A review of Anderson's lectures

- Extract key points

Review

- Why study operating systems?
- What is an operating system?
- Principles of operating system design
- History of operating systems

Why study OS?

- **Abstraction:** OS is a wizard, providing illusion of infinite CPUs, infinite memory, single worldwide computing, etc.
- **System Design:** tradeoffs between performance and simplicity, crosscutting, putting functionality in hardware vs. software, etc.
- **How computers work:** "look under the hood" of computer systems

What is an Operating System?

- Definition: An operating system implements a virtual machine that is (hopefully) easier to program than the raw hardware:
  - Application
    - » Virtual Machine Interface
  - Operating System
    - » Physical Machine Interface
  - Hardware

Just a software engineering problem?

- In some sense, OS is just a software engineering problem: how do you convert what the hardware gives you into something that the application programmers want?

Key questions in OS

- For any OS area (file systems, virtual memory, networking, CPU scheduling), begin by asking two questions:
  - what's the hardware interface? (the physical reality)
  - what's the application interface? (the nicer abstraction)

Same theme at higher levels

- what’s the programming language (e.g. Java) interface? (the programming reality)
- what’s the application interface? (the nicer abstraction)

Dual-mode operation

- when in the OS, can do anything (kernel-mode)
- when in a user program, restricted to only touching that program's memory (user-mode)
  - don’t need boundary between kernel and application if system is dedicated to a single application.

Portable operating system

- want OS to be portable, so put in a layer that abstracts out differences between different hardware architectures.
- OS
  - portable OS layer
  - machine dependent OS layer

Operating systems principles

- Meta-principle: OS design tradeoffs change as technology changes.

History

- **History Phase 1: hardware expensive, humans cheap**
- **History, Phase 2: hardware cheap, humans expensive**
- **History, Phase 3: hardware very cheap, humans very e x p e n s i v e**

Lecture 2: Concurrency: Threads, Address Spaces and Processes

- OS has to coordinate all the activity on a machine -- multiple users, I/O interrupts, etc.
- How can it keep all these things straight?
- Answer: Decompose hard problem into simpler ones. Instead of dealing with everything going on at once, separate so deal with one at a time.

Processes

- **Process:** Operating system abstraction to represent what is needed to run a single program (this is the traditional UNIX definition)
- Formally, a process is a sequential stream of execution in its own address space.

Processes

- **Two parts to a process:**
  - **sequential execution:** No concurrency inside a process -- everything happens sequentially.
  - **process state:** everything that interacts with process.
- **Process =? Program**
  - More to a process than just a program
  - Less to a process than a program

Threads

- **Thread**: a sequential execution stream within a process (concurrency)
- Sometimes called: a "lightweight" process.
- **Address space:** all the state needed to run a program (literally, all the addresses that can be touched by the program).
- **Multithreading:** a single program made up of a number of different concurrent activities

Thread state

- Some state shared by all threads in a process/address space: contents of memory (global variables, heap), file system
- Some state "private" to each thread -- each thread has its own program counter, registers, execution stack
- Threads encapsulate concurrency; address spaces encapsulate protection

Book

- Book talks about processes: when this
  - concerns concurrency, really talking about thread portion of a process; when this
  - concerns protection, really talking about address space portion of a process.

Lecture 3: Threads and Dispatching

- Each thread has illusion of its own CPU
  - Thread control block: one per thread execution state: registers, program counter, pointer to stack scheduling information, etc.
- **Dispatching Loop (scheduler.cc)**
  - LOOP
    - Run thread
    - Save state (into thread control block)
    - Choose new thread to run
    - Load its state (into TCB) and loop

Running a thread

- Load its state (registers, PC, stackpointer) into the CPU, and do a jump.
- How does dispatcher get control back? Two ways:
  - Internal events: IO, other thread, yield
  - External events: Interrupts, timer

Choosing a thread to run

- Dispatcher keeps a list of ready threads -- how does it choose among them?
  - Zero ready threads -- dispatcher just loops
  - One ready thread -- easy.
  - More than one ready thread:
    - LIFO
    - FIFO
    - Priority queue

Thread states

- Each thread can be in one of three states:
  - Running -- has the CPU
  - Blocked -- waiting for I/O or synchronization with another thread
  - Ready to run -- on the ready list, waiting for the CPU

Lecture 4: Independent vs. cooperating threads

- **Independent threads**: No state shared with other threads Deterministic -- input state determines result, Reproducible, Scheduling order doesn't matter
- **Cooperating threads:** Shared state, Non-deterministic, Non-reproducible

Why allow cooperating threads?

- **Why allow cooperating threads?**
- **Speedup**
- **Modularity** chop large problem up into simpler pieces
- Need:
  - **Atomic operation:** operation always runs to completion, or not at all. Indivisible, can't be stopped in the middle.

