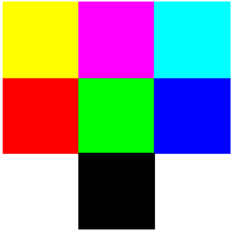
Implementing Synchronization Objects

Eric Li
Feb 4, 2021



Overview

- Goal: implement locks and condition variables (CV)

Overview

- Goal: implement locks and condition variables (CV)
- We need to atomically modify states
  - Lock: FREE / BUSY; queue of waiting threads
  - CV: queue of waiting threads

Overview

- Goal: implement locks and condition variables (CV)
- We need to atomically modify states
  - Lock: FREE / BUSY; queue of waiting threads
  - CV: queue of waiting threads
- Method: use hardware primitives
  - Disable interrupt (uniprocessor)
  - Atomic read-modify-write instructions (multiprocessor)

Overview

- Goal: implement locks and condition variables (CV)
- We need to atomically modify states
  - Lock: FREE / BUSY; queue of waiting threads
  - CV: queue of waiting threads
- Method: use hardware primitives
  - Disable interrupt (uniprocessor)
  - Atomic read-modify-write instructions (multiprocessor)
- Kernel mode vs. user mode

Uniprocessor Locks: Disabling Interrupts

- No other processors can change memory
- If no context switch, operations appear to be atomic

Uniprocessor Locks: Disabling Interrupts

- No other processors can change memory
- If no context switch, operations appear to be atomic
- Trivial implementation

```
Lock::acquire() { disableInterrupts(); }
Lock::release() { enableInterrupts(); }
```

Uniprocessor Locks: Disabling Interrupts

- No other processors can change memory
- If no context switch, operations appear to be atomic
- Trivial implementation

```
Lock::acquire() { disableInterrupts(); }
Lock::release() { enableInterrupts(); }
```

- Problem
  - Disable interrupt for a long time (starvation, not real-time)
  - Cannot allow user-level code to disable interrupts

Implementing Uniprocessor Queueing Locks

- Briefly disable interrupt to protect lock structure
- When lock is locked, context switch to another ready thread

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        chosenTCB = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        chosenTCB = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

- Do not set value = FREE in release() to prevent starvation

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    TCB *chosenTCB;

    disableInterrupts();
    if (value == BUSY) {
        waiting.add(runningThread);
        runningThread->state = WAITING;
        chosenTCB = readyList.remove();
        thread_switch(runningThread,
                      chosenTCB);
        runningThread->state = RUNNING;
    } else {
        value = BUSY;
    }
    enableInterrupts();
}
```

```
Lock::release() {
// next thread to hold lock
    TCB *next;

    disableInterrupts();
    if (waiting.notEmpty()) {
    // move one TCB from waiting
    // to ready
        next = waiting.remove();
        next->state = READY;
        readyList.add(next);
    } else {
        value = FREE;
    }
    enableInterrupts();
}
```

- Do not set value = FREE in release() to prevent starvation
- During call to `thread_switch`, interrupts are turned off

Implementing Multiprocessor Spinlocks

- Cannot turn off interrupt
- Use atomic read-modify-write instructions
  - Implementation: related to cache
  - Computer Architecture courses (ECS 154B / probably ECS 201)
  - "What Every Programmer Should Know About Memory" Figure 3.18

Implementing Multiprocessor Spinlocks

- Cannot turn off interrupt
- Use atomic read-modify-write instructions
  - Implementation: related to cache
  - Computer Architecture courses (ECS 154B / probably ECS 201)
  - "What Every Programmer Should Know About Memory" Figure 3.18
- Atomic test-and-set instruction

```c
int test_and_set(int* lockPtr, int newValue) {
    int oldValue;
    oldValue = *lockPtr;
    *lockPtr = newValue;
    return oldValue;
}
```

Implementing Multiprocessor Spinlocks

```
class SpinLock {
  private:
    int value = 0; // 0 = FREE; 1 = BUSY

  public:
    void acquire() {
        while (test_and_set(&value)) // while BUSY
            ; // spin
    }

    void release() {
        value = 0;
        memory_barrier();
    }
}
```

Implementing Multiprocessor Spinlocks

```
class SpinLock {
  private:
    int value = 0; // 0 = FREE; 1 = BUSY

  public:
    void acquire() {
        while (test_and_set(&value)) // while BUSY
            ; // spin
    }

    void release() {
        value = 0;
        memory_barrier();
    }
}
```

- Busy wait (assume locks are only held shortly)

Implementing Multiprocessor Queueing Locks

- Critical section length can be long
- Minimize busy waiting

Implementing Multiprocessor Queueing Locks

- Critical section length can be long
- Minimize busy waiting
- Class definitions

