Section 4

CSE 451: Asst. 1 Preparation
Locks

- `lock_acquire(lock)`
  - Wait until lock is free, then take it
- `lock_release(lock)`
  - Release lock, waking up someone waiting for it (if any)

- At most one lock holder at a time (safety)
- If no one holding, acquire gets lock (progress)
- If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)
Locks – Best Practices

• Lock is initially free
• Always acquire before accessing shared data structure
  • Beginning of procedure!
• Always release after finishing with shared data
  • End of procedure!
  • DO NOT throw lock for someone else to release
• Never access shared data without lock
  • Danger!
Lock::acquire() {
    disableInterrupts();
    spinlock.acquire();
    if (value == BUSY) {
        waiting.add(myTCB);
        suspend(&spinlock);
    } else {
        value = BUSY;
    }
    spinlock.release();
    enableInterrupts();
}

Lock::release() {
    disableInterrupts();
    spinlock.acquire();
    if (!waiting.Empty()) {
        next = waiting.remove();
        scheduler->makeReady(next);
    } else {
        value = FREE;
    }
    spinlock.release();
    enableInterrupts();
}
Spinlocks

- Processor waits in a loop for the lock to become free
  - Assumes lock will be held for a short time
  - Used to protect CPU scheduler and to implement waiting locks

- Uses read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware

- Need spinlocks to implement locks on multiprocessor machines
  - Turning off interrupts is not enough!
Condition Variables API

- Waiting inside a critical section
  - Called only when holding a lock

- cv_wait(cv, lock)
  - Atomically release lock and relinquish processor
  - Reacquire the lock when wakened

- cv_signal(cv, lock)
  - Wake up a waiter, if any

- cv_broadcast(cv, lock)
  - Wake up all waiters, if any
Condition Variables Cont.

• ALWAYS hold lock when calling wait, signal, broadcast
  • Condition variable is sync FOR shared state
  • ALWAYS hold lock when accessing shared state

• Condition variable is memoryless
  • If signal when no one is waiting, no op

• Wait atomically releases lock
  • What if wait, then release?
  • What if release, then wait?
Condition Variables Semantics

• When a thread is woken up from wait, it may not run immediately
  • Signal/broadcast puts thread(s) on ready list
  • When lock is released, anyone might acquire it

• Wait MUST be in a loop
  while (need_to_wait()) {
    cv.wait(lock);
  }

• Simplifies implementation
  • Of condition variables and locks
  • Of code that uses condition variables and locks
Wait Channels

• Primitive designed for holding sleeping threads
• Protected by a spinlock, not a lock
  • Caller must hold the spinlock for any wchan function call

• wchan_sleep(wc, 1k)
  • Put the current thread to sleep and place it in the wchan
  • Spinlock is released upon sleep

• wchan_wakeone(wc, 1k)
  • Pull a thread off of the wchan and place it in the ready list

• wchan_wakeall(wc, 1k)
  • Pull all threads off of the wchan and place them in the ready list
Semaphores

- Semaphores have a non-negative integer value
  - P() atomically waits for value to become $> 0$, then decrements
  - V() atomically increments value (waking up waiter if needed)

- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
    - If value is 1, two P’s will result in value 0 and one waiter

- Semaphores are useful for
  - “Stateful” wait: interrupt handlers, fork/join
  - But otherwise don’t use them
General Synchronization Advice

• When to synchronize?
  • Modifying global variable in different threads
  • Protecting state during forced sleep (i.e. I/O)

• Pick the right primitives
  • Locks – Critical sections, modifying shared state
  • CVs – Waiting for a condition to be satisfied
  • Semaphores – Stateful waiting
  • Mix and match when necessary

• Organize and limit conflicts
  • Try to modularize code to minimize critical sections
  • Keep related synchronization close together

• When in doubt, draw pictures
  • Draw graphs of resources and consumers
  • List the order in which things are acquired
  • Look for inconsistent orders of acquisition and circular dependencies
Thread Lifecycle

- `thread_create(thread, func, args)`
  - Create a new thread to run `func(args)`
  - `OS/161: thread_fork()`

- `thread_yield()`
  - Relinquish processor voluntarily

- `thread_join(thread)`
  - In parent, wait for forked thread to exit, then return
  - `OS/161: Your job`

- `thread_exit(ret)`
  - Quit thread and clean up, wake up joiner if any
Thread Join

- Parent thread creates child thread and calls `thread_join()`
  - Enters the waiting list
  - Can only join on (joinable) child threads
    - Cannot join on detached threads
- When a child finishes, `thread_join()` returns
  - Parent enters the ready list
  - Can only join once
- What should happened if...
  - The parent joins before the child finishes?
  - The parent joins after the child finishes?
  - The parent joins just as the child is finishing?
  - The parent joins before the child even starts?
sys161.conf

- Specifies the simulated hardware you are running on
  - mainboard ramsiz=524288 cpus=1
- cpus specifies number of cores (1 to 4)
- ramsiz specifies the amount of memory you have