CPS 310
Threads and Concurrency

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase cps310
Threads

• A thread is a **stream of control**….  
  – Executes a sequence of instructions.  
  – Thread identity is defined by CPU register context (PC, SP, …, page table base registers, …)  
  – Generally: a thread’s **context** is its register values and referenced memory state (stacks, page tables).

• Multiple threads can execute independently:  
  – They can run in parallel on multiple cores...  
    • **physical concurrency**  
    – …or arbitrarily interleaved on some single core.  
      • **logical concurrency**

• A thread is also an OS abstraction to spawn and manage a stream of control.

I draw my threads like this.
Some people draw threads as squiggly lines.
Portrait of a thread

In an implementation, each thread is represented by a data struct. We call it a “thread object” or “Thread Control Block”. It stores information about the thread, and may be linked into other system data structures.

Each thread also has a runtime stack for its own use. As a running thread calls procedures in the code, frames are pushed on its stack.
Processes and their threads

Each process has a virtual address space (VAS): a private name space for the virtual memory it uses.

The VAS is both a “sandbox” and a “lockbox”: it limits what the process can see/do, and protects its data from others.

Each process has a main thread bound to the VAS, with a stack.

If we say a process does something, we really mean its thread does it.

The kernel can suspend/restart a thread wherever and whenever it wants.

On real systems, a process can have multiple threads.

We presume that they can all make system calls and block independently.
Two threads sharing a CPU/core

concept

reality

context switch
Thread Abstraction

• Infinite number of processors
• Threads execute with variable speed
  – Programs must be designed to work with any schedule
Possible Executions

Thread 1  Thread 1
Thread 2  Thread 2
Thread 3  Thread 3

a) One execution  b) Another execution

c) Another execution

These executions are “schedules” chosen by the system.
Thread APIs (rough/pseudocode)

- Every thread system has an API for applications to use to manage their threads. Examples: Pthreads, Java threads, C#, Taos…
  - Create and start (“spawn” or “generate”) a thread.
  - Wait for a thread to complete (exit), and obtain a return value.
  - Break into the execution of a thread (optional).
  - Save and access data specific to each thread.
- References to thread objects may be passed around in the API.

Thread operations (parent)
a rough sketch:
```plaintext
\[
\begin{align*}
  t &= \text{create}(); \\
  t.\text{start}(\text{proc, argv}); \\
  t.\text{alert}(); \quad \text{(optional)} \\
  \text{result} &= t.\text{join}(); \quad \text{(wait)}
\end{align*}
\]
```

Self operations (child)
a rough sketch:
```plaintext
\[
\begin{align*}
  \text{exit}(\text{result}); \\
  \text{yield}(); \\
  t &= \text{self}(); \\
  \text{setdata}(\text{ptr}); \\
  \text{ptr} &= \text{selfdata}(); \\
  \text{alertwait}(); \quad \text{(optional)}
\end{align*}
\]
```

Details vary.
Threads lab (due 2/25)

#define STACK_SIZE 262144 /* size of each thread's stack */

typedef void (*thread_startfunc_t) (void *);

extern int thread_libinit(thread_startfunc_t func, void *arg);
extern int thread_create(thread_startfunc_t func, void *arg);
extern int thread_yield(void);
Shared vs. Per-Thread State

**Shared State**
- Heap
- Global Variables
- Code

**Per-Thread State**
- Thread Control Block (TCB)
  - Stack Information
  - Saved Registers
  - Thread Metadata

**Per-Thread State**
- Stack
- Stack
Thread context switch

Running code can suspend the current thread just by saving its register values in memory. Load them back to resume it at any time.
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>y = y + x;</td>
<td>y = y + x;</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td></td>
<td>thread is suspended</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y = y + x</td>
<td>y = y + x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>z = x + 5y</td>
<td>z = x + 5y</td>
</tr>
</tbody>
</table>
Example: Unix fork

The Unix `fork()` system call creates/launches a new thread, in its own fresh virtual address space: it creates a new process. (Thread + VAS == Process.)