```
class Lock {
  private:
    int value = FREE;
    SpinLock spinLock;
    Queue waiting;
  public:
    void acquire();
    void release();
}
class Scheduler {
  private:
    Queue readyList;
    SpinLock schedulerSpinLock;
  public:
    void suspend(SpinLock *lock);
    void makeReady(Thread *thread);
}
```

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
        // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
        // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- Suspending a thread

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
        // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- Suspending a thread
    1. Call scheduler.suspend without releasing Lock's spinLock

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
        // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- Suspending a thread
  1. Call scheduler.suspend without releasing Lock's spinLock
  2. disableInterrupts() to prevent thread being preempted

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        scheduler.suspend(&spinLock);
        // scheduler releases spinLock
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- Suspending a thread
  1. Call scheduler.suspend without releasing Lock's spinLock
  2. disableInterrupts() to prevent thread being preempted
  3. Acquire Scheduler's spinLock to protect readyList

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        spinLock.release();
        scheduler.suspend(&spinLock);
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    // lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- What if we release Lock's spinLock before calling scheduler.suspend?

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        spinLock.release();
        scheduler.suspend(&spinLock);
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    // lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- What if we release Lock's spinLock before calling scheduler.suspend?
    1. After release spinLock

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        spinLock.release();
        scheduler.suspend(&spinLock);
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    // lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- What if we release Lock's spinLock before calling scheduler.suspend?
    1. After release spinLock
    2. Another thread release lock, make this thread READY

Implementing Uniprocessor Queueing Locks

```
Lock::acquire() {
    spinLock.acquire();
    if (value != FREE) {
        waiting.add(runningThread);
        spinLock.release();
        scheduler.suspend(&spinLock);
    } else {
        value = BUSY;
        spinLock.release();
    }
}
Lock::release() {
    TCB *next;
    spinLock.acquire();
    if (waiting.notEmpty()) {
        next = waiting.remove();
        scheduler.makeReady(next);
    } else {
        value = FREE;
    }
    spinLock.release();
}
```

```
Scheduler::suspend(SpinLock *lock) {
    TCB *chosenTCB;
    disableInterrupts();
    schedulerSpinLock.acquire();
    // lock->release();
    runningThread->state = WAITING;
    chosenTCB = readyList.getNextThread();
    thread_switch(runningThread,
                  chosenTCB);
    runningThread->state = RUNNING;
    schedulerSpinLock.release();
    enableInterrupts();
}

Scheduler::makeReady(TCB *thread) {
    disableInterrupts();
    schedulerSpinLock.acquire();
    readyList.add(thread);
    thread->state = READY;
    schedulerSpinLock.release();
    enableInterrupts();
}
```

- What if we release Lock's spinLock before calling scheduler.suspend?
  1. After release spinLock
  2. Another thread release lock, make this thread READY
  3. Current thread calls suspend(), state = WAITING forever

Linux 2.6 Kernel Mutex Lock

- Optimized for the common case
- Assumption: most locks are FREE most of the time

Linux 2.6 Kernel Mutex Lock

- Optimized for the common case
- Assumption: most locks are FREE most of the time
- Acquire: fast path when lock is not already acquired

Linux 2.6 Kernel Mutex Lock

- Optimized for the common case
- Assumption: most locks are FREE most of the time
- Acquire: fast path when lock is not already acquired
- Release: fast path when no waiters on the lock

Implementing Condition Variables

- Similar to implementing locks

```
class CV {
  private:
    Queue waiting;
  public:
    void wait(Lock *lock);
    void signal();
    void broadcast();
}

void CV::wait(Lock *lock) {
    assert(lock.isHeld());
    waiting.add(myTCB);
    scheduler.suspend(&lock);
    lock->acquire();
}
```

```
void CV::signal() {
    if (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}

void CV::broadcast() {
    while (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}
```

Implementing Condition Variables

- Similar to implementing locks
- Still, pass spinLock to scheduler.suspend

```
class CV {
  private:
    Queue waiting;
  public:
    void wait(Lock *lock);
    void signal();
    void broadcast();
}

void CV::wait(Lock *lock) {
    assert(lock.isHeld());
    waiting.add(myTCB);
    scheduler.suspend(&lock);
    lock->acquire();
}
```

```
void CV::signal() {
    if (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}

void CV::broadcast() {
    while (waiting.notEmpty()) {
        thread = waiting.remove();
        scheduler.makeReady(thread);
    }
}
```

Implementing Application-level Synchronization

- Kernel-Managed Threads
  - Simple case: place Lock and CV in kernel space, app use syscalls

Implementing Application-level Synchronization

- Kernel-Managed Threads
  - Simple case: place Lock and CV in kernel space, app use syscalls
  - Sophisticated case: fast path in user space, slow path in kernel

Implementing Application-level Synchronization

- Kernel-Managed Threads
  - Simple case: place Lock and CV in kernel space, app use syscalls
  - Sophisticated case: fast path in user space, slow path in kernel
- User-Managed Threads
  - Implement most things at user level
  - Disabling interrupts $\rightarrow$ temporarily disable upcalls (usually supported by modern OS)

Thank you

Ref: Anderson & Dahlin, Operating Systems - Principles and Practice

Ref: `https://en.wikipedia.org/wiki/Test-and-set`

Thanks: LaTeX, Beamer, OBS