

Deadlock (2)

Dave Eckhardt
Roger Dannenberg

Synchronization

- Project 2 progress
 - Don't split the coding in a bad way
 - One popular bad way: Person A codes list/queue, syscall stubs
 - Person B codes everything else
 - Person A will probably be in big trouble on the exam

Synchronization

- Project 2 progress
 - Should have created a thread
 - Should have passed `startle`
 - Should be passing some mutex/cvar tests
 - Ok if exiting, cleanup aren't right yet
 - Ok if `thr_exit()` looks like: **`while(1) continue;`**
 - Not as good if it looks like: **`for(;;);`**
 - Ok if some components are “demo quality”

Outline

- Review
 - Prevention/Avoidance/Detection
- Today
 - Avoidance
 - Detection/Recovery

Deadlock - What to do?

- Prevention
 - *Pass a law* against one of four ingredients
- Avoidance
 - Processes *pre-declare usage patterns*
 - Request manager avoids “unsafe states”
- Detection/Recovery
 - Clean up only when trouble really happens

Deadlock Avoidance – Motivation

- Deadlock prevention *passes laws*
 - Unenforceable: shared CD-writers???
 - Annoying
 - Mandatory lock-acquisition order may induce starvation
 - Locked 23, 24, 25, ... 88, 89, now must lock 0...
 - *Lots* of starvation opportunities
- Do we really need such strict laws?
 - Couldn't we be more “situational”?

Deadlock Avoidance Assumptions

1. Processes pre-declare usage patterns

- Could enumerate all paths through allocation space
 - Request R1, Request R2, Release R1, Request R3, ...
 - or -
 - Request R1, Request R3, Release R3, Request R1, ...
- Easier: declare *maximal resource usage*
 - I will never need more than 7 tape drives and 1 printer

Deadlock Avoidance Assumptions

2. Processes proceed to completion

- Don't hold onto resources forever
 - Obvious how this helps!
- Complete in “reasonable” time
 - So it is ok, if necessary, to stall P2 until P1 completes
 - We will try to avoid this

Safe Execution Sequence

- $(P_1, P_2, P_3, \dots P_n)$ is a *safe sequence* if
 - Every process P_i can be satisfied using
 - currently-free resources F plus
 - resources currently held by $P_1, P_2, \dots P_i$
- P_i 's waiting is bounded by this sequence
 - P_1 will run to completion, release resources
 - P_2 can complete with $F + P_1$'s + P_2 's
 - P_3 can complete with $F + P_1$'s + P_2 's + P_3 's
 - P_i won't wait forever, so no wait cycle, no deadlock \square

Safe State

- System in a *safe state* iff...
 - there exists at least one safe sequence
- Worst-case situation
 - Every process asks for every resource at once
 - Follow the safe sequence (run processes serially)
 - Slow, but not as slow as a deadlock!
- Serial execution is *worst-case*, not typical
 - Usually execute in parallel

Request Manager - Naïve

- Grant request if
 - Enough resources are free now
- Otherwise, tell requesting process to *wait*
 - While *holding* resources
 - Which are *non-preemptible, ...*
- Easily leads to deadlock

Request Manager – Avoidance

- Grant request if
 - Enough resources are free now, *and*
 - Enough resources would *still* be free
 - For some process to complete and release resources
 - And then another one
 - And then you
- Otherwise, wait
 - While holding a smaller set of resources...
 - *...which we previously proved other processes can complete without*

Example (from text)

Who	Max	Has	Room
P0	10	5	5
P1	4	2	2
P2	9	2	7
System	12	3	—

Max declared
Has allocated
Room (Max-Has)

“Is it safe?”

“Yes, it's safe; it's very safe, so safe you wouldn't believe it.”