Lecture 5: Synchronization: Too Much Milk

- **Synchronization:** using atomic operations to ensure cooperation between threads
- **Mutual exclusion:** ensuring that only one thread does a particular thing at a time. One thread doing it *excludes* the other, and vice versa.
- **Critical section:** piece of code that only one thread can execute at once. Only one thread at a time will get into the section of code.

Lecture 5

- **Lock:** prevents someone from doing something.
  - Lock before entering critical section, before accessing shared d a t a
  - unlock when leaving, after done accessing shared data
  - wait if locked
- Key idea -- all synchronization involves waiting.

Too Much Milk Summary

- Have hardware provide better (higher-level) primitives than atomic load and store.
- Use locks as atomic building block and solution becomes easy:
  - lock->Acquire();
    - if (nomilk) buy milk;
  - lock->Release();

Lecture 6: Implementing Mutual Exclusion

- High level atomic operations (API)
  - locks, semaphores, monitors, send&receive
- Low level atomic operations (hardware)
  - load/store, interrupt disable, test&set

Lecture 7: Semaphores and Bounded Buffer

- Writing concurrent programs is hard because you need to worry about multiple concurrent activities writing the same memory; hard because ordering matters.
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state. What are the right synchronization abstractions, to make it easy to build correct concurrent programs?

Definition of Semaphores

- Semaphores are a kind of generalized lock, first defined by Dijkstra in the late 60's. Semaphores are the main synchronization primitive used in UNIX.
- ATOMIC operations
  - P = Down, waits for positive, decrements by 1
  - V = Up, increments by 1, waking up any waiting P

Two uses of semaphores

- **Mutual exclusion**
  - semaphore->P();
  - // critical section goes here
  - semaphore->V();
- **Scheduling constraints**
  - semaphores provide a way for a thread to wait for something.

Motivation for monitors

- Semaphores are a huge step up; But problem with semaphores is that they are dual purpose. Used for both mutex and scheduling constraints.
- This makes the code hard to read, and hard to get right.

Monitors

- Idea in monitors is to separate these concerns:
  - use locks for mutual exclusion and
  - condition variables for scheduling constraints
- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data

Lock

- Lock::Acquire -- wait until lock is free, then grab it
- Lock::Release -- unlock, wake up anyone waiting in Acquire

condition variables

- Key idea with condition variables: make it possible to go to sleep inside critical section, by **atomically** releasing lock at same time we go to sleep
- **Condition variable**: a queue of threads waiting for something **inside** a critical section

Difference between monitors and Java classes

- From Solomon: processes.html: First, instead of marking a whole class as monitor, you have to remember to mark each method as synchronized. Every object is potentially a monitor. Second, there are no explicit condition variables. In effect, every monitor has exactly one anonymous condition variable. Instead of writing c.wait() or c.notify(), where c is a condition variable, you simply write wait() or notify()

Synchronized Queue

```
• public synchronized void
  AddToQueue(Object item) {
        // add item to queue;
        notify();
  }
```

Synchronized Queue

```
public synchronized Object
  RemovFromQueue() {
    while ( is_empty())
       try {wait(); }
         catch(InterruptedException
                  ex) {};
    //remove item from queue;
  return item;}
```

Summary Lecture 7

- Monitors represent the logic of the program -- wait if necessary, signal if change something so waiter might need to wake up.

Synchronization in Java

- Monitors
- Separate core behavior from synchronization behavior

Basic behavior

```
public class GroundCounter {
  protected long count_;
  protected GroundCounter(long c){
    count_ = c;
  }
  protected long value_(){return count_;}
  protected void inc_() { ++ count_;}
  protected void dec_() { -- count_;}
}
```

Synchronization using Subclassing

```
public class BoundedCounterC extends GroundCounter
  implements BoundedCounter {
  public BoundedCounterC() { super(MIN); }
  public synchronized long value() { return value_();}
  public synchronized void inc() {
    while (value_() >= MAX)
      try {wait();} catch(InterruptedException ex){};
    inc_(); notifyAll();}
  public sychronized void dec() {
    while (value_() <= MIN)
      try {wait();} catch(InterruptedException ex) {};
    dec_(); notifyAll();}
  }
```

BoundedCounter interface

```
public interface BoundedCounter {
  public static final long MIN = 0;
  public static final long MAX = 10;
  public long value();// invariant:
     // MIN <= value() <= MAX
  public void inc(); // only when value()<MAX
  public void dec(); // only when value()>MIN
}
```

Advantage of separation

- Less tangling: separation of concerns.
- Can more easily use different synchronization policy; can use synchronization policy with other basic code.
- Avoids to mix variables used for synchronization with variables used for basic behavior.

Implementation rules

- For each condition that needs to be waited on, write a guarded wait loop.
- Ensure that every method causing state changes that affect the truth value of any waited-for condition invokes notifyAll to wake up any threads waiting for state changes.

