

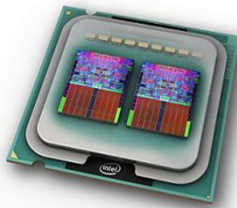
CMSC 330: Organization of Programming Languages

Multithreading

Multiprocessors

► Description

- Multiple processing units (**multiprocessor**)
- From single microprocessor to large compute clusters
- Can perform multiple tasks in parallel simultaneously



**Intel Core 2
Quad 6600**



**32 processor
Pentium Xeon**



**106K processor
IBM BlueGene/L**

Concurrency

- ▶ Important & pervasive topic in CS
- ▶ Currently covered in
 - CMSC 132 – object-oriented programming II
 - Java threads, data races, synchronization
 - CMSC 216 – low level programming / computer systems
 - C pthreads
 - CMSC 411/430 – architectures / compilers
 - Instruction level parallelism
 - CMSC 412 – operating systems
 - Concurrent processes
 - CMSC 424 – database design
 - Concurrent transactions
 - CMSC 433 – programming language technologies
 - Advanced synchronization and parallelization
 - CMSC 451 – algorithms
 - Parallel algorithms

CMSC 330

3

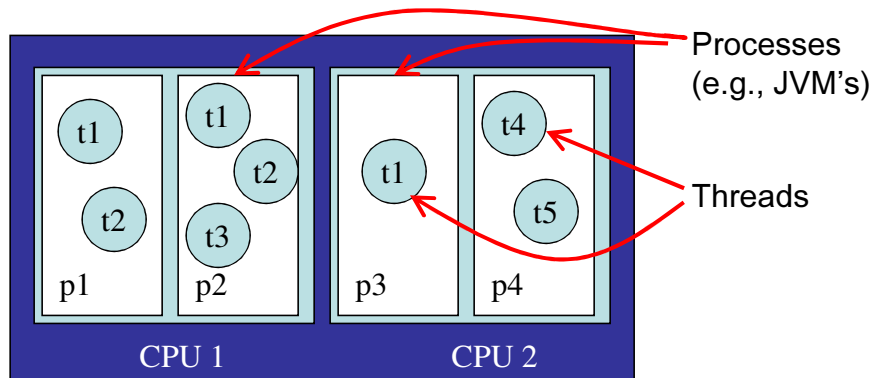
Parallelizable Applications of Interest

- ▶ Knowledge discovery: mine and analyze massive amounts of distributed data
 - Discovering social networks
 - Real-time, highly-accurate common operating picture, on small, power-constrained devices
- ▶ Simulations (games?)
- ▶ Data processing
 - NLP, vision, rendering, in real-time
- ▶ Commodity applications
 - Parallel testing, compilation, typesetting, ...