P1: $2 \Rightarrow 4$

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	2	2	\Rightarrow	P1	4	4	0
P2	9	2	7		P2	9	2	7
System	12	3	-	\Rightarrow	System	12	1	-

P1: Complete

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	4	0	⇒				
P2	9	2	7		P2	9	2	7
System	12	1	-	⇒	System	12	5	-

P0: $5 \Rightarrow 10$

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5	\Rightarrow	P0	10	10	0
P2	9	2	7		P2	9	2	7
System	12	5	-	\Rightarrow	System	12	0	-

P0: Complete

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	10	0	⇒				
P2	9	2	7		P2	9	2	7
System	12	0	-	⇒	System	12	10	-

“Run P1, P0, P2” is a *safe sequence*.
 So the system was in a *safe state*.

Example (from text)

Who	Max	Has	Room
P0	10	5	5
P1	4	2	2
P2	9	2	7
System	12	3	—

“Can P2 ask for more?”

“Is it safe?”

“No, it's not safe, it's very dangerous, be careful.”

P2: $2 \Rightarrow 3$?

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	2	2		P1	4	2	2
P2	9	2	7	\Rightarrow	P2	9	3	6
System	12	3	-	\Rightarrow	System	12	2	-

P2: $2 \Rightarrow 3$?

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	2	2		P1	4	2	2
P2	9	2	7	\Rightarrow	P2	9	3	6
System	12	3	-	\Rightarrow	System	12	2	-

Now only P1 can be satisfied without waiting.

P1: 2 \Rightarrow 4?

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	2	2	\Rightarrow	P1	4	4	0
P2	9	3	6		P2	9	3	6
System	12	2	-	\Rightarrow	System	12	0	-

P1: Complete

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	4	0	⇒				
P2	9	3	6		P2	9	3	6
System	12	0	-	⇒	System	12	4	-

P1: Complete

Who	Max	Has	Room		Who	Max	Has	Room
P0	10	5	5		P0	10	5	5
P1	4	4	0	⇒				
P2	9	3	6		P2	9	3	6
System	12	0	-	⇒	System	12	4	-

Problem: P0 and P2 are allowed to ask for >4.
 If both do, both sleep: **deadlock**.

Avoidance - Key Ideas

- Safe state
 - Some safe sequence exists
 - Prove it by *finding one*
- Unsafe state: No safe sequence exists
- Unsafe *may not be fatal*
 - Processes might exit early
 - Processes might not use max resources today

