CS111, Lecture 25

Modern Technologies and OSes

Key question: How do hardware advances impact the design of operating systems?

CS111 Topic 4: Virtual Memory

Modern Technologies and OSes - *How do hardware advances impact the design of operating systems?*

Why is answering this question important?

- Understand the full impact and utility of modern technologies we take for granted
- We can better understand the interplay between technology and OSes: OSes are at the hardware-software boundary

Learning Goals

- Learn about multicore CPUs and how they change scheduling and lock implementations
- Understand the benefits and drawbacks of flash storage and how flash storage can impact filesystem design

Plan For Today

- **Example 1:** Multicore CPUs
- **Example 2:** Flash Storage

Plan For Today

- **Example 1: Multicore CPUs**
  - Multicore scheduling
  - Multicore locks
- **Example 2:** Flash Storage

Multicore CPUs

- **True multitasking:** multiple cores let us run multiple threads simultaneously
- Starting mid-2000s, multicore processors more common in consumer devices
- OS manages these cores; new challenges!

Multicore CPUs

- Most modern consumer devices (phones, tablets, PCs) have multiple cores. *Examples:*
  - *Latest iPhone processors have 6 cores*
  - *Latest Snapdragon smartphone processors (common for Android devices) have 8 cores*
  - *Latest Intel processors have up to 24 cores*
- Now more common to have *different types of cores*; e.g. "performance" and "efficiency":
  - less-intensive tasks run on efficiency cores; more power-efficient
  - More intensive tasks run on performance cores; better performance
  - Apple, Intel + Qualcomm (major processor manufacturers) use this approach (Qualcomm at one point had 3 types of cores)
  - E.g. iPhone 16 has 2 P-cores, 4 E-cores, one Intel Core Ultra laptop chips have 4 P-cores, 4 E-cores

Aside: Other Hardware

- **GPU** is in charge of graphics
- **Newer Development:** NPU ("Neural Processing Unit") / "Neural Engine" powers machine learning / AI tasks

Multicore Challenges

OS management of multiple cores surfaces new challenges:

- **Example:** how does scheduling work with multiple CPUs?
- **Example:** how can we implement mutexes where there are multiple CPUs?

Plan For Today

- **Example 1: Multicore CPUs**
  - **Multicore scheduling**
  - Multicore locks
- **Example 2:** Flash Storage

Scheduling

Key Question: How does the operating system decide which thread to run next? (e.g. many **ready** threads).

Previously: First-Come-First-Serve, Round-Robin, SRPT, Priority-Based

What about when we have multiple cores to schedule threads on? (assume all cores equal)

Multicore Scheduling

Initial idea: one ready queue shared by **k** cores

- Share ready queue data structure across cores, lock to synchronize access
- One dispatcher per core
- Separate timer interrupts for each core
- Run the **k** highest-priority threads on the **k** cores
- When a new thread is marked "ready", compare its priority against lowest-priority running thread, preempt if new thread has higher priority.
- This works fine for 2 cores but breaks down with lots more cores. What is the main bottleneck with this approach when used with many cores?

Respond on PollEv:
pollev.com/cs111

What is the main bottleneck if we use 1 ready queue for many cores?

Nobody has responded yet.

Hang tight! Responses are coming in.

Multicore Scheduling

The single ready queue is a huge bottleneck - cores must wait for access!

Modification: have 1 ready queue *per core*.

Problem: how do we balance threads across different ready queues?

One idea: “work stealing”: if one core is free, take a thread from another core’s ready queue

- Maybe want to also do this prior to ready queue being empty? e.g. if one core has 1 ready thread and another core has 30 ready threads, the 30 threads will get less time than the 1 thread.

Another challenge: expensive to move a thread to another core.

Core Affinity

Another challenge: expensive to move a thread to another core.

- Cores have caches for data; if we move to a new core, won't have cached data
- Multiprocessor schedulers try to keep threads on same core – "core affinity"
- Maybe better in some cases to just wait for current core instead of moving?

Tension between <u>work stealing</u> (want to move often) and <u>core affinity</u> (don't want to move often)

Gang Scheduling

How should we approach scheduling if one process has several threads?

- threads may be coordinating / exchanging info
- “gang scheduling” – run all threads together on different cores.
    - Why? Thread progress may be intertwined. E.g. one thread holds lock then de-scheduled, another runs but soon needs to wait for that same lock.

Multicore Scheduling

In general: these systems all have good and bad situations – e.g. Linux scheduler had problems for many years, better now, but still some problems with load balancing and moving threads too rapidly between cores.

Plan For Today

- **Example 1: Multicore CPUs**
  - Multicore scheduling
  - **Multicore locks**
- **Example 2:** Flash Storage

Single-Core Locks

So far: our Mutex implementation relied on disabling interrupts to prevent race conditions.

```
class Lock {
    int locked = 0;
    ThreadQueue q;
};

void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

Multicore Locks

Problem: only works with single-core processors! If multiple cores, even if interrupts are disabled, some other thread could be running on another core.

How do we approach this on multicore systems?

- Turn off all other cores? Not a great option.

Key Idea: we *must* use a (small amount) of busy waiting (!!). We need a mechanism for cores to sync up before proceeding, and setting/checking a shared value is the only option.

- There's no other way to synchronize with the other cores; until we have synchronized, we can't even put a thread to sleep

Single-Core Locks

```
class Lock {
    int locked = 0;
    ThreadQueue q;
};

void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    int sync = 0;
};
```

```
void Lock::lock() {
    // try to change sync from 0 to 1
    while (true) {
        int old = sync;
        sync = 1;
        if (old == 0) break;
    }
    // we are only one proceeding now

    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        sync = 0;
        blockThread();
    }
}
```

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};
```

```
void Lock::lock() {
    // try to change sync from 0 to 1
    while (sync.exchange(1)) {}
    // we are only one proceeding now

    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        sync = 0;
        blockThread();
    }
}
```

exchange: an atomic operation that reads the memory value, replaces it with a given value, and returns the old value.