Waits

- While loop is essential: when an action is resumed, the waiting task does not know whether the condition is true: it must check again. Avoid busy waits like:

```
protected void spinWaitUntilCond(){
  while (!cond_)
    Thread.currentThread().yield();
}
```

Notifications

- Good to start with blanket notifications using notifyAll
- notifyAll is an expensive operation
- optimize later

Semaphores in Java

```
public final class CountingSemaphore {
  private int count_ = 0;
  public CountingSemaphore(int inC) {
    count_ = inC;}
  public void P() { // down
    while (count_ <= 0)
      try {wait();}
        catch (InterruptedException ex) {}
      --count_;}
  public void V() { // up
    ++count_; notify(); }
}
```

From: Concurrent Programming in Java by Doug Lea

Readers and Writers

```
public abstract class RW {
  protected int activeReaders_ = 0; //threads executing read_
  protected int activeWriters_ = 0; //always zero or one
  protected int waitingReaders_ = 0; //threads not yet in read_
  protected int waitingWriters_ = 0; // same for write_

  protected abstract void read_(); //implement in subclasses
  protected abstract void write_();//implement in subclasses
  public void read(){beforeRead(); read_();afterRead();}
  public void write(){beforeWrite(); write_();afterWrite();}
  protected boolean allowReader() {
    return waitingWriters_ == 0 && activeWriters_ == 0;}
  protected boolean allowWriter() {
    return activeReaders_==0 && activeWriters_ == 0;}
```

From: Concurrent Programming in Java by Doug Lea

Readers and Writers

```
// continued: public abstract class RW {
  protected synchronized void beforeRead() {
    ++ waitingReaders_;
    while (!allowReader())
      try {wait();} catch (InterruptedException ex) {}
    -- waitingReaders_;
    ++ activeReaders_; }
  protected synchronized void afterRead() {
    --activeReaders_;
    notifyAll();}
```

Readers and Writers

```
// continued: public abstract class RW {
  protected synchronized void beforeWrite() {
    ++ waitingWriters_;
    while (!allowWriter())
      try {wait();} catch (InterruptedException ex) {}
    -- waitingWriters_;
    ++ activeWriters_; }
  protected synchronized void afterWrite() {
    --activeWriters_;
    notifyAll();}
}
```

Threads and locks

- Java associates a lock with every object. The lock is used to allow only one thread at a time to execute a region of protected code.
- The synchronized statement synchronized(e) {b} (1) locks a lock associated with the object returned by e and (2) after executing b, it unlocks the same lock.

Threads and locks

- As a convenience, a method may be synchronized. Such a method behaves as if its method were synchronized in a synchronized statement.

```
synchronized void f(){b} =
void f(){ synchronized(this) {b};}
```

Threads and locks

- Code in one synchronized method may make self-calls to another synchronized method in the same object without blocking.
- Similarly for calls on other objects for which the current thread has obtained and not yet released a lock.
- Synchronization is retained when calling an unsynchronized method from a synchronized one.

Example

```
Class A{
  synchronized void f(){this.g();}
  synchronized void g(){…};
}
…
A a; a.f();
// The a-lock will be acquired
  twice and released twice.
```

Threads and locks

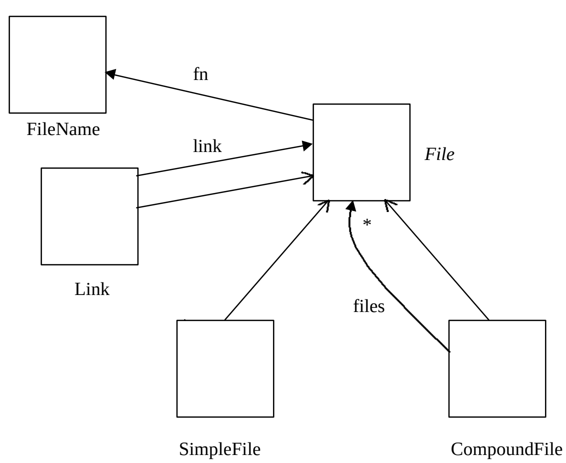
- Only one thread at a time is permitted to lay claim on a lock, and moreover a thread may acquire the same lock multiple times and doesn't relinquish ownership of it until a matching number of unlock actions have been performed.
- An unlock action by a thread T on a lock L may occur only if the number of preceding unlock actions by T on L is strictly less than the number of preceding lock action by T on L. (unlock only what it owns)

