CSE 451: Operating Systems

Section 5:
Synchronization

Project 2a is due on Wednesday

- Review the sthread code that we give you
  - If a function is there, then there is probably a good reason why
- Remember to write more test cases
- test-burgers: your shell may limit the number of threads you can run
  - ulimit –u
- Has anybody found bugs in the 64-bit code yet?

Synchronization support

- Processor level:
  - Disable / enable interrupts
  - Atomic instructions
- Operating system / library level:
  - Special variables: mutexes, semaphores, condition variables
- Programming language level:
  - Monitors, Java synchronized methods
Disabling / enabling interrupts

- Prevents context-switches during execution of critical sections
- Sometimes necessary
- Many pitfalls

Processor support

- Atomic instructions:
  - test-and-set
  - compare-exchange (x86)
- Use these to implement higher-level primitives
  - E.g. test-and-set on x86 (given to you for part 5) is written using compare-exchange
  - You’ll use test-and-set when implementing mutexes and condition variables (part 5)

Processor support

- Test-and-set using compare-exchange:

```c
compare_exchange(lock_t *x, int y, int z):
if(*x == y)
  *x = z;
  return y;
else
  return *x;
}
test_and_set(lock_t *lock) {
  ???
}
```

Processor support

- Test-and-set using compare-exchange:

```c
compare_exchange(lock_t *x, int y, int z):
if(*x == y)
  *x = z;
  return y;
else
  return *x;
}
test_and_set(lock_t *lock) {
  compare_exchange(lock, 0, 1);
}
```
**Preemption**

- You will need to use test-and-set and interrupt disabling in part 5 of project 2 (preemption)
- You can start thinking about this while completing the code for the first 2 parts: where are the critical sections where I shouldn’t be interrupted?

**Semaphores**

- Semaphore = a special variable
- Manipulated atomically via two operations
  - P (wait): tries to decrement semaphore
  - V (signal): increments semaphore
- Has a *queue* of waiting threads
  - If execute wait() and semaphore is available, continue
  - If not, block on the waiting queue
  - signal() unblocks a thread on queue

**Mutexes**

- What is a mutex?
  - A binary semaphore (semaphore initialized with value 1)
- Why use a mutex rather than a low-level lock?
  - Threads wait on a mutex by *blocking*, rather than *spinning*.

**How not to implement mutexes**

- Definitely not like this:
  ```c
  void sthread_user_mutex_lock(
      sthread_mutex_t lock) {
      while (lock->held) { ; }
      lock->held = true;
  }
  ```
- And also not like this:
  ```c
  void sthread_user_mutex_lock(
      sthread_mutex_t lock) {
      while (lock->held) {
          yield();
      }
      lock->held = true;
  }
  ```
Condition variables

- Let threads block until a certain event or condition occurs (rather than polling)
- Associated with some logical condition in a program:
  ```c
  sthread_mutex_lock(lock);
  while (x <= y) {
    sthread_cond_wait(cond, lock);
  }
  sthread_mutex_unlock(lock);
  ```

Condition variables

- Operations:
  - wait: sleep on wait queue until event happens
  - signal: wake up one thread on wait queue
  - broadcast: wake up all threads on wait queue
- signal or broadcast is called explicitly by the application when the event / condition occurs

Example synchronization problem

- Late-Night Pizza
  - A group of students study for CSE 451 exam
  - Can only study while eating 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
Late-night pizza

* Each student thread executes the following:
  while (must_study) {
    pick up a piece of pizza;
    study while eating the pizza;
  }

Late-night pizza

* Need to 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

Semaphore / mutex solution

* Shared data:
  semaphore_t pizza;  //Number of
  //available pizza
  //resources;
  //init to 0
  semaphore_t deliver; //init to 1
  int num_slices = 0;
  mutex_t mutex;  //protects accesses
  //to num_slices

Semaphore / mutex solution

student_thread {
  while (must_study) {
    wait(pizza);
    acquire(mutex);
    num_slices--;
    if (num_slices==0)
      signal(deliver);
    release(mutex);
    study();
  }
}
delivery_guy_thread {
  while (employed) {
    wait(deliver);
    make_pizza();
    acquire(mutex);
    num_slices=S;
    release(mutex);
    for (i=0;i<S;i++)
      signal(pizza);
  }
}
**Condition variable solution**

- **Shared data:**
  ```
  int slices=0;
  bool has_been_ordered;
  ```

- **Condition order:**
  ```
  // an order has been placed
  ```

- **Condition deliver:**
  ```
  // a delivery has been made
  ```

- **Lock mutex:**
  ```
  // protects "slices";
  // associated with both Condition variables
  ```

**Monitors**

- **An object that allows one thread inside at a time**

- **Contain a lock and some condition variables**
  - **Condition variables used to allow other threads to access the monitor while one thread waits for an event to occur**
Midterm!

The kernel

* Kernel mode vs user mode
  * How these modes differ conceptually and from the
    CPU's point of view
  * How we switch between the two
* Interrupts

System calls

* What they are
* What they do
* How they do it
* What hardware is involved
* Who uses them and when

Processes and threads

* Kernel processes, kernel threads, and user
  threads
  * How these differ from one another
* Context switching
* Process and thread states
* fork, exec, wait
**Synchronization**

- Critical sections
- Locks and atomic instructions
- Mutexes, semaphores, and condition variables
- Monitors and how they are implemented
- Ways to detect / avoid deadlock

**Scheduling**

- Different scheduling algorithms and their tradeoffs
- Average response time, various “laws”
- Starvation
- Cooperative vs. preemptive scheduling

**Tips**

- Focus on lecture slides
- Review textbook, section slides and project writeups to emphasize key concepts and fill in gaps
- On Monday, when taking the exam:
  - Arrive early
  - Focus on key points
  - Work quickly; finish easy problems first