CIS 505 – Software Systems

Notes for 1/26/2012

Matt Blaze

Our problem:

- Resources (such as memory) shared by > 1 thread
  - things can get amazingly (and subtly) screwed up because of, among other things, *race conditions*
    - strange answers, deadlocks, etc
- So we need some way to “keep order”
  - mutual exclusion
  - thread synchronization
- Maybe we can solve in the OS, or maybe in the applications, or maybe both

So far:

- Solution 1: Disable interrupts
  - big hammer
- Solution 2: busy wait with shared variables
  - 2a: Peterson's solution
    - tricky (3 variables), but requires no hardware support
    - works on virtually any conventional processor
  - 2b: TSL
    - simpler, but assumes TSL (or equiv) instruction
    - works even on multiprocessors
  - Can be generalized to > 2 threads

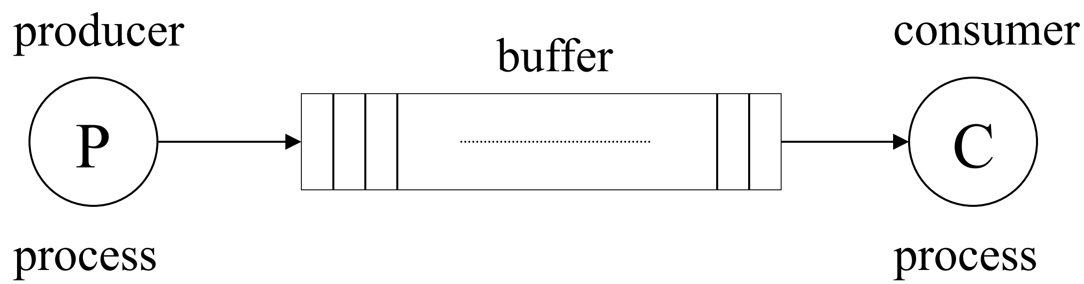
Limitations of Peterson/TSL

- Busy waiting!
  - "are we there yet?" in a tight loop
  - that is, the waiting thread doesn't *block*
- Designed specifically for mutual exclusion
  - you need other stuff to do general synchronization on top of them
- But, at least, they're very general
  - no help from OS, so useful *inside* OS

Approach 3: Sleep and Wakeup

- Thread blocking and unblocking synchronization
- Two abstract system calls (*not* actually what they're called in Unix systems!):
  - sleep()
    - blocks until someone calls wakeup
  - wakeup(process)
    - unblocks process, which should be blocked in sleep
- Remember the producer-consumer problem?
  - this helps solve the full / empty buffer problem
  - problem here is *synchronization*, not just mutual exclusion

Producer/Consumer Problems



- from time to time, the producer places an item in the buffer
- the consumer removes an item from the buffer
- careful synchronization required (they run simultaneously)
- the consumer must wait if the buffer empty
- the producer must wait if the buffer full
- typical solution would involve a shared variable count
- also known as the Bounded Buffer problem
- Example: in UNIX shell

  cat myfile.txt | lpr

Producer/Consumer (partial solution)

- Producer:

```
while (TRUE) {
  produce X…
  if (count == N)
          sleep;
  add_to_buffer(X)
  count = count + 1;
  /* need above to be atomic */
  if (count == 1)
     wakeup(Consumer);
}
```

- Consumer:

```
while (TRUE) {
  if (count==0)
     sleep;
 remove_from_buffer(X)
  count = count -1;
  /* need above to be atomic… */
  if (count == N-1)
      wakeup(Producer);
  consume X…
}
```

Doesn’t quite work

- Count is initially 0
  - Consumer reads the count
- Producer produces the item, inserts it, and increments count (to 1)
- Producer executes wakeup, but there is no waiting consumer (at this point)
- Now consumer continues its execution and goes to sleep
- Consumer stays blocked forever
  - Main problem: wakeup was lost

A new mechanism: Semaphores (Dijkstra)

- A semaphore **S** has a non-negative integer value
- Two operations
  - **up(S)** (aka V(S)) : increments the value of S
  - **down(S)** (aka P(S)) : decrements the value of S if S is positive, else makes the calling thread/process wait
- When S==0, down(S) moves the thread to sleep (blocked) state
  - no busy waiting
- If S==0, up(S) also wakes up one sleeping process (if there are any)
  - uses an internal list of sleeping processes
- up and down calls must be *atomic* actions

Can we do mutual exclusion with Semaphores?

