Reminders
- Homework 3 due next Monday
- Synchronization
- Project 2 parts 1, 2, 3 due next Wednesday
- Threads, synchronization
- Office hour at 3:30, not 4:30 today

Today:
- Project 2 continued (parts 2, 3)
- Synchronization

Project 2 Part 1 Questions
- Any questions about part 1?
- Some common issues:
  - sthread_create doesn’t immediately run the new thread
  - sthread_exit can ignore its ret argument
  - How do you clean up an exiting thread?
    - Must switch to another thread
    - Clean up in all places after sthread_switch()
    - Have a special GC thread

Synchronization Solutions

High-level
- Monitors
  - Java synchronized method

OS-level support
- Special variables – mutexes, semaphores, condition vars
- Message passing primitives

Low-level support
- Disable/enable interrupts
- Atomic instructions

Disabling/Enabling Interrupts

Thread A:
- disable_interupts()
- critical_section()
- enable_interrupts()

Thread B:
- disable_interupts()
- critical_section()
- enable_interrupts()

- Prevents context-switches during execution of CS
- Sometimes necessary
  - E.g. to prevent further interrupts during interrupt handling
  - Many problems

Hardware support

- Atomic instructions:
  - Test and set
  - Swap
  - Compare-exchange (x86)
- Use these to implement higher-level primitives
  - E.g. test-and-set on x86 (given to you for part 4) is written using compare-exchange.
    - compare_exchange(lock_t *,x,y,int z):
      - if(x == y)
        - *x = z;
        - return y;
      - else return *x;
      - test_and_set(lock_t *l) {
        ...
      }

Looking ahead: preemption

- You can start inserting synchronization code
  - disable/enable interrupts
  - atomic_test_and_set
  - Where would you use these?
  - Example:
    ...
    nextTCB = sthread_dequeue(readyQ);
    switch to nextTCB;
    ...

Semaphore review

- Semaphore = a special variable
  - Manipulated atomically via two operations:  
    - P (wait)
    - V (signal)
- Has a counter = number of available resources
  - P decrements it
  - V increments it
- Has a queue of waiting threads
  - If execute wait() and semaphore is free, continue
  - If not, block on that waiting queue
- signal() unblocks a thread if it’s waiting

Synchronization in Project 2

- Part 2: write two synchronization primitives
- Implement mutex (binary semaphore)
  - How is it different from spinlock?
  - Need to keep track of lock state
  - Need to keep waiting threads on a queue
  - In lock(), may need to block current thread
    - Don’t put on ready queue
    - Do run some other thread
  - For unlock(), need to take a thread off the waiting queue if available

Condition Variable

- A "place" to let threads wait for a certain event to occur while holding a lock (often a monitor lock).
- It has:
  - Wait queue
  - Three functions: wait, signal, and broadcast
    - wait – sleep until the event happens
    - signal – event/condition has occurred. If wait queue nonempty, wake up one thread, otherwise do nothing
    - Do not run the sleep on thread right away
    - FIFO determines who wakes up
  - broadcast – just like signal, except wake up all threads
  - In part 2, you implement all of these

Condition Variables 2

- How are CVs different from semaphores?
- More about:
  - cond_wait(pthread_cond_t cond, pthread_mutex_t lock)
    - Called while holding lock!
    - Should do the following atomically:
      - Release the lock (to allow someone else to get in)
      - Add current thread to the waiters for cond
      - Block thread until awakened
    - After woken up, a thread should reacquire its lock before continuing
  - Good explanation: man pthread_cond_wait
    - We follow the same spec for wait, signal, broadcast

Monitors: preview

- One thread inside at a time
- Lock + a bunch of condition variables (CVs)
- CVs used to allow other threads to access the monitor while one thread waits for an event to occur

Part 3 problem

- N cooks produce burgers & place on stack
- M students grab burgers and eat them
- Provide correct synchronization
  - Check with your threads and pthreads!
- Print out what happens!
- sample output (rough draft):
  - cook 2 produces burger #5
  - cook 2 produces burger #6
  - cook 3 produces burger #7
  - student 1 eats burger #7
  - student 2 eats burger #6
  - cook 1 produces burger #8
  - student 1 eats burger #8
  - student 1 eats burger #5
  - ...
Synchronization is necessary when multiple threads access the same shared data
- Can't use some primitives in interrupt handlers
  - Why? Which ones?
- Don't forget to release lock, semaphore, etc
  - Check all paths
- Synchronization bugs can be very difficult to find
  - Read your code

Late-Night Pizza
- Synchronize student threads and pizza delivery thread
- Avoid deadlock
- When out of pizza, order it exactly once
- No piece of pizza may be consumed by more than one student

Sample synchronization problem
**Late-Night Pizza**
- A group of students study for cse451 exam
- Can only study while eating pizza
- Each student thread executes the following:
  - while (1) {
    pick up a piece of pizza;
    study while eating the pizza;
  }
- If a student finds pizza is gone, the student goes to sleep until another pizza arrives
- First student to discover pizza is gone orders a new one.
- Each pizza has S slices.

Semaphore solution
- Shared data:
  - semaphore pizza; (counting semaphore, init to 0, represent number of available pizza resources)
  - semaphore deliver; (init to 1)
  - semaphore mutex; (init to 1) // guard updating of num_slices

**Condition Variable Solution**

Student()

```c
while(diligent) {
    P(pizza);
    P(mutex);
    num_slices--;
    if (num_slices==0)
        // took last slice
    V(deliver);
    V(mutex);
    study();
}
```

deliveryGuy()

```c
while(employed) {
    P(deliver);
    make_pizza();
    P(mutex);
    num_slices=S;
    V(mutex);
    for (int i=0; i<S; i++) {
        V(pizza);
    }
}
```