Example: With Shared data

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  static volatile int counter = 0;
7
8  // mythread()
9  //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter)
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n",
39            counter);
40     return 0;
41 }
```

With shared data – What happens?

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)

prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

What do we expect? Two threads, each increments counter by 10^7, so 2X10^7

Sometimes, a lower value. Why?

Assembly code

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Assembly code of

```
counter = counter + 1
```

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

Race condition & Synchronization

What just happened is called a race condition –Concurrent execution can lead to different results (functional output depends on order of execution)

Non-deterministic scheduling can change results

- **Critical section:** portion of code that can lead to race conditions

Usually critical sections access shared resources (eg. Variable on heap)

- What we need: mutual exclusion–Only one thread should be executing critical section at any time

- What we need: atomicity of the critical section –**The critical section should execute like one uninterruptible instruction**

One Solution – Atomic operations

Ask hardware to support a new instruction that does all these instructions atomically

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

No context switch between such sets of instructions => No race condition

But we cannot keep creating atomic versions of complex set of instructions

Instead, create a generic template for mutual exclusion with HW support.

Summary

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Any solution should satisfy the following requirements

– Mutual Exclusion : No more than one process in critical section at a given time

– Progress : When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay

– No starvation (bounded wait): There is an upper bound on the number of times a process enters the critical section, while another is waiting

Locks

Consider update of shared variable **`balance = balance + 1`**

We can use a special lock variable to protect it

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

All threads accessing a critical section share a lock

- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner
- Pthreads library in Linux provides such locks

shared variable

```
int counter=5;
lock_t L;
```

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

Building a Lock

Goals of a lock implementation

– Mutual exclusion

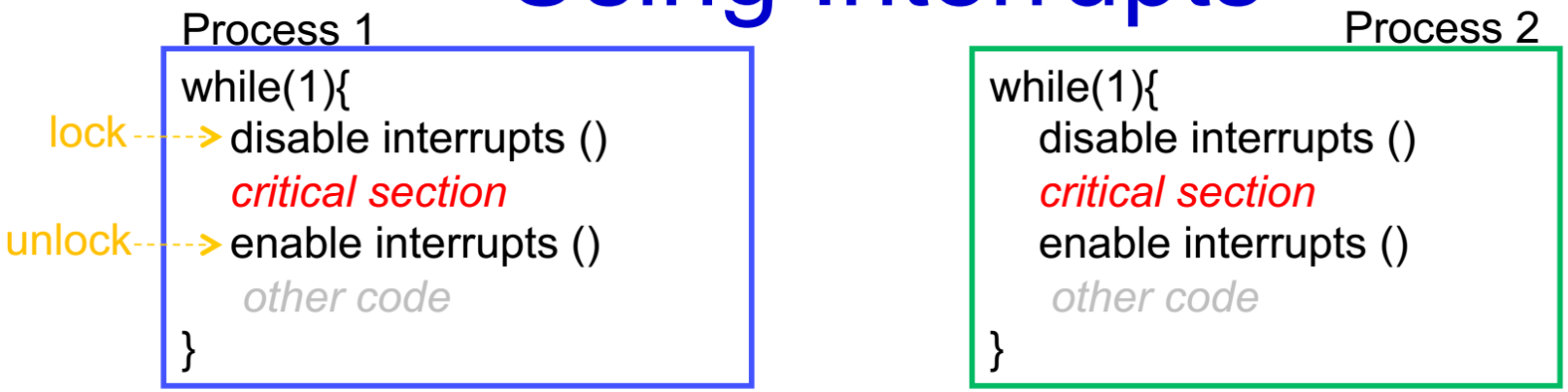
– Fairness: all threads should eventually get the lock, and no thread should starve

– Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources

• Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code

• Implementing locks needs support from hardware and OS

Using Interrupts



- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Is disabling interrupts enough ?

Is this enough?

- No, not always!
- Many issues here:

– Disabling interrupts is a privileged instruction and program can misuse it (e.g., run forever)

– Will not work on multiprocessor systems, since another thread on another core can enter critical section

- This technique is used to implement locks on single processor systems inside the OS

– Need better solution for other situations

Software Solution - Attempt -1

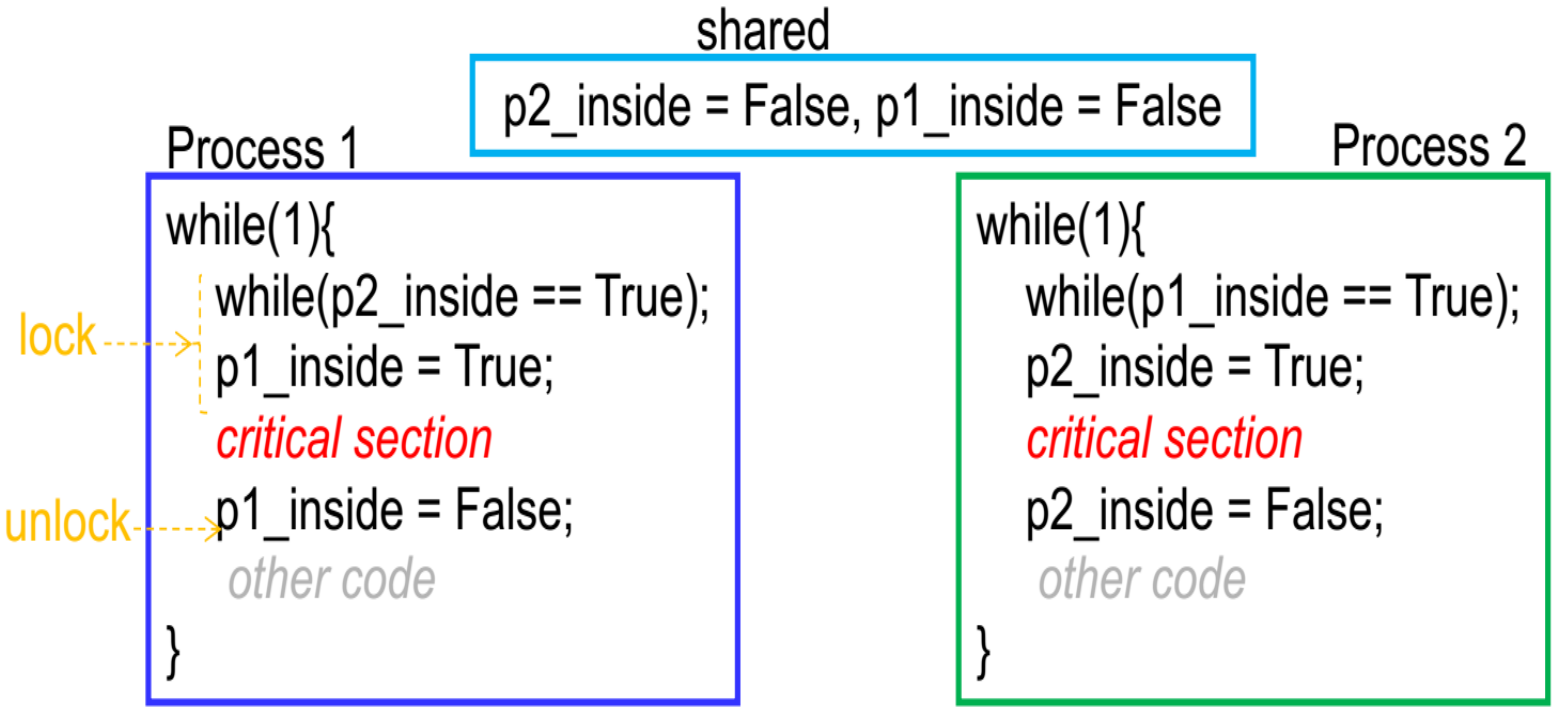
- Lock: spin on a flag variable until it is unset, then set it to acquire lock
- Unlock: unset flag variable

