Lock Implementation

Mendel Rosenblum

Lock Implementation

Optional readings:

Operating Systems: Principles and Practice: Section 5.7

Lock Implementation

How to implement locks and condition variables (inside the operating system)?

Starting with uniprocessor (single core):

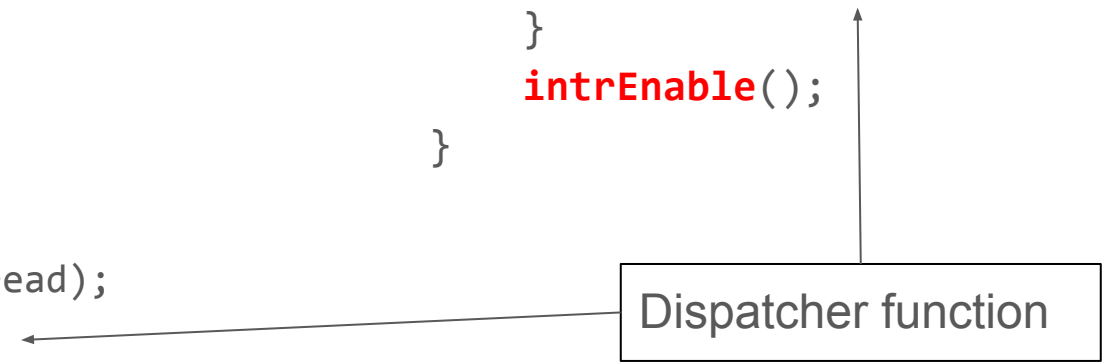
- Only traps or interrupts can cause a thread to be switched out
- Disabling interrupts give us a critical section

Uniprocessor Lock Implementation

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
};

void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = true;
    } else {
        q.add(currentThread);
        blockThread();
    }
    intrEnable();
}
```

```
void Lock::unlock() {
    intrDisable();
    if (q.empty()) {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```



Why must blockThread be invoked with interrupts disabled?

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
};

void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = true;
        intrEnable();
    } else {
        q.add(currentThread);
        intrEnable();
        blockThread();
    }
}
```

```
void Lock::unlock() {
    intrDisable();
    if (q.empty()) {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```

Uniprocessor Lock Implementation

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
};

void Lock::lock() {
    intrDisable();
    if (!locked) {
        locked = true;
    } else {
        q.add(currentThread);
        blockThread();
    }
    intrEnable();
}
```

Need this to be atomic!

```
void Lock::unlock() {
    intrDisable();
    if (q.empty()) {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    intrEnable();
}
```

Announcements

Multiprocessor Locks

- Need help from hardware to do efficiently
- atomic read-modify-write instruction
  - Atomic exchange (e.g. Intel x86 xchg)
    - Return the old value of a variable while setting it at same time

Locks for Multi-Core, v1

```
class Lock {
    Lock() {}
    std::atomic<bool> locked(false);
};

void Lock::lock() {
    while (locked.exchange(true)) {
        /* Do nothing */
    }
}
```

```
void Lock::unlock() {
    locked = false;
}
```

```
0x1140 <main()+16>:   mov   $0x1,%al
0x1142 <main()+18>:   xchg  %al,-0x1(%rsp)
0x1146 <main()+22>:   test  $0x1,%al
0x1148 <main()+24>:   jne   0x1140 <main()+16>
```

Busy waiting

Locks for Multi-Core, v2

```
class Lock {
    Lock() {}
    std::atomic<bool> locked(false);
    ThreadQueue q;
};

void Lock::lock() {
    if (locked.exchange(true)) {
        q.add(currentThread);
        blockThread();
    }
}
```

```
void Lock::unlock() {
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
}
```

What's wrong with approach?

ThreadQueue atomicity

Locks for Multi-Core, v3

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        spinlock = false;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
}
```

Why is this busy waiting acceptable?

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        spinlock = false;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
}
```

Busy waiting is unavoidable on multicores.

Try to minimize it.

Still has race conditions

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        spinlock = false;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
}
```

Need this to be atomic!

Where is race?

Locks for Multi-Core, v4

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
}
```

Change interface to dispatcher

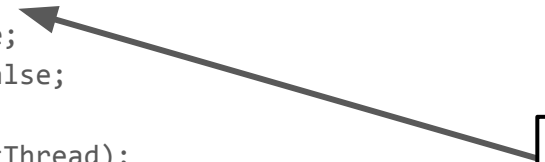
Linux approach

Locks for Multi-Core, v4

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
}
```

```
void Lock::unlock() {
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
}
```



What happens if we get an interrupt here?

Locks for Multi-Core, v5

```
class Lock {
    Lock() {}
    bool locked = false;
    ThreadQueue q;
    std::atomic<bool> spinlock;
};

void Lock::lock() {
    intrDisable();
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (!locked) {
        locked = true;
        spinlock = false;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = false;
        redispatch();
    }
    intrEnable();
}
```

```
void Lock::unlock() {
    intrDisable();
    while (spinlock.exchange(true)) {
        /* Do nothing */
    }
    if (q.empty() {
        locked = false;
    } else {
        unblockThread(q.remove());
    }
    spinlock = false;
    intrEnable();
}
```