CMSC 330

4

Computation Abstractions

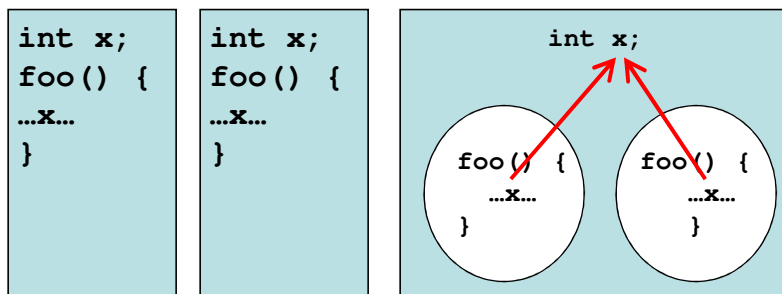


A computer

CMSC 330

5

Processes vs. Threads



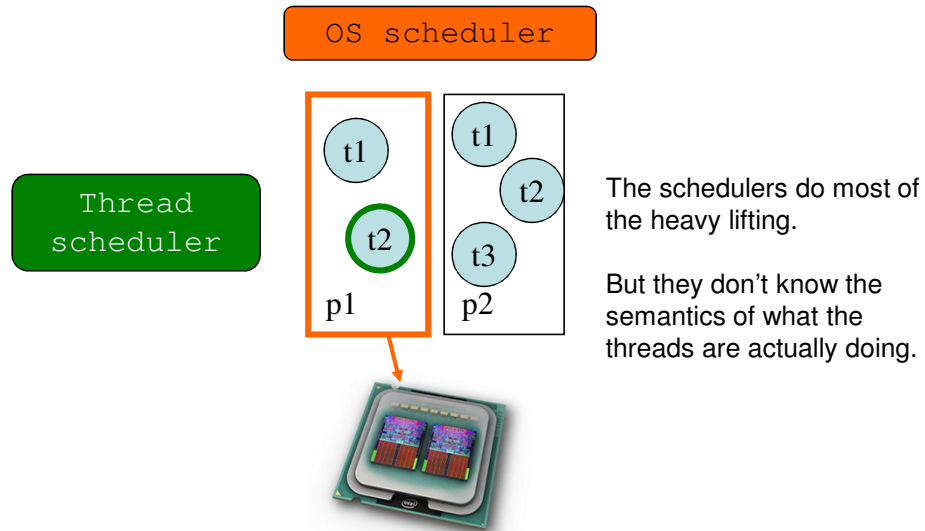
Processes do not
share data

Threads share data
within a process

CMSC 330

6

Scheduling



CMSC 330

7

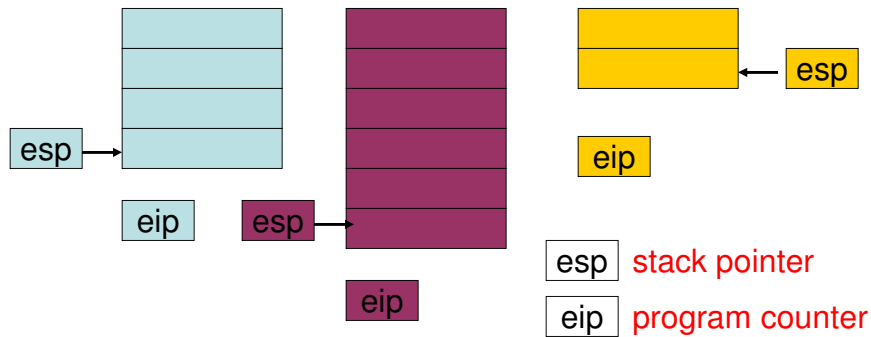
So, What Is a Thread?

- ▶ Fundamental unit of execution
 - All programs have at least one thread (main)
- ▶ Implementation view
 - A program counter and a stack
 - Heap and static area are shared among all threads

CMSC 330

8

Implementation View



- ▶ Per-thread stack and instruction pointer
 - Saved in memory when thread suspended
 - Put in hardware esp/eip when thread resumes

CMSC 330

9

Programming Processes

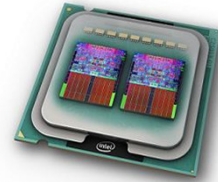
- ▶ Process creation is expensive
 - Stack, heap, PC, code, OS state
- ▶ Highly scalable
 - Virtually unlimited
- ▶ Processes may reside on separate processors
 - Sharing memory typically too expensive
- ▶ Message-passing programming paradigm
 - I/O streams, sockets, network, files
 - Cooperation key to communication (send/recv)



CMSC 330

10

Programming Threads



- ▶ Thread creation is inexpensive
 - Stack, program counter, scheduler state
- ▶ Traditionally, they don't scale well (10s of threads)
 - Multicore architectures relax this limitation
- ▶ Threads reside on same physical processor
 - So memory sharing is cheap
- ▶ Shared-memory programming paradigm
 - Everything except thread local variables are shared
 - Threads communicate via shared data
 - Synchronization used to avoid data races

CMS 330

11

What Is the Big Deal About Threads?

- ▶ Conventional wisdom: threads are hard
- ▶ Main reason: non-determinism
 - Different threads execute at different speeds
 - This leads to unpredictable results
 - When you program with threads, you have to be ready for unexpected results
- ▶ The goal of the PL designer is to make this easier

CMS 330

12

Programming Languages & Threads

- ▶ Old: libraries
 - pthreads
 - Could use different libraries for different properties
- ▶ New: primitives
 - Java, Ruby, OCaml
 - Can utilize special keywords, syntax
 - Not dependent on operating system

Which is better?

CMSC 330

13

Example

- ▶ $x = 0$ initially. Then these threads are executed:

T1	$y = x;$	T2	$z = x;$
	$x = y+1;$		$x = z+2;$

- ▶ What is the value of x afterward 3 1 2

T1	$y = x;$	T2	
	$x = y+1;$		
			$z = x;$
			$x = z+2;$

T1		T2	$z = x;$
			$x = z+2;$
	$y = x;$		
	$x = y+1;$		

T1	$y = x;$	T2	
			$z = x;$
			$x = z+2;$
	$x = y+1;$		

T1		T2	$z = x;$
	$y = x;$		
	$x = y+1;$		
			$x = z+2;$

Data Races

- ▶ That was an example of a **data race**
 - Threads are “racing” to read, write x
 - The value of x depends on who “wins” (3, 1, 2)
- ▶ Languages rarely specify who wins data races
 - The outcome is nondeterministic
- ▶ So programmers restrict certain outcomes
 - Synchronization with locks, condition variables
- ▶ And they often mess up
 - Deadlocks, bugs that are hard to track down...

