

Announcements

- ✘ No class on Friday, July 4th - holiday!
(also no office hours, no project session)
- ✘ Project sessions: not every week
 - one for each project
 - other help? newsgroup, office hours
- ✘ Project #1 due next Tuesday 10pm

Questions on material so far?

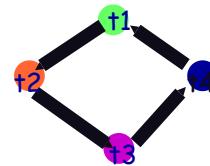
- ✘ Processes and threads
 - what they are? address space? CPU sharing? process switch vs. thread switch vs. function call?
- ✘ Concurrency/parallelism
 - why/how race conditions arise?
 - how to solve them? atomicity! what/why?
- ✘ Synchronization
 - how to achieve atomicity on uniprocessor, multiprocessor
 - synchronization primitives: lock, semaphore, monitor
 - producer/consumer problem & synchronization

Past & Present

- ✘ Have looked at two constraints:
 - mutual exclusion constraint between two events is a requirement that they not overlap in time**
enforced using scheduling, locks, semaphores, monitors
 - precedence constraint between two events is a requirement that one completes before the other**
(usually) enforced using scheduling or semaphores
- ✘ Synchronization primitive ordering:
 - atomic instructions can implement locks, locks can implement semaphores (lock + integer counter) or monitors (one implicit lock), and vice versa (of course)
- ✘ Today:
 - Deadlock: what to do when many threads and no progress**

Deadlock: parallelism's pox

- ✘ Graphically, caused by a directed cycle in inter-thread dependencies
 - e.g., T1 holds resource R1 and is waiting on R2, T2 holds R2 and is waiting on R1



No progress possible.
Even simple cases can be non-trivial to diagnose.

Deadlock example

- ✘ Given two threads, what sequence of calls cause the following to deadlock?
- ```

/* transfer x dollars from a to b */
void transfer(account *a, account *b, int x)
! P(a->sema);
! P(b->sema);
! a->balance += x;
! b->balance -= x;
! V(a->sema);
! V(b->sema);

```
- t1

transfer(?,?)

t2

transfer(?,?)


## Deadlock generalized

- ✘ Does deadlock require locks? No. Just circular constraints.
- Example:** consider two threads that send and receive data to each other using two circular buffers (buf size = 4096 bytes). Recall: a full buffer causes the sender to block until the receiver removes data.
- |                                                                                                        |                                                                                                                  |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <p>T1:</p> <pre> send n bytes to T2 while(receive data)   display data exit                     </pre> | <p>T2:</p> <pre> while(receive 4K of data)   process data   send 4K result to T1 exit                     </pre> |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
- What size of n will cause this to break?

## Deadlock Conditions: Need all four

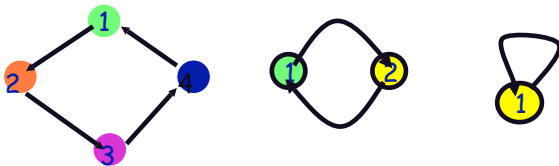
- ◆ Limited access:  
Resource can only be shared with finite users.
- ◆ No preemption:  
once resource granted, cannot be taken away.
- ◆ Multiple independent requests (hold and wait):  
don't ask all at once (wait for next resource while holding current one)
- ◆ Circularity in graph of requests
- ◆ Two approaches to dealing with deadlock:  
pro-active: prevention  
reactive: detection + corrective action  
(inactive: do nothing?)

## Deadlock Prevention: Eliminate 1 condition

- ◆ Limited access:  
Buy more resources, split into pieces, or virtualize to make "infinite" copies
- ◆ Non-preemption: create copies or virtualize  
Threads: threads have copy of registers = no lock  
Physical memory: virtualized with VM, can take physical page away and give to another process!
- ◆ Hold + wait: acquire resources "all at once"  
(wait on many without locking any, must know all needed)
- ◆ Circularity:   
Single lock for entire system: (problems?)  
Partial ordering of resources (next)

## Partial orders: simple deadlock control

- ◆ Order resources (lock1, lock2, ...)
- ◆ Acquire resources in strictly increasing (or decreasing) order
- ◆ Intuition:  
number all nodes in graph  
to form a cycle there has to be at least one edge from high to low number and low to high (or to same node)



## Two phase locking: simple deadlock control


- ◆ Acquire all resources, if block on any, release all, and retry  

```
print_file:
 lock(file);
 acquire printer
 acquire disk;
 - do work -
 release all
```
- ◆ Pro: dynamic, simple, flexible
- ◆ Con:  
cost with number of resources?  
length of critical section?  
Abstraction breaking: hard to know what's needed a priori

## Detection + correction

- ◆ Terminate threads and release resources  
repeat until deadlock goes away  
Con: Blowing away threads leaves system in what state?  
Wild guess: we don't know, but it's probably gross.  
Stylized use: acquire all locks, then modify state. Can always blow away. (Basically two-phase w/out explicit thread release of resources)
- ◆ More fancy: roll back actions of deadlocked threads  
acquire locks however  
only modify state using invertible (undoable) actions  
get stuck? System kills thread ("bad thread") and inverts actions. Repeat as necessary.  
"transactions" = very easy for programmer  
problem: tracking actions, constructing inverses, scalability/forward progress?

## Dirty secret: the most common schemes

- ◆ Prevention: Test  
pro: no complex machinery. Everyone understands testing.  
Con: interleavings = huge space.
- ◆ Kill app  
throw deadlock in the same box as infinite loops. Do what you usually do.  
  
Pro: works for some applications (which?) Just rerun.  
Con: works for some applications (which?).

## Synchronization in the real world

---

- ◆ Synchronization whenever > 1 user of resource
  - Use same sol'ns in real world: lock (on door), scheduling (appointments), duplicate resource (everyone has laptop)
  - Some differences: vision & physics (mass)
- ◆ Examples:
  - Contagious disease race conditions
  - One road, multiple cars: traffic lights (scheduling-based synchronization), two lanes ("duplicate" state -- trade less utilization for simpler coordination)
  - Bathroom: door (lock), men's/women's (duplicate state)
  - You & partner: lock = "hacking synch.cc" unlock = "done"
  - Parking space: use a physical object (car in space), explicit parking assignment (lock always: no concurrency = bad utilization) or permit ("sort of" lock that allows concurrency)

## Concurrency Summary

---

- ◆ Concurrency errors:
  - one way to view: thread checks condition(s)/examines value(s) and continues with the implicit assumption that this result still holds while another thread modifies.
- ◆ Fixes?
  - Rule 1: don't do concurrency (poor utilization or impossible)
  - Rule 2: don't share state (may be impossible)
  - Rule 3: if you violate 1 & 2 use one big lock (coarse-grain)
- Worst: many locks (fine-grain: good parallelism but error prone; typical solution for OS and parallel programs; also may be hard to debug *performance* "bugs").