- Yep, we sure can:
  - Set semaphore S with initial value of 1
  - To enter critical section
    - down(S)
  - To exit critical section
    - up(S)
- That’s it.
  - Does this really work? Why?

Neat tricks with Semaphores

- Simplest semaphore examples are binary (semaphore value S is either 1 or 0),
  - used here as an “improved” (blocking) TSL
  - to get strict mutual exclusion, set initial S=1
    - maximum number in critical section = 1
- Initial values of S > 1 can be used to allow an arbitrary maximum number of threads to be active somewhere
  - other applications besides mutual exclusion (e.g., resource load control)

Basic Semaphore Implementation

```
typedef struct {
     int value;
     *pid_t wait_list; /* list of processes *;
       } semaphore;

down(semaphore S){
    if (S.value >0) S.value--;
    else {  add this process to S.wait_list;
       sleep;
          }

up(semaphore S){
    if (S.wait_list==null) S.value++;
    else {   select and remove a process P from S.wait_list;
            wakeup(P);
          }
```

up and **down** have to be protected as critical sections

Producer-Consumer, with semaphores

```
semaphore:  full = 0      /* number of full slots */
            empty = SLOTS   /* number of empty slots */
            mutex = 1   /* critical section (binary) */
```

Producer code:

```
while (TRUE) {
  /* produce */
  down (empty)
  down (mutex)
  /* add to buffer */
  up (mutex)
  up (full)
}
```

Consumer code:

```
while (TRUE) {
  down (full)
  down (mutex)
  /* remove from buffer */
  up (mutex)
  up (empty)
  /* consume */
}
```

Mutual exclusion
For accessing buffer

Producer-Consumer does it work yet?

```
semaphore:  full = 0      /* number of full slots */
            empty = SLOTS /* number of empty slots */
            mutex = 1     /* critical section (binary) */
```

Producer code:

```
while (TRUE) {
   /* produce */
   down (empty)
   down (mutex)
  /* add to buffer */
   up (mutex)
   up (full)
}
```

Consumer code:

```
while (TRUE) {
   down (full)
   down (mutex)
  /* remove from buffer */
   up (mutex)
   up (empty)
   /* consume */
}
```

What happens if we switch the order of down(empty) and down(mutex) ?
What happens if we switch the order of up(mutex) and up(full) ?

Making it work in practice: semaphores in real systems

Semaphores in actual UNIX/POSIX systems

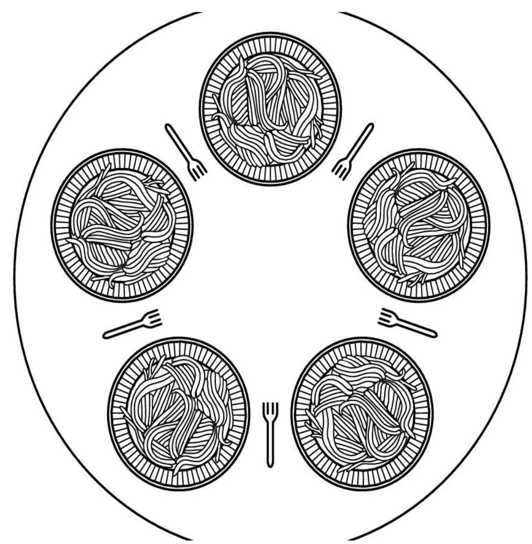
- There aren't system calls called *up* or *down*
  - naturally, it's more complicated than that
  - semaphores have traditionally been an internal feature of the UNIX kernel, but not available to user processes
- Instead, there's a family of POSIX standard system calls that implement semaphores
  - one data type: **`sem_t data`**
  - five system calls: **`sem_init`**, **`sem_destroy`**, **`sem_wait`**, **`sem_trywait`**, **`sem_post`**