Strangely, the new ("child") process is an exact clone of the calling ("parent") process.

Oh Ghost of Walt, please don’t sue me.
int pid = fork();
Create a new process that is a clone of its parent. Return child process ID (pid) to parent, return 0 to child.

exit(status);
Exit with status, destroying the process. Status is returned to the parent. 
Note: this is not the only way for a process to exit!
int pid;
int status = 0;
if (pid = fork()) {
    /* parent */
    ..... 
} else {
    /* child */
    ..... 
    exit(status);
}
int pid;
int status = 0;

if (pid = fork()) {
    /* parent */
    ....
    pid = wait(&status);
} else {
    /* child */
    ....
    exit(status);
}
5.4 Process Synchronization
Another process control system call

\[ \text{processid} = \text{wait}( ) \]
causes its caller to suspend execution until one of its children has completed execution. Then \textit{wait} returns the \textit{processid} of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. \textit{Wait} may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination
Lastly,

\[ \text{exit (status)} \]
Terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the \textit{wait} primitive, the indicated \textit{status} is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes

Note: in modern Unix systems the \textit{wait} syscall has many variants and options.
A simple program: parallel

```
... int main(...arg N...) {
    for 1 to N
dofork();
    for 1 to N
    wait(...);
}

void child() {
    BUSYWORK {x = v;}
    exit(0);
}
...
```

Parallel creates \( N \) child processes and waits for them all to complete.

Each child performs a computation that takes, oh, 10-15 seconds, storing values repeatedly to a global variable, then it exits.

How does \( N \) affect completion time?

```
chase$ cc –o parallel parallel.c
chase$ ./parallel
???
chase$
```
A simple program: parallel

Three different machines

Completion time (ms)

N (# of children)
Parallel: some questions

- Which machine is fastest?
- How does the **total work** grow as a function of $N$?
- Does **completion time** scale with the **total work**? Why?
- Why are the lines flatter for low values of $N$?
- How many cores do these machines have?
- Why is the timing roughly linear, even for “odd” $N$?
- Why do the lines have different slopes?
- Why would the completion time ever drop with higher $N$?
- Why is one of the lines smoother than the other two?
- Can we filter out the noise?
Thread states and transitions

If a thread is in the **ready** state thread, then the system may choose to run it “at any time”. The kernel can switch threads whenever it gains control on a core, e.g., by a timer interrupt. If the current thread takes a fault or system call trap, and blocks or exits, then the scheduler switches to another thread. But it could also preempt a **running** thread. From the point of view of the program, dispatch and preemption are **nondeterministic**: we can’t know the **schedule** in advance.

These **preempt** and **dispatch** transitions are controlled by the kernel scheduler.

**Sleep** and **wakeup** transitions are initiated by calls to internal sleep/wakeup APIs by a running thread.
Thread Lifecycle

Init → Ready
  Thread Creation
  e.g., sthread_create()

Ready → Running
  Scheduler Resumes Thread
  e.g., sthread_yield()

Running → Waiting
  Thread Waits for Event
  e.g., sthread_join()

Waiting → Ready
  Event Occurs
  e.g., other thread calls sthread_join()

Running → Finished
  Thread Yields/Scheduler Suspends Thread
  e.g., sthread_exit()
What cores do

- scheduler
  - getNextToRun()
  - nothing?

- ready queue (runqueue)
  - got thread
  - put thread
  - timer quantum expired?

- idle loop
  - switch in
  - run thread
  - switch out

- pause
  - sleep? exit?
  - got thread

- put thread

- idle

What causes a context switch?

There are three possible causes:

1. Preempt (yield). The thread has had full use of the core for long enough. It has more to do, but it’s time to let some other thread “drive the core”.
   - E.g., timer interrupt, quantum expired → OS forces yield
   - Thread enters Ready state, goes into pool of runnable threads.

