Lecture 6: Mutual Exclusion

Phuong Ha

Based on and including slides from Otto J. Anshus, Tore Brox-Larsen, Kai Li, Thomas Plagemann, A. S. Tanenbaum, A. Silberschatz, M. Herlihy, N. Shavit

Outline

- Preemptive scheduling
- Interprocess communication
  - Background
  - Mutual exclusion
    - Disable interrupt
    - Utilize atomic instructions
- Spin-locks and contention
  - Basic spin-locks
  - Bus-based architecture
  - TAS-based spin-locks revisited
  - Exponential backoff
  - Queue locks
    - Anderson's, CLH, MCS

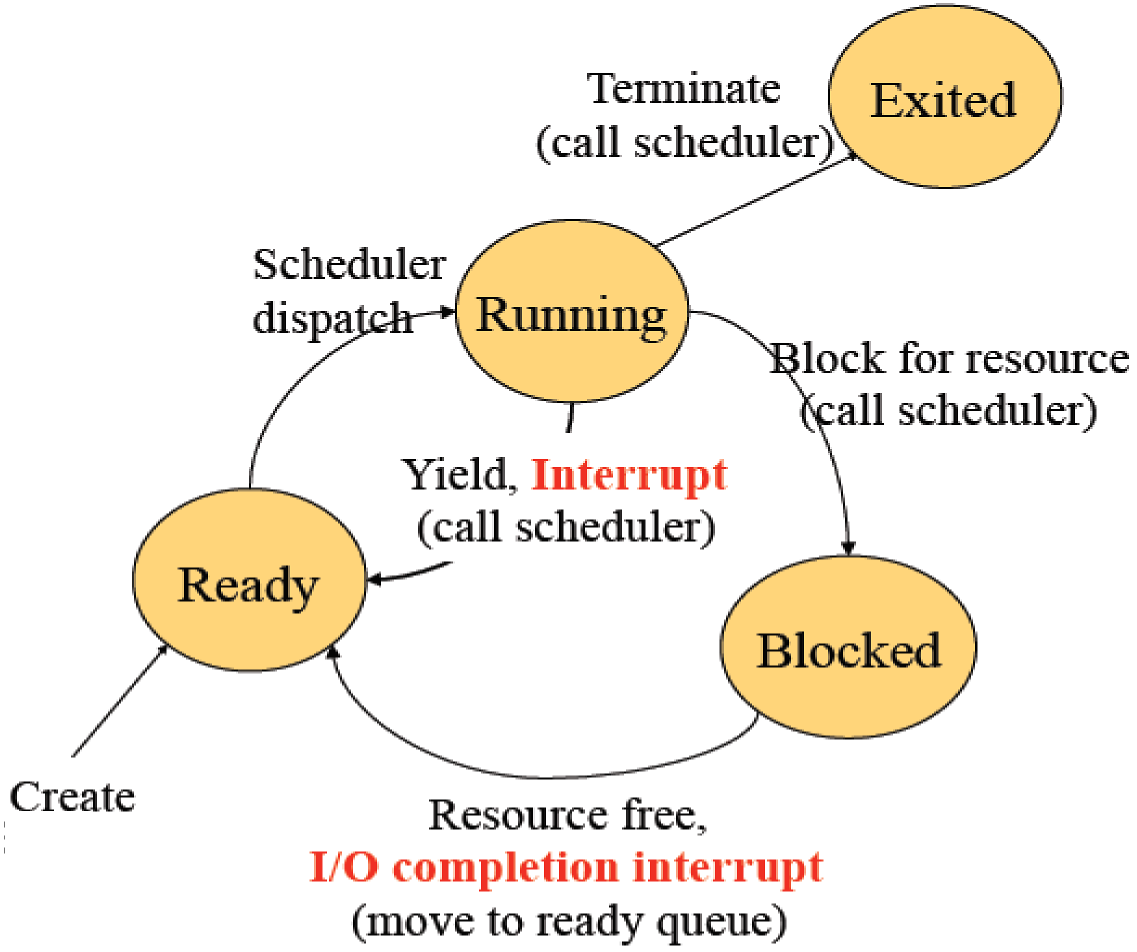
When to Schedule?