POSIX Semaphore Syscalls

- *sem_t:* data type for semiphores
- *int sem_init(sem_t *sp, unsigned int count, int type):* Initialize semaphore pointed to by sp to count. (*type* controls details of semaphore behavior)
- *int sem_destroy(sem_t *sp):* destroys any state related to the semaphore pointed to by sp.
- *int sem_wait(sem_t *sp):* blocks the calling thread until the semaphore count pointed to by sp is greater than zero, and then it atomically decrements the count.
- *int sem_trywait(sem_t *sp):* atomically decrements the semaphore count pointed to by sp if it is greater than zero; otherwise, it returns an error.
- *int sem_post(sem_t *sp);* atomically increments the semaphore count pointed to by sp. If there are any threads blocked on the semaphore, one is unblocked.

The infamous Dining Philosophers…

- Assumptions:
  – philosophers can only afford to eat rice
  – worse, they only have one chopstick each!
    - but need two to eat
  – alternate between two states:
    - complaining (or looking for work)
    - trying to eat
- Sadistic computer scientist arranges them in circle:



Dining Philosophers?

- Philosophers have good table manners
  - must acquire two chopsticks to eat
- Useful abstraction / “standard problem”:
  - soundness
    - every chopstick is held by <= 1 philosopher at a time
  - deadlock-freedom
    - no state where no one gets to eat
  - starvation-freedom
    - solution guarantees that *all* philosophers occasionally eat

First try at a solution

- Obvious solution is to represent each chopstick by a semaphore (semaphore chopstick[N]).
- Down before picking it up & up after using it.

```
 while (TRUE) {  /* philosopher i loop */
    DOWN(chopstick[i]);
    DOWN(chopstick[i+1 mod N]);
    eat();
    UP(chopstick[i]);
    UP(chopstick[i+1 mod N]);
    lookforjob()
    complain();
}
```

Did that work?

- No

What was wrong?

- Solution is *sound*
  – the semaphores indeed guarantee that every chopstick is in use by <= 1 philosopher
- But vulnerable to *deadlock*
  – Bad scenario: everyone runs in lockstep
  – first everyone picks up their chopstick[i]
    - that works
  – then everyone tries to get chopstick[i+1 mod N]
    - but that's someone else's chopstick[i], which is in use
    - so they all block
  – kaboom. deadlock.

Possible solutions (do they work?)

- ? Make get chopstick[i] and get chopstick[i+1 mod N] atomic by surrounding with a binary ("mutex") semaphore ?
- ? Reverse the order of get chopstick[i] and get chopstick[i+1 mod N] for every other philosopher ?
- ? Forget the chopstick semaphores ?
  - ? protect eat() with mutex semaphore ?

A better solution

- Add a binary mutex semaphore
- Each philosopher in one of three states
  - *hungry (*wants to eat*)*, *eating* (holding 2 chopsticks) or *complaining*
- To eat, add this code:
  - down(mutex)
  - state[i]=hungry
  - state[i]=eating if neither neighbor is eating
  - up(mutex)
- To finish eating, if either neighbor is hungry, up() their chopstick
- See Tannenbaum figure 2-33 for details

Some “standard” problems (learn them!)