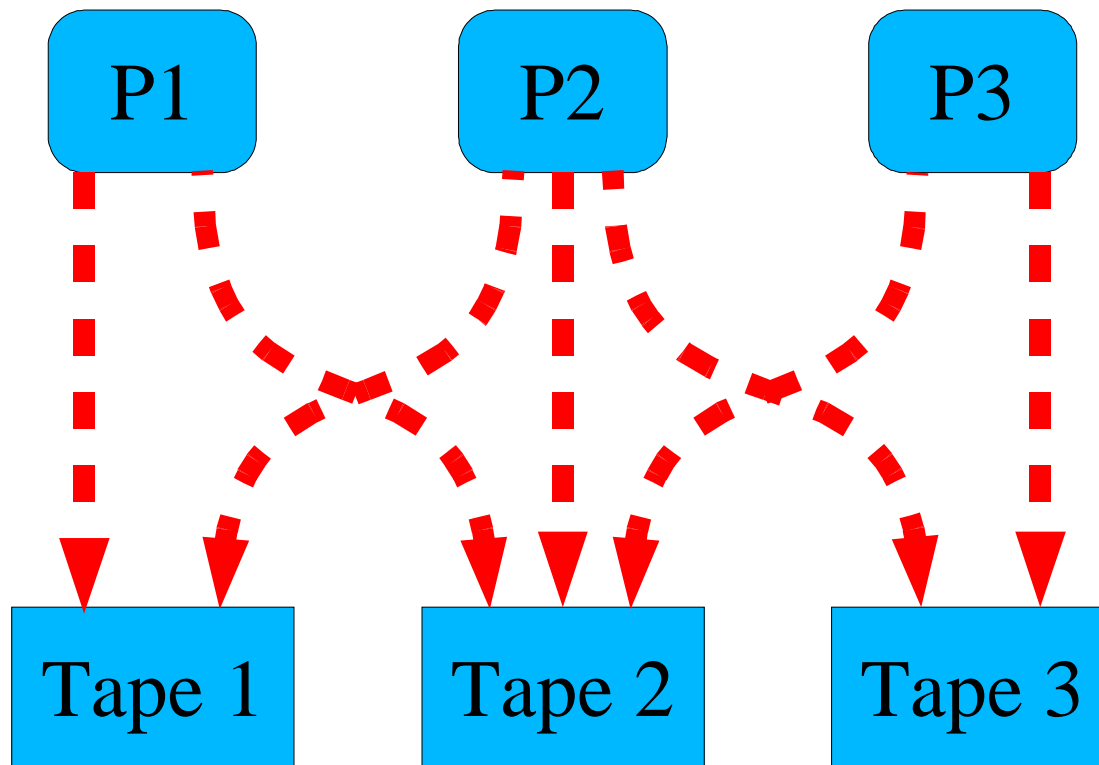
Avoidance – Tradeoff

- Allowing only safe states is more flexible than Prevention
- But rejecting *all* unsafe states reduces efficiency
 - System *could* enter unsafe state and then return to safety...
 - How often would the system “retreat from disaster”?
- Hmm...

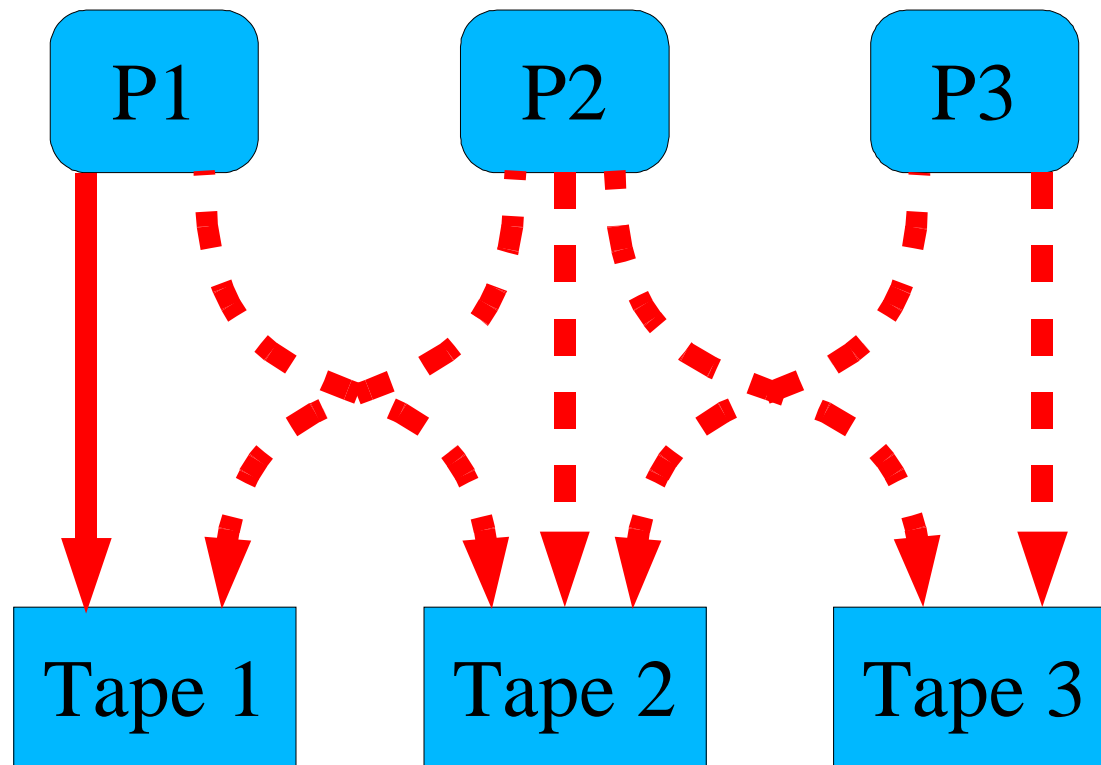
Avoidance - Unique Resources

- Unique resources instead of multi-instance?
 - Graph algorithm
- Three edge types
 - Claim (future request)
 - Request
 - Assign

