CSC369 Lecture 3

Larry Zhang, September 28, 2015

Assignment 1 submission

- ➔ Use the **web interface**, submit just “interceptor.c”
- ➔ Subversion repository provided separately, independent of submission
- ➔ On MarkUs, still need to create group or select “work individually” so we can create corresponding SVN repo.

Assignment 1 tips

- ➔ Understand the code base, which takes time
- ➔ Read the comments in the code (they are quite comprehensive), and write your own comments when necessary (good coding style)
- ➔ Only do testing in virtual machine
- ➔ Backup code regularly
- ➔ Must compile without error or warning
- ➔ Find usage of kernel functions by going to kernel files or Googling.
- ➔ If you haven’t started yet, do it soon!

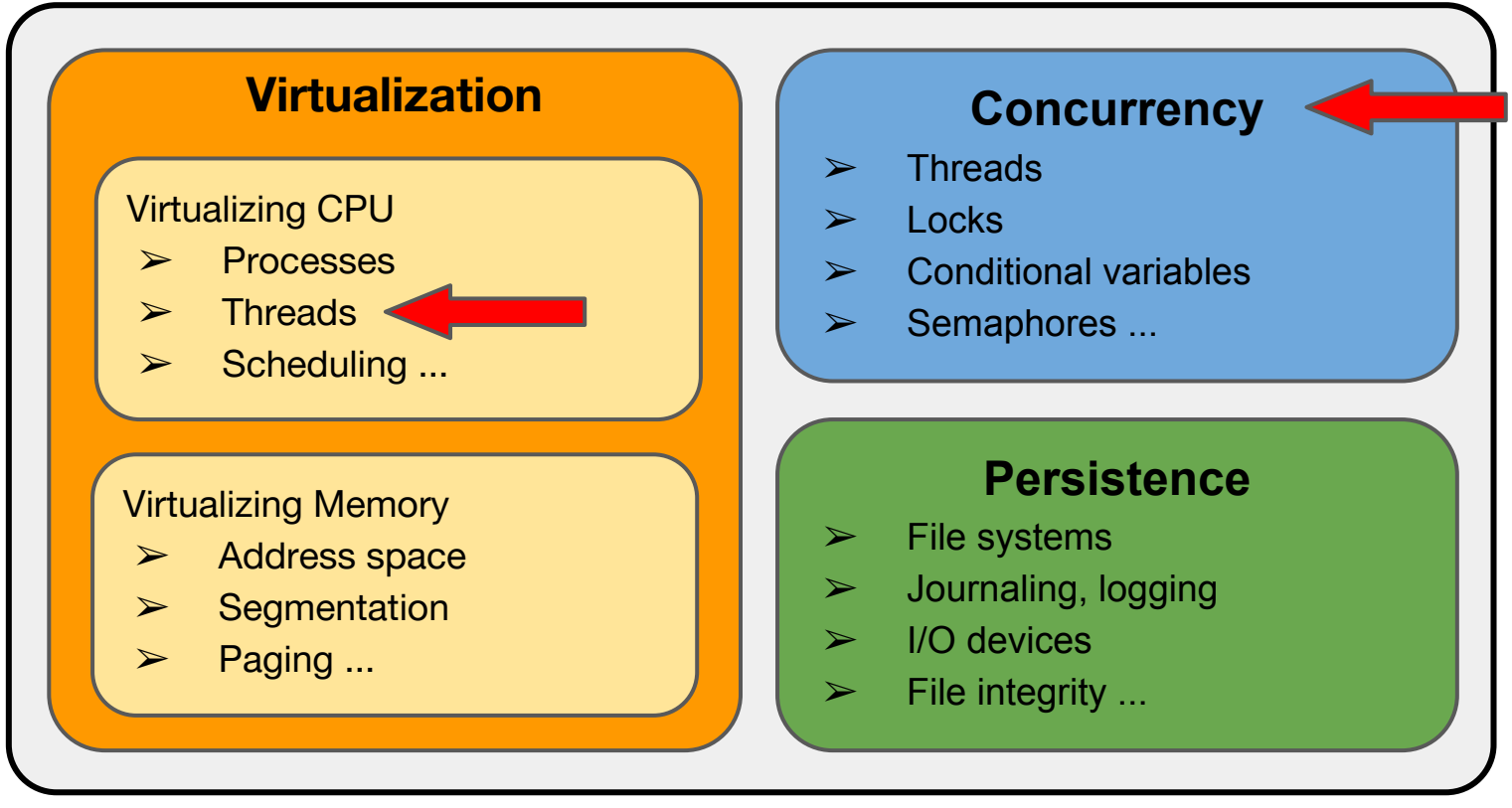
C Programming Video Lectures

https://mcs.utm.utoronto.ca/~pcrs/C-programming/

Today’s outline

- ➔ Thread
- ➔ Concurrency

We are here


Virtualization
Virtualizing CPU
Processes
Threads
Scheduling ...
Virtualizing Memory
Address space
Segmentation
Paging ...
Concurrency
Threads
Locks
Conditional variables
Semaphores ...
Persistence
File systems
Journaling, logging
I/O devices
File integrity ...

Review: What happens when a process calls fork() ?

- ➔ Creates a new PCB and a new address space
- ➔ Initialize the **new** address space which a **copy** of entire content of the parent's address space
- ➔ Initialize resources to point to parent's resources (e.g., open file)

These things take time.

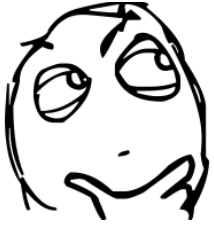
Review: What happens in a context switch from Process A to Process B?

