CS111, Lecture 26
Modern Technologies and OSes

_masks recommended_
Key question: How do hardware advances impact the design of operating systems?
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 has 3 types of cores)
  • E.g. iPhone 15 has 2 P-cores, 4 E-cores, one Intel Core i5 chip has 4 P-cores, 8 E-cores
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
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. **Why?**

• Single ready queue is main bottleneck: cores waiting for access
Multicore Scheduling

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 work stealing (want to move often) and core affinity (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 it’s waiting for that 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
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());
    }
}
**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’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.
class Lock {
    std::atomic<int> locked(0);
};

void Lock::lock() {
    while (locked.exchange(1)) {}  
}

void Lock::unlock() {
    locked = 0;
}

std::atomic is a C++ type that provides atomic operations for its contained data. We use it here for the atomic exchange operation.
class Lock {
    std::atomic<int> locked(0);
};

void Lock::lock() {
    while (locked.exchange(1)) {} // This is a functionally correct implementation.
}

void Lock::unlock() {
    locked = 0;
}

This is a functionally correct implementation. But what is one problem with this approach?
What is one problem with this approach?

Nobody has responded yet.

Hang tight! Responses are coming in.
class Lock {
    std::atomic<int> locked(0);
};

void Lock::lock() {
    while (locked.exchange(1)) {}
}

void Lock::unlock() {
    locked = 0;
}

This is a functionally correct implementation (called a “spin lock”). But what is one problem with this approach? **Busy waiting (bad!)**

We want to put threads to sleep while waiting for a lock, like in our single-core mutex.
class Lock {
    std::atomic<int> locked(0);
    ThreadQueue q;
};

void Lock::lock() {
    if (locked.exchange(1)) {
        q.add(currentThread);
        blockThread();
    }
}

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

Problem: race conditions possible! E.g., multiple threads could access + modify queue at the same time.
How can we synchronize access to our queue without a sleep mutex, and with interrupt disabling insufficient?

Must use busy-waiting.

• When synchronizing with other cores, at some level, there must be busy waiting!
• There’s no other way to synchronize with the other cores; until we have synchronized, we can’t even put a thread to sleep
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

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

void Lock::unlock() {
    while (spinlock.exchange(1)) {}
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
        spinlock = 0;
    }
}
Busy-Waiting

The only way to implement a multi-core “sleep lock” is with busy waiting.

• The busy-waiting is very short; just long enough to manipulate lock structure
class Lock {
  int locked = 0;
  ThreadQueue q;
  std::atomic<int> spinlock;
};

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

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

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

We need to somehow block before unlocking the spin lock??
Somehow, we need to block and *then* unlock the spin lock??

- **Key insight**: we don’t need to block prior to unlocking the spin 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).
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

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

void Lock::unlock() {
    while (spinlock.exchange(1)) {};
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
}
One last change – we must disable interrupts.

• E.g. if the timer fires right after we acquire the spin lock, another thread trying to get that spinlock would just busy wait, wasting resources.

```c
void Lock::lock() {
    while (spinlock.exchange(1)) {}  // Example line
    if (!locked) {
        locked = 1;
        spinlock = 0;
    } else {
        q.add(currentThread);
        currentThread->state = BLOCKED;
        spinlock = 0;
        blockThreadIfNecessary();
    }
}
```
class Lock {
    int locked = 0;
    ThreadQueue q;
    std::atomic<int> spinlock;
};

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

void Lock::unlock() {
    IntrGuard guard;
    while (spinlock.exchange(1)) {};
    if (!q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
    spinlock = 0;
}
Plan For Today

• Example 1: Multicore CPUs
• Example 2: Flash Storage
Flash Storage

• **Much faster than hard disks:** no moving parts (no seek 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 256Kbytes big.

• **Write:** modify one *page*, can only clear bits to 0. The storage is also divided up into pages, typically 512 bytes or 4Kbytes big.
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.
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, waste capacity, no direct access to raw hardware, unnecessary layers

Lots of interesting questions about what filesystems would look like if designed with flash storage in mind, without an FTL. Many research projects!

Other storage technologies in the future?
Recap

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

**Lecture 26 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.

Next time: wrap-up / life after CS111