# Lecture 11: Concurrency Part I

Professor G. Sandoval

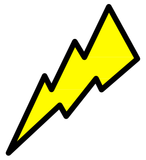
Some slides derived from : William Stallings, Tanenbaum/Bo, John Regehr and Brendan Dolan-Gavitt
Thanks !!

- Motivating Example:  Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# What is Synchronization?

- **<u>Question</u>: How do you control the behavior of ″cooperating″ processes that share resources?**

| Time | You | Your roomate |
|---|---|---|
| 3:00 | Arrive home | |
| 3:05 | Check fridge □□ no milk | |
| 3:10 | Leave for grocery | |
| 3:15 | | Arrive home |
| 3:20 | Buy milk | Check fridge □□ no milk |
| 3:25 | Arrive home, milk in fridge | Leave for grocery |
| 3:30 | | |
| 3:35 | | Buy milk |
| 3:40 | | Arrive home, milk in fridge! |



# Shared Memory Synchronization

- Threads share memory
- Preemptive thread scheduling is a major problem
  - Context switch can occur at any time, even in the middle of a line of code (e.g., "X = X + 1;")
    - » Unit of atomicity **Machine instruction**
    - » Cannot assume anything about how fast processes make progress
  - Individual processes have little control over order in which processes run
- Need to be paranoid about what scheduler might do
- Preemptive scheduling introduces non-determinism

# Issues in Concurrency

- We want to run things *concurrently* for performance reasons
  - Particularly if we have multiple processors (even phones now typically have 2+ CPU cores)
  - Some of the Intel i7s have 6 cores. (i7-990x)
- But when multiple concurrent tasks (processes or threads) need to operate on some shared resources, things can get messy

# Race Condition

- **Two (or more) processes run in parallel and output depends on order in which they are executed**
- **ATM Example**
  - **SALLY: balance += $50;  BOB: balance -= $50;**
  - **Question: If initial balance is $500, what will final balance be?**

```
SALLY
X = ReadBalance(500)
X = X + 50
WriteBalance(X)
```

```
BOB
Y = ReadBalance(550)
Y = Y - 50
WriteBalance(Y)
```

**This (or reverse) is what you' d normally expect to happen.**

**Net: $500**

# Race Conditions

- **Two (or more) processes run in parallel and output depends on order in which they are executed**

- **ATM Example**
  - **SALLY: balance += $50; BOB: balance -= $50;**
  - **Question: If initial balance is $500, what will final balance be?**

**However, this (or reverse) can happen due to a race condition.**

| SALLY | BOB |
|---|---|
| `X = ReadBalance(500)` | |
| | `Y = ReadBalance(500)` |
| `X = X + 50` | |
| | `Y = Y - 50` |
| `WriteBalance(X)` | |
| | `WriteBalance(Y)` |

**Net: $450**

# Race Conditions

- If two processes or threads need to update some data at the same time, we may have a *race condition*
  - The name comes from the idea that the two are both *racing* each other to be the first to write or read the data
- These correctness problems are notoriously difficult to debug – problem only occurs when the timing is just right and is hard to reproduce
- Note that we don't actually need true parallelism here for a mistake to occur!
- Just *preemption* at the wrong time
- Multiple processors do make this sort of problem more likely to manifest, though

- Motivating Example:  Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# Mutual Exclusion

- The key trouble we ran into was that the read-update-write sequence on a shared resource could be interleaved between two processes

  ($read_1$-$read_2$-$update_2$-$write_2$-$update_1$-$write_1$)

