CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

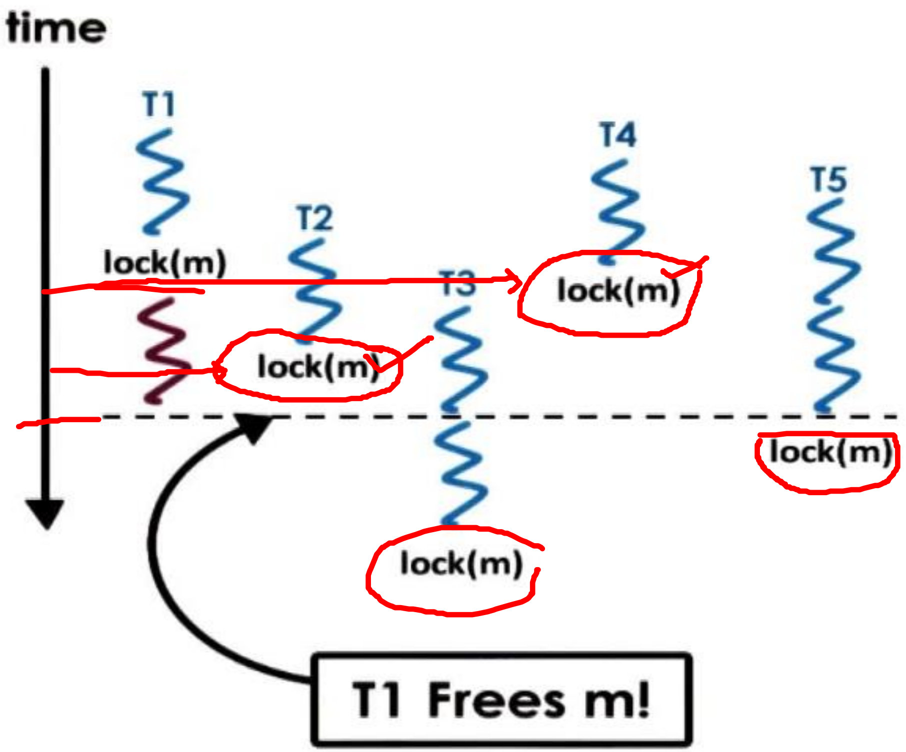
gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

Mutex Quiz

Threads T1-T5 are contending for a mutex m. T1 is the first to obtain the mutex. Which thread will get access to m after T1 releases it? Mark all that apply.

- [ ] T2
- [ ] T4
- [ ] T3
- [ ] T5



Feb 1, Locks Contd... Implementation Attempt -1

- Lock: spin on a flag variable until it is unset, then set it to acquire lock
- Unlock: unset flag variable

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
        ; // spin wait (do nothing)
    mutex->flag = 1;         // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Attempt -2

shared

`p2_inside = False, p1_inside = False`

Process 1

lock → (lines 2–3) · unlock → (line 5)

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

time

CPU	p1_inside	p2_inside
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

Both p1 and p2 can enter into the critical section at the same time

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
     other code
}
```

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
     other code
}
```

Attempt -3

globally defined

```
p2_wants_to_enter, p1_wants_to_enter = False
```

Process 1

```
        while(1){
           p1_wants_to_enter = True
lock  -->  while(p2_wants_to_enter = True);
           critical section
unlock --> p1_wants_to_enter = False
           other code
        }
```

Process 2

```
while(1){
   p2_wants_to_enter = True
   while(p1_wants_to_enter = True);
   critical section
   p2_wants_to_enter = False
   other code
}
```