- Process/thread creation
- Process/thread exit
- Blocking on I/O or synchronization
- I/O interrupt
- Clock interrupt (preemptive scheduling)

Preemptive Scheduling

- Scheduler select a READY process and sets it up to run for a maximum of some fixed time *(time-slice)*
- Scheduled process computes happily, oblivious to the fact that a maximum time-slice was set by the scheduler
- Whenever a running process exhausts its time-slice, the scheduler needs to suspend the process and select another process to run (assuming one exists)
- To do this, the scheduler needs to be running!
  - Clock interrupt must occur at the end of the time slice.

Preemptive vs. Non-Preemptive Scheduling



Preemptive vs. Non-Preemptive Scheduling

- Non-Preemptive Scheduling ("Yield")
  - Current process or thread has exclusive control until it explicitly yields
    - No other thread executes until yield
    - Access to shared resources simplified
- Preemptive scheduling
  - Current process or thread may be preempted at any time without even noticing.
    - Other threads will progress concurrently
    - Access to shared resources becomes more complicated
    - Some sort of coordination among the threads is needed

Outline

- Preemptive scheduling
- Interprocess communication
    - Background
    - Mutual exclusion
        - Disable interrupt
        - Utilize atomic instructions
- Spin-locks and contention
    - Basic spin-locks
    - Bus-based architecture
    - TAS-based spin-locks revisited
    - Exponential backoff
    - Queue locks
        - Anderson's, CLH, MCS

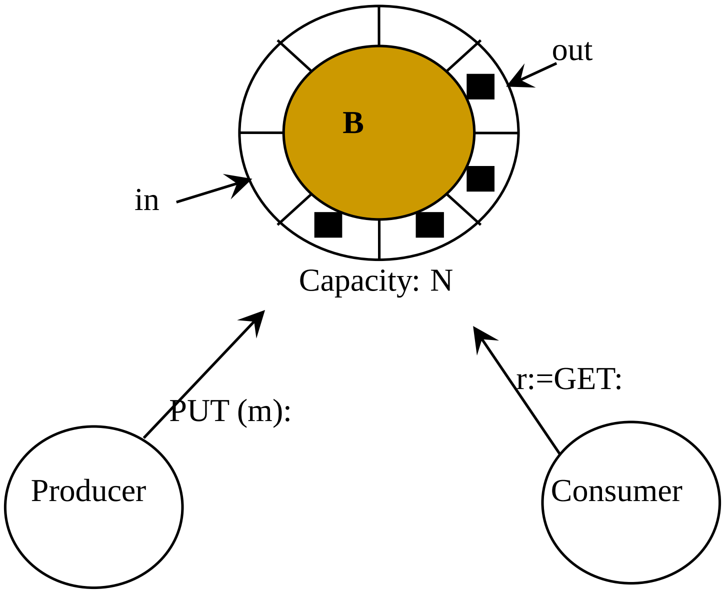
Background

- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Suppose that we wanted to provide a solution to the producer-consumer problem that fills the buffer.
    - We can do so by having an integer **count** that keeps track of the number of items.
    - Initially, count is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.

Ex: Producer-consumer problem



Rules for the buffer B:

•No Get when empty

•No Put when full

•B shared

Producer

```
while (true) {

          /*  produce an item and put in nextProduced  */
          while (count == BUFFER_SIZE)
                    ; // do nothing
           buffer [in] = nextProduced;
           in = (in + 1) % BUFFER_SIZE;
           count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
              ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

      /*  consume the item in nextConsumed */
}
```

Race Condition

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

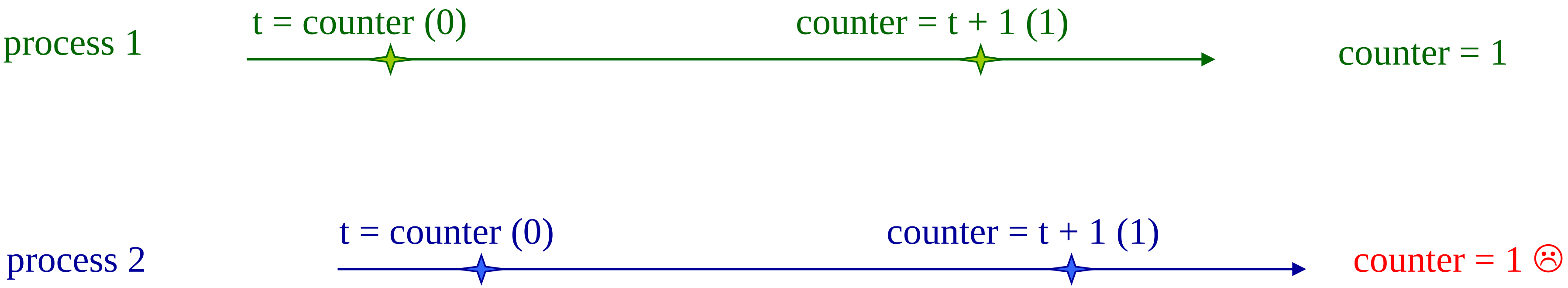
- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

