CSCC 69H

Operating Systems
Sina Meraji
U of T

1

Announcement

- Check discussion board for announcements

- A1 is posted
- Tutorials
  - Ahmed-Eysman   BA1180
  - Fan- Leung   BA3008
  - Li-Sarmiento   BA2155
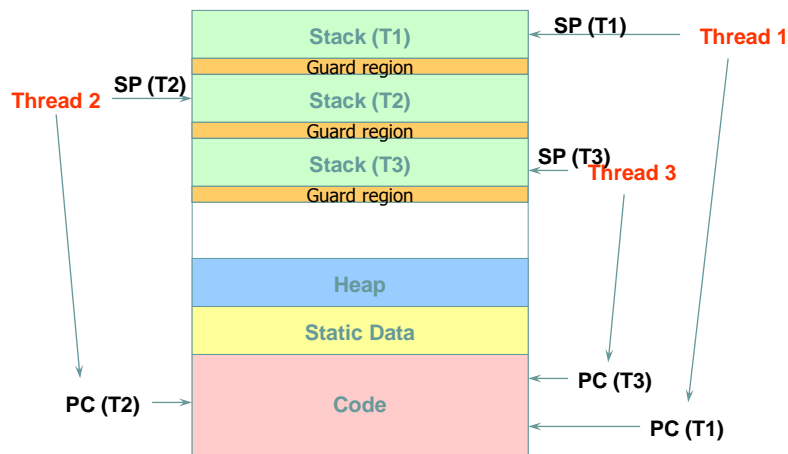 - Semenkov-Zou   BA2139

2

Recap:
Process Creation: Unix

- In Unix, processes are created using fork()
  int fork()
- fork()
  - Creates a new address space
  - Initializes the address space with a **copy** of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)

3

Recap: Threads

4

TODAY:

- System Calls
- Intro to Synchronization

5

Bootstrapping

- Hardware stores small program in non-volatile memory
    - BIOS – Basic Input Output System
    - Knows how to access simple hardware devices
        - Disk, keyboard, display
- When power is first supplied, this program executes
- What does it do?
    - Checks that RAM, keyboard, and basic devices are installed and functioning correctly
    - Scans buses to detect attached devices and configures new ones
    - Determines boot device (tries list of devices in order)
    - Reads first sector from boot device and executes it (bootloader)
    - Bootloader reads partition table, finds active partition, reads secondary bootloader
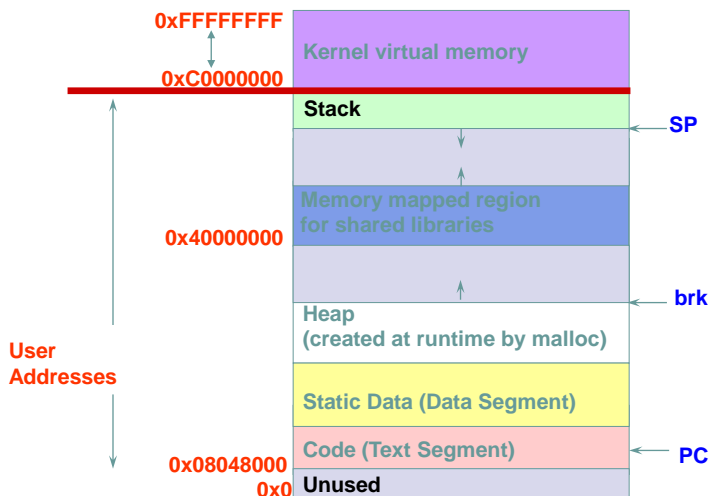   - Secondary bootloader reads OS into memory and executes it

6

Operating System Startup

- Machine starts in system mode, so kernel code can execute immediately
- OS initialization:
  - Initialize internal data structures
    - Machine dependent operations are typically done first
  - Create first process
  - Switch mode to user and start running first process
  - Wait for something to happen
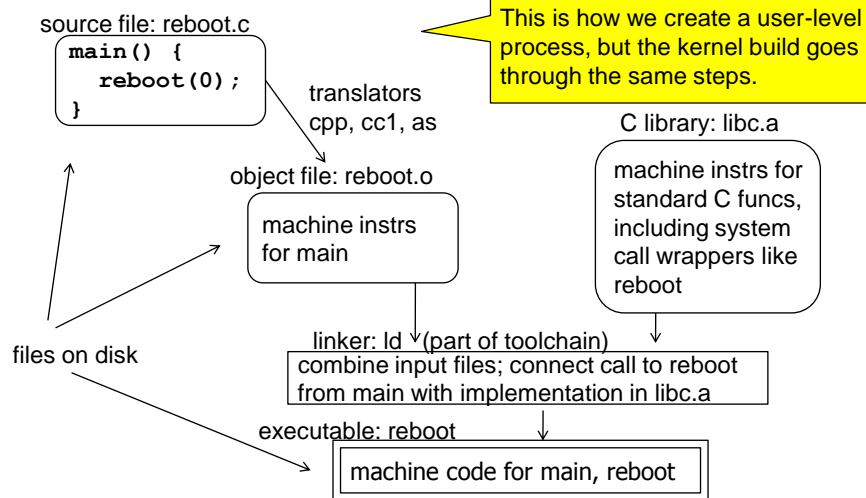   - OS is entirely driven by external events