No Progress

time

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved

Each process is waiting for the other

this is a deadlock

time

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

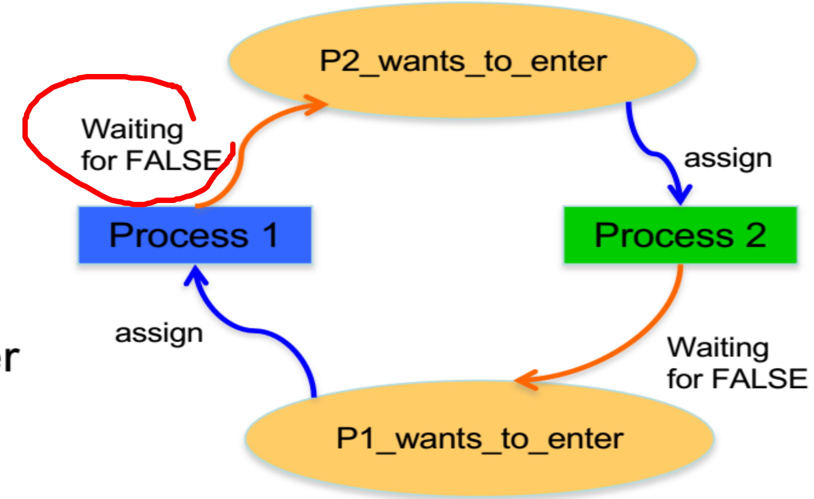
There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved

Each process is waiting for the other

this is a deadlock



Peterson's Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

lock

```
while(1){
  p1_wants_to_enter = True
  favored = 2

  while (p2_wants_to_enter AND
         favored = 2);
  critical section
  p1_wants_to_enter = False
  other code
}
```

unlock

If the second process wants to enter. favor it. (be nice !!!)

favored is used to break the tie when both p1 and p2 want to enter the critical section.

favored can take only two values : 1 or 2

(* the process which sets favored last looses the tie *)

Break the deadlock with a 'favored' process

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
            favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

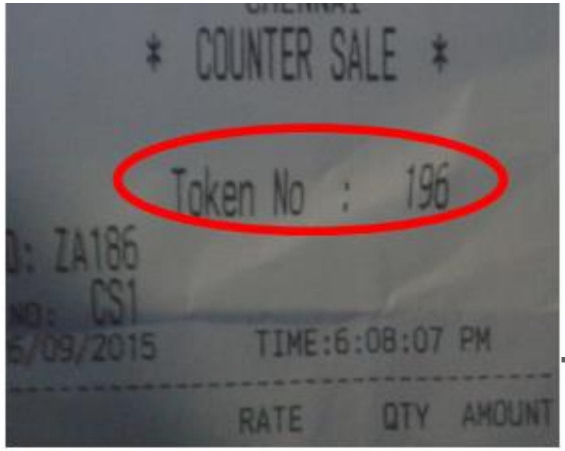
```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
            favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- Deadlock broken because favored can only be 1 or 2.
  - Therefore, tie is broken. Only one process will enter the critical section
- Solves Critical Section problem for two processes

Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport

Eat
when 196 displayed

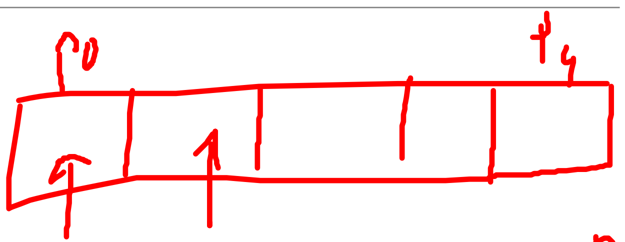


wait your turn!!

http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

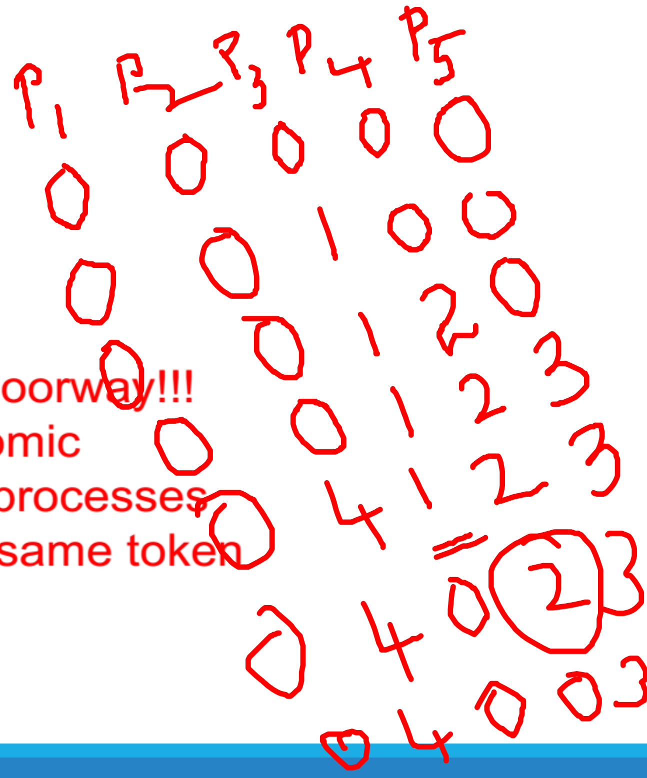
Simplified Bakery Algorithm

- Processes numbered 0 to N-1 ✓ 5
- num is an array N integers (initially 0).
  - Each entry corresponds to a process