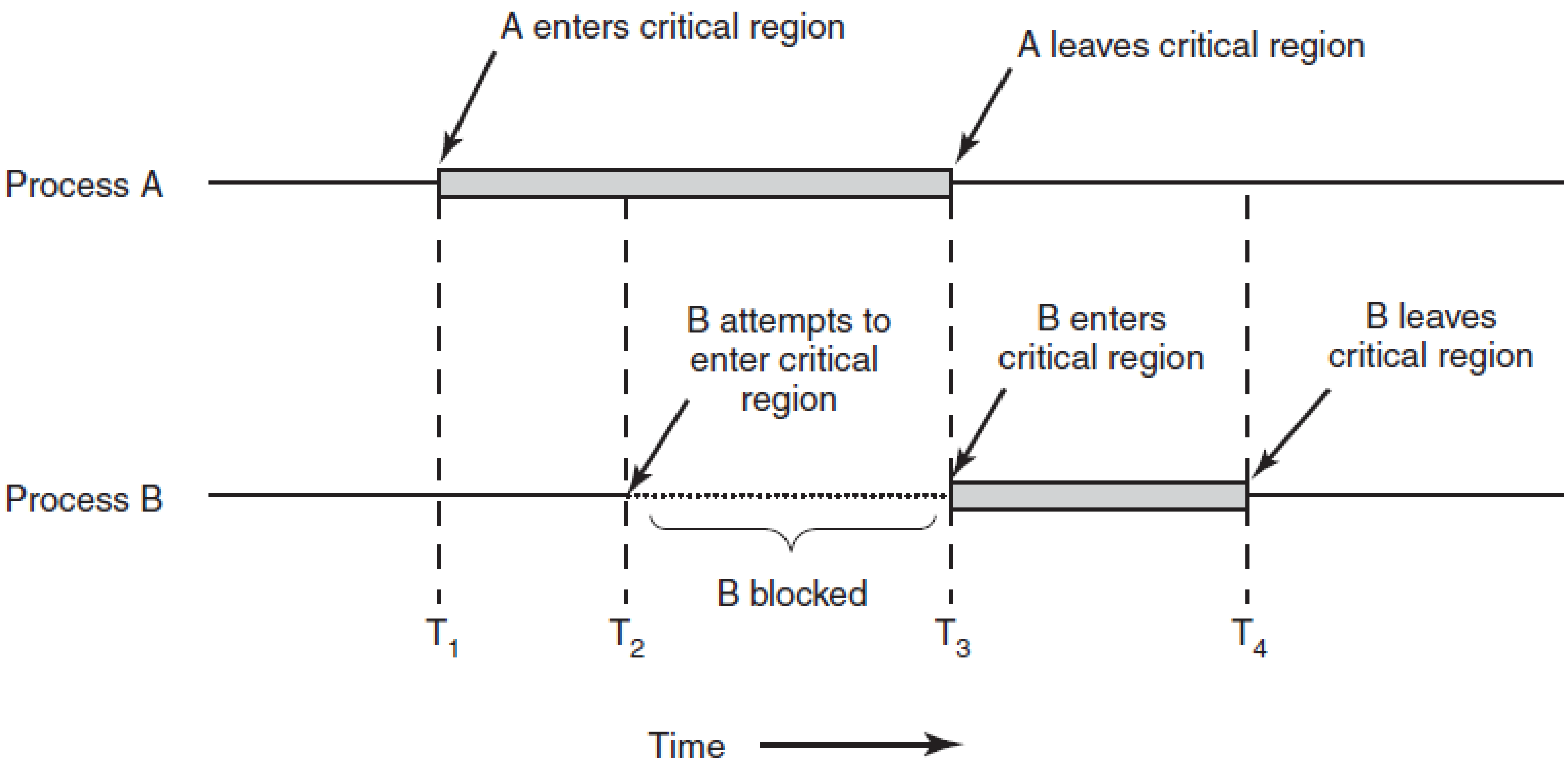
- The way to solve this is through *mutual exclusion* –making sure that only one process has access to the shared resource at a time

# Critical Regions/Sections

- We can formulate the problem by classifying what programs do into two parts
  - The majority of the time they do things that **don't require synchronization**; the things they do only affect non-shared resources
  - Some of the time, they need to access shared memory or files; we call this a *critical region* or *critical section* of the program
- If we can arrange it so that two programs are never in a critical section at the same time, we can avoid race conditions

# Requirements

1. No two processes may be simultaneously inside their critical regions
2. No assumptions may be made about speed or the number of CPUs
3. No process running outside its critical region may block any process
4. No process should have to wait forever to enter its critical region


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
$T_1$
$T_2$
$T_3$
$T_4$
Time

- Motivating Example: Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# Disabling Interrupts

- If we have only one processor, the only time things can go wrong is if we get preempted in the middle of our critical section
- Preemption depends on an interrupt (e.g., the timer interrupt) occurring
- Simple idea: disable interrupts in critical sections
    - Why is this a bad idea?

# Lock Variable

```
void acquire(struct spinlock *lk) {
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

- To enter a critical region, wait until the lock variable is 0, then set it to 1
- To leave the critical region, just set the lock back to 0
- What's wrong with this idea?

# Peterson's Algorithm

```c
#define FALSE 0
#define TRUE 1
#define N 2                              /* number of processes */

int turn;                                /* whose turn is it? */
int interested[N];                       /* all values initially 0 (FALSE) */

void enter_region(int process);          /* process is 0 or 1 */
{
  int other;                             /* number of the other process */
  other = 1 - process;                   /* the opposite of process */
  interested[process] = TRUE;            /* show that you are interested */
  turn = process;                        /* set flag */
  while (turn == process && interested[other] == TRUE) {
    // Do nothing
  }
}

void leave_region(int process) {         /* process: who is leaving */
  interested[process] = FALSE;           /* indicate departure from
                                            critical region */
}
```

# Peterson's Algorithm

- Each process indicates its *interest* by setting its entry in the "interested" array

- Then, it sets the global *turn* variable to its own process number

- Finally, loop until *turn* indicates that it's our turn *and* we see that the other process is no longer interested

- There is still a race – but regardless of the winner, only one process will get to enter its critical region

# Hardware Support

- We can have a simpler solution if the hardware helps us out a bit with an *atomic instruction*

- For example, "test and set lock": TSL RX, LOCK

  - ATOMICALLY reads the memory at address LOCK into RX and then stores a nonzero value back into LOCK

  - No other processor is allowed to access the memory at address LOCK until TSL is done

# Using TSL for Locks