CMS3 330

15

Thread API Concepts

- ▶ Thread management
 - Creating, killing, joining (waiting for) threads
 - Sleeping, yielding, prioritizing
- ▶ Synchronization
 - Controlling order of execution, visibility, atomicity
 - **Locks**: Can prevent data races, but watch out for deadlock!
 - **Condition variables**: supports communication between threads
- ▶ Most languages have similar APIs, details differ

CMS3 330

16

Java – Creating Threads

- ▶ Thread.create(Runnable r)
 - Or subclass the Thread class
 - Java makes it hard to create threads that access local variables (since it does not have closures)
- ▶ In practice, there are better ways
 - Use thread pools, to separate the idea of creating a thread from creating a (Runnable) task
 - May have N threads execute M > N jobs
- ▶ We'll stick with the simple idea here

CMSC 330

17

Thread Creation Example

```
public class MyT implements Runnable {
    public void run( ) {
        ...           // particular work for this thread
    }
}

Thread t = new Thread(new MyT( )); // create thread
t.start(); // begin running thread
...           // thread executing in parallel
t.join();     // waits for thread to exit
```

CMSC 330

18

Locks

- ▶ Language designers limit non-determinism by introducing **concurrency-control** constructs
 - Make some parts deterministic = more predictable
 - Trade-off: reducing concurrency can reduce performance
- ▶ Common concurrency-control feature: **locks**
 - They “guard” shared resources that shouldn’t be accessed by more than one thread at a time
- ▶ The gist
 - At most one owner at a time
 - If someone else owns it, you block
 - When you’re done with it, release ownership

CMSC 330

19

Java Locks (1.4)

- ▶ Objects each have an associated lock
- ▶ Use **synchronized** keyword to acquire lock
 - Code blocks – `synchronized (o) { ... }` // lock for Object o
 - Methods – `synchronized foo( ) { ... }` // lock for this
- ▶ Thread blocks when lock held
 - Thread returns when lock is finally acquired
 - May **deadlock** if threads try to acquire each other’s lock
- ▶ Locks sometimes referred to as **mutexes**

CMSC 330

20

Why Locks?

- ▶ #1 concern: prevent data races
- ▶ Patterns of use:
 - Enforce atomicity of shared data
 - Rule of thumb 1: You must hold a lock when accessing shared data
 - Rule of thumb 2: You must not release a lock until shared data is in a valid state
 - Overuse use of synchronization can create deadlock
 - Rule of thumb: No deadlock if only one lock
- ▶ Synchronization also used to ensure ordering and visibility
 - The last is due to memory models in modern arch's

CMSC 330

21

Synchronization Example (Java 1.4)

```
public class Example extends Thread {  
    private static int cnt = 0;  
    public void run() {  
        synchronized (this) {  
            int y = cnt;  
            cnt = y + 1;  
        }  
    }  
    ...  
}
```

Acquires the lock
associated w/ current
object; only succeeds if
lock not held by another
thread, otherwise blocks

Releases the lock

CMSC 330

22

Producer / Consumer Problem

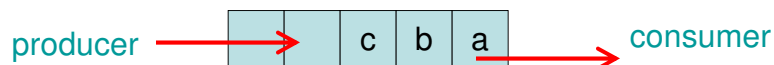
- ▶ Suppose we are communicating with a shared variable
 - E.g., a fixed size buffer holding messages
- ▶ One thread **produces** input to the buffer
- ▶ One thread **consumes** data from the buffer
- ▶ Rules
 - Producer can't add input to the buffer if it's full
 - Consumer can't take input from the buffer if it's empty

CMS3 330

23

Producer / Consumer Idea

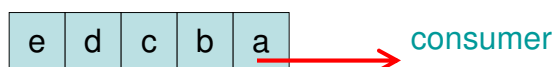
- ▶ If buffer is partially full, producer or consumer can run