Threads and locks

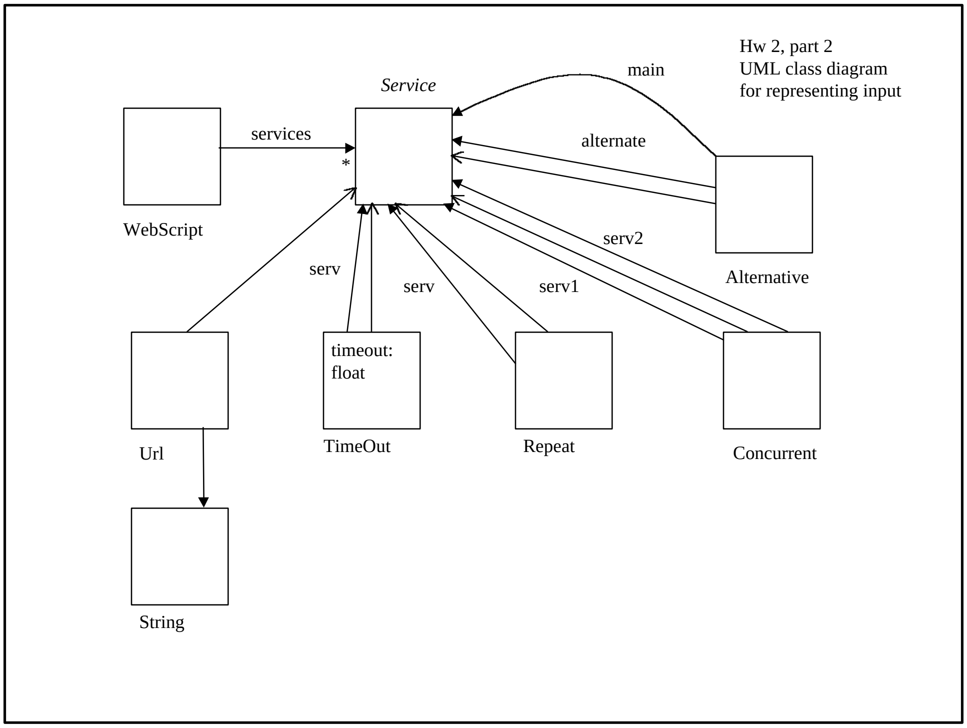
- A notify invocation on an object results in the following:
  - If one exists, an arbitrarily chosen thread, say T, is removed by the Java runtime system from the internal wait queue associated with the target object.
  - T must re-obtain the synchronization lock for the target object which will always cause it to block at least until the thread calling notify releases the lock.
  - T is then resumed at the point of its wait.