```
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



- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

No mutual exclusion

time

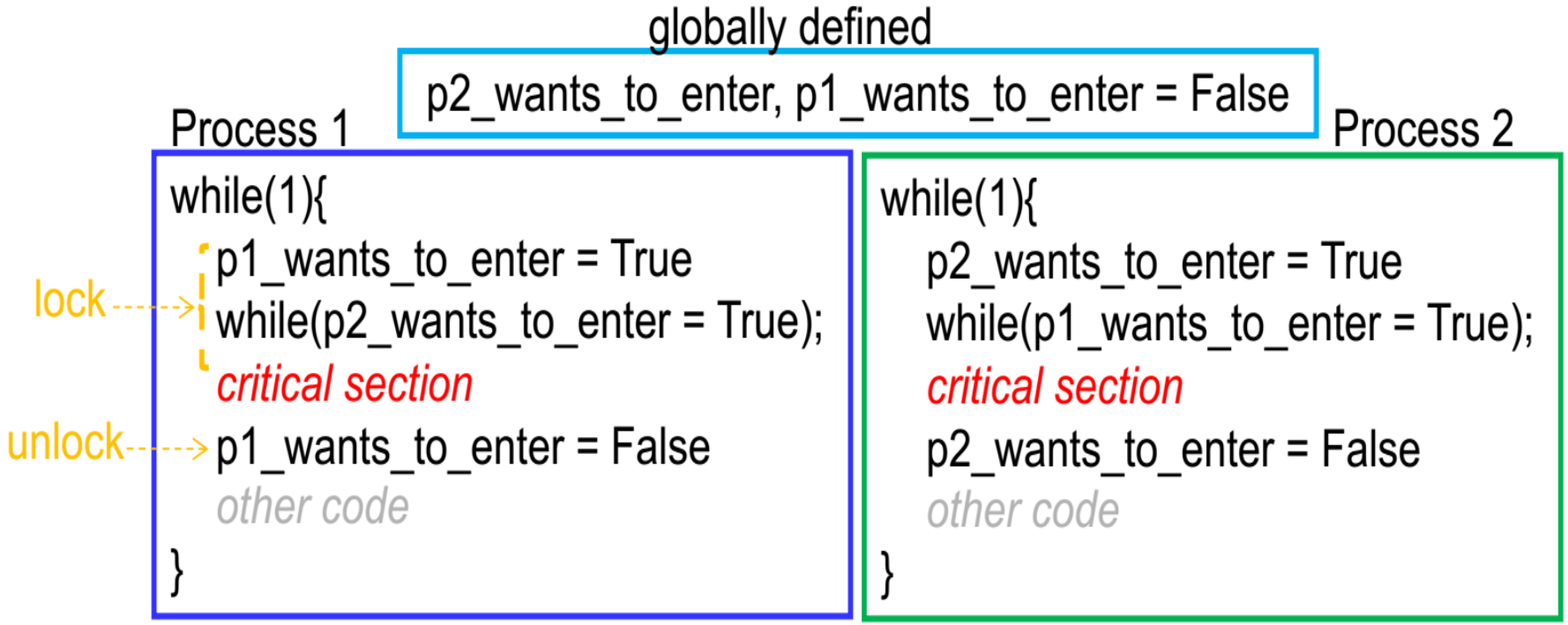
CPU	p1_inside	p2_inside
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