- ➔ Save reg(A) to kernel stack A
- ➔ In kernel mode, save reg(A) to PCB(A)
- ➔ Restore reg(B) from PCB(B)
- ➔ Switch to kernel stack B
- ➔ Restore reg(B) from kernel stack B

These things also take time.

What we have learned so far

- ➔ An OS that **virtualizes** its CPU into abstractions of program executions, i.e., **processes**
- ➔ We can do pretty much everything we need using **processes**, maybe that's all the CPU-virtualization we need. Right?
- ➔ NOT REALLY, it not enough yet.



Example


```
while (1) {
    int sock = accept();
    if (0 == fork()) {
        handle_request();
        close(sock);
        exit(0);
    }
}
```

A web server

- ➔ a process listens for requests
- ➔ when a new request comes in, create a new child process to handle this request
- ➔ multiple requests can be handled at the same time by different child processes

Example Web Server


```
while (1) {
    int sock = accept();
    if (0 == fork()) {
        handle_request();
        close(sock);
        exit(0);
    }
}
```

Truth is: this implementation is very **inefficient**.

Why?

Web Server using fork() is inefficient

- ➔ **Time**: creating a **new process** for each incoming request **take time**. Lots of **context switches** between processes also **take time**.
- ➔ **Space**: each child process has their **own copy** of the address space, but they are almost the **same**.
- ➔ **Inter-process communication (IPC)**: the child process may generate a response which will be used by the parent process. **Extra overhead** to make this happen (use a file, pipe, socket, or shared memory).

There must be a better way!