- ▶ If buffer is empty, only producer can run



- ▶ If buffer is full, only consumer can run



CMS3 330

24

Broken Producer/Consumer Example

```
boolean valueReady = false;  
Object value;
```

```
void produce(object o) {  
    synchronized (this) {  
        while (valueReady)  
            ;  
        value = o;  
        valueReady = true;  
    }  
}  
  
Object consume() {  
    synchronized (this) {  
        while (!valueReady)  
            ;  
        Object o = value;  
        valueReady = false;  
        return o;  
    }  
}
```

Threads wait with lock held – no way to make progress

CMSC 330

25

Broken Producer/Consumer Example

```
boolean valueReady = false;  
Object value;
```

```
void produce(object o) {  
    while (valueReady)  
        ;  
    synchronized (this) {  
        value = o;  
        valueReady = true;  
    }  
}  
  
Object consume() {  
    while (!valueReady)  
        ;  
    synchronized (this) {  
        Object o = value;  
        valueReady = false;  
        return o;  
    }  
}
```

valueReady accessed without a lock held – data race

CMSC 330

26

Inefficient Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

Constantly acquiring / releasing lock — busy wait

```
void produce(Object o) {
    boolean done = false;
    while (!done) {
        synchronized (this) {
            if (!valueReady) {
                value = o;
                valueReady = true;
                done = true;
            }
        }
    }
}

Object consume() {
    Object o = null;
    boolean done = false;
    while (!done) {
        synchronized (this) {
            if (valueReady) {
                o = value;
                valueReady = false;
                done = true;
            }
        }
    }
    return o;
}
```

CMSC 330

27

Solving Producer / Consumer Problem

- ▶ Difficult to use locks directly
 - Very hard to get correct (or efficient) solution
 - Problems often very subtle
- ▶ Proper approach – use **signaling**
- ▶ Common signaling scenario
 1. You get the lock
 2. You realize it's no good to you yet (buffer is empty)
 3. Go to sleep: “wake me up when there's work to do”
- ▶ Virtually every threading model supports this
 - Condition variables
 - Wait / notify

CMSC 330

28

Condition Variables

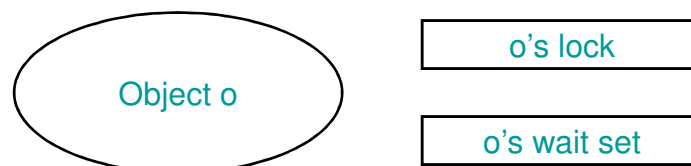
- ▶ Each condition variable represents those threads waiting for a condition to become true
 - Implemented, at least conceptually, as a wait set associated with the condition variable
- ▶ Since different threads access the variable at once, we must protect the wait set contents with a lock

CMS 330

29

Condition Variables in Java 1.4

- ▶ Use **synchronized** on object to get associated lock



- ▶ Objects also have an associated condition variable (and thus a wait set)

CMS 330

30

Wait and NotifyAll

- ▶ `o.wait()`
 - Must hold lock associated with `o`
 - Release that lock
 - And no other locks
 - Adds this thread to wait set for lock
 - Blocks the thread
- ▶ `o.notifyAll()`
 - Must hold lock associated with `o`
 - Resumes all threads on lock's wait set
 - Those threads must reacquire lock before continuing
 - This is part of the function; you don't need to do it explicitly

CMS3 330

31

Producer/Consumer Example

```
boolean valueReady = false;  
Object value;
```

```
void produce(Object o) {  
    synchronized (this) {  
        while (valueReady)  
            wait();  
        value = o;  
        valueReady = true;  
        notifyAll();  
    }  
}  
  
Object consume() {  
    synchronized (this) {  
        while (!valueReady)  
            wait();  
        Object o = value;  
        valueReady = false;  
        notifyAll();  
        return o;  
    }  
}
```

CMS3 330

32

Using Conditions Correctly

- ▶ `wait()` **must** be called in a while loop
 - Conditions may not be met when wait returns
 - Some other thread may have awoken first
 - ...and changed condition (e.g., consumed item in buffer)
- ▶ Avoid holding other locks when waiting
 - `wait()` only gives up lock on object you are waiting on
 - Reduces possibility of deadlock

CMS330

33

Broken Producer/Consumer Example

```
boolean valueReady = false;  
Object value;
```

```
void produce(Object o) {  
    synchronized (this) {  
        if (valueReady)  
            wait();  
        value = o;  
        valueReady = true;  
        notifyAll();  
    }  
}  
  
Object consume() {  
    synchronized (this) {  
        if (!valueReady)  
            wait();  
        Object o = value;  
        valueReady = false;  
        notifyAll();  
        return o;  
    }  
}
```