2. Exit. Thread is finished: “park the core” and die.

3. Block/sleep/wait. The thread cannot make forward progress until some specific occurrence takes place.
   - Thread enters Blocked state, and just lies there until the event occurs. (Think “stop sign” or “red light”.)
## Two threads call yield

<table>
<thead>
<tr>
<th>Thread 1’s instructions</th>
<th>Thread 2’s instructions</th>
<th>Processor’s instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
<tr>
<td>call thread_yield</td>
<td>call thread_yield</td>
<td>call thread_yield</td>
</tr>
<tr>
<td>save state to stack</td>
<td>save state to stack</td>
<td>save state to stack</td>
</tr>
<tr>
<td>save state to TCB</td>
<td>save state to TCB</td>
<td>save state to TCB</td>
</tr>
<tr>
<td>choose another thread</td>
<td>choose another thread</td>
<td>choose another thread</td>
</tr>
<tr>
<td>load other thread state</td>
<td>load other thread state</td>
<td>load other thread state</td>
</tr>
<tr>
<td>return thread_yield</td>
<td>return thread_yield</td>
<td>return thread_yield</td>
</tr>
</tbody>
</table>

...
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <loops>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
Two executions of this code, so: $x$ is incremented by two.
Interleaving matters

<table>
<thead>
<tr>
<th>load</th>
<th>x, R2</th>
<th>; load global variable x</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R2, 1, R2</td>
<td>; increment: x = x + 1</td>
</tr>
<tr>
<td>store</td>
<td>R2, x</td>
<td>; store global variable x</td>
</tr>
</tbody>
</table>

In this schedule, \( x \) is incremented only once: last writer wins. The program breaks under this schedule. This bug is a race.
Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the “random walk” through the space of possible program states.

RTG is useful to depict all possible executions of multiple threads. I draw them for only two threads because slides are two-dimensional.

RTG for N threads is N-dimensional.

Thread \( i \) advances along axis \( i \).

Each point represents one state in the set of all possible system states.

Cross-product of the possible states of all threads in the system.
Resource Trajectory Graphs

This RTG depicts a schedule within the space of possible schedules for a simple program of two threads sharing one core.

Blue advances along the y-axis.
Purple advances along the x-axis.

Every schedule starts here.

EXIT

context switch

Every schedule ends here.

The diagonal is an idealized parallel execution (two cores).

The scheduler chooses the path (schedule, event order, or interleaving).

From the point of view of the program, the chosen path is nondeterministic.
A race

This is a valid schedule.

But the schedule interleaves the executions of “x = x+ 1” in the two threads.

The variable x is shared (like the counter in the pthreads example).

This schedule can corrupt the value of the shared variable x, causing the program to execute incorrectly.

This is an example of a race: the behavior of the program depends on the schedule, and some schedules yield incorrect results.
The need for mutual exclusion

The program may fail if the schedule enters the grey box (i.e., if two threads execute the critical section concurrently).

The two threads must not both operate on the shared global $x$ “at the same time”.

\[ x = x + 1 \]
Using a lock/mutex

The program may fail if it enters the grey box.

A lock (mutex) prevents the schedule from ever entering the grey box, ever: both threads would have to hold the same lock at the same time, and locks don’t allow that.
“Lock it down”

Use a lock (mutex) to synchronize access to a data structure that is shared by multiple threads.

A thread **acquires** (locks) the designated mutex before operating on a given piece of shared data.

The thread **holds** the mutex. At most one thread can hold a given mutex at a time (**mutual exclusion**).

Thread **releases** (unlocks) the mutex when done. If another thread is waiting to acquire, then it wakes.