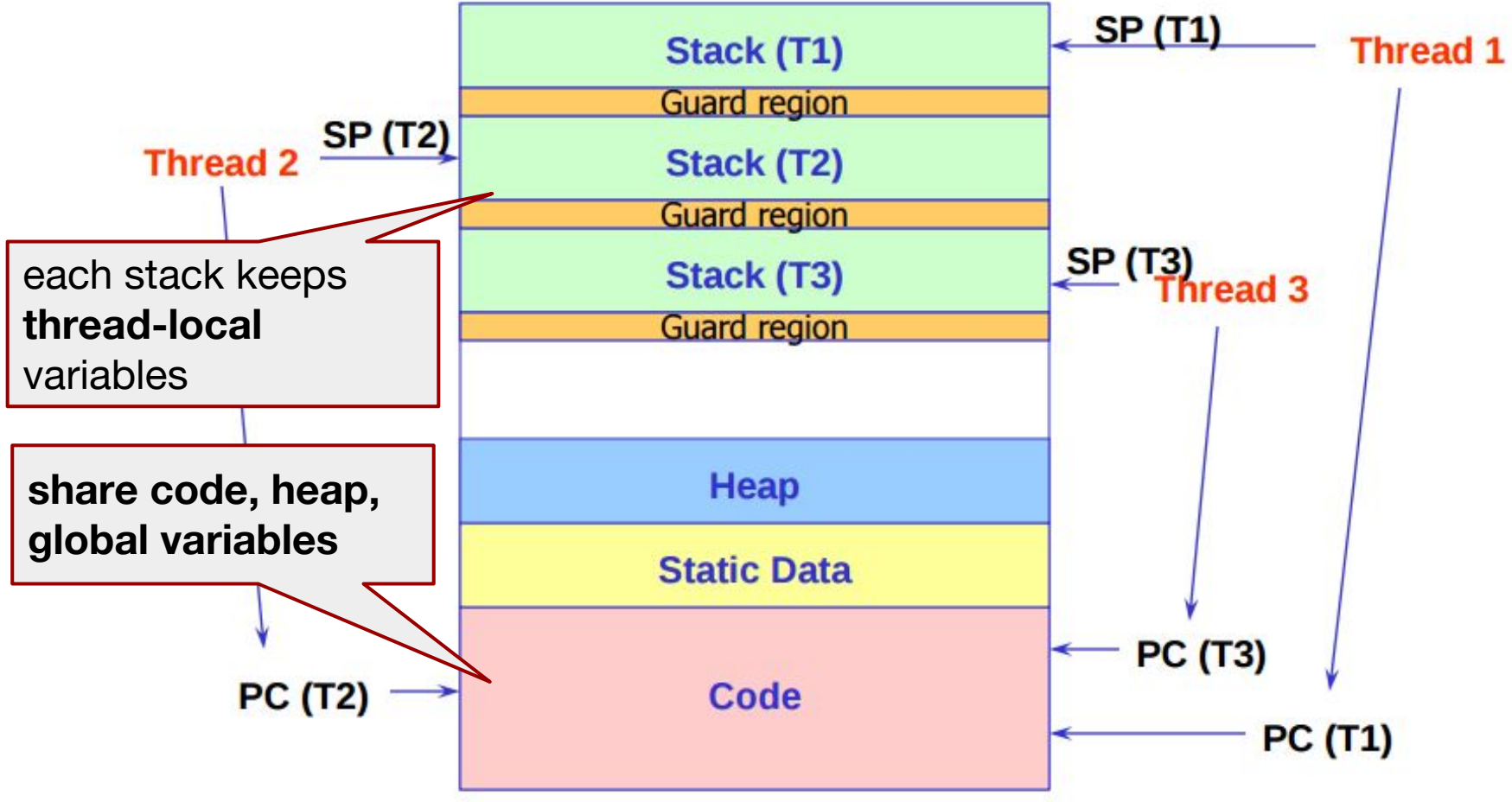
Thread

a new abstraction of program execution

What's new about threads

- ➔ A normal process is a running program with a **single point of execution**, i.e., a single PC.
- ➔ A **multi-threaded program** has **multiple points of execution**, i.e., multiple PCs.
  - ◆ One thread is a single **flow control** through a program
- ➔ Multiple threads share the **same address space**
  - ◆ no need to copy address space when creating thread,
  - ◆ no need to switch address space when context switch
- ➔ There are **multiple stacks** in the address space, one for each thread.