A simple concurrent program

Task: ***Count the number of running processes***

```
shared counter=0;
Increment() {
    t = counter;
    counter = t + 1;
    return;
}
```



process 1 t = counter (0) counter = t + 1 (1) counter = 1

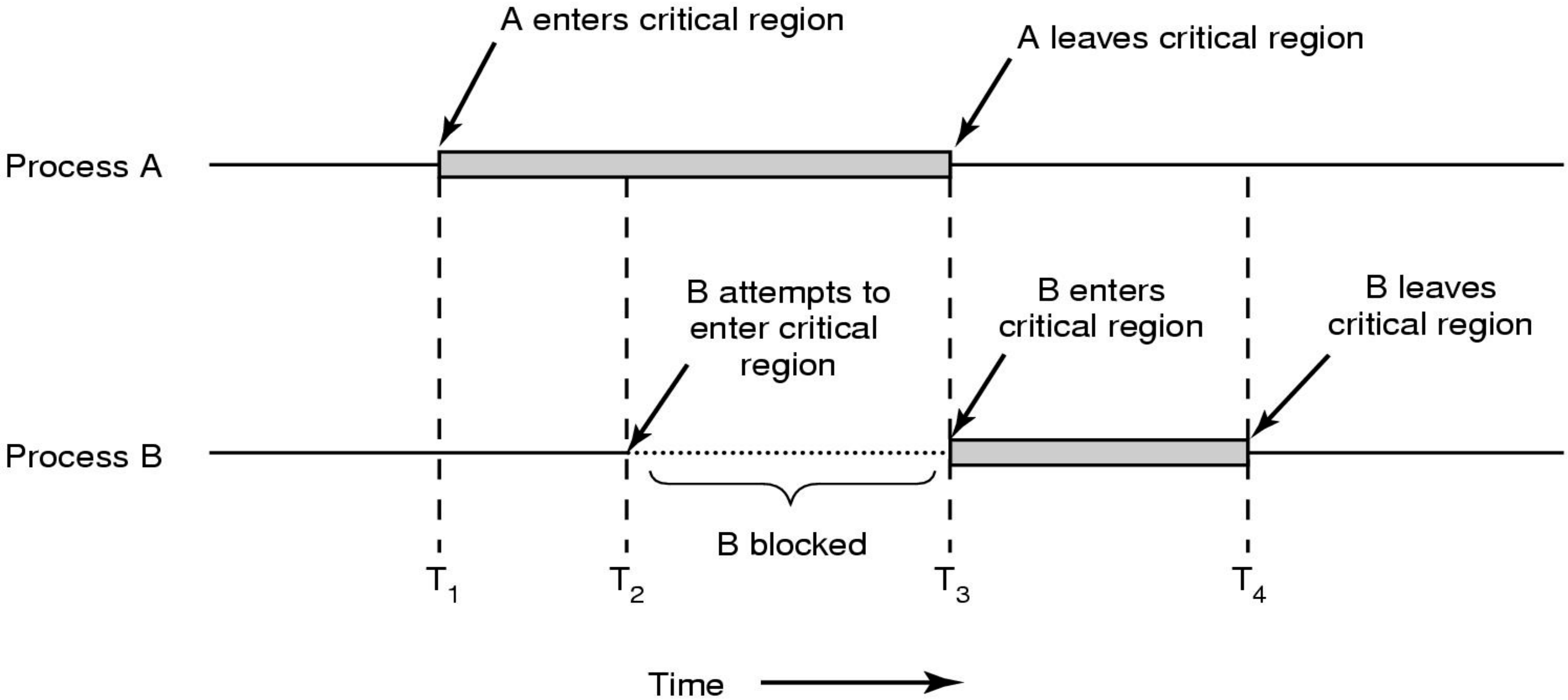
process 2 t = counter (0) counter = t + 1 (1) counter = 1 ☹

Critical Regions

Conditions required to avoid race condition:

- Mutual exclusion:
  - No two processes may be simultaneously inside their critical regions.
- Progress:
  - No process running outside its critical region may block other processes.
- Bounded waiting:
  - No process should have to wait forever to enter its critical region.
- No assumptions may be made about speeds or the number of CPUs.