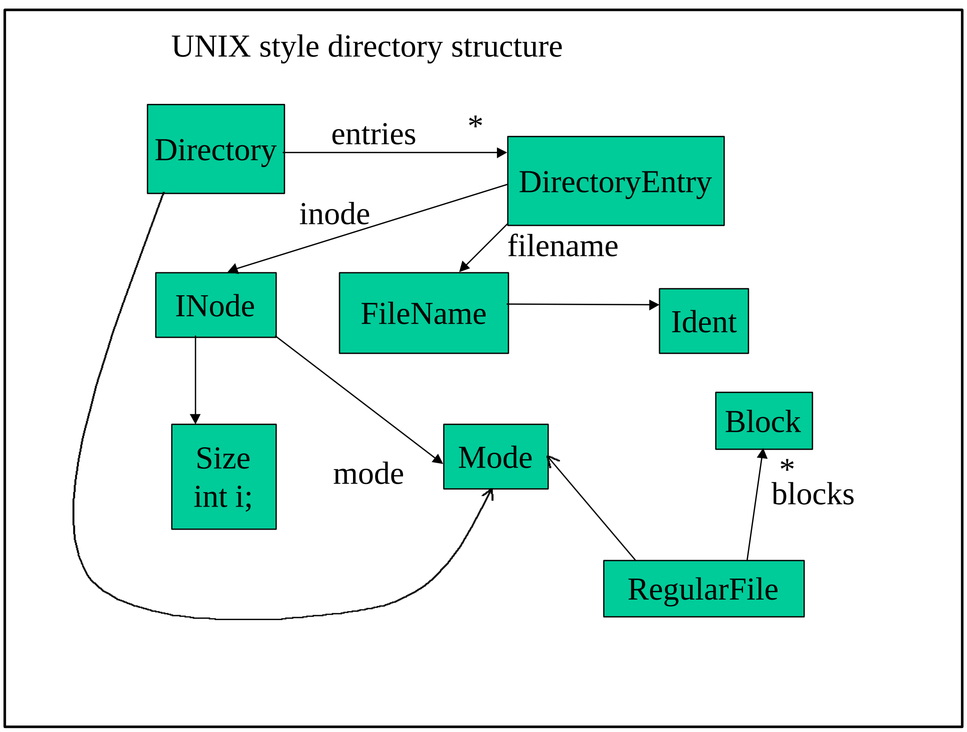
HW related viewgraphs

- UML class diagram
- Law of Demeter

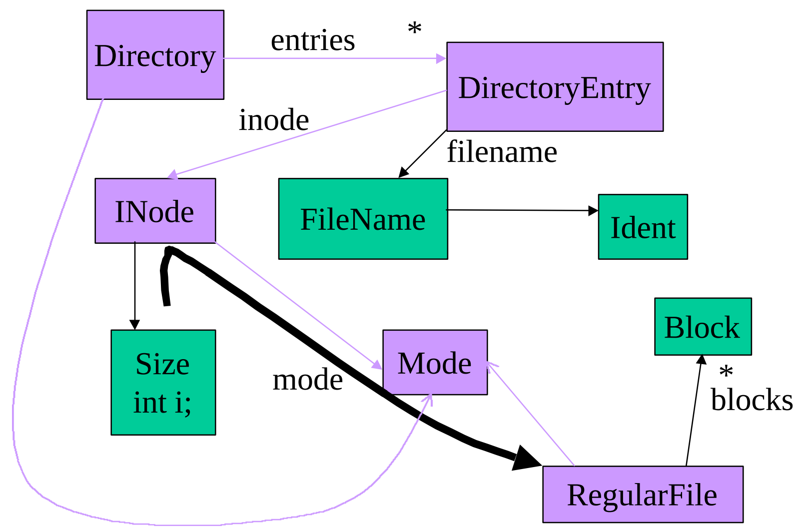


Hw 2, Part 1 UML class diagram
For representing file system
italics: abstract class


Hw 2, part 2
UML class diagram
for representing input
Service
main
services
*
alternate
WebScript
serv2
serv
serv
serv1
Alternative
timeout:
float
Url
TimeOut
Repeat
Concurrent
String

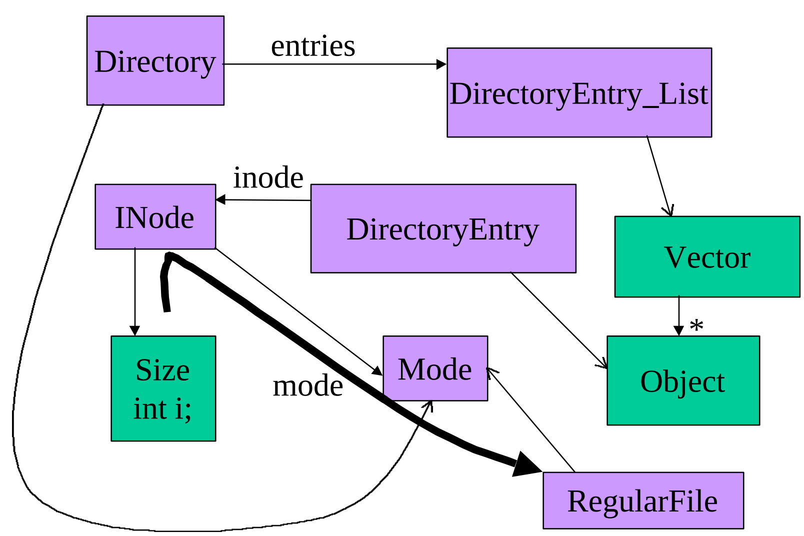

UNIX style directory structure
Directory
entries
*
DirectoryEntry
inode
filename
INode
FileName
Ident
Block
*
blocks
Size
int i;
mode
Mode
RegularFile

Maximum RegularFile size



Traversal: from Directory to RegularFile

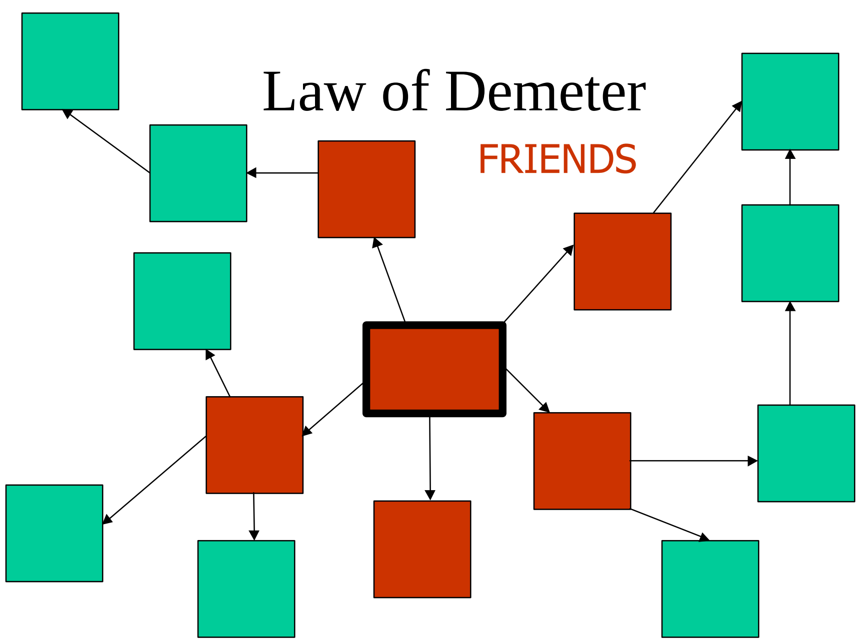
Maximum RegularFile size: implementation structure



Law of Demeter Principle

- **Each unit should only use a limited set of other units: only units "closely" related to the current unit.**
- "Each unit should only talk to its friends." "Don't talk to strangers."
- Main Motivation: Control information overload. We can only keep a limited set of items in short-term memory.