Multi-threaded Process Address Space



Web Server using Threads

```
int global_counter = 0;
web_server() {
  while (1) {
    int sock = accept();
    thread_create(handle_request, sock);
  }
}
handle_request(int sock) {
  process request…
  ++global_counter;
  close(sock);
}
```

Different threads share and modify this global variable.

The counter thing doesn't work using fork(). Why?

```
int global_counter = 0;
while (1) {
    int sock = accept();
    if (0 == fork()) {
        handle_request();
        ++global_counter;
        close(sock);
        exit(0);
    }
}
```

counter is always 0 in the parent process, since changes only happen in children.

If you really want to make this work using fork(), need **inter-process communication (IPC)**

API: POSIX threads, a.k.a. **pthreads**

- ➔ Standardized C language thread programming API
  - ◆ A tutorial: https://computing.llnl.gov/tutorials/pthreads/
- ➔ pthreads specifies the interface of using threads, but not how threads are implemented in OS
  - ◆ different implementations include: kernel-level threads, user-level threads, or hybrid
  - ◆ Read this article if interested: http://goo.gl/RCNDGI
  - ◆ Linux pthreads using kernel-level threads

Summary of threads

- ➔ Lighter weight
  - ◆ faster to create and destroy
  - ◆ faster context switch
- ➔ Sharing
  - ◆ threads can solve a single problem concurrently and can easily share code, heap and global variables
- ➔ Concurrent programming performance gains
  - ◆ overlapping computations and I/O

If POSIX threads are a good thing, perhaps I don't want to know what they're better than.

-- Rob Pike

Some people, when confronted with a problem, think, "I know, I'll use regular expressions." Now they have two problems.

Some people, when confronted with a problem, think, "I know, I'll use threads!" Now they have 10 problems.

-- Bill Schindler

“… a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of ***multithreaded*** *systems be* ***insane****. Were they sane, they could not understand their programs.”*

-- Edward A. Lee

let’s learn about

concurrency

Motivating example: bank machine


```
Withdraw(account, amount):
1    balance = get_balance(account)
2    balance = balance - amount
3    put_balance(account, balance)
4    return balance
```

```
Deposit(account, amount):
5    balance = get_balance(account)
6    balance = balance + amount
7    put_balance(account, balance)
8    return balance
```

Let's say I have **$1000** originally, then I called

`Withdraw(acc, 100)`
and
`Deposit(acc, 100)`
concurrently

How much money do I have now?

Problem: **interleaved** execution of the two processes

Possible schedule A

```
1   balance = get_balance(acc)
2   balance = balance - 100
```

```
5   balance = get_balance(acc)
6   balance = balance + 100
7   put_balance(acc, balance)
```

```
3   put_balance(acc, balance)
```

balance = 900

Possible schedule B

```
1   balance = get_balance(acc)
2   balance = balance - 100
```

```
5   balance = get_balance(acc)
6   balance = balance + 100
```

```
3   put_balance(acc, balance)
```

```
7   put_balance(acc, balance)
```

balance = 1100

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 behaviour, in a **sane** way
  - ◆ We need **synchronization**.

Concurrency problems can occur at very small scale

- ➔ Suppose two threads T1 and T2 share variable X
- ➔ T1 does **X = X + 1**
- ➔ T2 does **X = X - 1**
- ➔ Could these two one-liners have concurrency problem?
  - ◆ **YES**! Because at the **machine instruction** level we have

```
T1: LOAD X from memory
    INCR in register
    STORE X to memory
```

```
T2: LOAD X from memory
    DECR in register
    STORE X to memory
```

Exactly the same situation as the bank account!