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



- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

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

Deadlock

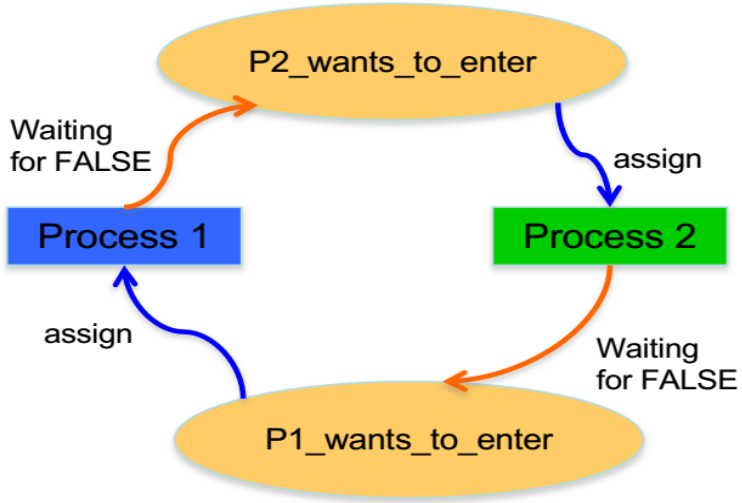
time

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

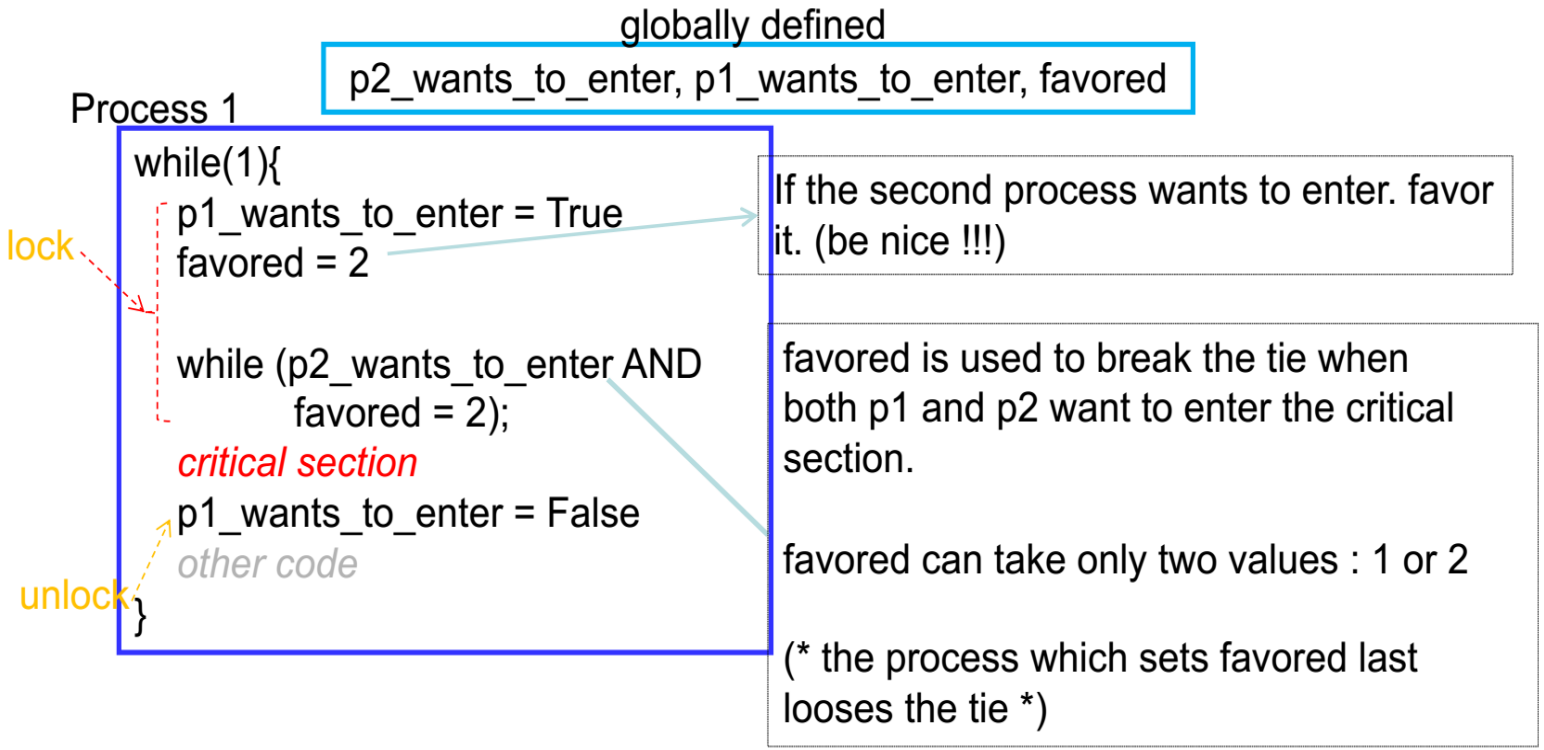
There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved
- Each process is waiting for the other
- this is a deadlock



Peterson's Solution



Break the deadlock with a ‘favored’ process

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
}
```

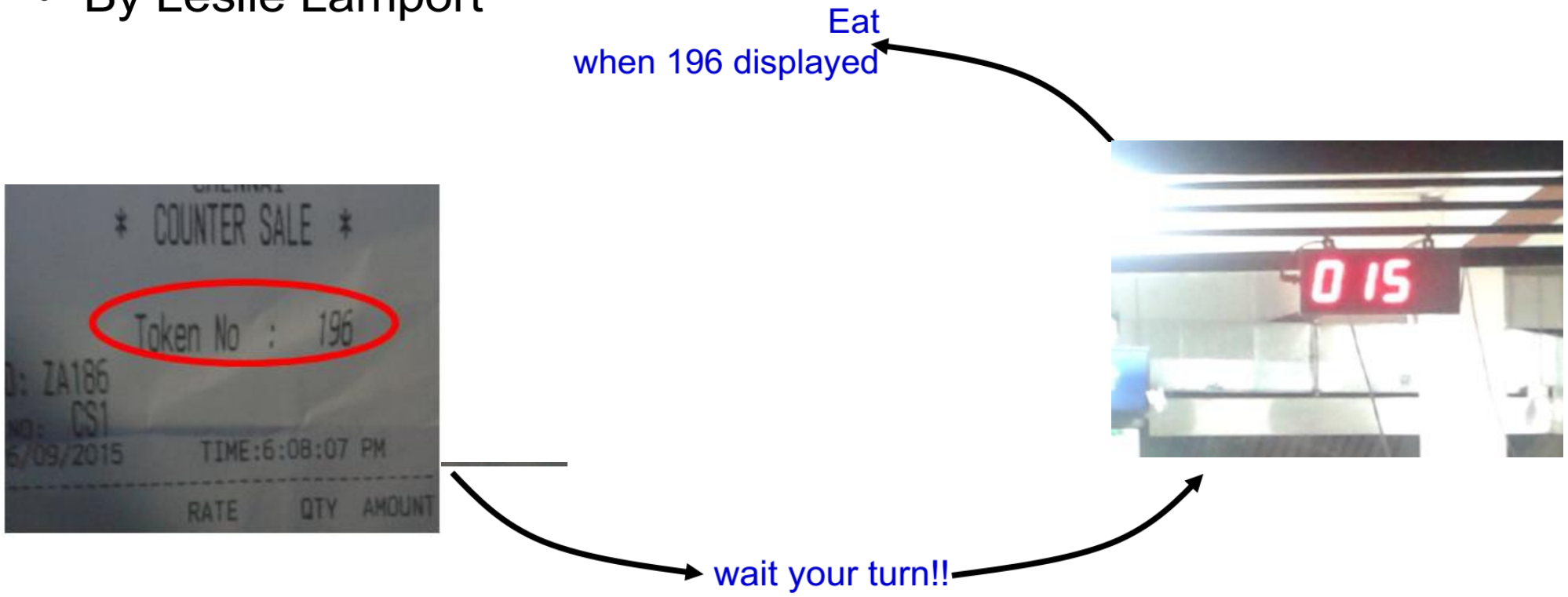
Process 2

