Use of CVs in Project 1

- Incorrect use of condition variables:

```c
while (cond) {
    cv.signal()
    cv.wait()
}
```

- Thread going to sleep should not be of interest to other threads
Interactions between threads

- Threads must synchronize access to shared data
- **High-level synchronization primitives:**
  - Locks
  - Condition variables
  - Monitors
  - Semaphores

- Threads share the same CPU
- Then what is a non-running thread?
States of a Thread

New → Running
Create thread → Thread completes execution

Running → Terminated
Wait on lock, wait, or down → Another thread calls unlock, signal, or up

Blocked

What if there are more threads than CPUs?
States of a Thread

Why no transition from Ready to Blocked?

- New → Ready: Create thread
- Ready → Blocked: Another thread calls unlock, signal, or up
- Blocked → Running: Wait on lock, wait, or down
- Running → Ready: CPU to spare
- Ready → Terminated: Switch CPU to another thread
- Terminated: Thread completes execution
What to do with thread while it’s not running?
- Must save its private state somewhere

Thread “context” stored in a “thread control block” (TCB) when thread isn’t running

What should be stored in TCB?
Process Address Space

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code
- Data Segment

Thread 1
PC (T1)

Thread 2
PC (T2)

Thread 3
PC (T3)
Thread context

- To save space in TCB
  - Share code among all threads and store only PC
  - Use multiple stacks and copy only SP to TCB

- Keep track of ready threads (e.g., on a queue)

- Any thread can be in one of three states
  - Running on the CPU
  - TCB is in ready queue
  - Blocked: TCB is in waiting queue of synchronization primitive
Project 2 is out

- Implement a thread library
  - Create threads
  - Switch between threads
  - Manage interactions (locks and CVs)
  - Schedule threads on CPUs

- Due Feb 17th
  - Start early!

- Everyone should now be in a group
Two Perspectives to Execution

- **Thread view:**
  - Running $\rightarrow$ Paused $\rightarrow$ Resume

- **CPU view:**
  - Thread 1 $\rightarrow$ Thread 2 $\rightarrow$ Thread 1
Steps in Switching threads

- Current thread returns (yields) control to OS
- OS chooses next thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How does thread return control back to OS?
Returning control to OS

- Three types of internal events:
  - Thread calls wait(), lock(), etc.
  - Thread requests OS to do some work (e.g., I/O)
  - Thread voluntarily gives up CPU with yield()

- Are these enough?

- Also need external events:
  - Interrupts are hardware events that transfer control from CPU to OS’s interrupt handler
Steps in Switching threads

- Current thread returns control to OS
- **OS chooses next thread to run**
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How does the OS choose the next thread to run?
Choosing next thread to run

- 1 ready thread
  - What if thread calls yield?

- >1 ready thread
  - FIFO
  - Priority

- What should CPU do if no ready threads?
  - Modern CPUs suspend their execution and resume on an interrupt
    - `interrupt_enable_suspend()` in Project 2
Steps in Switching threads

- Current thread returns control to OS
- OS chooses new thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How do you save the state of the current thread?
Saving state of current thread

- Save registers, PC, stack pointer
- Tricky to get right!
  - Why won’t the following code work?
    100 save PC
    101 switch to next thread

- Involves tricky assembly-language code
- In Project 2, we’ll use Linux’s `swapcontext()`
Steps in Switching threads

- Current thread returns control to OS
- OS chooses new thread to run
- OS saves state of current thread from CPU to its thread control block
- OS loads context of next thread from its thread control block
- OS runs next thread

How do you load the TCB of the next thread and run?
Loading context of next thread and running it

- How to load registers?
- How to load stack?
- How to resume execution?
- Who is carrying out these steps?
- How does thread that gave up control run again?
Example of thread switching

Thread 1

print "start thread 1"
yield()
print "end thread 1"

Thread 2

print "start thread 2"
yield()
print "end thread 2"
yield()