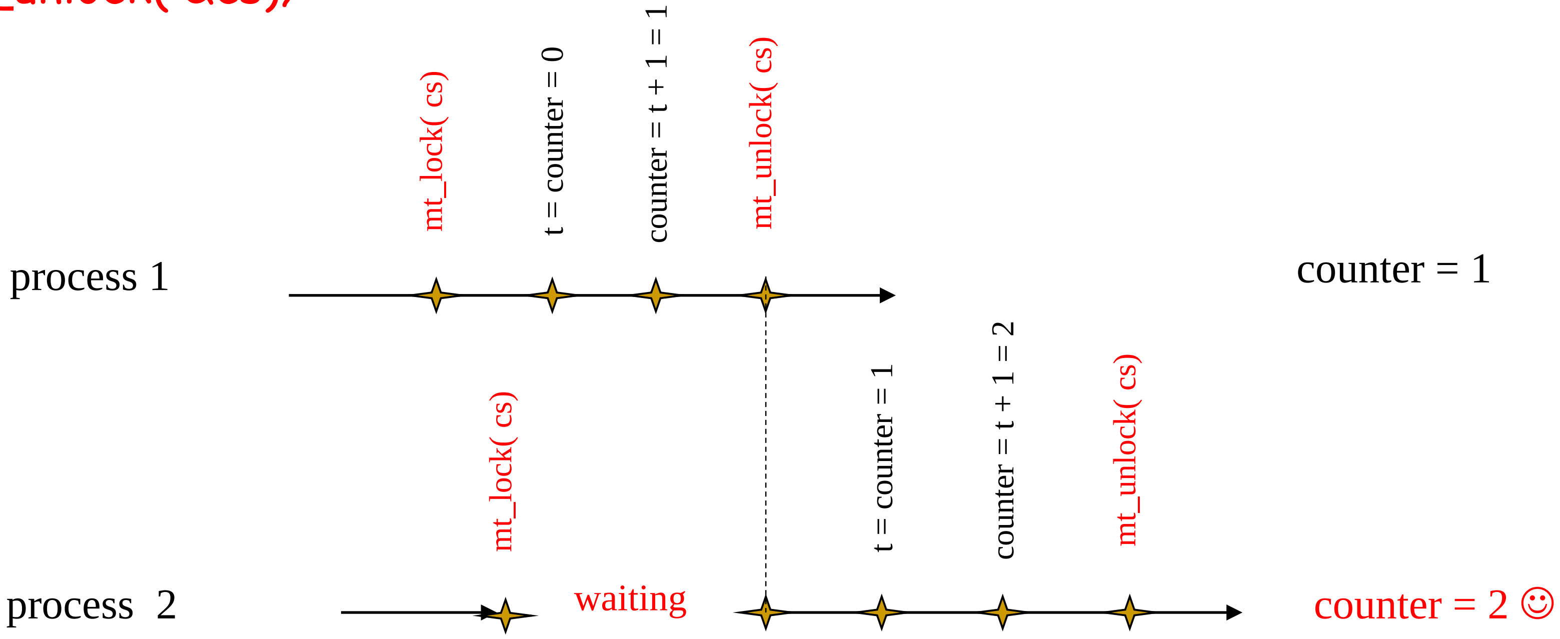
Mutual exclusion using critical regions



.

Mutual exclusion example

```
shared counter = 0; cs = free;
Increment() {
    mutex_lock( &cs);  //synch. point
    t = counter;
    counter = t + 1;
    mutex_unlock( &cs);
    return;
}
```



process 1

mt_lock(cs)

t = counter = 0

counter = t + 1 = 1

mt_unlock(cs)

counter = 1

process 2

mt_lock(cs)