```
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



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

Simplified Bakery Algorithm

- 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]);
    }
}
```

critical section

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

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

- 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

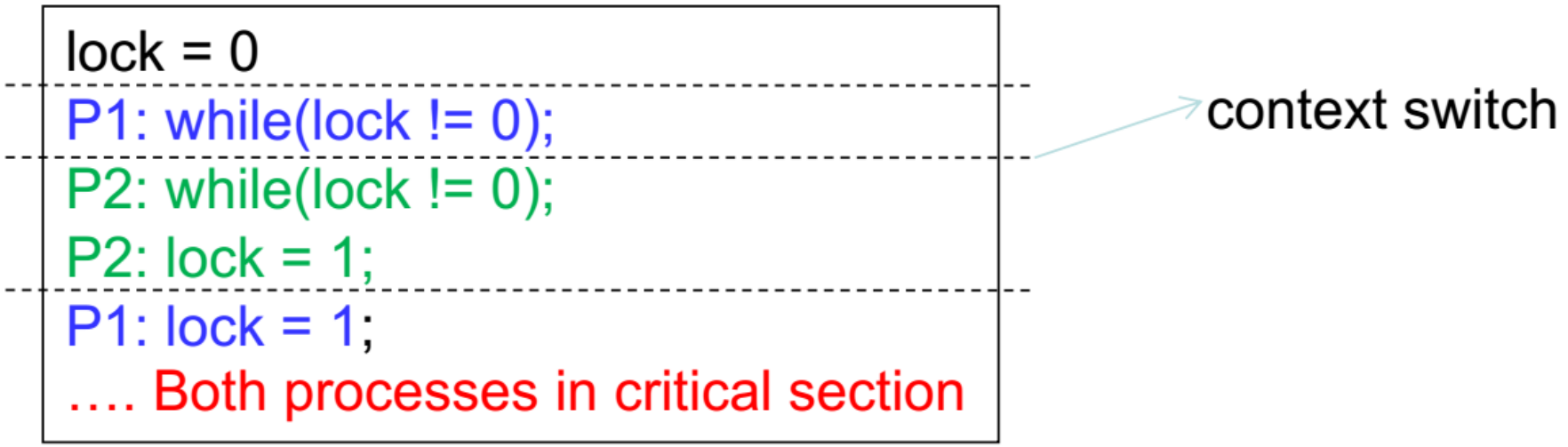
Process 1

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

Process 2

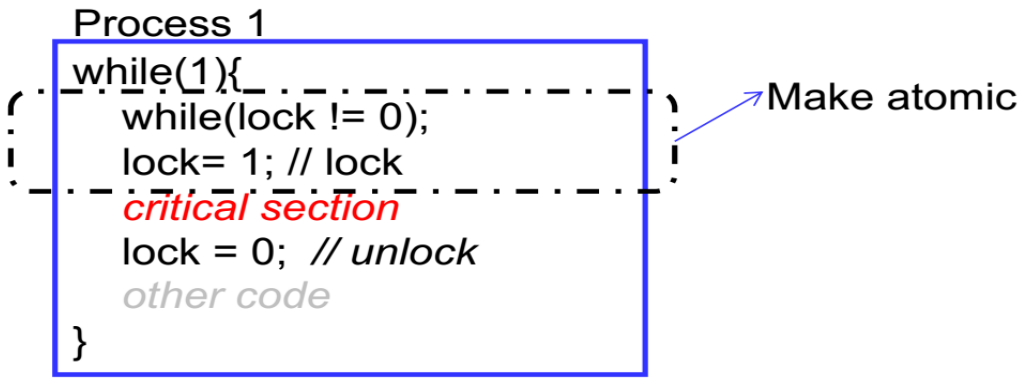
```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No


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

If only ..

- We could make this operation atomic



Hardware to the rescue….

Hardware atomic instructions

Very hard to ensure atomicity only in software

- Modern architectures provide hardware atomic instructions
- Example of an atomic instruction: test-and-set

–Update a variable and return old value, all in one hardware instruction

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

Simple lock using test-and-set

If TestAndSet(flag,1) returns 1, it means the lock is held by someone else, so wait busily
•This lock is called a spinlock–spins until lock is acquired

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Hardware Support (Test & Set)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