```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1 ✓
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]); ✓
    }
}
```

This is at the doorway!!!
It has to be atomic
to ensure two processes
do not get the same token



critical section

```
unlock(i){
    num[i] = 0;
}
```

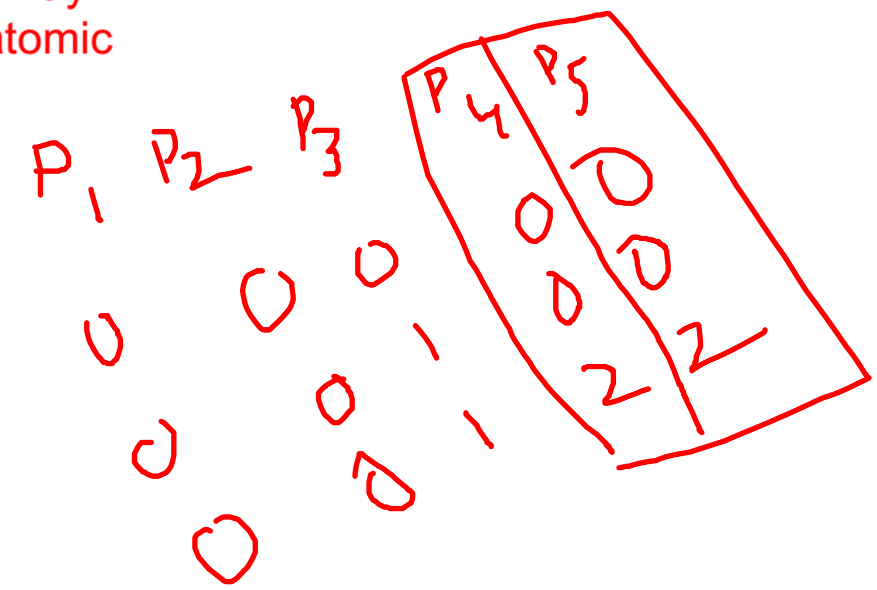
- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

This is at the doorway!!!
Assume it is not atomic

critical section

```
unlock(i){
    num[i] = 0;
}
```



Original Bakery Algorithm

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

Favor one process when there is a conflict.
If there are two processes, with the same num value, favor the process with the smaller id (i)

critical section

```
unlock(i){
    num[i] = 0;
}
```

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

- Does this scheme provide mutual exclusion?

lock=0

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No

```
lock = 0
-----------------------------
P1: while(lock != 0);
-----------------------------  context switch
P2: while(lock != 0);
P2: lock = 1;
-----------------------------
P1: lock = 1;
.... Both processes in critical section
```