waiting

t = counter = 1

counter = t + 1 = 2

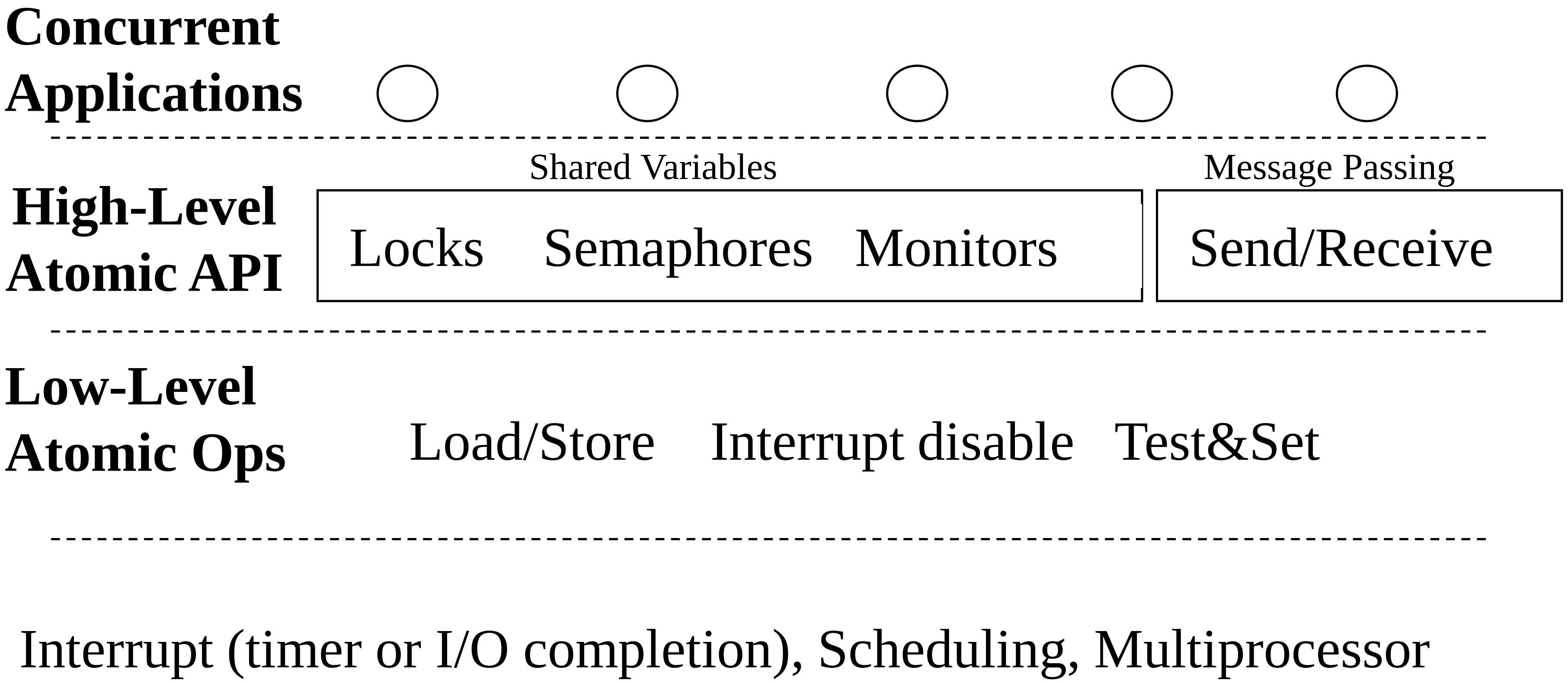
mt_unlock(cs)

counter = 2 ☺

Outline

- Preemptive scheduling
- Interprocess communication
  - Background
  - Mutual exclusion
    - Disable interrupt
    - Utilize atomic instructions
- Spin-locks and contention
  - Basic spin-locks
  - Bus-based architecture
  - TAS-based spin-locks revisited
  - Exponential backoff
  - Queue locks
    - Anderson's, CLH, MCS

Implementation of Synchronization Mechanisms



Concurrent Applications

High-Level Atomic API

Shared Variables: Locks Semaphores Monitors

Message Passing: Send/Receive

Low-Level Atomic Ops

Load/Store Interrupt disable Test&Set

Interrupt (timer or I/O completion), Scheduling, Multiprocessor

Hardware Support for Mutex

- Atomic load and atomic store from/to memory
  - Assumed by Dijkstra (CACM 1965): Shared memory w/atomic R and W operations issued in program order
  - L. Lamport, "A Fast Mutual Exclusion Algorithm," ACM Trans. on Computer Systems, 5(1):1-11, Feb 1987.
- Disabling Interrupts
- Atomic read-modify-write
  - IBM/360: Test-And-Set (TAS) proposed by J. Dirac for IBM S/360 (1963)
  - IBM/370: Generalized Compare-And-Swap (CAS) (1970)

For Shared Memory Multiprocessor w/only atomic read and atomic write (Michael Fischer)

Executed by process no. i.

X is shared memory.

<op> is an Atomic operation, no more complex than load or store, (no test-and-set or similar)

”While x =/= 0 do skip;”

Or could block? How?