The mutex bars entry to the grey box: the threads cannot both hold the mutex.
```c
pthread_mutex_t m;
volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&m);
        counter++;
        Pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

“Lock it down.”
Concurrent control

- The scheduler (and the machine) select the execution order of threads
- Each thread executes a sequence of instructions, but their sequences may be arbitrarily interleaved.
  - E.g., from the point of view of loads/stores on memory.
- Each possible execution order is a schedule.
- A thread-safe program must exclude schedules that lead to incorrect behavior.
- It is called synchronization or concurrency control.
This is not a game

But we can think of it as a game.

1. You write your program.
2. The game begins when you submit your program to your adversary: the scheduler.
3. The scheduler chooses all the moves while you watch.
4. Your program may constrain the set of legal moves.
5. The scheduler searches for a legal schedule that breaks your program.
6. If it succeeds, then you lose (your program has a race).
7. You win by not losing.

\[ x = x + 1 \]
A Lock or Mutex

Locks are the basic tools to enforce mutual exclusion in conflicting critical sections.

- A lock is a special data item in memory.
- API methods: Acquire and Release.
- Also called Lock() and Unlock().
- Threads pair calls to Acquire and Release.
- Acquire upon entering a critical section.
- Release upon leaving a critical section.
- Between Acquire/Release, the thread holds the lock.
- Acquire does not pass until any previous holder releases.
- Waiting locks can spin (a spinlock) or block (a mutex).
- Also called a monitor: threads enter (acquire) and exit (release).
Definition of a lock (mutex)

- Acquire + release ops on $L$ are strictly paired.
  - After acquire completes, the caller holds (owns) the lock $L$ until the matching release.

- Acquire + release pairs on each $L$ are ordered.
  - Total order: each lock $L$ has at most one holder at any given time.
  - That property is mutual exclusion; $L$ is a mutex.
void PingPong() {
    while(not done) {
        ...
        if (blue)
            switch to purple;
        if (purple)
            switch to blue;
    }
}
void PingPong() {
    while(not done) {
        Mx->Acquire();
        ...
        Mx->Release();
    }
}
Mutexes don’t work for ping-pong
Waiting for conditions

- Ping-pong motivates more general synchronization primitives.
- In particular, we need some way for a thread to **sleep until some other thread wakes it up**.
- This enables explicit signaling over any kind of condition, e.g., changes in the program state or state of a shared resource.
- Ideally, the threads don’t have to know about each other explicitly. They should be able to coordinate around shared objects.
Waiting for conditions

- In particular, a thread might wait for some logical condition to become true. A **condition** is a predicate over state: it is any statement about the “world” that is either true or false.
  - Wait until a new event arrives; wait until event queue is not empty.
  - Wait for certain amount of time to elapse, then wake up at time t.
  - Wait for a network packet to arrive or an I/O operation to complete.
  - Wait for a shared resource (e.g., buffer space) to free up.
  - Wait for some other thread to finish some operation (e.g., initializing).
Condition variables

• A condition variable (CV) is an object with an API.
  – **wait**: block until some condition becomes true
    • Not to be confused with Unix `wait*` system call
  – **signal** (also called **notify**): signal that the condition is true
    • Wake up one waiter.

• Every CV is bound to exactly one mutex, which is necessary for safe use of the CV.
