CS 3204
Operating Systems

Lecture 12

Godmar Back

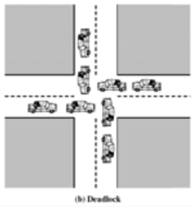
Announcements

- Project 2 due Oct 20
  - Should have initial steps in **3.2 Suggested Order of Implementation** by about now, but completed no later than Oct 8.
- Will return midterm on Thursday
- Office Hours this week:
  - Back on Wed 2pm-4pm
  - Ryan on Wed 4pm-6pm
  - No hours on Thursday

Deadlock

Deadlock (Definition)

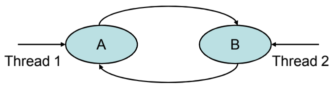
- A situation in which two or more threads or processes are blocked and cannot proceed
- “blocked” either on a resource request that can’t be granted, or waiting for an event that won’t occur
  - Possible causes: resource-related or communication-related
- Cannot easily back out



Deadlock Canonical Example (1)

```
pthread_mutex_t A;
pthread_mutex_t B;
…
pthread_mutex_lock(&A);
pthread_mutex_lock(&B);
…
pthread_mutex_unlock(&B);
pthread_mutex_unlock(&A);
```

```
pthread_mutex_lock(&B);
pthread_mutex_lock(&A);
…
pthread_mutex_unlock(&A);
pthread_mutex_unlock(&B);
```



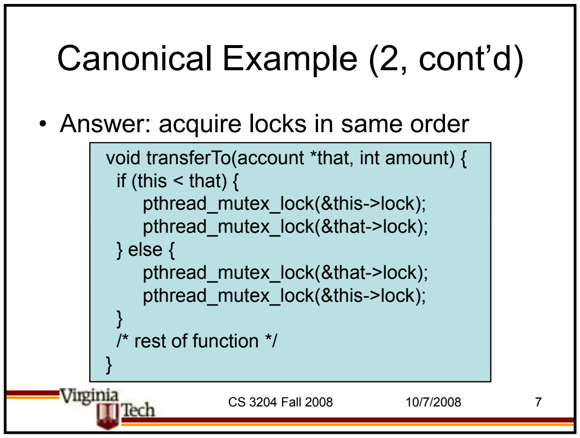
Canonical Example (2)

```
account acc1(10000, "acc1");
account acc2(10000, "acc2");

// Thread 1:
 for (int i = 0; i < 100000; i++)
     acc2.transferTo(&acc1, 20);
// Thread 2:
 for (int i = 0; i < 100000; i++)
     acc1.transferTo(&acc2, 20);
```

```
class account {
  pthread_mutex_t lock; // protects balance
  int balance;  const char *name;
public:
  account(int balance, const char *name) :
   balance(balance), name(name)  { pthread_mutex_init(&this->lock, NULL); }
  void transferTo(account *that, int amount) {
    pthread_mutex_lock(&this->lock);
    pthread_mutex_lock(&that->lock);
    cout << "Transferring $" << amount << " from "
         << this->name << " to " << that->name << endl;
    this->balance -= amount;
    that->balance += amount;
    pthread_mutex_unlock(&that->lock);
    pthread_mutex_unlock(&this->lock);
  }
};
```

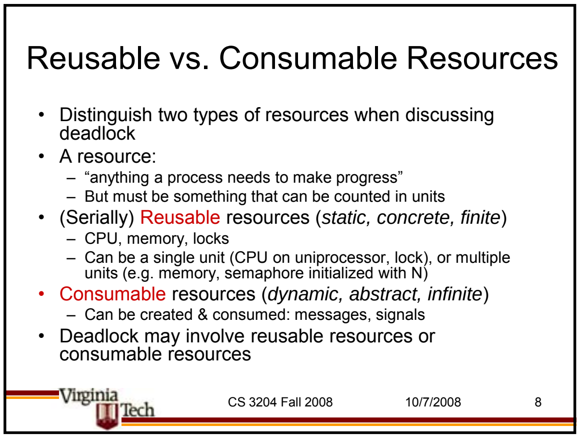
Q.: How to fix?