Entry:

```
Repeat
    await <x=0>;
    <x := i>;
    <delay>;
until <x = i>;
```

Critical Region

use shared resource

Exit

```
<x := 0>;
```

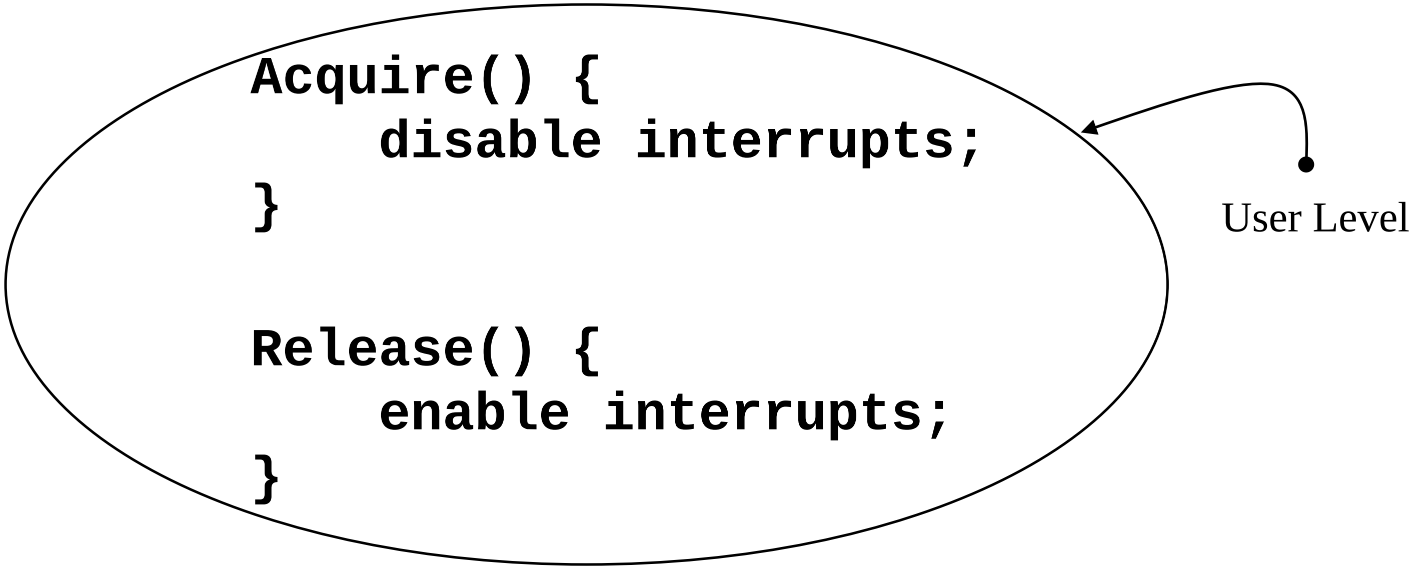
We are assuming that COMMON CASE will be fast and that all processes will get through eventually

L. Lamport. A Fast Mutual Exclusion Algorithm, 1986.

Disable Interrupts

- Model
  - Single-processor system
- CPU scheduling
  - Internal events
    - Threads do something to relinquish the CPU
  - External events
    - Interrupts cause rescheduling of the CPU
- Disabling interrupts
  - **Delay** handling of external events
    - and make sure we have a safe ENTRY or EXIT

Does This Work?



```
Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}
```

- Kernel cannot let **users** disable interrupts
- Kernel can provide two system calls, Acquire and Release, but need ID of critical region
- Remember: Critical sections can be arbitrary long, OS must be able to preempt process in critical section
- Disabling interrupts is insufficient on multiprocessors

Disable Interrupts w/Busy Wait & Lock

```
Acquire(lock) {
  disable interrupts;
  while (lock != FREE)

Spins
          ;
  lock = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  lock = FREE;
  enable interrupts;
}
```

- Why do we need to *disable* interrupts at all?
- Would this work?

Disable Interrupts Briefly w/Busy Wait

```
Acquire(lock) {
  disable interrupts;
  while (lock != FREE){
    enable interrupts;
    disable interrupts;
  }
  lock = BUSY;
  enable interrupts;
}
```

Spins

```
Release(lock) {
  disable interrupts;
  lock = FREE;
  enable interrupts;
}
```

- Why do we need to enable interrupts inside the loop in `Acquire`?
- Would this work for multiprocessors?

