Semaphores vs. CVs

- Condition variables:
  - User-level condition
  - Developer must control access to variables
- Semaphores:
  - Access to value is thread-safe
  - Only condition is "(value == 0)"
- How to implement custom waiting condition with semaphores?

Producer-consumer with semaphores (monitor style)

```c
Producer
mutex.down()
while (numCokes == MAX) {
    semaphore s = 0
    waitingProducers.push(&s)
    mutex.up()
    s.down()
    mutex.down()
}
add coke to machine
numCokes++
if (!waitingConsumers.empty()) {
    waitingConsumers.front()->up()
    waitingConsumers.pop()
}
mutex.up()
```

```c
Consumer
mutex.down()
while (numCokes == 0) {
    semaphore s = 0
    waitingConsumers.push(&s)
    mutex.up()
    s.down()
    mutex.down()
}
take coke out of machine
numCokes--
if (!waitingProducers.empty()) {
    waitingProducers.front()->up()
    waitingProducers.pop()
}
mutex.up()
```

Implementing condition variables with semaphores

```c
wait(mutex m) {
    // create a new semaphore
    semaphore s = 0;
    m.up();
    // add new semaphore to waiting list
    waiters.insert(&s);
    // go to sleep
    s.down();
    m.down();
}
signal() {
    // nothing to do if no waiters
    if (!waiters.empty()) {
        return;
    }
    // wake up one of the waiters
    semaphore s = waiters.front();
    s.up();
    // remove waiter from queue
    waiters.pop();
}
```

Queue waiters = {}; // list of semaphores inserted by waiting threads

Homework exercise

- Given an implementation of a condition variable, how to implement a semaphore?

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
Project 2 is out

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

- Due October 22nd
  - Start early!

- Everyone should now be in a group

Interactions between threads

- Threads must synchronize access to shared data

  - High-level synchronization primitives:
    - Locks
    - Condition variables
    - Monitors
    - Semaphores

- Threads share the same CPU

States of a Thread

- New
- Running
- Blocked
- Terminated

Create thread
Thread completes execution
Wait on lock, wait, or down
Another thread calls unlock, signal, or up

Why no transition from Ready to Blocked?

- Ready threads
  - 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)

- Stack
- Heap
- Static Data
- Code

- PC (T1)
- PC (T2)
- 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
  - TCB is in waiting queue of synchronization primitive

Two Perspectives to Execution

- Thread view:
  - Running → Paused → Resume
- CPU view:
  - Thread 1 → Thread 2 → Thread 1

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 does thread return control back to OS?

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>Thread 1 output</th>
<th>Thread 2 output</th>
</tr>
</thead>
<tbody>
<tr>
<td>print “start thread 1”</td>
<td>print “start thread 2”</td>
</tr>
<tr>
<td>yield()</td>
<td>yield()</td>
</tr>
<tr>
<td>print “end thread 1”</td>
<td>print “end thread 2”</td>
</tr>
<tr>
<td>start thread 1</td>
<td>start thread 2</td>
</tr>
<tr>
<td>start yield: thread 1</td>
<td>start yield: thread 2</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>print “start thread 2”</td>
<td></td>
</tr>
<tr>
<td>yield()</td>
<td></td>
</tr>
<tr>
<td>print “end thread 2”</td>
<td></td>
</tr>
<tr>
<td>end thread 1</td>
<td>end thread 2</td>
</tr>
<tr>
<td>end thread 2</td>
<td></td>
</tr>
</tbody>
</table>

Thread 1 output

- `start thread 1`
- `start yield: thread 1`
- `end thread 1`

Thread 2 output

- `start thread 2`
- `start yield: thread 2`
- `end thread 2`

Example of thread switching

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>print “start thread 1”</td>
<td>print “start thread 2”</td>
</tr>
<tr>
<td>yield()</td>
<td>yield()</td>
</tr>
<tr>
<td>print “end thread 1”</td>
<td>print “end thread 2”</td>
</tr>
<tr>
<td>start thread 1</td>
<td>start thread 2</td>
</tr>
<tr>
<td>start yield: thread 1</td>
<td>start yield: thread 2</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
</tr>
<tr>
<td>print “start thread 2”</td>
<td></td>
</tr>
<tr>
<td>yield()</td>
<td></td>
</tr>
<tr>
<td>print “end thread 2”</td>
<td></td>
</tr>
<tr>
<td>end thread 1</td>
<td>end thread 2</td>
</tr>
<tr>
<td>end thread 2</td>
<td></td>
</tr>
</tbody>
</table>