- See Tanenbaum, *Modern Operating Systems*, ch 2
- Dining philosophers
- Readers and Writers
  - access to a shared database
  - deadlocks, starvation, consistency
- Sleeping Barber
  - one barber, one chair, n chairs in waiting area
    - barber sleeps when no customers
    - customers arrive asynchronously
  - queuing, race conditions, starvation, deadlock

Mutual Exclusion Toolkit

- Solution 1: Disable interrupts
  - big hammer
- Solution 2: busy wait with shared variables
  - 2a: Peterson’s solution
    - complex but no hardware support required
    - works on virtually all uniprocessors
  - 2b: TSL
    - simpler but assumes TSL (or equiv) instruction
    - works even on multiprocessors
- Solution 3: blocking system calls
  - 3a sleep/wakeup
    - simple; useful building block
    - “lost wakeup problem”
  - 3b Semaphores
    - generalized abstraction
    - can solve lost wakeup problem
  - can be built atop interrupt disabling and busy waiting inside OS

OK, new topic:

Deadlocks

Deadlocks and Resource Allocation

- The deadlock-vulnerable environment
  - finite *resources*, each of which can be assigned to a limited number of processes/threads (usually at most one)
  - processes can request multiple resources
    - several at once or one at a time
    - allowed to accumulate more while holding
    - block until requested resource is available
  - processes can release resources when finished with them
- A *deadlock* is a situation where there is a set of blocked processes waiting for resources such that there is no way to satisfy any of their requests
  - even if all the *unblocked* processes release theirs
- We've already encountered deadlocks
  - dining philosophers, etc

Our (not-so-great) deadlock toolkit so far

- Approach 1: Ignore problem
- Approach 2: Prevention
  - 2a: Exhaustive Search of Possible States
  - 2b: Ad Hoc Genius
- Approach 3: Detection
  - 3a: Ad Hoc Genius
- We'll be focusing on Prevention, Detection
  - making our toolkit more systematic
- Also a new idea: Deadlock *Avoidance*

Approach 1: Ignore Problem

- Deadlocks aren't always so bad
  - sometimes we can just live with the possibility
  - when things seem to stop, just start over
  - fine for many applications
- But unfortunately, sometimes they *are* bad
  - inside the OS, databases, transaction systems
- Usual name: "Ostrich Algorithm"
  - "stick head in sand"
  - (but real ostriches aren't actually that dumb)

Approach 2: Deadlock Prevention

- 2a: Ad hoc approach
  - uses ingenious insight to notice potential deadlocks
  - works well only for very smart people
  - example: dining philosophers
- 2b: Exhaustive search approach
  - go through every possible state, look for deadlocks
  - exponential, gets too much to deal with quickly
- We'd prefer something more systematic

Approach 3: Deadlock Detection & Recovery

- Ad hoc approach
  - notice that a deadlock has occurred, then wildly panic
    - start killing off processes until things start to move again
  - have to be careful that killed off processes can be rolled back / restarted safely
  - may have to reboot OS
- We'd prefer something more systematic

Approach 2, redux: Deadlock Prevention

(a more systematic approach)

The four preconditions for deadlock

- #1 *Mutual Exclusion*
  - resources being held by one process can’t simultaneously be used by another
- #2 *No resource preemption*
  - holders keep resources until relinquished
- #3 *Hold and wait*
  - resource holders can “accumulate” (request additional resources and block waiting for them)
- #4 *Circular waiting*
  - circular chain of two (or more) processes/threads
  - each waiting for resource held by next in chain

The *entire secret* to deadlock prevention:

- Just eliminate *any one* of the four preconditions
  - doesn’t matter which one!
  - if you do this, deadlock can’t possibly happen, ever!
  - problem solved!
- That’s all there is to it!
  - but… how might we do this?

Removing Precondition 1: *Mutual Exclusion*

- Mutual Exclusion: Resources held by one process can’t simultaneously be used by another
- This is hard to avoid -- usually you either need mutual exclusion or you don’t
- Sometimes mutex can be avoided by making resource use effectively “atomic”
  - use resource in a burst of exclusive access
  - spooling

Removing Precondition 2: *No Resource Preemption*

- No Resource Preemption: Resource holders keep their resources until they relinquish them
- Maybe can eliminate with some kind of policy
  - e.g., if high priority process wants a resource, it gets it and the resource is taken away from lower priority
- But no good way to implement this in practice
  - complex recovery when resource is taken away

Removing Precondition 3: *Hold and Wait*

- Hold and Wait: Resources can be accumulated (requested and blocked for while holding others)
- Maybe can eliminate by requiring complete set of resources to be requested together
  - must release all to request new set
- Inefficient
  - and what about resources whose identity can't be determined in advance (e.g., file names)?