Disable Interrupts w/Blocking Queue

```
Acquire(lock) {
  disable interrupts;
  while (lock == BUSY) {
    insert(caller, lock_queue);
    BLOCK caller;
  } else
    lock = BUSY;
  enable interrupts;
}
```

```
Release(lock) {
  disable interrupts;
  if (nonempty(lock_queue)) {
    remove(tid, lock_queue);
    insert(tid, ready_queue);
  }
  lock = FREE;
  enable interrupts;
}
```

- When must Acquire *re-enable* interrupts in going to sleep?
  - Before insert()?
  - After insert(), but before block?
- Would this work on multiprocessors?

Starvation possible, at least unfairness

Deadlock possible because then **Release** can be executed by another thread *right* before we can do the BLOCK, and then we do the BLOCK, and we will never be awakened again

So enable inside BLOCK, and must involve Kernel

Outline

- Preemptive scheduling
- Interprocess communication
  - Background
  - Mutual exclusion
    - Disable interrupt
    - Utilize atomic instructions
- Spin-locks and contention
  - Basic spin-locks
  - Bus-based architecture
  - TAS-based spin-locks revisited
  - Exponential backoff
  - Queue locks
    - Anderson's, CLH, MCS

Atomic Read-Modify-Write Instructions

- What we want: **Test&Set**(lock): Returns TRUE if lock is closed; else returns FALSE and closes lock.
- Exchange (**xchg**, x86 architecture)
  - Swap register and memory
- Compare and Exchange (**cmpxchg**, 486 or Pentium)
  - cmpxchg d,s: If Dest = (**`al,ax,eax`**), Dest = SRC;
    else (**`al,ax,eax`**) = Dest
- LOCK prefix in x86
- Fetch&Add or Fetch&Op
  - Atomic instructions for large shared memory multiprocessor systems
- Load-linked and store-conditional (MIPS, Alpha)
  - Read value in one instruction, do some operations
  - When store, check if value has been modified. If not, ok; otherwise, jump back to start

Used to implement USER level locks

Examples of Read-Modify-Write

-
  ```
  test&set (&address) {         /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
  ```

-
  ```
  swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
  ```

-
  ```
  compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
  ```

-
  ```
  load-linked&store-conditional(&address) {
      /* R4000, alpha */
      loop:
          ll r1, M[address];
          movi r2, 1;     /* Can do arbitrary comp */
          sc r2, M[address];
          beqz r2, loop;
  }
  ```

A Simple Solution with Test&Set

INITIALLY: Lock := FALSE; /*0: OPEN */

TAS (lock):

```
{result := lock;
 lock := TRUE; /*1*/
 return result;}
```

Spin until lock = open →

```
Acquire(lock) {
  while (TAS(lock))
     ;
}
```

```
Release(lock) {
   lock = FALSE;
}
```

- Waste CPU time (busy waiting by all threads)
- Low priority threads may never get a chance to run
  - starvation possible because other threads always grabs the lock, but can be lucky…): No Bounded Waiting (a MUTEX criteria)
- No fairness, no order, random who gets access

Test&Set with Minimal Busy Waiting

CLOSED = TRUE
OPEN = FALSE

```
Acquire(lock) {
  while (TAS(lock.guard))
    ;
  if (lock.value) {
   enqueue the thread;
    block and lock.guard:=OPEN;
    %Starts here after a Release()
  }
  lock.value:=CLOSED;
  lock.guard:=OPEN;
}
```

```
Release(lock) {
  while (TAS(lock.guard))
    ;
  if (anyone in queue) {
    dequeue a thread;
    make it ready;
  } else lock.value:=OPEN;
  lock.guard:=OPEN;
}
```

NB: Lock is kept closed!

- Two levels: Get inside a mutex, then check resource availability (and block (remember to open mutex!) or not).
- Still busy wait, but only for a short time
- Use `yield()` inside the while loop on uniprocessors
- Works with multiprocessors

A Solution without Busy Waiting?

```
Acquire(lock) {
  while (TAS(lock)) {
    enqueue the thread;
    block;
  }
}
```

```
Release(lock) {
  if (anyone in queue) {
    dequeue a thread;
    make it ready;
  } else
  lock:=OPEN;
}
```

- BUT: No mutual exclusion on the thread queue for each lock: queue is shared resource
  - Need to solve another mutual exclusion problem

Using System Call **Block/Unblock**

```
Acquire(lock) {
  while (TAS(lock))
    Block( lock );
}
```

```
Release(lock) {
  lock = 0;
  Unblock( lock );
}
```