- ▶ Illegal access if **multiple** producers or consumers

CMS330

34

Lock Interface (Java 1.5)

```
interface Lock {  
    void lock();  
    void unlock();  
    ... /* Some more stuff, also */  
}  
class ReentrantLock implements Lock { ... }
```

- ▶ Explicit Lock objects
 - Same as implicit lock used by synchronized keyword
- ▶ Only one thread can hold a lock at once
 - lock() causes thread to **block** (become suspended) until lock can be acquired
 - unlock() allows lock to be acquired by different thread

CMSC 330

35

Synchronization Example (Java 1.5)

```
public class Example extends Thread {  
    private static int cnt = 0;  
    static Lock lock = new ReentrantLock();  
    public void run() {  
        lock.lock();  
        int y = cnt;  
        cnt = y + 1;  
        lock.unlock();  
    }  
    ...  
}
```

Lock, for protecting the shared state

Acquires the lock; only succeeds if lock not held by another thread, otherwise blocks

Releases the lock

CMSC 330

36

ReentrantLock Class (Java 1.5)

```
class ReentrantLock implements Lock { ... }
```

- ▶ Reentrant lock
 - Can be reacquired by same thread by invoking `lock()`
 - Up to 2147483648 times
 - To release lock, must invoke `unlock()`
 - The **same** number of times `lock()` was invoked
- ▶ Reentrancy is useful
 - Each method can acquire/release locks as necessary
 - No need to worry about whether callers already have locks
 - Discourages complicated coding practices
 - To determine whether lock has already been acquired

CMSC 330

37

Reentrant Lock Example

```
static int count = 0;
static Lock l =
    new ReentrantLock();

void inc() {
    l.lock();
    count++;
    l.unlock();
}
```

```
void returnAndInc() {
    int temp;

    l.lock();
    temp = count;
    inc();
    l.unlock();
}
```

- ▶ Example
 - `returnAndInc()` can acquire lock and invoke `inc()`
 - `inc()` can acquire lock without having to worry about whether thread already has lock

CMSC 330

38

Condition Interface (Java 1.5)

```
interface Lock { Condition newCondition(); ... }  
interface Condition {  
    void await();  
    void signalAll(); ... }
```

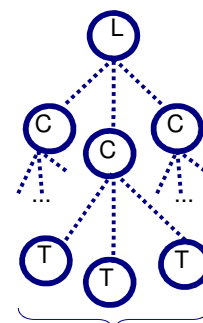
- ▶ Explicit condition variable objects
 - Condition variable **C** is created from a Lock object **L** by calling **L.newCondition()**
 - Condition variable **C** is then associated with **L**
- ▶ Multiple condition objects per lock
 - Allows different wait sets to be created for lock
 - Can wake up different threads depending on condition

CMS 330

39

Condition – await() and signalAll()

- ▶ Calling **await()** w/ lock held
 - Releases the lock
 - But not any other locks held by this thread
 - Adds this thread to **wait set** for condition
 - Blocks the thread
- ▶ Calling **signalAll()** w/ lock held
 - Resumes all threads in condition's wait set
 - Threads must reacquire lock
 - Before continuing (returning from await)
 - Enforced automatically; you don't have to do it



CMS 330

40

Producer / Consumer Solution (Java 1.5)

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean bufferReady = false;
Object buffer;
```

```
void produce(Object o) {      Object consume() {
    lock.lock();              lock.lock();
    while (bufferReady)      while (!bufferReady)
        ready.await();        ready.await();
    buffer = o;              Object o = buffer;
    bufferReady = true;      bufferReady = false;
    ready.signalAll();       ready.signalAll();
    lock.unlock();           lock.unlock();
}                             return o; }
```

- Uses single condition per lock (as in Java 1.4)

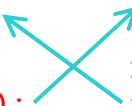
CMSC 330

41

Producer / Consumer Solution (Java 1.5)

```
Lock lock = new ReentrantLock();
Condition producers = lock.newCondition();
Condition consumers = lock.newCondition();
boolean bufferReady = false;
Object buffer;
```

```
void produce(Object o) {      Object consume() {
    lock.lock();              lock.lock();
    while (bufferReady)      while (!bufferReady)
        producers.await();    consumers.await();
    buffer = o;              Object o = buffer;
    bufferReady = true;      bufferReady = false;
    consumers.signalAll();    producers.signalAll();
    lock.unlock();           lock.unlock();
}                             return o; }
```



- Uses 2 conditions per lock for greater efficiency

CMSC 330

42