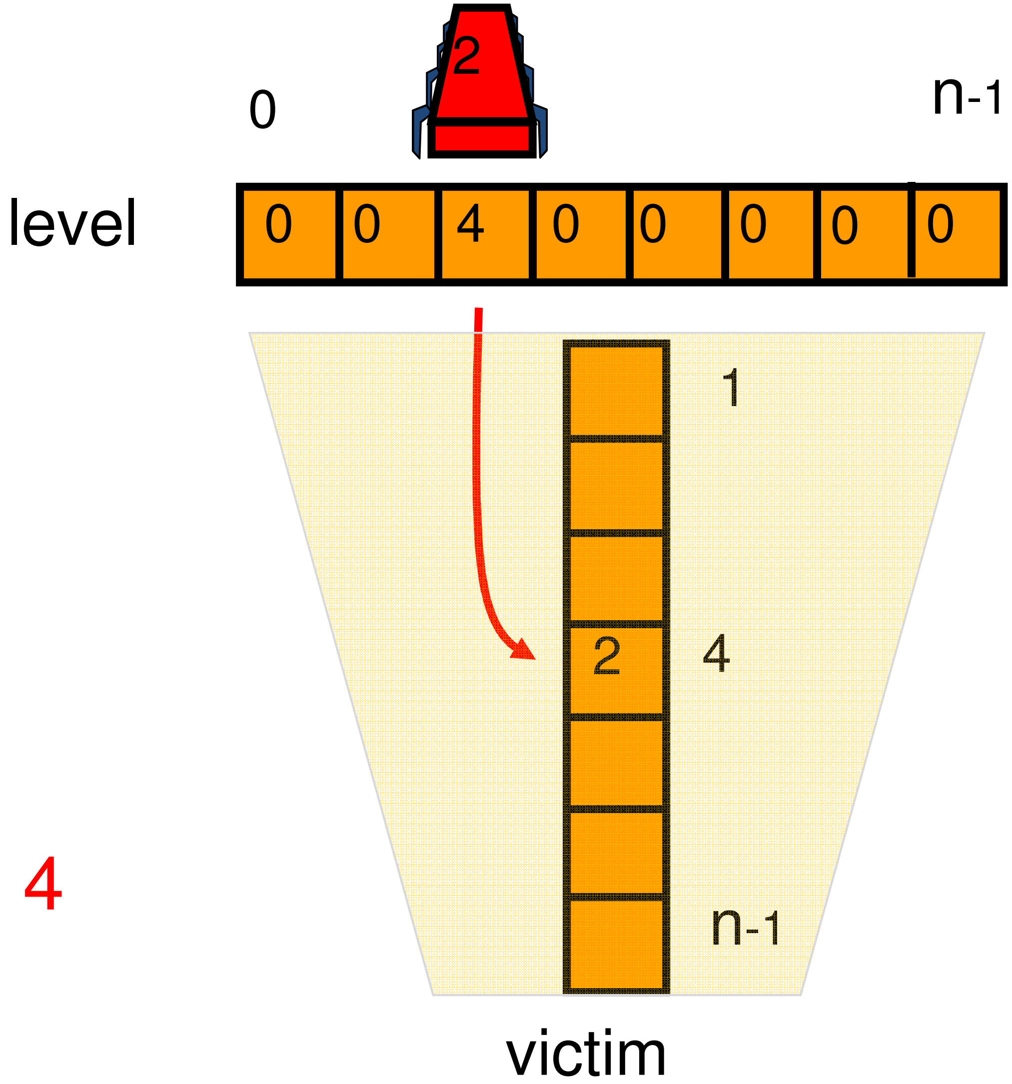
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