Eliminating Hold-and-Wait: Two-Phase Locking

- Processes manage resources in two phases
  - locking
    - attempt to acquire locks
    - if any attempt would block, release and start over
  - using
    - use any resource only after all locks acquired
    - release locks when finished
- Safe, but still inefficient for many applications
  - similar to spin locks
  - used occasionally in database

Removing Precondition 4: *Circular Waiting*

- Circular Waiting: circular chain of processes, each waiting for resource held by next process in chain
- May be able arrange requests to break cycles
  - this better work, it’s the only prevention idea left
- Central idea: **Hierarchical Allocation**
  - number resources; only allow obtaining resources with higher numbers than those currently held
  - requires finding a suitable numbering

Hierarchical Allocation and Dining Philosophers

- Obvious solution is to represent each chopstick by a semaphore (semaphore chopstick[N]).
- Down before picking it up & up after using it.

```
while (TRUE) {  /* philosopher i loop */
    DOWN(chopstick[i]);
    DOWN(chopstick[i+1 mod N]);
    eat();
    UP(chopstick[i]);
    UP(chopstick[i+1 mod N]);
    lookforwork()
    complain();
}
```

Now, what about this…

- ? Reverse the order of get chopstick[i] and get chopstick[i+1 mod N] for every other philosopher ?
  - what does this do?

or….

- if i+1 mod N == 0 , get chopsticks in reverse order
  - fixes deadlock (not starvation, though)

Bottom line on Deadlock Prevention

- It sounds so easy!
  - you only have to eliminate any *one* of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard
  - *Hold-and-Wait* (#3) can sometimes be eliminated with two-phase locking
  - *Circular waiting* (#4) can sometimes be eliminated in practice through Hierarchical Allocation

Our improved deadlock toolkit (so far)

- Approach 1: Ignore problem
- Approach 2: Prevention
  - 2a: Exhaustive Search of Possible States
  - 2b: Ad Hoc Genius
  - **2c: New! Two-Phase Locking**
  - **2d: New! Hierarchical Allocation**
- Approach 3: Detection and Recovery
  - 3a: Ad Hoc Genius
  - **3b: ??? coming up next ???**
- Approach 4, also still to come: Deadlock *Avoidance*

Approach 3 Revisited: Deadlock Detection/Recovery

(a more systematic approach)

Deadlock Detection

- Even if you can't prevent deadlocks, perhaps you can systematically *detect* them
  - should be coupled with a systematic recovery scheme
- So what we need are
  - algorithms for detecting deadlocks
    - graph theory to the rescue
  - a way of "undoing" what we did before the deadlock occurred
    - "rollback"

Are there good deadlock detection algorithms?

- Basic idea: represent resource allocation as a graph, look for cycles
  - no cycles == no deadlocks
  - cycles == deadlocks
- Cycle detection! Sounds like a typical graph theory problem
  - can this be done efficiently?
- Also a systems problem
  - when to check?