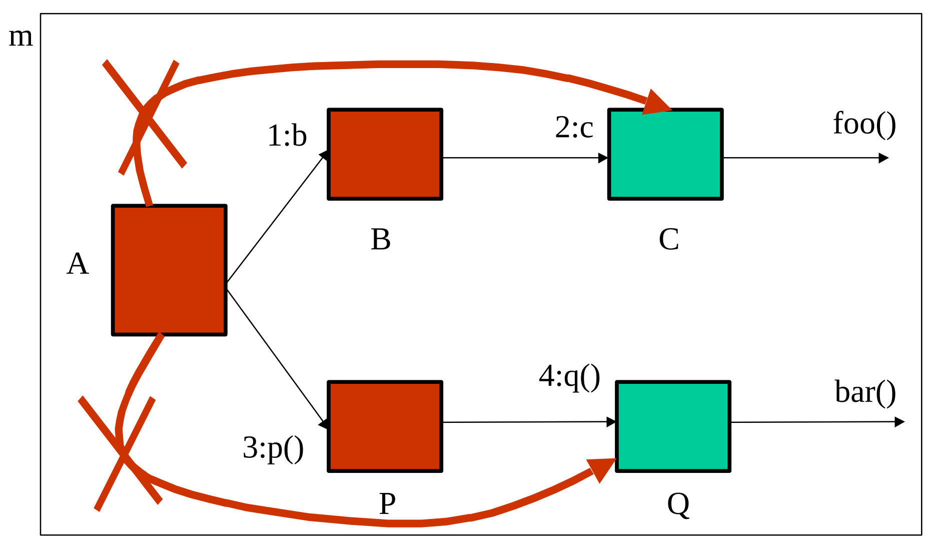
Law of Demeter



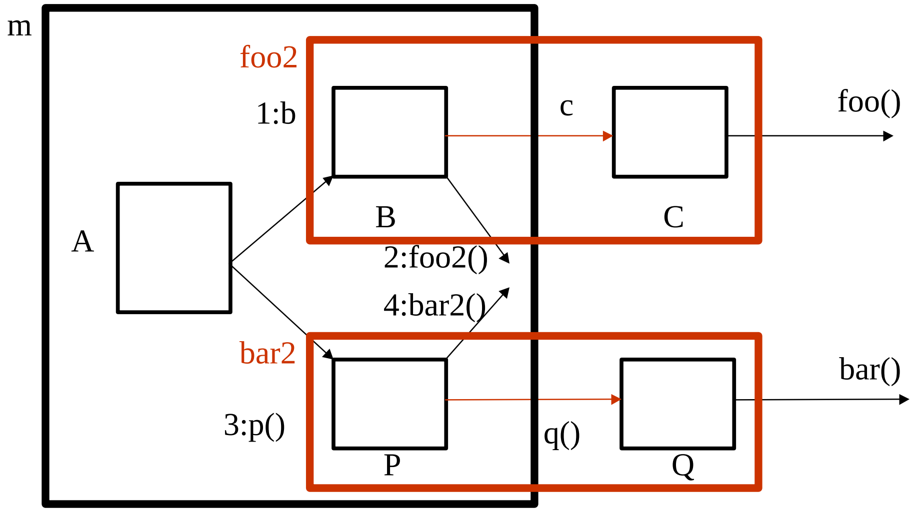
Application to OO

- Unit = method
  - closely related =
    - methods of class of `this/self` and other argument classes
    - methods of immediate part classes (classes that are return types of methods of class of `this/self`)
- In the following we talk about this application of the Law of Demeter Principle to OO: example follows in a few slides.