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

Usage for locking

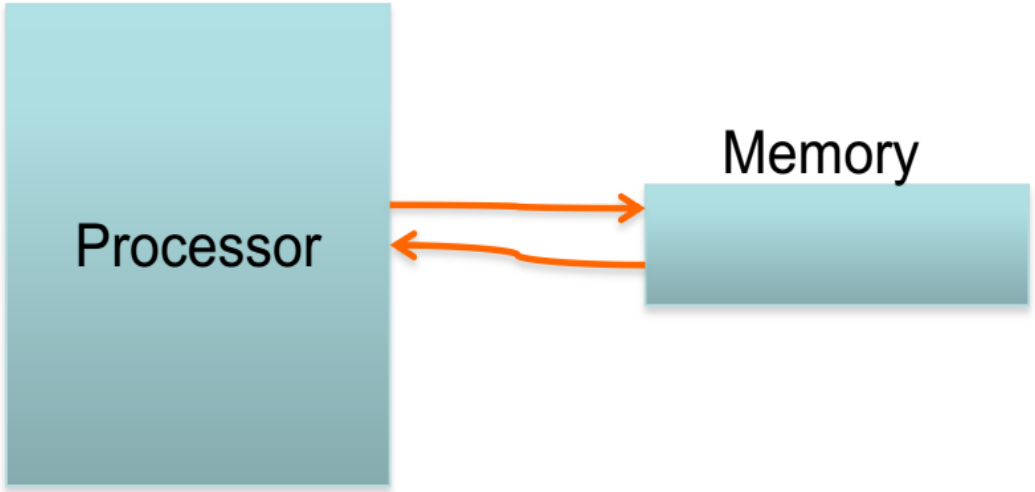
Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

Test & Set Instruction

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```



equivalent software representation
(the entire function is executed atomically)

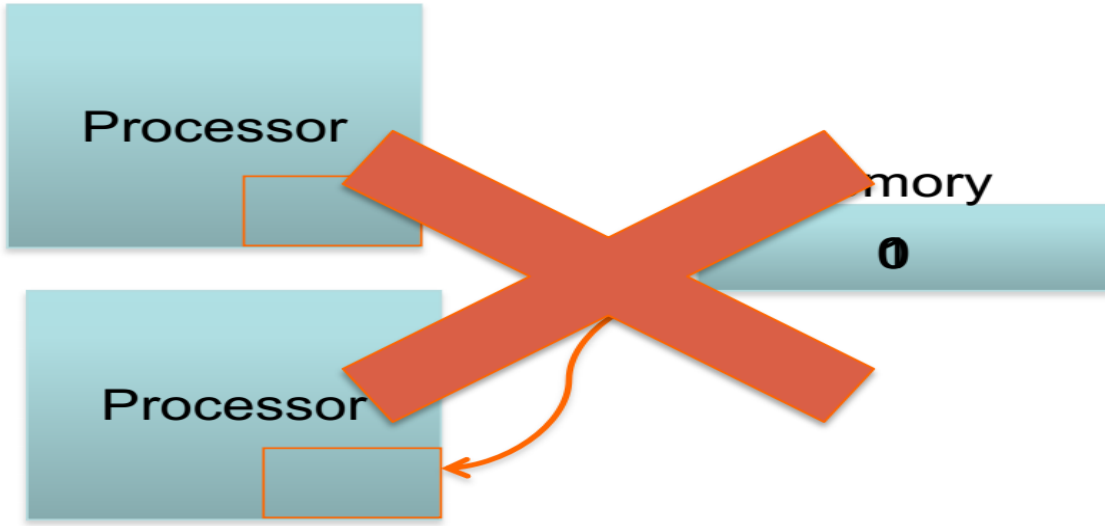
Test and Set instruction

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
    int prev = *L;
    *L = 1;
    return prev;
}
```

equivalent software representation (the entire function is executed atomically)



Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

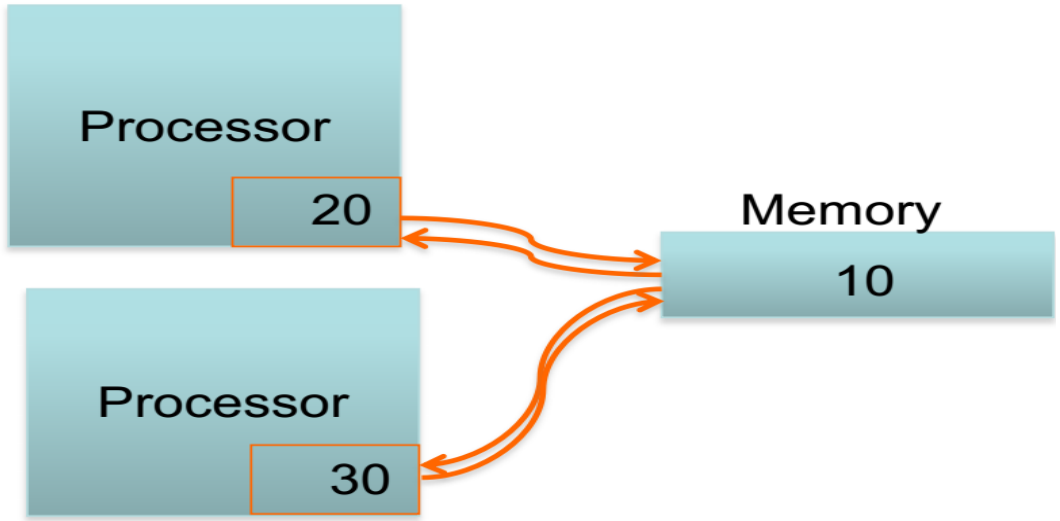
Intel Hardware Support (xchg instruction)

- Write to a memory location, return its old value

atomic