7

Memory Layout (Linux, x86)

0xFFFFFFFF	Kernel virtual memory
0xC0000000	Stack ← SP
	Memory mapped region for shared libraries
0x40000000	
	Heap (created at runtime by malloc) ← brk
	Static Data (Data Segment)
0x08048000	Code (Text Segment) ← PC
0x0	Unused

User Addresses

10

From Program to Process… 1

source file: reboot.c

```
main() {
  reboot(0);
}
```

This is how we create a user-level process, but the kernel build goes through the same steps.

translators
cpp, cc1, as

C library: libc.a

object file: reboot.o

machine instrs for main

machine instrs for standard C funcs, including system call wrappers like reboot

files on disk

linker: ld (part of toolchain)
combine input files; connect call to reboot from main with implementation in libc.a

executable: reboot

machine code for main, reboot

CSC369   12

Unix Shells

```
while (1) {
  char *cmd = read_command();
  int child_pid = fork();
  if (child_pid == 0) {
   exec(cmd); //cmd=executable name(reboot)
  } else {
     wait(child_pid);
  }
}
```

Process Creation: Unix (2)

- Wait a sec ... How do we actually start a new program?
  int exec(char *prog, char *argv[])
- exec()
  - Stops the current process
  - Loads the program "prog" into the process' address space
  - Initializes hardware context and args for the new program
  - Places the PCB onto the ready queue
  - Note: It **does not** create a new process

Requesting OS Services

- Operating System and user programs are isolated from each other
- But OS provides service to user programs…
- So, how do they communicate?

User Process

Memory Mgmt	Scheduling	File System	I/O

15
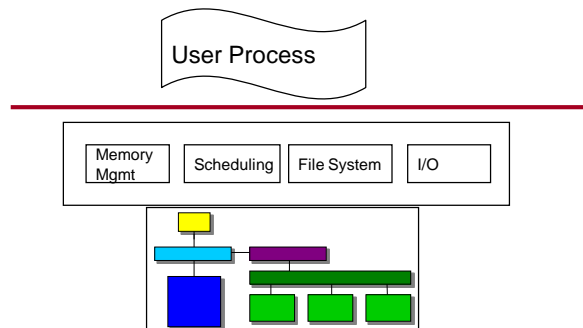
Boundary Crossings

- Getting to kernel mode
  - Boot time (not really a crossing, starts in kernel)
  - Explicit system call – request for service by application
  - Hardware interrupt
  - Software trap or exception
  - Hardware has table of "Interrupt service routines"
- Kernel to user
  - Jumps to next application instruction

16

System Calls for Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some of the major system calls.

17

System Calls for File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some of the major system calls.

18
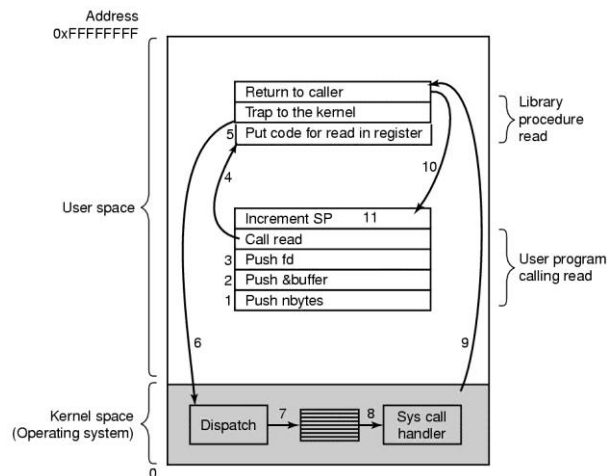
System Calls

Read(fd, buffer, nbytes).