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



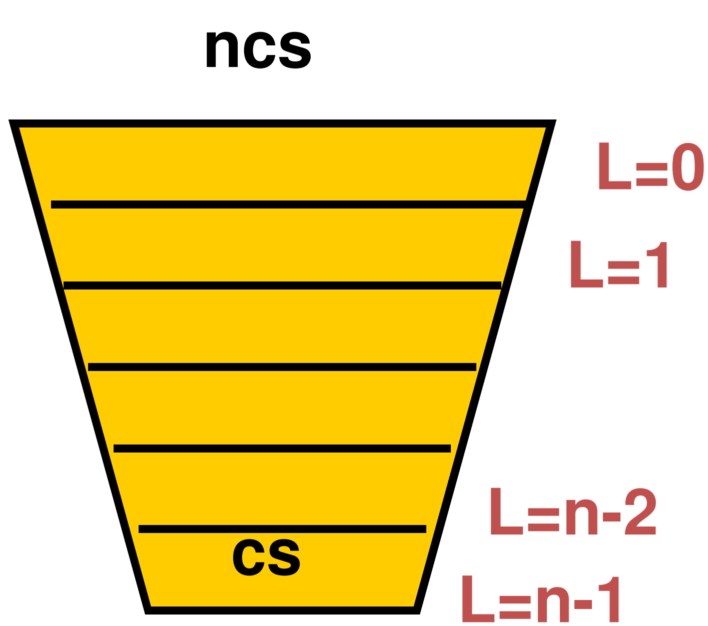
Thread 2 at level 4

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

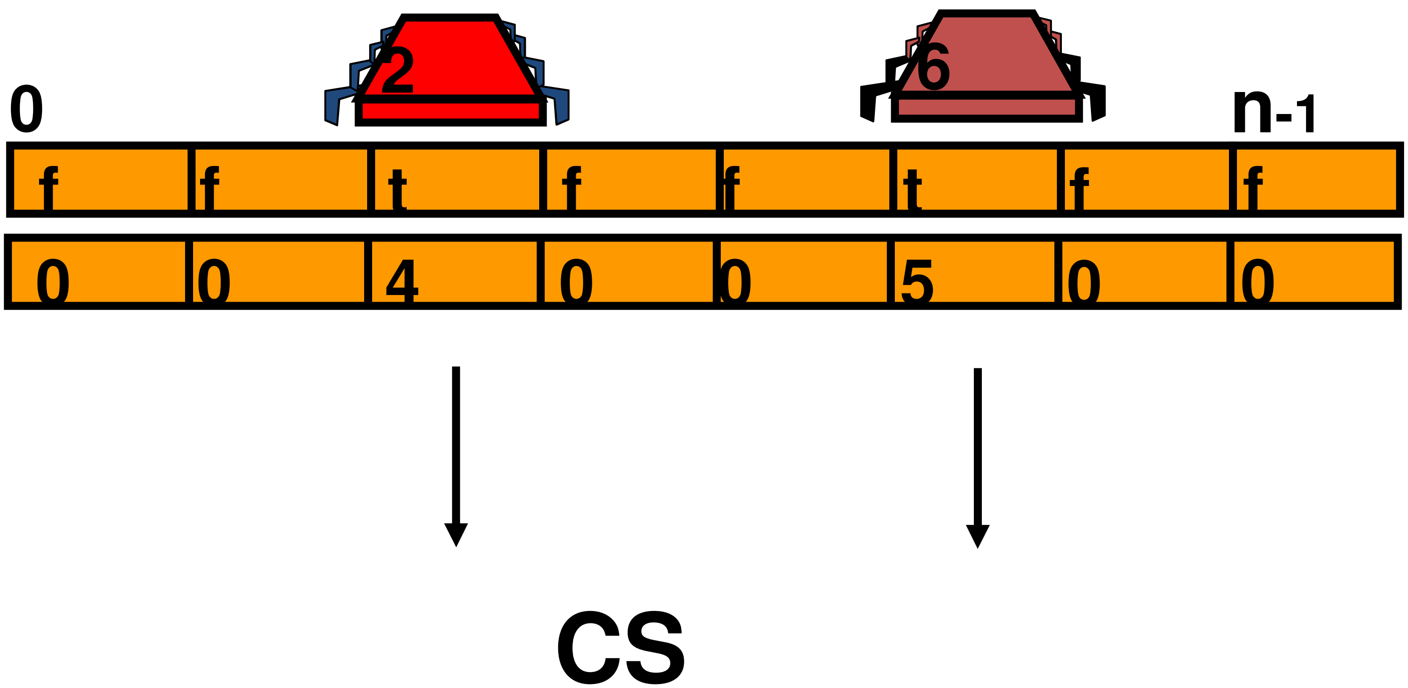
- Similar to Bakery Shop
- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - $(a,i) > (b,j)$
    - If $a > b$, or $a = b$ and $i > j$

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;
```