CS162
Operating Systems and Systems Programming
Lecture 8

Tips for Working in a Project Team/ Cooperating Processes and Deadlock

February 20, 2008

Prof. Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Tips for Programming in a Project Team
- Discussion of Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

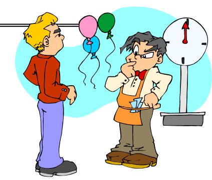
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Tips for Programming in a Project Team

"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    - » Empire state building finished in one year: staging iron production thousands of miles away
    - » Or the Hoover dam: built towns to hold workers
  - Is it OK to miss deadlines?
    - » We make it free (slip days)
    - » Reality: they're very expensive as time-to-market is one of the most important things!

Big Projects



- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)!
    - » This is why we bug you about starting the project early
    - » Had a grad student who used to say he just needed "10 minutes" to fix something. Two hours later…
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    - » Time reaches a minimum bound
    - » With complex interactions, time increases!
  - Mythical person-month problem:
    - » You estimate how long a project will take
    - » Starts to fall behind, so you add more people
    - » Project takes even more time!

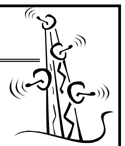
Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
  - Problem: Lots of communication across APIs
    - » If B changes the API, A may need to make changes
    - » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most CS162 project teams are functional, but people have had success with task-based divisions

Defensive Programming

- Like defensive driving, but for code:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Communication



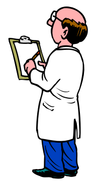
- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - » You would never be able to schedule meetings otherwise
  - Why do we require 4 people minimum?
    - » You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: phase 4, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document
  - Why Document?
    - » Expose decisions and communicate to others
    - » Easier to spot mistakes early
    - » Easier to estimate progress
  - What to document?
    - » Everything (but don't overwhelm people or no one will read)
  - Standardize!
    - » One programming format: variable naming conventions, tab indents,etc.
    - » Comments (Requires, effects, modifies)—javadoc?



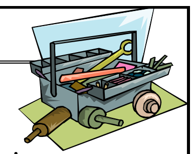
Suggested Documents for You to Maintain

- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Taming Complexity with Abstractions

- Break large, complex system into *independent* components
  - Goal: independently design, implement, and test each component
  - Added benefit: better security through isolation
  - But, components must work together in the final system
- We need interfaces (specs) between the components
  - The boundaries between components (and people)
  - To isolate them from one another
  - To ensure the final system actually works
- The interfaces must not change (much)!
  - Otherwise, development is not parallel

Use Software Tools



- Source revision control software
  - (CVS, Subversion, others…)
  - Easy to go back and see history/undo mistakes
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use CVS's features)
- Use automated testing tools
  - Write scripts for non-interactive software
  - Use "expect" for interactive software
  - JUnit: automate unit testing
  - Microsoft rebuilds the Vista kernel every night with the day's changes. Everyone is running/testing the latest software
- Use E-mail and instant messaging consistently to leave a history trail

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    - » Let's people test continuously, but more work
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
- Testing types:
  - Unit tests: check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?

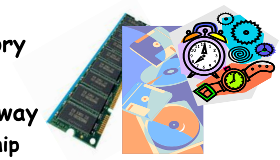
Administrivia

- Midterm I next week:
  - Wednesday, 2/27, 10 Evans 6-7:30pm
  - Closed book, no notes, no calculators/PDAs
  - Topics: Everything including today (lectures, book, readings, projects)
  - Email cs162 with conflicts (academic only)
- *No class on day of Midterm*
- I will post extra office hours for people who have questions about the material (or life, whatever)
- Midterm I review session Monday 2/25 after class
  - 120 Latimer, 6-7:30pm

Resource Contention and Deadlock

Resources

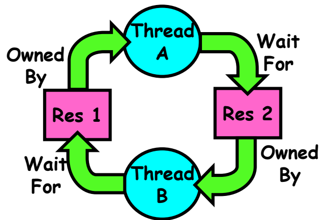
- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - » CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    - » Disk space, plotter, chunk of virtual address space
    - » Mutual exclusion – the right to enter a critical section



- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - » Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock $\Rightarrow$ Starvation but not vice versa
    - » Starvation can end (but doesn't have to)
    - » Deadlock can't end without external intervention

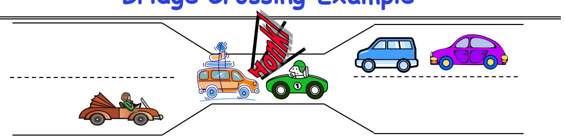
Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

Thread A	Thread B
`x.P();`	`y.P();`
`y.P();`	`x.P();`
`y.V();`	`x.V();`
`x.V();`	`y.V();`

  - Deadlock won't always happen with this code
    - » Have to have exactly the right timing ("wrong" timing?)
    - » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

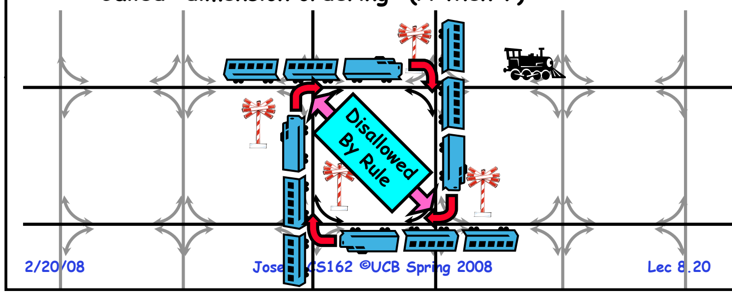
Bridge Crossing Example



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast $\Rightarrow$ no one goes west

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

Four requirements for Deadlock

- **Mutual exclusion**
  - Only one thread at a time can use a resource.
- **Hold and wait**
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption**
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- **Circular wait**
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

Safe and Unsafe States

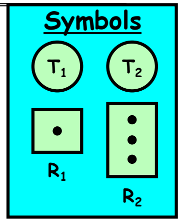
- *Safe state* – system will not enter a deadlock condition
- *Unsafe state* – system *may* enter a deadlock condition
- Being in an unsafe state does not guarantee that the system will deadlock
  - Thread requests A resulting in an unsafe state
  - Then it releases B which would prevent circular wait
  - The system is in an unsafe state, but not in deadlock

BREAK

Resource-Allocation Graph

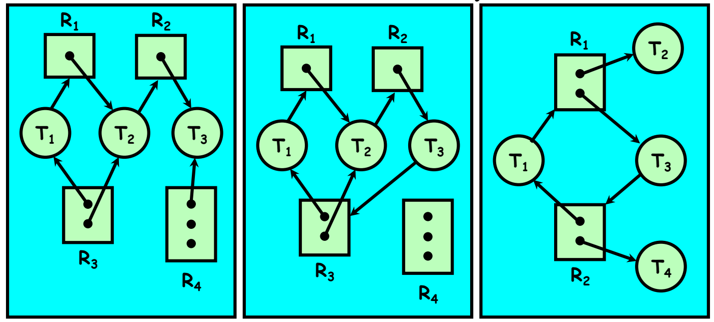
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - *CPU cycles, memory space, I/O devices*
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - » Request() / Use() / Release()
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Symbols



Resource Allocation Graph Examples

- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

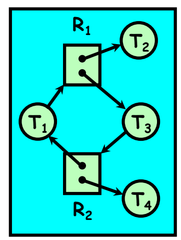
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

```
[FreeResources]:    Current free resources each type
[Request_X]:        Current requests from thread X
[Alloc_X]:          Current resources held by thread X
```

  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

Techniques for Preventing Deadlock (con't)

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max remaining that might be needed by any thread
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \le [Avail]$) for ($[Request_{node}] \le [Avail]$) Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker's Algorithm Example

- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Summary

- Suggestions for dealing with Project Partners
  - Start Early, Meet Often
  - Develop Good Organizational Plan, Document Everything, Use the right tools, Develop Comprehensive Testing Plan
  - (Oh, and add 2 years to every deadline!)
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress
- Next Time: Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this