Canonical Example (2, cont'd)

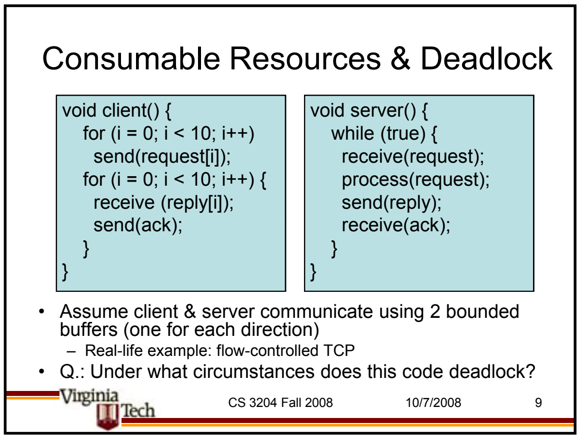
- Answer: acquire locks in same order

```
void transferTo(account *that, int amount) {
  if (this < that) {
       pthread_mutex_lock(&this->lock);
       pthread_mutex_lock(&that->lock);
  } else {
       pthread_mutex_lock(&that->lock);
       pthread_mutex_lock(&this->lock);
  }
  /* rest of function */
}
```



Reusable vs. Consumable Resources

- Distinguish two types of resources when discussing deadlock
- A resource:
  - "anything a process needs to make progress"
  - But must be something that can be counted in units
- (Serially) Reusable resources (*static, concrete, finite*)
  - CPU, memory, locks
  - Can be a single unit (CPU on uniprocessor, lock), or multiple units (e.g. memory, semaphore initialized with N)
- Consumable resources (*dynamic, abstract, infinite*)
  - Can be created & consumed: messages, signals
- Deadlock may involve reusable resources or consumable resources

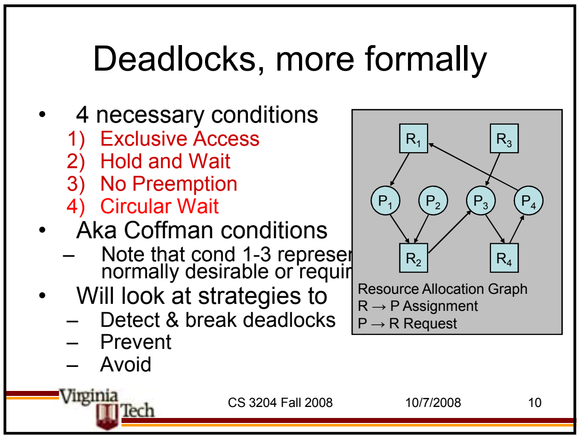


Consumable Resources & Deadlock

```
void client() {
    for (i = 0; i < 10; i++)
      send(request[i]);
    for (i = 0; i < 10; i++) {
      receive (reply[i]);
      send(ack);
    }
}
```

```
void server() {
    while (true) {
      receive(request);
      process(request);
      send(reply);
      receive(ack);
    }
}
```

- Assume client & server communicate using 2 bounded buffers (one for each direction)
  - Real-life example: flow-controlled TCP
- Q.: Under what circumstances does this code deadlock?

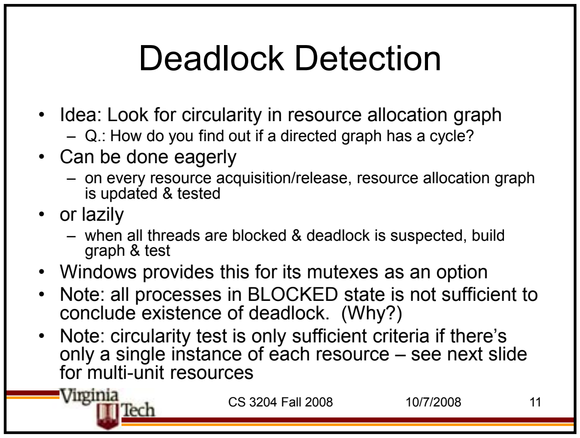


Deadlocks, more formally

- 4 necessary conditions
  1) Exclusive Access
  2) Hold and Wait
  3) No Preemption
  4) Circular Wait
- Aka Coffman conditions
  - Note that cond 1-3 represer normally desirable or requir
- Will look at strategies to
  - Detect & break deadlocks
  - Prevent
  - Avoid

