EECS 482
Introduction to Operating Systems
Winter 2018

Harsha V. Madhyastha
Monitors vs. Semaphores

- Monitors:
  - Custom user-defined conditions
  - 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?
Implementing condition variables with semaphores

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

wait(mutex m) {
    // create a new semaphore
    semaphore s = 0;

    // add new semaphore to waiting list
    waiters.insert(&s);

    m.up(); // 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();
}
Exercise to try ...

- Given implementations of mutex and condition variable, how to implement a semaphore?
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 → Terminated

Create thread

Thread completes execution

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: Wait on lock, wait, or down
- Blocked -> Ready: Another thread calls unlock, signal, or up
- Ready -> Running: Switch CPU to another thread
- Running -> Ready: CPU to spare
- Running -> Terminated: Wait on lock, wait, or down
- Terminated: Thread completes execution
Ready threads

- What to do with thread while it’s not running?
  - Essentially, a “paused” execution
  - Must save its private state somewhere

- Thread control block (TCB)
  - Per-thread OS data structure for thread info
  - Store thread “context” when not running

- What context should be stored in TCB?
Process Address Space

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

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
  ✦ Also need to store general-purpose registers

● Keep track of ready threads (e.g., queue of TCBs)

● Now, any thread can be
  ✦ Running on the CPU
  ✦ Ready with TCB on ready queue
  ✦ Blocked with TCB in waiting queue of lock, CV, etc.
Project 2 is out

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

- Due February 17\textsuperscript{th}
  - Start right away!

- Everyone should now be in a group
Differences compared to P1

- Much harder!
- 15% of grade
- Test cases will be graded
- Hand grading to check for good coding practices, efficiency, no duplication of code, etc.
Two Perspectives to Execution

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

- **CPU view:**
  - Thread 1 $\rightarrow$ Thread 2 $\rightarrow$ Thread 1
Context switch

1. **Current thread returns control** to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. 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 (e.g., timer, I/O)
  ◆ are hardware events that transfer control from thread to OS interrupt handler
Interrupts

- Hardware events (implemented by CPU)
- Stop current execution (e.g., thread function)
- Start running OS interrupt handler

- OS registers handlers in interrupt vector table
- Example: timer interrupt
  - OS may set timer to go off every 10 ms
  - Guarantees that it will get control back in <= 10 ms
Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread
Choosing next thread to run

- 1 ready thread: just run it
  - What if only thread that exists calls yield?

- >1 ready thread: need to make a decision
  - CPU’s scheduling policy
  - Lots of options: FIFO, priority, round robin, etc.

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

1. Current thread returns control to OS
2. OS chooses new thread to run
3. **OS saves current thread state:** CPU to TCB
4. **OS loads context of next thread:** TCB to CPU
5. 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()`
Context switch

1. Current thread returns control to OS
2. OS chooses new thread to run
3. OS saves current thread state: CPU to TCB
4. OS loads context of next thread: TCB to CPU
5. OS runs next thread
Load context and run

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

Thread 2

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

yield()

print “start yield: thread %d”

switch to next thread (swapcontext)

print “end yield: thread %d”
Creating a new thread

- Create a running thread? Seems challenging

- Instead, create paused thread
  - Key idea: pretend it was running, put on readyq
  - Then just wait for it to be scheduled!

- Implication:
  - Construct TCB as if it were paused at thread start
Recipe for creating a thread

1. Allocate and initialize TCB
   - Set PC to start of thread function
   - Set general-purpose registers to func parameters

2. Allocate and initialize stack
   - What goes on stack?
   - Set TCB stack pointer to stack top
   - `getcontext()` and `makecontext()` in Project 2

3. Add TCB to ready queue

- Note: `fork()` creates a process, discuss later
How to use new thread

- Creating a thread is like an asynchronous procedure call
Synchronizing with child

- What if parent wants to work for a while, then wait for child to finish?
Synchronizing with child

parent()
  create child thread
  print “parent works”
  ...
  print “parent continues”
  ...

child()
  ...
  print “child is done”

When would this work?

Desired output
parent works
child is done
parent continues

OR
child is done
parent works
parent continues
Synchronizing with child

parent()
    create child thread
    print "parent works"
    ...
    \textbf{yield}()
    print "parent continues"
    ...

child()
    ...
    print "child is done"

\textbf{Desired output}
parent works
child is done
parent continues

OR
child is done
parent works
parent continues

Does this work?
Synchronizing with join

parent()
create child thread
print “parent works”
...
childThread.join()
print “parent continues”
...

child()
...
print “child is done”

Desired output
parent works
child is done
parent continues

OR
child is done
parent works
parent continues

How to make do without join?
Synchronizing with monitors

parent()
    childDone = 0
    create child thread
    print “parent works”
    lock()
    while (!childDone)
        wait()
    unlock()
    print “parent continues”

child()
    print “child is done”
    lock()
    childDone = 1
    signal()
    unlock()

Desired output
parent works
child is done
parent continues

OR

child is done
parent works
parent continues
Project 2 update

- You can now do a substantial part of project 2
  - Thread create
  - Context switch
  - Thread join

- Next topic: implementing synchronization
  - Need to protect OS data structures (ready queue)
  - Need to block without (much) busy waiting