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};
```

```
void Lock::lock() {
    // try to change sync from 0 to 1
    while (sync.exchange(1)) {}
    // we are only one proceeding now

    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        sync = 0;
        blockThread();
    }
}
```

std::atomic is a C++ type that provides atomic operations for its contained data. We use it here for the atomic *exchange* operation.

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};
```

```
void Lock::unlock() {
    // try to change sync from 0 to 1
    while (sync.exchange(1)) {}
    // we are only one proceeding now

    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    sync = 0;
}
```

exchange: an atomic operation that reads the memory value, replaces it with a given value, and returns the old value.

Multicore Locks

Key idea: we’ll rely on atomic instructions provided by hardware to avoid race conditions when we have multiple cores.

Example: **exchange:** atomically read memory value, replace it with a given value, and get old value.

Additionally: single-word references and assignments (e.g., assigning ints, pointers, chars) are atomic on almost all systems.

Busy waiting unavoidable! However, it’s very short – just long enough to manipulate the lock structure.

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};

void Lock::lock() {
    while (sync.exchange(1)) {}
    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        sync = 0;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (sync.exchange(1)) {};
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    sync = 0;
}
```

Multicore Locks, V1

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};

void Lock::lock() {
    while (sync.exchange(1)) {}
    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        sync = 0;
        blockThread();
    }
}
```

```
void Lock::unlock() {
    while (sync.exchange(1)) {};
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    sync = 0;
}
```

Problem: there's an air gap in between unlocking the lock and blocking. Another thread could call unlock here, unblocking us, and then we block forever ☹

Multicore Locks

We won’t worry about these, but there are a few more steps/tweaks needed (specifically; tweaking how we block to fix race condition and continuing to use IntrGuard to disable interrupts). (*See optional slides at end if you’re interested!)*

Key overarching ideas:

- On multicore, disabling interrupts is not sufficient to eliminate race conditions
- Instead, we must rely on brief busy-waiting and provided atomic operations (**exchange**) to sync up cores before proceeding.

Plan For Today

- **Example 1:** Multicore CPUs
- **Example 2: Flash Storage**

Flash Storage

- **Much faster than hard disks:** no moving parts (no delays from platters/head!), smaller, faster
- Flash storage has become more common with increase in mobile devices, nowadays common in PCs too.
- Can buy separately, or some devices have non-removable storage (e.g., many mobile devices)
- New opportunities and challenges with managing filesystem designs for flash - has own quirks

Flash Storage Quirks

Quirk #1: Writing Data: flash storage doesn’t support just writing arbitrary data to a portion of the storage. Instead, it supports two operations that combined allow us to write data:

- **Erase:** set all bits of an *erase unit* to 1. The storage is divided up into erase units, typically 1-8MB big.
- **Write:** modify one *page*, can only clear bits to 0 (can specify some, not all). The storage is also divided up into pages, now typically 4Kbytes big. Reading data also happens in units of pages.

Flash Storage Quirks

Quirk #2: Wear-out: after erasing an erase unit many times, it no longer reliably stores data (!). Typically, around 100K.

Wear Leveling: want erase units to erase at same rate everywhere (rather than having some parts wear out before others). Ideas about moving “hot” (short-lived) and “cold” (long-lived) data around to even out storage usage.

Flash Storage and Filesystem Design

- A common approach has been to abstract away these quirks and include software in the Flash Storage that makes it look like a hard disk.
  - "Flash Translation Layer" – software that manages flash device, built in to drive, typically mimics disk interface (read/write blocks)
  - OS has no visibility into erase units, etc. – looks like a disk! **Virtualization**.
  - Advantage: use existing filesystem software
  - Disadvantages: sacrifice performance, no direct access to raw hardware, unnecessary layers / duplication
- Lots of interesting questions about what filesystems would look like if designed with flash storage in mind, without an FTL.
- Other storage technologies in the future?

Recap

- **Example 1:** Multicore CPUs
  - Multicore scheduling
  - Multicore locks
- **Example 2:** Flash Storage

Lecture 25 takeaway: Operating systems and hardware changes are tightly intertwined; multicore processors and flash storage provide two examples of the impact of hardware changes on OS implementations.

Extra Slides

Multicore Locks, V2

Somehow, we need to block and *then* unlock the lock??

- **Key insight***:* we don’t need to **block** prior to unlocking the lock; we just need to be *marked as blocked*.
- **Solution (awkward):** let’s change the interface of our thread scheduler/dispatcher to allow us to separately mark a thread as blocked and context switch. (Linux does something like this).

Multicore Locks, V2

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};

void Lock::lock() {
    while (sync.exchange(1)) {}
    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        sync = 0;
        blockThreadIfNecessary();
    }
}
```

```
void Lock::unlock() {
    while (sync.exchange(1)) {};
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    sync = 0;
}
```

Multicore Locks, Final Version

One last change – we must disable interrupts.

- **E.g.** if the timer fires right after we acquire the int, another thread trying to get it would just busy wait, wasting resources.

```
void Lock::lock() {
    while (sync.exchange(1)) {}
    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        sync = 0;
        blockThreadIfNecessary();
    }
}
```

Multicore Locks, Final Version

```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> sync(0);
};

void Lock::lock() {
    IntrGuard guard;
    while (sync.exchange(1)) {}
    if (!locked) {
        locked = 1;
        sync = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        sync = 0;
        blockThreadIfNecessary();
    }
}
```

```
void Lock::unlock() {
    IntrGuard guard;
    while (sync.exchange(1)) {};
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    sync = 0;
}
```