Detecting Deadlocks (simple case)

- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Solution: Build a graph: Resource Allocation Graph (RAG)
  - There is a node for every process and a node for every resource
  - If process P currently has resource R, then put an edge from R to P
  - If process P is requesting resource R, then put an edge from P to R
- There is a deadlock if and only if RAG has a cycle

Resource Allocation Graph Cycle Detection Algorithms

- Can be generalized to multiple instances of resources
- A little slow (quadratic) but still often feasible
- Still leaves us with the problem of *recovery*…

After Detection: Deadlock Recovery

- Preemption
  - Take away a resource from current owner
  - Frequently impossible
- Rollback
  - Checkpointing periodically to save states
  - Reset to earlier state before acquiring resource
  - Used in database systems
- “Soprano’s” Algorithm: Manual Killing
  - Crude but simple
  - Keep killing processes in cycle until cycle is broken

Our deadlock toolkit (so far)

- Approach 1: Ignore problem
- Approach 2: Prevention
  - 2a: Exhaustive Search of Possible States
  - 2b: Ad Hoc Genius
  - **2c: New! Two-Phase Locking**
  - **2d: New! Hierarchical Allocation**
- Approach 3: Detection and Recovery
  - 3a: Ad Hoc Genius
  - **3b: New! Cycle Detection (with recovery)**
- Now: Approach 4: Deadlock *Avoidance*

Approach 4: Deadlock Avoidance

(another approach)

Preventing Deadlock vs. *Avoiding* Deadlock

- Deadlock *prevention* depends on designing the system/application to deny at least one of the preconditions for deadlock
  - advantage: won't ever deadlock no matter what, e.g., the OS scheduler does
  - disadvantage: may not be possible/efficient for some kinds of resources
- Another approach, called deadlock *avoidance*, asks the OS to schedule resource use (at *runtime*) in a "safe" way
  - "unsafe" schedules are those that risk deadlock

Safe States and Deadlock

- Environment: there are *multiple, inter-changeable* instances of resources
    - example: memory
    - this is *not* the case with, e.g. mutexes, chopsticks in DP
  - processes can state requirements in advance
  - system free to make choices at runtime about which resources to allocate (and who to unblock when)
    - but no preemption - once resource given, it can't be taken back until the process gives it back
- "Safe" states result from making these allocation choices *conservatively*
  - never allocate in a way that could result in deadlock

(Very) Conservative Allocation (fundamentalist, in fact)

Example: total resources: 12

process	maximum requirement
A	6
B	11
C	7

Looks like we can only run one at a time, no matter how many resources are actually used by each process at any given moment.

Too bad! But maybe that's just the price you pay for being so risk averse…

How conservative is *too* conservative here?

- It may be possible to be conservative in allocating resources without assuming that everyone is always using their maximum
  - is there a more *efficient* way to be conservative?
- Consider a bank that issues lines of credit:
  - greedy: wants as many customers as possible
    - customers will eventually repay their debts, but may need to max out their credit line first
      - customers willing to wait to get their loans
    - total lines of credit may exceed bank's actual assets (but actual money loaned out is always <= total assets
  - conservative: must never, ever get deadlocked

Banker's (Dijkstra's) Algorithm

- Environment: n process P1, …. Pn and m resources R1 …. Rm
- Every process declares (in advance) its *claim*---the maximum number of resources it will ever need
  - Sum of claims of all processes could exceed total number of resources
- To avoid deadlocks, OS maintains the *allocation state*
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi
  - Claims matrix M: M[i,j] is the maximum number of instances of Rj that process Pi will ever request
  - Availability vector A: A[j] is the number of instances of Rj currently free.
- Suppose process Pi requests certain number resources. Let Req be the request vector (Req[j] is number of requested instances of Rj)
  - Valid request if Req <= M[i]-C[i] (i.e. it should be in accordance with claim)
  - If Req <= A, then it is possible for OS to grant the request
  - Avoidance strategy: Deny the request if the resulting state will be *unsafe*

Safe states

- An allocation state is ***safe*** if there is an ordering of processes (a ***safe sequence****)* such that:
  - the first process can finish for sure
    - there are enough available resources to satisfy all of its claim
  - once the first process releases its resources, the second process can finish for sure (even if it asks all its claim)
  - and so on.
- This is "safe" because the OS always can avoid deadlock simply by blocking *new* requests until some or all of the current processes have finished
  - check to see if a new request would be unsafe
    - if so, block it until someone finishes, then check again

Example

(One resource class only)

total resources: 12

unallocated: 2

process	holding	max claims
A	4	6
B	4	11
C	2	7

safe sequence: A,C,B

Another example

(One resource class only)

total resources: 12

unallocated: 2

process	holding	max claims
A	4	6
B	4	11
C	2	9

safe sequence: none!