  – The mutex protects shared state associated with the condition

• A mutex may have any number of CVs bound to it.

• CVs also define a **broadcast (notifyAll)** primitive.
  – Signal all waiters.
Lab #1 API

int thread_lock(unsigned int lock)
int thread_unlock(unsigned int lock)
int thread_wait(unsigned int lock, unsigned int cond)
int thread_signal(unsigned int lock, unsigned int cond)
int thread_broadcast(unsigned int lock, unsigned int cond)

A lock is identified by an unsigned integer (0 - 0xffffffff). Each lock has a set of condition variables associated with it (numbered 0 - 0xffffffff), so a condition variable is identified uniquely by the tuple (lock number, cond number). Programs can use arbitrary numbers for locks and condition variables (i.e., they need not be numbered from 0 - n).
void PingPong() {
    mx->Acquire();
    while(not done) {
        ...
        cv->Signal();
        cv->Wait();
    }
    mx->Release();
}
Waiting for conditions

• You can use condition variables (CVs) to represent any condition in your program (Q empty, buffer full, op complete, resource ready…).
  – We can use CVs to implement any kind of synchronization object.

• Associate the condition variable with the mutex that protects the state relating to that condition!
  – **Note:** CVs are not variables. But you can associate them with whatever data you want, i.e., the state protected by its mutex.

• A caller of CV wait must hold its mutex (be “in the monitor”).
  – This is crucial because it means that a waiter can wait on a logical condition and know that it won’t change until the waiter is safely asleep.
  – Otherwise, due to nondeterminism, another thread could change the condition and signal before the waiter is asleep! The waiter would sleep forever: the missed wakeup or wake-up waiter problem.

• **Wait** atomically releases the mutex to sleep, and reacquires it before returning.
A note on the terms “wait” and “signal”

Don’t confuse CV wait() with the Unix wait* system call. Also, don’t confuse CV signal() with Unix signals. It is easy to confuse concepts when they have the same name.

The similarity of names is not an accident. Both kinds of “wait” put the calling thread to sleep until some other thread notifies it of a particular condition or event. And both kinds of “signal” are ways to notify a thread that something of interest has happened, so that it can respond.

In the case of wait* syscall, the condition or event to wait for is specifically an exit (or other change of status) of a child process. CV wait is more general: it can be used to implement wait/signal behavior for any kind of condition, for any kind of synchronization object, including but not limited to the internal implementation of the wait* syscalls.

Note that a wait* syscall does not always put the calling thread to sleep. For example, waitpid() syscall with the WNOHANG option does not put the caller to sleep. There are other cases in which a wait* syscall does not wait. (Refer to the manpage or slides on that topic.)

In contrast to the wait* syscall, CV wait() always puts the caller to sleep. The blocked thread will/must sleep until it is awakened by some subsequent signal/notify or broadcast/notifyAll on that specific CV.

As for Unix signals, a Unix signal is posted to a process and is delivered by redirecting control of some thread in the process into a registered signal handler, whatever that thread was doing before. In contrast, CV signal() can only affect a thread that is sleeping in wait() on a specific CV. It could wake up any thread sleeping on that CV. Once it wakes up, the signaled thread just continues executing according to its program.
Java uses mutexes and CVs

Every Java object has a mutex ("monitor") and condition variable ("CV") built in. You don’t have to use it, but it’s there.