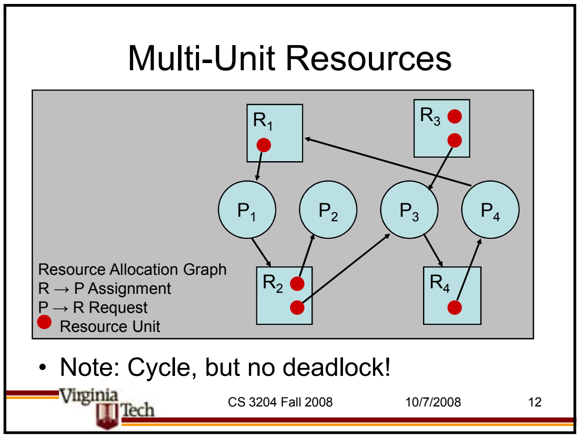
R_1
R_3
P_1
P_2
P_3
P_4
R_2
R_4

Resource Allocation Graph
R → P Assignment
P → R Request



Deadlock Detection

- Idea: Look for circularity in resource allocation graph
  - Q.: How do you find out if a directed graph has a cycle?
- Can be done eagerly
  - on every resource acquisition/release, resource allocation graph is updated & tested
- or lazily
  - when all threads are blocked & deadlock is suspected, build graph & test
- Windows provides this for its mutexes as an option
- Note: all processes in BLOCKED state is not sufficient to conclude existence of deadlock. (Why?)
- Note: circularity test is only sufficient criteria if there's only a single instance of each resource – see next slide for multi-unit resources



Multi-Unit Resources

Resource Allocation Graph
R → P Assignment
P → R Request
● Resource Unit

- Note: Cycle, but no deadlock!

Deadlock Detection

- For reusable resources
  - If each resource has exactly one unit, deadlock iff cycle
  - If each resource has multiple units, existence of cycle may or may not mean deadlock
    - Must use reduction algorithm to determine if deadlock exists (Intuition: remove processes that don't have request edges, return their resource units and remove assignment edges, assign resources to remove request edges, repeat until out of processes without request edges. – If entire graph reduces to empty graph, no deadlock.)
- For consumable resources
  - analog algorithm possible
- Q.: What to do once deadlock is detected?

Deadlock Recovery

Increasing Severity

- Preempt resources (if possible)
- Back processes up to a checkpoint
  - Requires checkpointing or transactions (typically expensive)
- Kill processes involved until deadlock is resolved
- Kill all processes involved
- Reboot

Task Manager Warning

WARNING: Terminating a process can cause undesired results including loss of data and system instability. The process will not be given the chance to save its state or data before it is terminated. Are you sure you want to terminate the process?

Killing Threads or Processes

- Can be difficult issue:
  - When is it safe to kill a thread/process?
- Consider:

```
thread_func()
{
  while (!done) {
    lock_acquire(&lock);
    // access shared state
    lock_release(&lock);
  }
}
```

What if thread is killed here?

```
thread_func()
{
  while (!done) {
    lock_acquire(&lock);
    p = queue1.get();
    queue2.put(p);
    lock_release(&lock);
  }
}
```

- Must guarantee
  - full resource reclamation (e.g., locks held must be unlocked)
  - consistency of all surviving system data structures (e.g., no packets dropped)

Safe Termination

- System code must make sure that no shared data structures are left in an inconsistent state when thread is terminated
  - Same assurance must hold even if thread terminates itself (as in Pintos project 2)
- General strategy:
  - Allow termination of user processes at any point in time
    - Note: this does not protect user's data structures
  - Restrict kernel code to certain termination points (where process checks if termination request is pending):
    - E.g., after schedule(); before returning from syscall
    - Protects kernel's data structures

Deadlock Prevention (1)