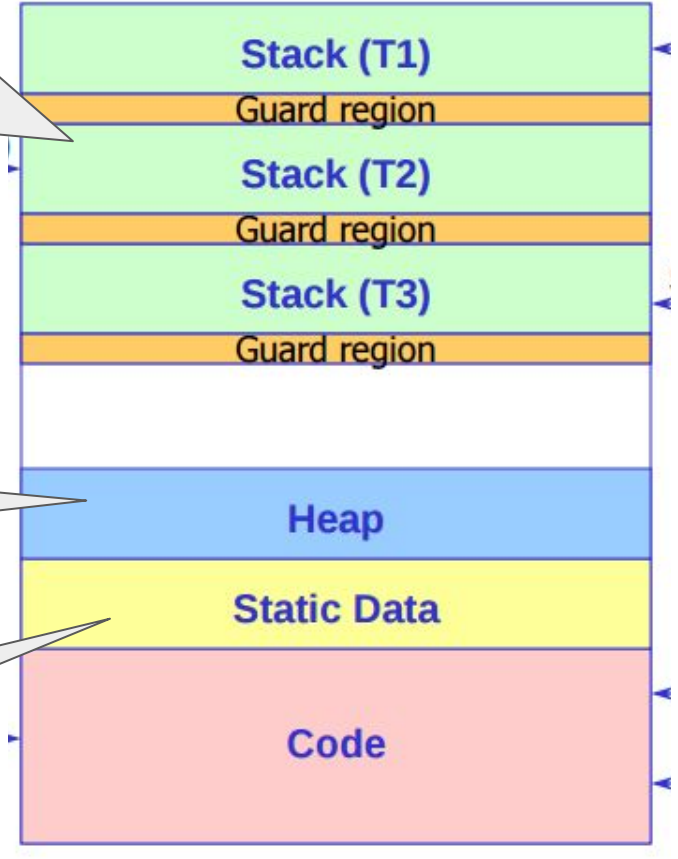
What is shared / not shared in a multithreaded program?

Local variables are **not shared** (private)

- ➔ each thread has its own stack
- ➔ local variables are allocated on its own stack
- ➔ Never pass / share / store a pointer to a local variable on another thread's stack!

Dynamic objects are **shared**, allocated from **heap** with malloc/free or new/delete, accessible by any thread.

Global variables are **static objects** are **shared**, accessible by any thread.



The Critical Section Problem

a systematic model for synchronization problems

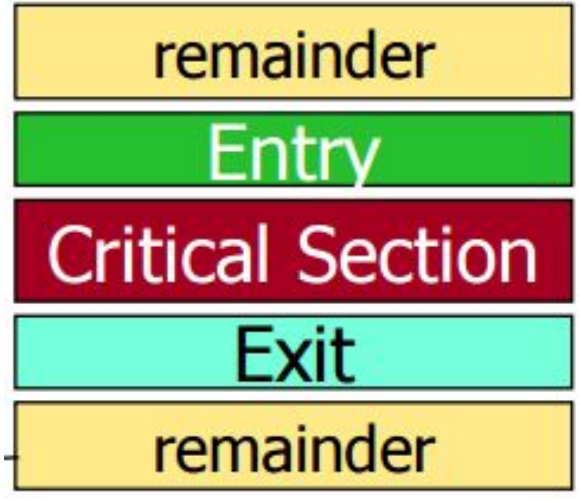
The critical section problem

- ➔ Given
  - ◆ a set of n threads, T_0, T_1, …, T_{n-1}
  - ◆ a set of resources shared between threads
  - ◆ **Critical Section (CS)**: a segment of code which accesses the shared resources
- ➔ A **race condition** arises when multiple threads enter the **critical section** roughly at the same time, both attempt to update the shared data, lead to surprising outcomes.
- ➔ An **indeterminate** program consists of one or more **race conditions**; the output of the program varies from run to run, depending on which thread runs when.