```java
public class Object {
    void notify();    // signal
    void notifyAll(); // broadcast
    void wait();
    void wait(long timeout);
}

public class PingPong extends Object {
    public synchronized void PingPong() {
        while(true) {
            notify();
            wait();
        }
    }
}
```

A thread must own an object’s monitor ("synchronized") to call wait/notify, else the method raises an IllegalMonitorStateException.

Wait(*) waits until the timeout elapses or another thread notifies.
Mutual exclusion in Java

- Mutexes are built in to every Java object.
  - no separate classes

- Every Java object is/has a monitor.
  - At most one thread may “own” a monitor at any given time.

- A thread becomes owner of an object’s monitor by
  - executing an object method declared as synchronized
  - executing a block that is synchronized on the object

```java
public synchronized void increment()
{
    x = x + 1;
}
```

```java
public void increment()
{
    synchronized(this)
    {
        x = x + 1;
    }
}
```
Roots: monitors

A **monitor** is a module in which execution is serialized. A module is a set of procedures with some private state.

At most one thread runs in the monitor at a time. Other threads wait until the monitor is free.

Java **synchronized** just allows finer control over the entry/exit points. Also, each Java object is its own “module”: objects of a Java class share methods of the class but have private state and a private monitor.

[Brinch Hansen 1973]
[C.A.R. Hoare 1974]
Monitors and mutexes are “equivalent”

• Entry to a monitor (e.g., a Java `synchronized` block) is equivalent to **Acquire** of an associated mutex.
  – Lock on entry

• Exit of a monitor is equivalent to **Release**.
  – Unlock on exit (or at least “return the key”…)

• Note: exit/release is implicit and automatic if the thread exits synchronized code by a Java exception.
  – Much less error-prone than explicit release
  – Can’t “forget” to unlock / “return the key”.
  – **Language-integrated support is a plus for Java.**
Monitors and mutexes are “equivalent”

- **Well:** mutexes are more flexible because we can choose which mutex controls a given piece of state.
  - E.g., in Java we can use one object’s monitor to control access to state in some other object.
  - Perfectly legal! So “monitors” in Java are more properly thought of as mutexes.

- **Caution: this flexibility is also more dangerous!**
  - It violates modularity: can code “know” what locks are held by the thread that is executing it?
  - Nested locks may cause **deadlock** (later).

- **Keep your locking scheme simple and local!**
  - Java ensures that each Acquire/Release pair (**synchronized** block) is contained within a method, which is good practice.
Using monitors/mutexes

Each monitor/mutex protects specific data structures (state) in the program. Threads hold the mutex when operating on that state.

The state is consistent iff certain well-defined invariant conditions are true. A condition is a logical predicate over the state.

Example invariant condition
E.g.: suppose the state has a doubly linked list. Then for any element e either e.next is null or e.next.prev == e.

Threads hold the mutex when transitioning the structures from one consistent state to another, and restore the invariants before releasing the mutex.
Monitor wait/signal

We need a way for a thread to wait for some condition to become true, e.g., until another thread runs and/or changes the state somehow.

At most one thread runs in the monitor at a time.

A thread may wait (sleep) in the monitor, exiting the monitor.

A thread may signal in the monitor.

Signal means: wake one waiting thread, if there is one, else do nothing.

The awakened thread returns from its wait and reenters the monitor.
Condition variables are equivalent

- A **condition variable (CV)** is an object with an API.
- A CV implements the behavior of monitor conditions.
  - interface to a CV: **wait** and **signal** (also called **notify**)
- Every CV is bound to exactly one mutex, which is necessary for safe use of the CV.
  - “holding the mutex” $\leftrightarrow$ “in the monitor”
- A mutex may have any number of CVs bound to it.
  - (But not in Java: only one CV per mutex in Java.)
- CVs also define a **broadcast (notifyAll)** primitive.
  - Signal all waiters.
Monitor wait/signal

Design question: when a waiting thread is awakened by signal, must it start running immediately? Back in the monitor, where it called wait?

At most one thread runs in the monitor at a time.

Two choices: yes or no.
If yes, what happens to the thread that called signal within the monitor? Does it just hang there? They can’t both be in the monitor.

If no, can’t other threads get into the monitor first and change the state, causing the condition to become false again?
Design question: when a waiting thread is awakened by `signal`, must it start running immediately? Back in the monitor, where it called `wait`?

Mesa semantics: no. An awakened waiter gets back in line. The `signal` caller keeps the monitor.

So, can’t other threads get into the monitor first and change the state, causing the condition to become false again?

Yes. So the waiter must recheck the condition: “Loop before you leap”.
Handing off a lock

First I go.

release

Handoff

The nth release, followed by the (n+1)th acquire

Then you go.

serialized
(one after the other)
Locking and blocking

If thread T attempts to acquire a lock that is busy (held), T must spin and/or block (sleep) until the lock is free. By sleeping, T frees up the core for some other use. Just spinning is wasteful!

Note: H is the lock holder when T attempts to acquire the lock.
Implementing threads

• Thread_fork(func, args)
  – Allocate thread control block
  – Allocate stack
  – Build stack frame for base of stack (stub)
  – Put func, args on stack
  – Put thread on ready list
  – Will run sometime later (maybe right away!)

• stub(func, args): Pintos switch_entry
  – Call (*func)(args)
  – Call thread_exit()