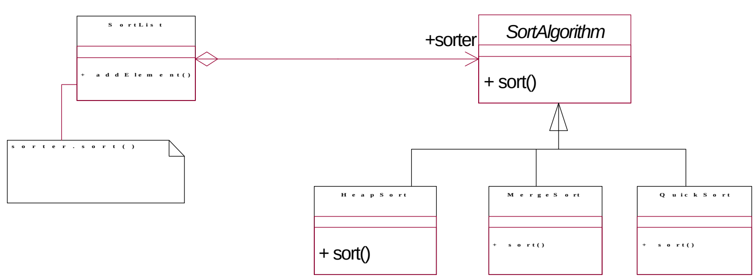
Violations: Dataflow Diagram



OO Following of LoD



Strategy - Example



Lecture 9: **Concurrency Conclusion**

- **Every** major operating system built since 1985 has provided threads -- Mach, OS/2, NT (Microsoft), Solaris (new OS from SUN), OSF (DEC Alphas). Why? Makes it easier to write concurrent programs, from Web servers, to databases, to embedded systems.
- **Moral: threads are cheap, but they're not free.**

Lecture 10: Deadlock

- Necessary conditions:
  - Limited access (for example: mutex or bounded buffer)
  - No preemption (if someone has resource, can't take it away)
  - Multiple independent requests -- "wait while holding"
  - Circular chain of requests

Solutions to Deadlock

- **Detect deadlock and fix**
- scan graph of threads and resources
- detect cycles
- fix them // this is the hard part!
  - Shoot thread, force it to give up resources.
  - Roll back actions of deadlocked threads (transactions)

Solutions to Deadlock

- **Preventing deadlock**
  - Need to get rid of one of the four conditions
  - **Banker's algorithm**:(request can be granted if some sequential ordering of threads is deadlock free)

Lecture 11: CPU Scheduling

- **Scheduling Policy Goals:**
  - **Minimize response time**
  - **Maximize throughput**: operations (or jobs) per second
  - **Fair**: share CPU among users in some equitable way

Scheduling Policies

- **FIFO**
- **Round Robin**
- **STCF**: shortest time to completion first.
- **SRTCF**: shortest remaining time to completion first. Preemptive version of STCF
- **Multilevel feedback**
- **Lottery scheduling (for fairness)**

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Lecture 12: Protection: Kernel and Address Spaces

- How is protection implemented?
- Hardware support:
  - address translation
  - dual mode operation: kernel vs. user mode

Lecture 13: Address Translation

- **Paging**
- Allocate physical memory in terms of fixed size chunks of memory, or **pages**.
- allows use of a bitmap.
- Operating system controls mapping: any page of virtual memory can go anywhere in physical memory.

Lecture 14: Caching and TLBs

- **Cache**: copy that can be accessed more quickly than original. Idea is: make frequent case efficient, infrequent path doesn't matter as much. Caching is a fundamental concept used in lots of places in computer systems. It underlies many of the techniques that are used today to make computers go fast

Caching

- **Translation Buffer, Translation Lookaside Buffer**:
  – hardware table of frequently used translations, to avoid havingto go through page table lookup in common case.
- Thrashing: cache contents tossed out even if still needed

Writes

- Two options:
  – **write-through**: update immediately sent through to next level in memory hierarchy
  – **write-back**: (delayed write-through) update kept until item is replaced from cache, then sent to next level.

Localities

- **Temporal locality**: will reference same locations as accessed in the recent past
- **Spatial locality**: will reference locations near those accessed in the recent past
- **When does caching break down?**
  – Whenever programs don’t exhibit enough spatial or temporal locality

Coordination aspect

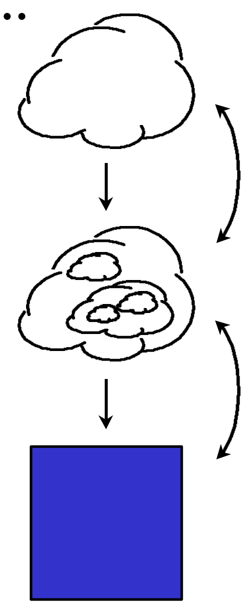
- Review of AOP
- Summary of threads in Java
- COOL (COOrdination Language)
  – Design decisions
  – Implementation at Xerox PARC

the goal is a clear separation of concerns

we want:

– natural decomposition
– concerns to be cleanly localized
– handling of them to be explicit
– in <u>both</u> design and implementation

achieving this requires...



• synergy among
  – problem structure and
  – design concepts and
  – language mechanisms

"natural design"

"the program looks like the design"

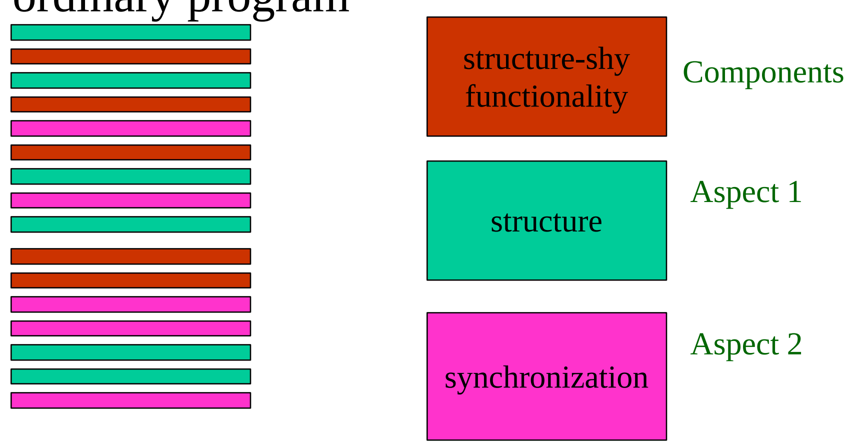
What is an aspect?