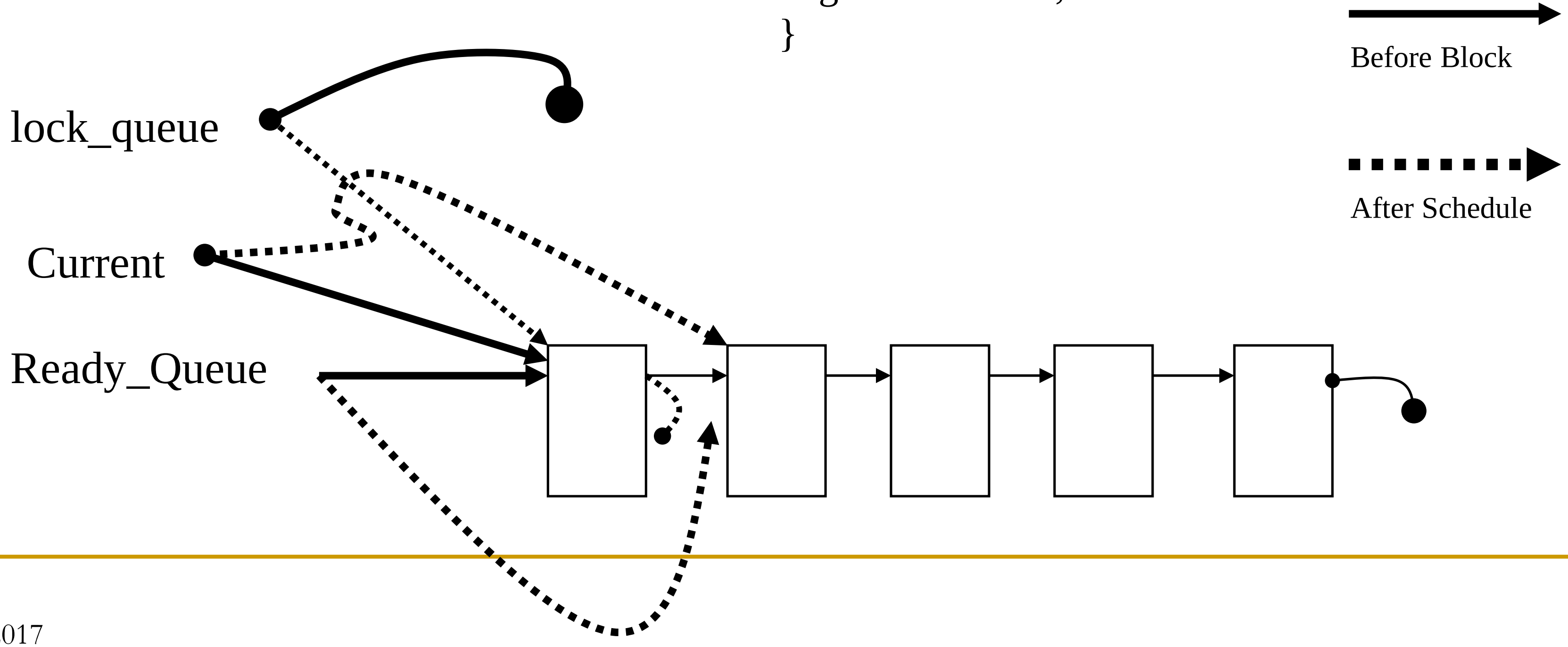
- Block/Unblock are implemented as system calls
- How would you implement them?
  - Minimal waiting solution

Block and Unblock

Context is already saved by Trap Handler because we did a system call

```
Block (lock) {
    spin on lock.guard;
    insert (current, lock_queue, last);
    clear lock.guard;
    goto scheduler;
}
```

```
Unblock (lock) {
    spin on lock.guard;
    insert (out (lock_queue, first), Ready_Queue, last);
    clear lock.guard;
    goto scheduler;
}
```



References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.
- M. Herlihy et. al., The Art of Multiprocessor Programming
- Via Wikipedia
  - Mutual exclusion
  - Dekker's algorithm
  - Peterson's algorithm
  - Lamport's bakery algorithm
  - Szymansky's algorithm
  - Taubenfeldts black and white bakery algorithm
- L. Lamport, A Fast Mutual Exclusion Algorithm
  - http://research.microsoft.com/users/lamport/pubs/fast-mutex.pdf

Thanks for your attention!

Questions?