```
int xchg(int  *L, int v){
    int prev = *L;
    *L = v;
    return prev;
}
```

equivalent software representation (the entire function is executed atomically)



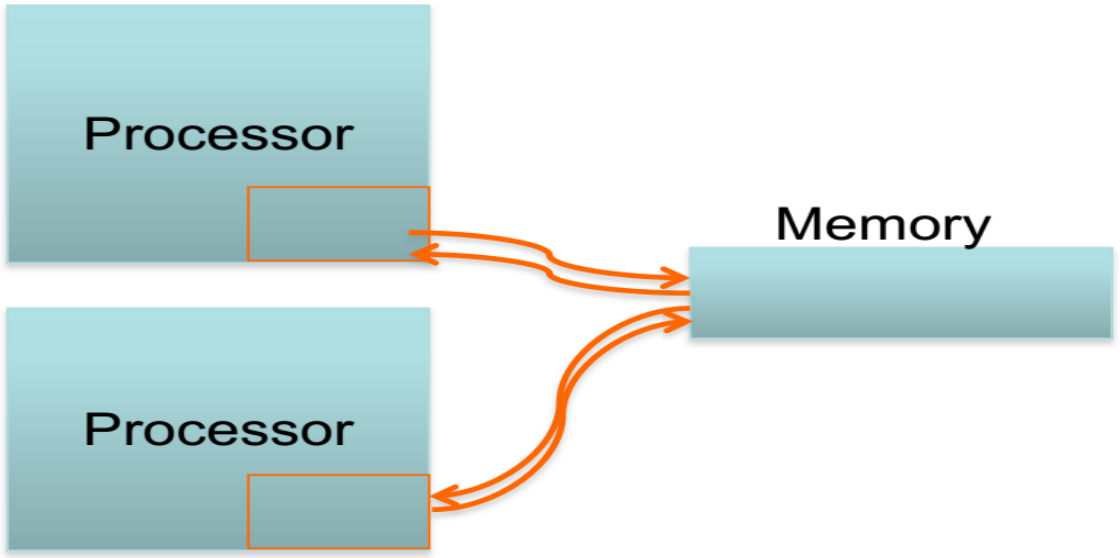
Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

xchg instruction

Note. %eax is returned

typical usage :

xchg reg, mem

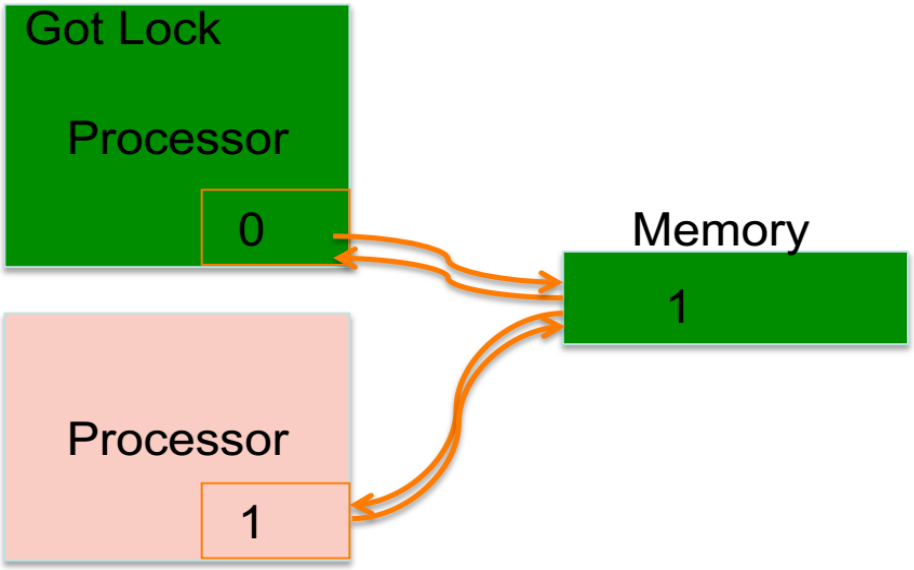



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem

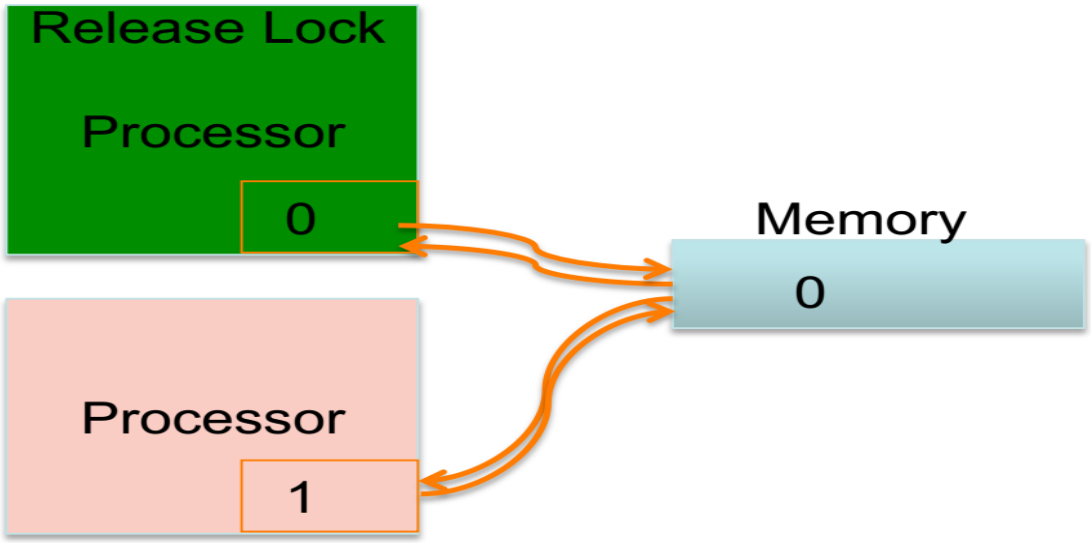



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem

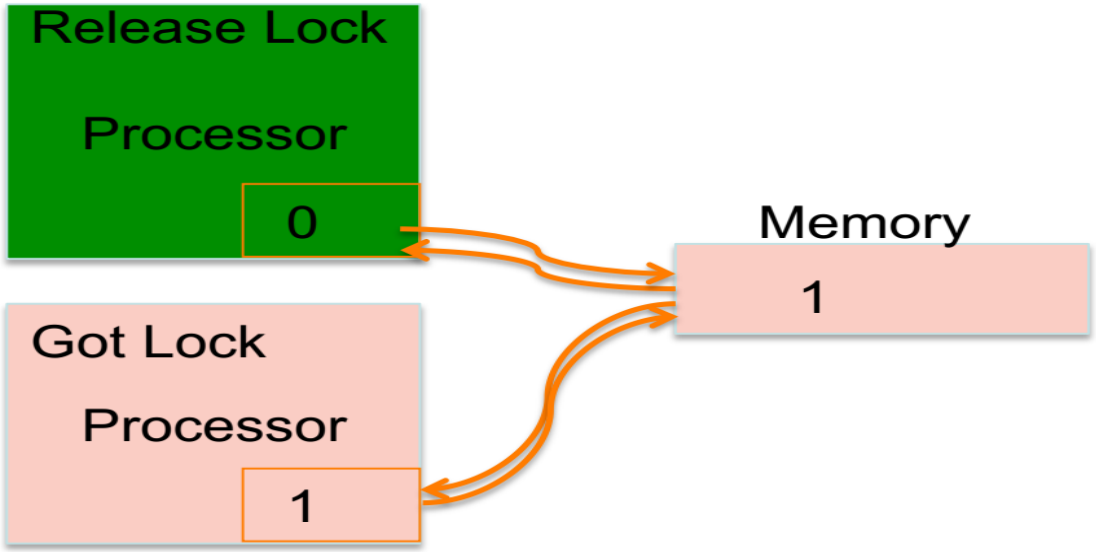



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem

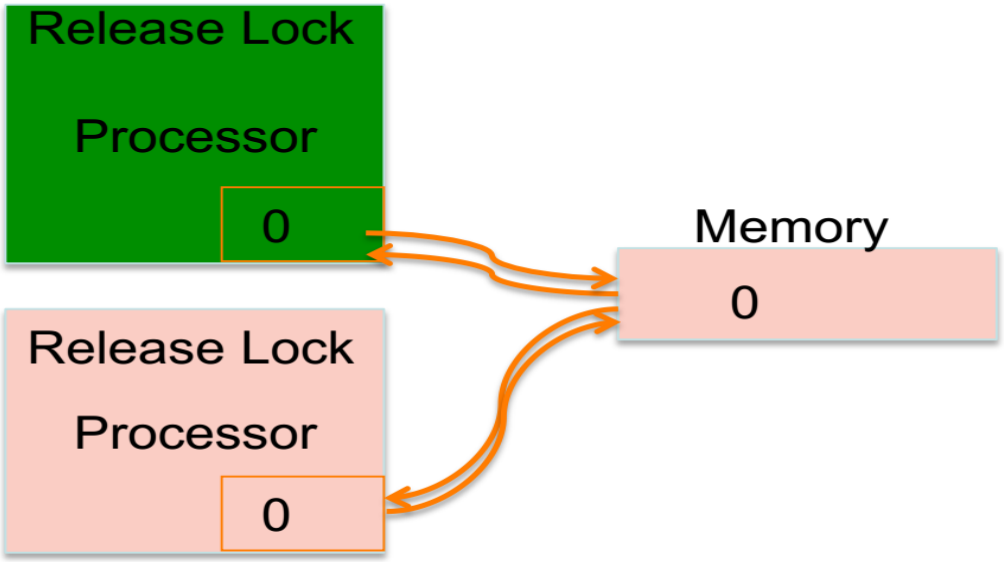