19

System Call Interface

- User program calls C library function with arguments
- C library function arranges to pass arguments to OS, including a system call identifier
- Executes special instruction to trap to system mode
  - Interrupt/trap vector transfers control to a system call handling routine
- Syscall handler figures out which system call is needed and calls a routine for that operation
- How does this differ from a normal C language function call? Why is it done this way?
  - Extra level of indirection through system call handler, rather than direct control flow to called function
  - Hardware support is needed to enforce separation of userspace and kernel

20

System Call Operation

- Kernel must verify arguments that it is passed
  - Why?
- A fixed number of arguments can be passed in registers
  - Often pass the address of a user buffer containing data (e.g., for write())
  - Kernel must copy data from user space into its own buffers
- Result of system call is returned in register

21

Intro to Synchronization

23

Cooperating Processes

- A process is *independent* if it cannot affect or be affected by the other processes executing in the system
- No data sharing $\Rightarrow$ process is independent
- A process is *cooperating* if it is not independent
- Cooperating processes must be able to communicate with each other and to synchronize their actions

24

Interprocess Communication

- Cooperating processes need to exchange information, using either
  - Shared memory (e.g. fork())
  - Message passing
- Message passing models
  - Send(P, msg) – send msg to process P
  - Receive(Q, msg) – receive msg from process Q

25

Motivating Example

- Suppose we write functions to handle withdrawals and deposits to a bank account:

```
Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    return balance;
}
```

```
Deposit(account, amount) {
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    return balance;
}
```

- Idea: Create separate threads for each action, which may run at the bank's central server

- What's wrong with this implementation?
  - Think about potential schedules for these two threads

Motivating Example

- Suppose we write functions to handle withdrawals and deposits to a bank account:

```
Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    return balance;
}
```

```
Deposit(account, amount) {
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    return balance;
}
```

- Suppose you share this account with someone and the balance is $1000
- You each go to separate ATM machines - you withdraw $100 and your S.O. deposits $100

27

Interleaved Schedules

- The problem is that the execution of the two processes can be interleaved:

Schedule A

```
balance = get_balance(acct);
balance = balance - amt;
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
```
Context switch

```
put_balance(acct, balance);
```

- What is the account balance now?
- Is the bank happy with our implementation?
  - Are you?

28

Interleaved Schedules

- The problem is that the execution of the two processes can be interleaved:

Schedule A

```
balance = get_balance(acct);
balance = balance - amt;
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
```

```
put_balance(acct, balance);
```

Context switch

Schedule B

```
balance = get_balance(acct);
balance = balance - amt;
```

```
balance = get_balance(acct);
balance = balance + amt;
```

```
put_balance(acct, balance);
```

```
put_balance(acct, balance);
```

- What is the account balance now?
- Is the bank happy with our implementation?
  - Are you?

29

What Went Wrong

- Two concurrent threads manipulated a *shared resource* (the account) without any synchronization
  - Outcome depends on the order in which accesses take place
    - This is called a *race condition*
- We need to ensure that only one thread at a time can manipulate the shared resource
  - So that we can reason about program behavior
  - → We need *synchronization*

30

Example continued …

- Could the same problem occur with a simple shared variable:
  - T_1 and T_2 share variable X
  - T_1 increments X   (X := X+1)
  - T_2 decrements X   (X := X-1)
  - At the machine level, we have:

```
T₁:     LOAD X
        INCR
        STORE X
```

```
T₂:     LOAD X
        DECR
        STORE X
```

- Same problem of interleaving can occur!

31

Mutual Exclusion

- Given:
  - A set of n threads, $T_0, T_1, …, T_n$
  - A set of resources shared between threads
  - A segment of code which accesses the shared resources, called the *critical section, CS*

```
Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    return balance;
}
```
CS

- We want to ensure that:
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

32

14

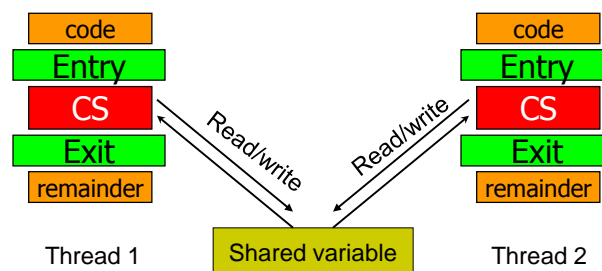# Aside: What program data is shared between threads?