Banker's algorithm

- Maintain claims M, current allocation C and current availability A
- Suppose process Pi requests Req such that Req <= A and Req+C[i] <= M[i]
  - Consider the state resulting from granting this request (i.e. by adding Req to C[i] and subtracting Req from A).
  - Check if the new state is a safe state. If so, grant the request, else deny it.

Checking Safety

- How do we check if an allocation state is safe?
  - Current allocation matrix C
  - Maximum claims matrix M
  - Availability vector A
- Same as running a deadlock detection algorithm assuming that every process has requested maximum possible resources
  - Choose Requests Matrix R to be M – C, and see if the state is deadlocked (is there an order in which all of these requests can be satisfied).

Variations on the Banker's Algorithm

- Multiple resource types
  - just run it for each resource
- "increased" and "decreased" claims at runtime

When is Deadlock Avoidance practical?

- Requires that processes know (and state) their maximum requirements *up front*
  - even if not using them all at once
  - like a “credit line”
- But general processes may not be able to do that for most of the kinds of resources they use
- Useful mainly for
  - specialized applications that share a few classes of inter-changeable resources
  - OS-based resources like memory & processor allocation

Evaluating our deadlock toolkit

- Approach 1: Ignore
- Approach 2: Prevention
  - System design rules
  - 2a: Exhaustive Search of Possible States
  - 2b: Ad Hoc Genius
  - **2c: New! Two-Phase Locking**
    - inefficient (spin locks)
  - **2d: New! Hierarchical Allocation**
    - maybe, if you can find global ordering
- Approach 3: Detection and Recovery
  - Runtime techniques
  - 3a: Ad Hoc Genius
  - **3b: New! Cycle Detection (with recovery)**
    - OK, but then what? need safe rollback mechanism
- Approach 4: Deadlock Avoidance
  - runtime techniques
  - **4a: New! Dijkstra's "Banker's Algorithm"**
    - not generally useful, but OK for specialized applications & OS services

OK, what do we do? Which tools are best?

- Deadlock prevention via hierarchical allocation seems in some sense "best"
  - guarantees that a deadlock can never occur as long as rules are followed
  - no other system support required
- But there's a significant limitation: all we have is a set of rules for writing *new* programs, not a way to test an arbitrary system for deadlocks
  - in fact, that would be provably impossible!
    - remember the halting problem
    - same reason we don't have SJF schedulers

Approach 1 (ignore problem) looking better…

- Most general computing systems don’t do deadlock prevention or avoidance as a system service
  - instead they leave applications to fend for themselves
  - avoidance and recovery techniques must be done directly by any applications that require it
- Deadlock is an issue to the OS designer mainly for managing resources *inside* the OS
  - deadlocks within the OS are very disruptive

What about Starvation?

Starvation != Deadlock

- *Deadlock* is a situation where 2 or more processes are “locked” in a situation where neither *can ever* proceed, no matter what the OS does
  - e.g., circular waiting, etc.
- *Starvation*, on the other hand, is a situation where a process that *could* get service just never *does*
  - e.g., the scheduler never picks it, some other process is always selected ahead of it
- A system can be provably free of deadlock but still be subject to starvation!

So… deadlock prevention may not prevent starvation

- Mechanisms like semaphores and schedulers don’t usually make guarantees about who gets service in what order
- In practice, we avoid starvation in three ways:
  – algorithms that take into account starvation
    - e.g., the dining philosophers solution in Tanenbaum
  – “fair” schedules
    - e.g, based on FCFS RR queues
    - doesn’t always work
  – random system behavior that perturbs “bad” cycles
    - no guarantees, but often works in practice