```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

typical usage :

xchg reg, mem




```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Spinlocks

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

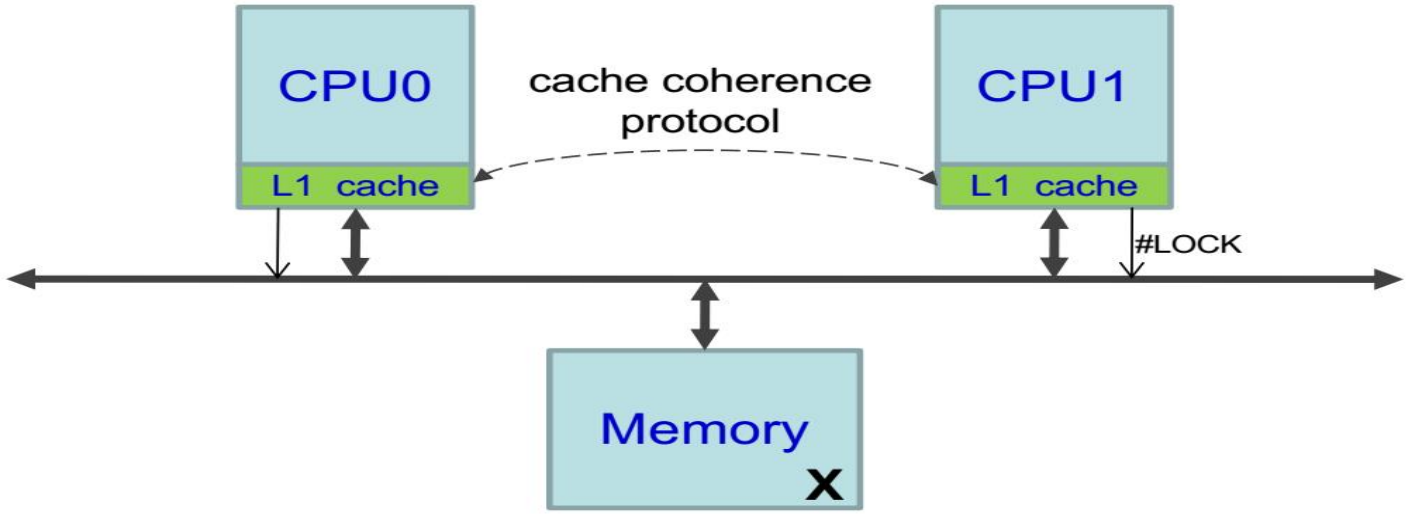
Issues with Spinlock

xchg %eax, X

- No compiler optimizations should be allowed
  - Should not make X a register variable
    - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
    - Use serialized instructions (which forces instructions not to be reordered)
    - Luckly xchg is already implements serialization

Issues with Spinlock

xchg %eax, X



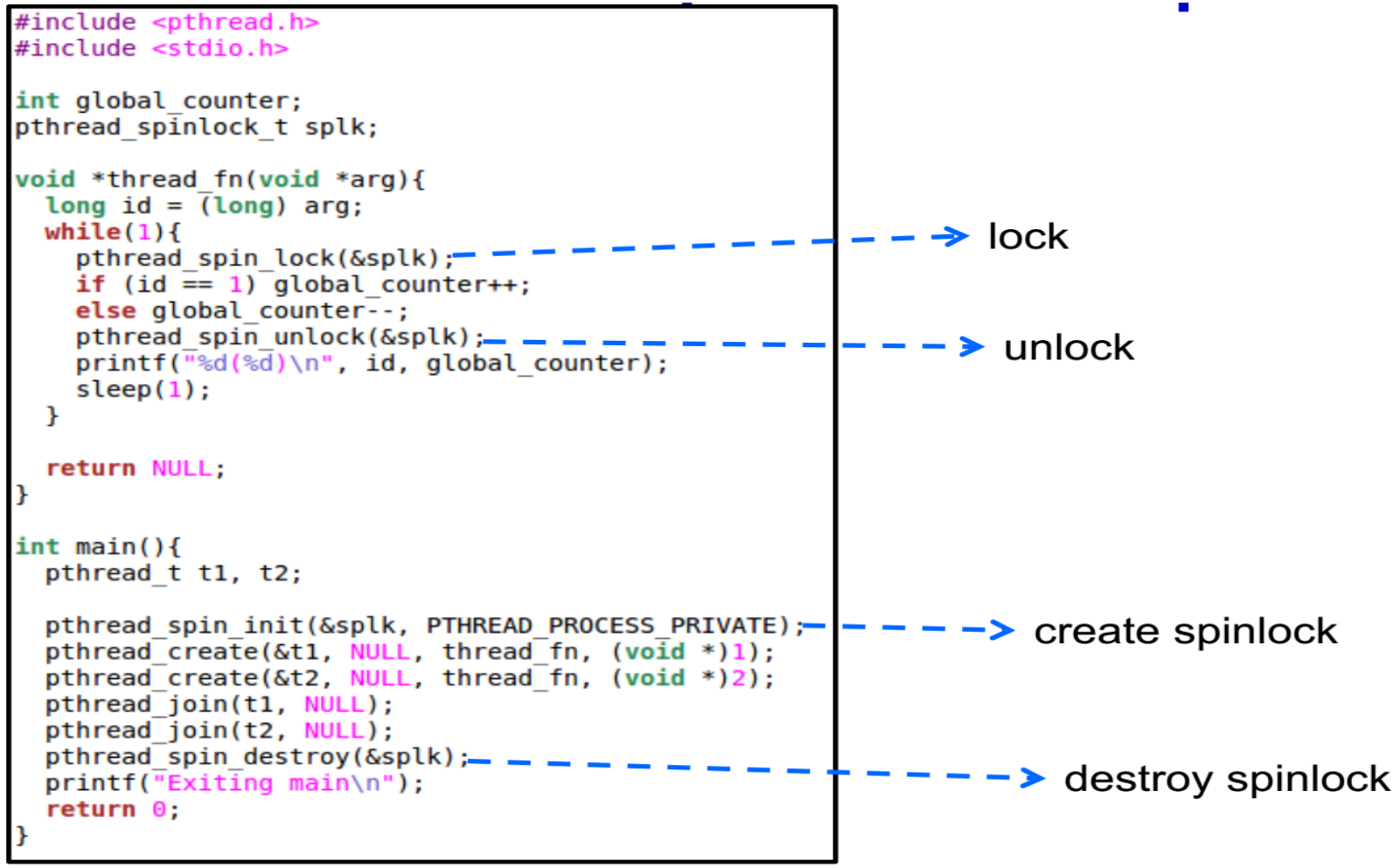
- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a ‘locked ‘ memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

When to use Spinlocks?

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

Pthread example