- An aspect is a modular unit that cross-cuts the structure of other modular units.
- An aspect is a unit that encapsulates state, behavior and behavior enhancements to other units.

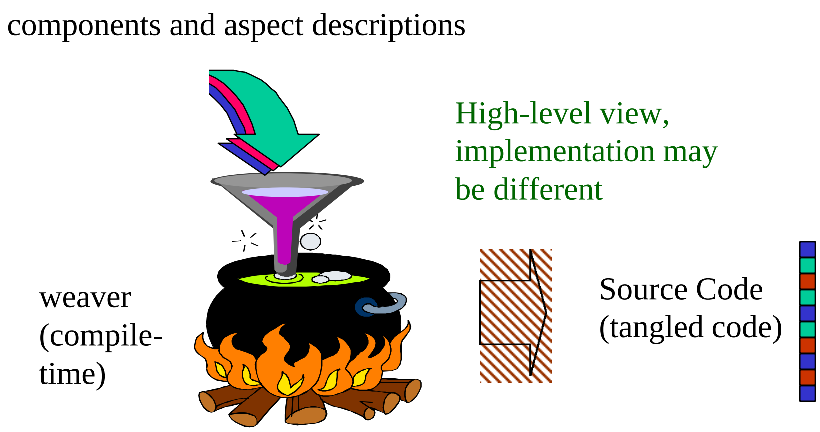
Cross-cutting of components and aspects

ordinary program

better program



Aspect-Oriented Programming



Coordination aspect

- Put coordination code about thread synchronization in one place.
- Threads are synchronized through methods.
- Method synchronization
  - Exclusion sets
  - Method managers

Problem with synchronization code: it is tangled with component code

```
class BoundedBuffer {
  Object[] array;
  int putPtr = 0, takePtr = 0;
  int usedSlots = 0;
  BoundedBuffer(int capacity){
    array = new Object[capacity];
  }
```

Tangling

```
synchronized void put(Object o) {
  while (usedSlots == array.length) {
    try { wait(); }
    catch (InterruptedException e) {};
  }
  array[putPtr] = o;
  putPtr = (putPtr +1 ) % array.length;
  if (usedSlots==0) notifyall();
  usedSlots++;
  // if (usedSlots++==0) notifyall();
}
```

Solution: tease apart basics and synchronization

- write core behavior of buffer
- write coordinator which deals with synchronization
- use weaver which combines them together
- simpler code
- replace `synchronized, wait, notify` and `notifyall` by coordinators

With coordinator: basics

```
BoundedBuffer {
public void put (Object o) (@
  array[putPtr] = o;
  putPtr = (putPtr+1)%array.length;
  usedSlots++; @)
public Object take() (@
  Object old = array[takePtr];
  array[takePtr] = null;
  takePtr = (takePtr+1)%array.length;
  usedSlots--;
  return old; @)
```

Using Demeter/COOL, put into *.cool file

Coordinator

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take}
  condition empty=true, full=false;
```

exclusion sets

coordinator variables

Coordinator

method managers with *requires* clauses and *entry/exit* clauses

```
  put requires (!full) {
    on exit {empty=false;
      if (usedSlots==array.length)
        full=true; }}
  take requires (!empty) {
    on exit {full=false;
      if (usedSlots==0)
        empty=true; }}
}
```

plain Java

```
public class Shape {
  protected double x_ = 0.0;
  protected double y_ = 0.0;
  protected double width_ = 0.0;
  protected double height_ = 0.0;

  double x() { return x_(); }
  double y() { return y_(); }
  double width(){
    return width_();
  }
  double height(){
   return height_();
  }
  void adjustLocation() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
  void adjustDimensions() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}
```

COOL Shape

```
coordinator Shape {
  selfex {adjustLocation,
          adjustDimensions}
  mutex {adjustLocation,x}
  mutex {adjustLocation,y}
  mutex {adjustDimensions,
          width}
  mutex {adjustDimensions,
          height}
}
```

Remaining Lectures

- See original notes

Some courses in Software Engineering Track

- Adaptive Object-Oriented Software Development (COM 3360)
- Object-Oriented Design (COM 3230, Professor Lorenz)
- Component-Based Programming (COM 3240, Professor Lorenz)

The End

- Nothing lasts …
- Everything arises and passes away

Hoping to see you in COM3360