- Idea: remove one of the necessary conditions!
- (C1) (Don't require) Exclusive Access
  - Duplicate resource or make it shareable (where possible)
- (C2) (Avoid) Hold and Wait
  - a) Request all resources at once
    - hard to know in modular system
  - b) Drop all resources if additional request cannot be immediately granted – retry later
    - requires "try_lock" facility
    - can be inefficient if lots of retries

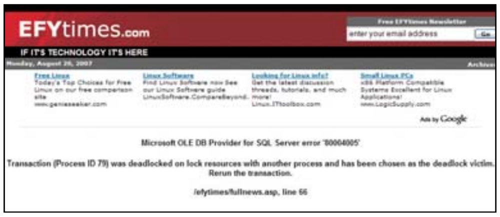
Deadlock Prevention (2)

- (C3) (Allow) Preemption
  - Take resource away from process
    - Difficult: how should process react?
  - Virtualize resource so it can be taken away
    - Requires saving & restoring resource's state
- (C4) (Avoid) Circular Wait
  - Use partial ordering
    - Requires mapping to domain that provides an ordering function (addresses often work!)

Deadlock Avoidance

- Don't grant resource request if deadlock could occur in future
  - Or don't admit process at all
- Banker's Algorithm (Dijkstra 1965, see book)
  - Avoids "unsafe" states that might lead to deadlock
  - Need to know what future resource demands are ("credit lines" of all customers)
  - Need to capture all dependencies (no additional synchronization requirements – "loans" can be called back if needed)
- Mainly theoretical
  - Impractical assumptions
  - Tends to be overly conservative – inefficient use of resources

Deadlock In The Real World



EFYtimes.com

Free EFYtimes Newsletter

enter your email address Go

IF IT'S TECHNOLOGY IT'S HERE

Monday, August 20, 2007 Archives

Free Linux
Today's Top Choices for Free Linux on our free comparison site
www.geniesseeker.com

Linux Software
Find Linux Software now See our Linux Software guide
LinuxSoftware.CompareBeyond.

Looking for Linux info?
Get the latest discussion threads, tutorials, and much more!
Linux.ITtoolbox.com

Small Linux PCs
x86 Platform Compatible Systems Excellent for Linux Applications!
www.LogicSupply.com

Ads by Google

Microsoft OLE DB Provider for SQL Server error '80004005'

Transaction (Process ID 79) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

/efytimes/fullnews.asp, line 66

Deadlock in the Real World

- Most common strategy of handling deadlock
  - Test: fix all deadlocks detected during testing
  - Deploy: if deadlock happens, kill and rerun (easy!)
    - If it happens too often, or reproducibly, add deadlock detection code
- Weigh cost of preventing vs cost of (re-) occurring
- Static analysis tools detects some kinds of deadlocks before they occur
  - Example: Microsoft Driver Verifier
  - Idea: monitor order in which locks are taken, flag if not consistent lock order

Deadlock vs. Starvation

- Deadlock:
  - No matter which policy the scheduler chooses, there is no possible way for processes to make forward progress
- Starvation:
  - There is a possible way in which threads can make possible forward progress, but the scheduler doesn't choose it
    - Example: strict priority scheduler will never scheduler lower priority threads as long as higher-priority thread is READY
    - Example: naïve reader/writer lock: starvation may occur by "bad luck"

Informal uses of term `deadlock'

- 4 Coffman conditions apply specifically to deadlock with definable resources
- Term deadlock is sometimes informally used to also describe situations in which not all 4 criteria apply
  - See interesting discussion in Levine 2003, Defining Deadlock
  - Consider: *When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.*
  - Does it meet the 4 conditions?
- However, even under informal/extended view, not all "lack of visible progress" situations can reasonably be called deadlocked
  - e.g., an idle system is not usually considered deadlocked

Summary

- Deadlock:
  - 4 necessary conditions: mutual exclusion, hold-and-wait, no preemption, circular wait
- Strategies to deal with:
  - Detect & recover
  - Prevention: remove one of 4 necessary conditions
  - Avoidance: if you can't do that, avoid deadlock by being conservative