```c
#include <pthread.h>
#include <stdio.h>

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
  long id = (long) arg;
  while(1){
    pthread_spin_lock(&splk);
    if (id == 1) global_counter++;
    else global_counter--;
    pthread_spin_unlock(&splk);
    printf("%d(%d)\n", id, global_counter);
    sleep(1);
  }

  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_spin_init(&splk, PTHREAD_PROCESS_PRIVATE);
  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_spin_destroy(&splk);
  printf("Exiting main\n");
  return 0;
}
```



Alternative to Spinning

- Alternative to spinlock: a (sleeping) mutex
- Instead of spinning for a lock, a contending thread could simply give up the CPU and check back later – yield() moves thread from running to ready state

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
       sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

pthread Mutex

- pthread_mutex_lock
- pthread_mutex_unlock

```
pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int rc = pthread_mutex_lock(&a_mutex);
if (rc) { /* an error has occurred */
   perror("pthread_mutex_lock");
   pthread_exit(NULL);
}
/* mutex is now locked - do your stuff. */
.
.
```

```
rc = pthread_mutex_unlock(&a_mutex);
if (rc) {
   perror("pthread_mutex_unlock");
   pthread_exit(NULL);
}
```

Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

Spinlock vs Sleeping Mutex

- Most userspace lock implementations are of the sleeping mutex kind
  - CPU wasted by spinning contending threads
  - More so if a thread holds spinlock and blocks for long
- Locks inside the OS are always spinlocks – Why? Who will the OS yield to?
- When OS acquires a spinlock:
  - It must disable interrupts (on that processor core) while the lock is held. Why? An interrupt handler could request the same lock, and spin for it forever.
  - It must not perform any blocking operation–never go to sleep with a locked spinlock!
- In general, use spinlocks with care, and release as soon as possible

How locks should be used?

A lock should be acquired before accessing any variable or data structure that is shared between multiple threads of a process–“Thread-safe” data structures

•All shared kernel data structures must also be accessed only after locking

•Coarse-grained vs. fine-grained locking: one biglock for all shared data vs. separate locks

–Fine-grained allows more parallelism

–Multiple fine-grained locks may be harder to manage

•OS only provides locks, correct locking discipline is left to the user

Homework

The program introduces two employees competing for the "employee of the day" title, and the glory that comes with it.

To simulate that in a rapid pace, the program employs 3 threads: one that promotes Danny to "employee of the day", one that promotes Moshe to that situation, and a third thread that makes sure that the employee of the day's contents is consistent (i.e. contains exactly the data of one employee).

Two versions of the program are given to you. One that uses a mutex, and one that does not.

In this HW, you will report the output of each program and will analyze and write the differences between both the programs

Link : https://drive.google.com/drive/folders/1RrLsvKJJjkjKTefnqMxQ1vdozaLMHIzS