The critical section problem

➔ Design a protocol that threads can use to cooperate

- Each thread requests permission to enter CS in its **Entry** section.
- CS may be followed by an **Exit** section
- Remaining code is called the **Remainder** section



Critical section problem **solution**: the **requirements**

- ➔ **Mutual Exclusion**
  - ◆ if one thread is in the CS, then no other is
- ➔ **Progress**
  - ◆ threads waiting for access to CS are not prevented from entering by threads not in the CS; threads do not wait indefinitely.
- ➔ **Bounded waiting (no starvation)**
  - ◆ All waiting threads are guaranteed to eventually get access to the CS
- ➔ **Performance**
  - ◆ The overhead of entering and exiting the CS is small compared to the work done within it.

Assumptions & Notations

- ➔ Assume no special hardware instructions, no restrictions on the number of processors (for now).
- ➔ Assume that basic machine instructions (LOAD, STORE, etc) are **atomic**:
  - ◆ nothing can cut into the middle of an **atomic** instruction.
- ➔ If in 2-thread scenarios, number them as $\mathbf{T_0}$ and $\mathbf{T_1}$
  - ◆ so if $\mathbf{T_i}$ is one of the thread, then $\mathbf{T_{1-i}}$ is the other

now try different solutions to the
Critical Section Problem
and see if they satisfy our requirements

2-Thread Solution #1

➔ Use a shared variable ***turn*** to indicate whose turn it is

- if ***turn*** = 0, T0 is allowed to enter CS, if ***turn*** = 1, T1 to enter CS

```
my_work(id_t id):          # id can be 0 or 1
   …                       # remainder section
   while (turn != id):     # entry section, wait if not my turn
       pass
   do_work()               # critical section
   turn = 1 - id;          # exit section, the other's turn
   …                       # remainder section
```

➔ **Mutual exclusion**: only one thread in CS at a time
- **Satisfied**

➔ **Progress**: no thread waits indefinitely
- **NOT satisfied**, could happen that turn=0 (initial value), T1 entering, T0 still in remainder, T1 waits indefinitely.

2-Thread Solution #2

➔ Replace ***turn*** with a **shared flag** for **each** process

- **{ T0 in CS, T1 in CS}**
- Initial value: { false, false }

```
my_work(id_t id):          # id can be 0 or 1
   …                       # remainder section
   while (flag[1-id]):     # entry section, wait if the other is in CS
       pass
   flag[id] = true;        # now I am in CS
   do_work()               # critical section
   flag[id] = false;       # exit section, now I'm out of CS
   …                       # remainder section
```

➔ **Mutual exclusion**: only one thread in CS at a time

- **NOT satisfied**
- T0 passes while loop, but before T0 sets its flag to true, T1 also passes while loop, and set it flag to true, so we get **{ true, true }**

2-Thread Solution #3

➔ Basically, combine Solution #1 and #2

- 2 threads share ***turn*** and ***flag***

```
my_work(id_t id):          # id can be 0 or 1
   …                       # remainder section
   flag[id] = true         # I wanna go into CS
   turn = id               # call it my turn
   while (turn == id && flag[1-id]):  # entry section
       pass
   do_work()               # critical section
   flag[id] = false;       # exit section, now I'm out of CS
   …                       # remainder section
```

Wait only if **both** threads wanted to go into CS (set flag). and both called it their turn.
turn == id means I was the **second** one who set ***turn*** (so the final value of ***turn*** was set my me)
i.e., I lost the race when calling the turn, so I have to wait.

➔ This algorithm is called **Peterson's Algorithm** (1981)

➔ Convince yourself this really works ...

Multiple-Thread Solutions

➔ Peterson's Algorithm can be extended to N threads

➔ Another approach is Lamport's **Bakery Algorithm**

- Upon entering each customer (thread) get a **number**
- The customer with the lowest number is served next
- No guarantee that 2 threads do not get the same number
  - In case of a tie, thread with the lowest id is served first
  - Thread id's are unique and totally ordered

Now you get a bit how multithreaded programming feels like

Higher-level Abstractions for CS problems

➔ **Semaphores**

- Basic, easy to understand, hard to program with

➔ **Locks**

- Very primitive, minimal semantics

➔ **Condition Variables / Monitors**

- High-level, ideally has language support (Java)

Semaphores

Invented by Edsger W. Dijkstra (1962)

We have to fight chaos, and the most effective way of doing that is the prevention of its emergence.