```
enter_region:
        TSL REGISTER,LOCK        | copy lock to register and set lock to 1
        CMP REGISTER,#0          | was lock zero?
        JNE enter_region         | if it was not zero, lock was set, so loop
        RET                      | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0             | store a 0 in lock
        RET                      | return to caller
```

# x86 Atomic Locking

- A similar instruction exists on x86: **xchg REG, MEM**
- Atomically exchanges the contents of a register and a memory location
- You can see that this is equivalent to TSL if the register is set to 1

# xv6 Lock using XCHG

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

- Motivating Example:  Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# Busy Waiting

- Whenever a process is waiting to enter a critical section under these schemes, it sits in an infinite loop
- This wastes CPU time
- It can also interact badly with scheduling

# Priority Inversion

- Suppose we're using priority scheduling, and we have a high-priority process **H** and a low priority process **L**
    - So whenever **H** is runnable, it will always be chosen over **L**
- Now, **L** enters a critical region, but is then preempted to run **H**
- **H** wants to enter the critical region, but the lock is already held by **L**, so it enters a busy wait loop
- But now **H** is *always* runnable, and will *always* be chosen over **L**, so **L** can never leave its critical region and the system is stuck

# The Martian Inversion

The Mars Pathfinder mission used the VxWorks realtime operating system with priority scheduling

Meterological data gathering was low-priority task, communication task was medium-priority

Both shared a data bus, controlled by a lock

This caused a classic priority inversion, hanging the lander until the watchdog timer reset it

# The Martian Inversion

This was debugged from 140 million miles away by examining system log data

Fixed by uploading a snippet of C code that turned on *priority inheritance* for the lock

Priority inheritance says that a process holding a lock is elevated to the highest priority of anything waiting for the lock

# Sleep and Wakeup

- Instead of waiting in a loop, wasting CPU, we would like to put the waiting process to sleep, waking up when the lock is released
- Often, sleep and wakeup take as parameters the address of some variable so we can match up sleeps with wakeups
- E.g., in xv6:
  - sleep(void *chan)
  - wakeup(void *chan)

# Producer-Consumer

- Imagine we have two tasks, one that *produces* items and places them in a fixed-size buffer, and one that *consumes* them
  - An example you have seen already – a pipe!
- If the buffer is full, the producer sleeps until there's space
- If the buffer is empty, the consumer sleeps until there's data available

# The Producer-Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer); /* was buffer empty? */
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count – 1;              /* decrement count of items in buffer */
                if (count == N – 1) wakeup(producer); /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

# The Lost Wakeup Problem

- Suppose the buffer is empty, and consumer checks that count is 0
- Just before consumer actually goes to sleep, it gets preempted, and producer puts something in the buffer
- Since count is now 1, producer tries to wake up consumer, but consumer isn't asleep yet, so it does nothing
- Control returns to the consumer, who goes to sleep, and never wakes up – it has missed its wakeup

# Solving the Lost Wakeup Problem

- We can have the producer and the consumer share a lock
- The consumer acquires the lock, checks the value of count, and goes to sleep, passing the lock to the sleep function, which releases it
- The producer acquires the lock before calling wakeup; if the process is not yet fully asleep, it will wait, ensuring that the wakeup is sent *after* the process is actually asleep

- Motivating Example:  Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# Mutexes

- A mutex is a way to have mutual exclusion without busy waiting
- Implementation is very similar to the busy-wait version of critical regions, but instead of looping, **we yield the CPU**

# Mutex Code

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again
ok:     RET                         | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET                         | return to caller
```

- Motivating Example:  Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Mutexes
- Locks and Interrupts

# Locks and Interrupts

- Here is an a non-reentrant interrupt handler. How can you fix it:

```
char temp;

void interrupt_handler(void) {
    char x1, x2;

    x1 = inb(0x120); // read from I/O port
    x2 = inb(0x121); // read from I/O port

    acknowledge_interrupt();

    temp = x1;
    x1 = x2;
    x2 = temp;

    printf("Got and swapped x1=%d x2=%d\n", x1, x2);
}
```

# Locks and Interrupts

- It's tempting to just add a lock around the swap:

```
char temp;

void interrupt_handler(void) {
    char x1, x2;

    x1 = inb(0x120); // read from I/O port
    x2 = inb(0x121); // read from I/O port

    acknowledge_interrupt();

    acquire(&lock);
    temp = x1;
    x1 = x2;
    x2 = temp;
    release(&lock);

    printf("Got and swapped x1=%d x2=%d\n", x1, x2);
}
```

# Locks and Interrupts

- But now consider what happens if we get an interrupt that calls interrupt_handler after we acquire the lock
- We re-enter interrupt_handler, which tries to acquire the lock...
- But it can't! The lock is held by the earlier call to the interrupt handler, which

# Avoiding Interrupt Deadlocks

- To get around this problem, we must make sure that when a lock is held by an interrupt handler we disable interrupts
- xv6 actually goes further – all locks in the kernel disable interrupts on acquire and re-enable on release