- Local variables are not shared (*private*)
  - Each thread has its own stack
  - Local vars are allocated on this private stack
- Global variables and static objects are *shared*
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objs are *shared*
  - Allocated from heap with malloc/free or new/delete

33

The Critical Section Problem

- Design a protocol that threads can use to cooperate


```
code          code
Entry         Entry
CS    Read/write  Read/write   CS
Exit          Exit
remainder     remainder
Thread 1   Shared variable   Thread 2
```
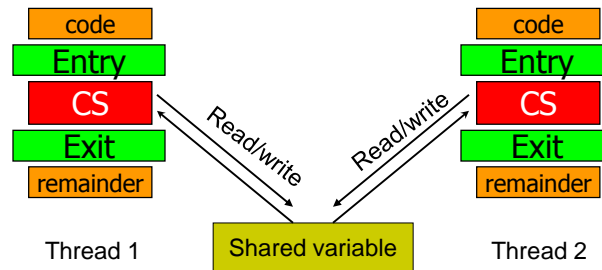
- Each thread must request permission to enter its CS, in its *entry* section
- CS may be followed by an *exit* section
- Remaining code is the *remainder* section

34

Critical Section Requirements (1)

- Design a protocol that threads can use to cooperate

code		code
Entry		Entry
CS		CS
Exit		Exit
remainder		remainder

Read/write ← → Read/write

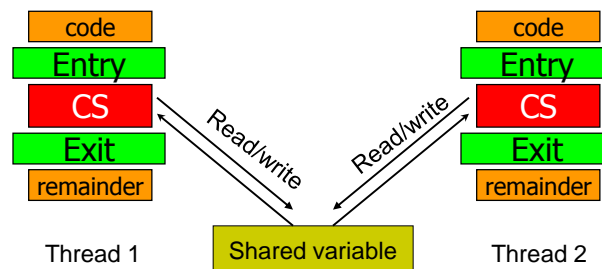Thread 1   Shared variable   Thread 2

1) Mutual Exclusion
  - If one thread is in the CS, then no other is

35

Critical Section Requirements (2)

- Design a protocol that threads can use to cooperate

code		code
Entry		Entry
CS		CS
Exit		Exit
remainder		remainder

Read/write ← → Read/write

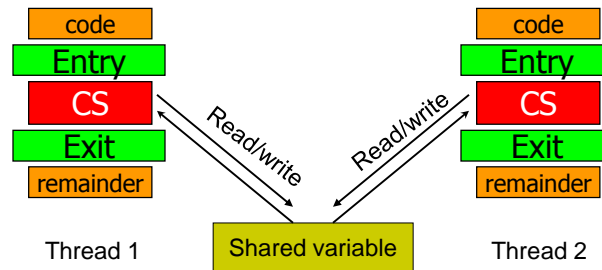Thread 1   Shared variable   Thread 2

2) Progress
  - If no thread is in the CS, and some threads want to enter CS, it should be able to enter in definite time

36

16

Critical Section Requirements (3)

- Design a protocol that threads can use to cooperate

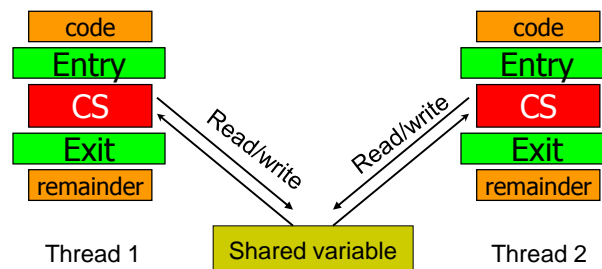|		
code		code
Entry		Entry
CS	Read/write Read/write	CS
Exit		Exit
remainder		remainder
Thread 1	Shared variable	Thread 2

3) Bounded waiting (no starvation)

- If some thread T is waiting on the CS, then there is a limit on the number of times other threads can enter CS before this thread is granted access

37

Critical Section Requirements (4)

- Design a protocol that threads can use to cooperate

code		code
Entry		Entry
CS	Read/write Read/write	CS
Exit		Exit
remainder		remainder
Thread 1	Shared variable	Thread 2

4) Performance

- The overhead of entering and exiting the CS is small with respect to the work being done within it

38

Critical Section Requirements

1) Mutual Exclusion
   - If one thread is in the CS, then no other is
2) Progress
   - If no thread is in the CS, and some threads want to enter CS, it should be able to enter in definite time
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the CS, then there is a limit on the number of times other threads can enter CS before this thread is granted access
- Performance
   - The overhead of entering and exiting the CS is small with respect to the work being done within it

39

Some Assumptions & Notation

- Assume no special hardware instructions, no restrictions on the # of processors (for now)
- Assume that basic machine language instructions (LOAD, STORE, etc.) are *atomic:*
   - If two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order
- If only two threads, we number them T_0 and T_1
   - Use T_i to refer to one thread, T_j for the other (j=1-i) when the exact numbering doesn't matter
- Let's look at one solution…