-- Edsger W. Dijkstra

What is a semaphore?

➔ A semaphore is an object with an **integer value** that we can manipulate with two routines:

- **`sem_wait()`**: **decrement** the value of semaphore and **wait** if its value is negative
- **`sem_post()`**: **increment** the value of semaphore and **wake up** a waiting thread, if any.
- it keeps a queue of **waiting** threads

➔ Semaphores provide **synchronization**

Two types of semaphore

➔ Binary Semaphore

➔ Counting Semaphore

depending on what the initial value is.

Binary Semaphore (Locks)

Functions like a lock

- ➔ **sem_wait**: decrement and wait until value is non-negative
- ➔ **sem_post**: increment and wake up a waiting thread

```
sem_t m;
sem_init(&m, 0, X); //init to value X

sem_wait(&m); //lock
do_work();    //critical section
sem_post(&m); //unlock
```

What value should **X** be?

X = 1

value = 1; T0 sem_wait, value = 0, enter CS;
T1 sem_wait, value = -1, wait until value = 0;
T0 finishes work, sem_post, value = 0, wake up T0.

Counting Semaphore

- ➔ A semaphore with initial value N > 1
- ➔ Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
- ➔ Multiple threads can pass the semaphore
- ➔ Max number of threads is determined by semaphore's initial value N
  - ◆ N = 1: mutex / lock (only one thread can pass the semaphore at a time)

Hardware support

➔ Semaphore would only work if **`sem_wait()`** and **`sem_post()`** are **atomic**.

- ◆ No two threads can do **`sem_wait()`** or **`sem_post()`** at the same time; must be one after another.

➔ Hardware support

- ◆ use lower-level primitive (e.g., locks, conditional variables, which we will learn)
- ◆ Uniprocessor: disable interrupts
- ◆ Multiprocessor: need special atomic instructions

When dealing with concurrency, we always need to know clearly what the hardware has to offer, then build upon it.

An atomic instruction: `test-and-set`

➔ The semantics of **`test-and-set(v)`**

- record the current value of v
- set v to some non-zero value
- return the old value
- All above done **atomically**, even with multiprocessors

`test-and-set(bool v)` returns **false** if the value of v changed from **false to true**.
Otherwise, just return **true,** v value **unchanged**.

Implement a **lock** using test-and-set

Implement a lock

```
bool locked;
void acquire(bool *locked) {
    while (test_and_set(locked));
}
void release(bool *locked) {
    *locked = false;
}
```

Block when the value of locked stays true, pass when value change from false to true (unlocked to locked, i.e., lock acquired).

This is a **spinlock**

locked = false, i.e., unlock

Spinlock

- ➔ **Busy waiting**: thread continually executes while loop in acquire(), consumes CPU cycle. Bad if there is only one CPU.
- ➔ Starvation is possible, if scheduler makes bad choices
- ➔ Works reasonably well for scenarios where the critical section is short, and/or there are multiple processors.
  - ◆ For example, in Assignment 1

Sleep lock

➔ Instead of spinning, put thread into “blocked” state while waiting to acquire lock, and keep a queue of waiting threads.

- Little spinning needed
- No starvation
- Need a system call

use a lock

Using locks for the bank account problem

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);
```

Summary

- ➔ Thread, a new way of virtualizing CPU
- ➔ Threads make things more efficient, but also make things more complicated -- concurrency issues
- ➔ Synchronization
  - ◆ Critical Section Problem, solutions
  - ◆ Semaphores
  - ◆ Locks

Next

➔ More on concurrency

➔ Scheduling

This week’s tutorial

➔ More practice on semaphores and locks