40

2-Thread Solutions: 1st Try

- Let the threads share an integer variable *turn* initialized to 0 (or 1)
- If *turn=i* , thread T_i is allowed into its CS

```
My_work(id_t id) {   /* id_t can be 0 or 1 */
        ...
        while (turn != id) ;/* entry section */
        /* critical section, access protected resource */
        turn = 1 - id;        /* exit section */
        ...                   /* remainder section */
}
```

✓ Only one thread at a time can be in its CS
✗ Progress is not satisfied
  - Requires strict alternation of threads in their CS: if *turn=0*, T_1 may not enter, even if T_0 is in the code section

41

2-Thread Solutions: 2nd Try

- First attempt does not have enough info about state of each process. It only remembers which process is allowed to enter its CS
- Replace turn with a shared flag for each thread
  - **boolean** flag[2] = {false, false}
  - Each thread may update its own flag, and read the other thread's flag
  - If flag[i] is true, T_i is ready to enter its CS

42

A Closer Look at 2nd Attempt

```
My_work(id_t id) {   /* id can be 0 or 1 */
      ...
      while (flag[1-id]) ;/* entry section */
      flag[id] = true;   /* indicate entering CS */
      /* critical section, access protected resource */
      flag[id] = false;  /* exit section */
      ...                /* remainder section */
}
```

- Mutual exclusion is not guaranteed
  - Each thread executes *while* statement, finds *flag* set to false
  - Each thread sets own *flag* to *true* and enters CS
- Can't fix this by changing order of testing and setting *flag* variables (leads to *deadlock*)

43

2-Thread Solutions: 3rd Try

- Combine key ideas of first two attempts for a correct solution
- The threads share the variables *turn* and *flag* (where *flag* is an array, as before)

```
Enter_region(id_t id) {    /* id can be 0 or 1 */

      flag[id] = true;    /* indicate entering CS */
      turn = id;
      while (turn == id && flag[other] == true);
}
```

```
Leave_region(id_t id) {    /* id can be 0 or 1 */

      flag[id] = false;
}
```

44

2-Thread Solutions: 3rd Try

- Imagine two threads i and j execute `Enter_region()` at the same time:

Thread i	Thread j

`flag[i] = true;`
`turn = i;`
`while(turn==i && flag[j]==true);`
	`flag[j] = true;`
	`turn = j;`
	`while(turn==j && flag[i]==true);`

- Basic idea: if both try to enter at the same time, *turn* will be set to both 0 and 1 at roughly the same time. Only one assignments will last. The final value of *turn* decides who gets to go first.

- This is the basis of *Peterson's Algorithm*

45

Peterson's Solution

```
#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) /* null statement */ ;
}

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

Peterson's solution for achieving mutual exclusion.

46

Higher-level Abstractions for CS's

- Locks
  - Very primitive, minimal semantics
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
- Messages
  - Simple model for communication & synchronization
  - Direct application to distributed systems

47

Synchronization Hardware

- To build these higher-level abstractions, it is useful to have some help from the hardware
- On a uniprocessor:
  - Disable interrupts before entering critical section
  - Prevents context switches
  - Doesn't work on multiprocessor
- Need some special atomic instructions

48

Atomic Instructions:
Test-and-Set Lock (TSL)

- Test-and-set uses a *lock* variable
  - Lock == 0   => nobody is using the lock
  - Lock == 1   => lock is in use
  - In order to acquire lock, must change it's value from 0=>1

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
```

- Hardware executes this atomically!

49

Atomic Instructions: Test-and-Set

- The semantics of test-and-set are:
  - Record the old value of the variable
  - Set the variable to some non-zero value
  - Return the old value

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
```

- *lock* is always *True* on exit from test-and-set
  - Either it was *True* (locked) already, and nothing changed
  - or it was *False* (available), but the caller now holds it
- Return value is either *True* if it was locked already, or *False* if it was previously available

50

A Lock Implementation

- There are two operations on locks: *acquire()* and *release()*

```
boolean lock;

void acquire(boolean *lock) {
        while(test_and_set(lock));
}

void release(boolean *lock) {
        *lock = false;
}
```

- This is a *spinlock*
  - Uses *busy waiting* - thread continually executes *while* loop in *acquire()*, consumes CPU cycles

51

Using Locks

Function Definitions

```
Withdraw(acct, amt) {

    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

```
Deposit(account, amount) {

    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

Possible schedule

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;
```

```
acquire(lock);
```

```
put_balance(acct, balance);
release(lock);
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
release(lock);
```

52

Next Week

- More on Synchronization

53

Announcement

- Check course website regularly

- Attend Tutorials

54