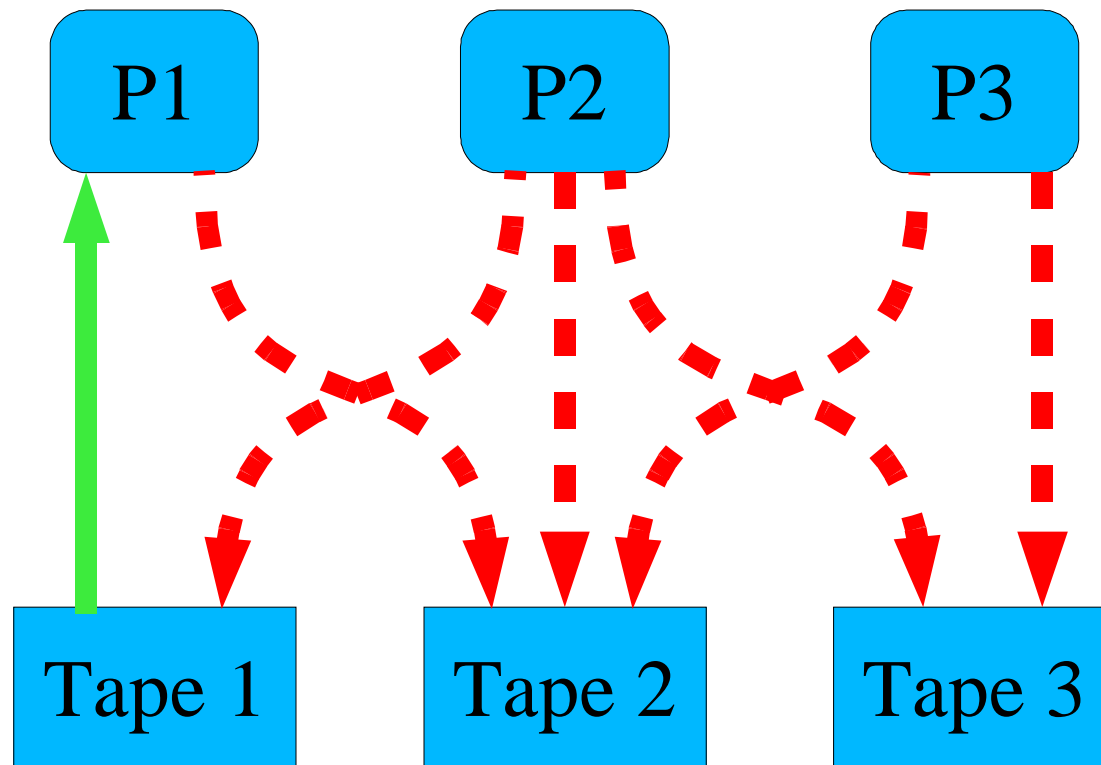
“Claim” (Future-Request) Edges



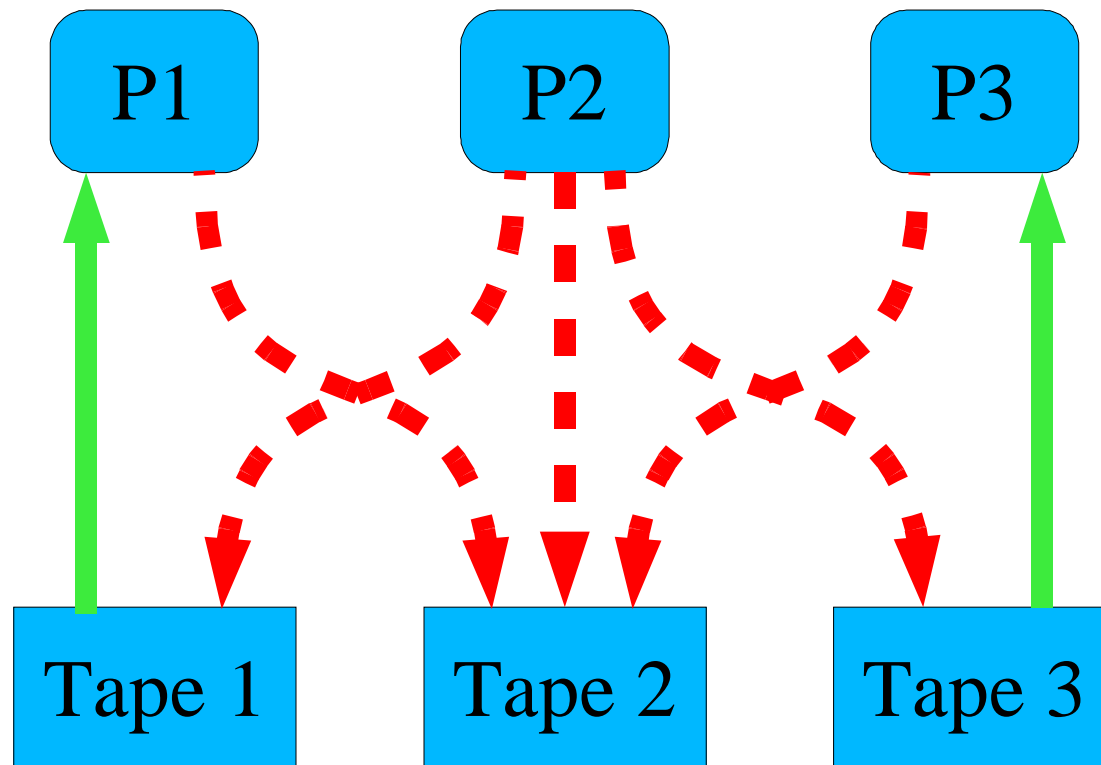
Claim \Rightarrow Request



Request \Rightarrow Assignment



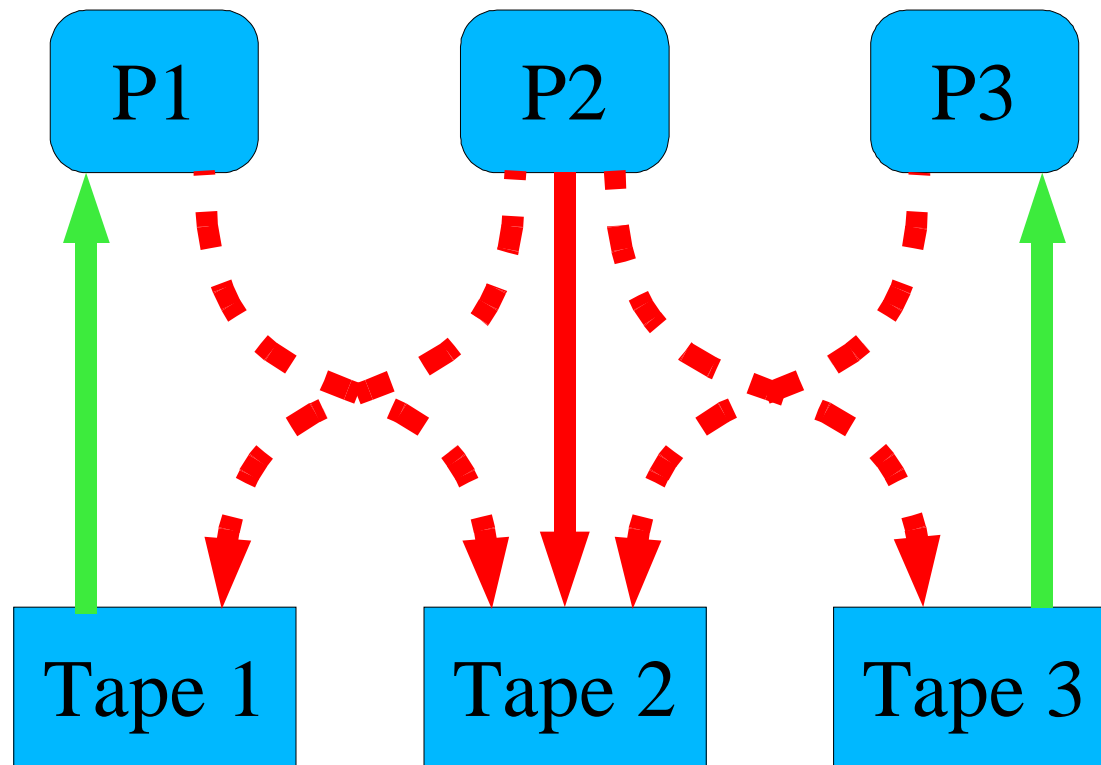
Safe: No Cycle



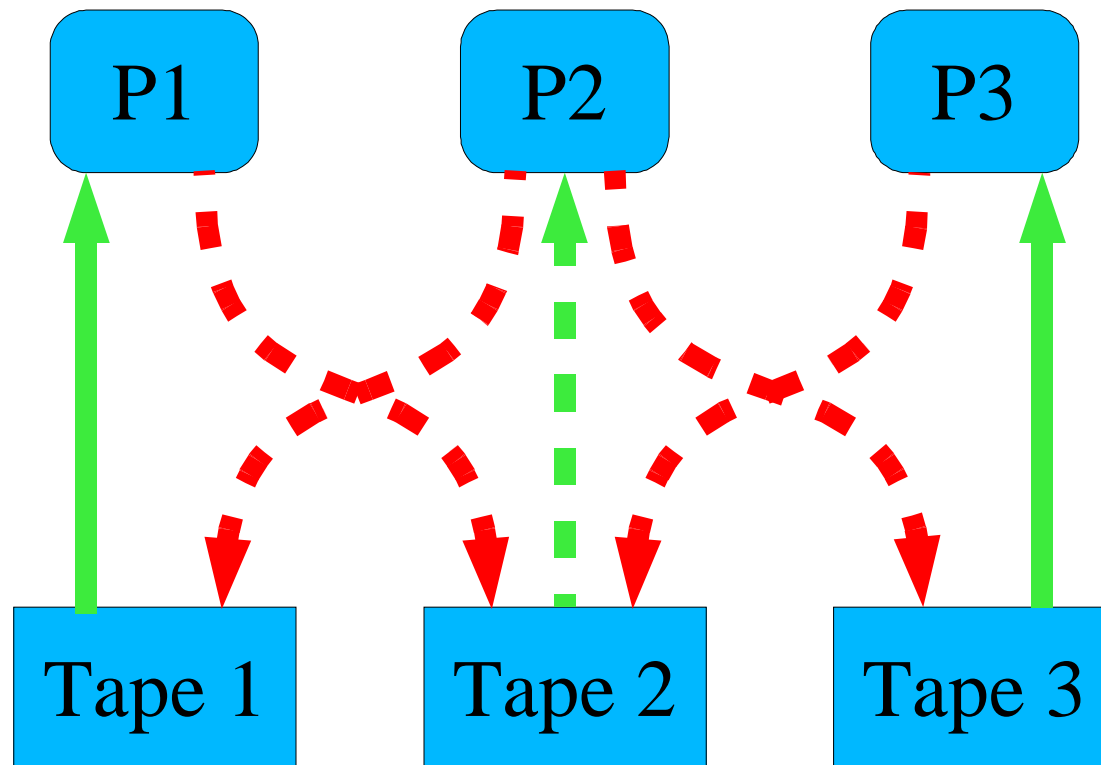
Which Requests Are Safe?

- Pretend to satisfy request
- Look for cycles in resultant graph

A Dangerous Request



See Any Cycles?



Are “Pretend” Cycles Fatal?

- Must we worry about *all* cycles?
 - Nobody is waiting on a “pretend” cycle
 - Lots of the edges are only *potential request* edges
 - We don't have a deadlock
- “Is it safe?”

Are “Pretend” Cycles Fatal?

- *No* process can, without waiting
 - Acquire maximum-declared resource set
- So *no process* can acquire, complete, release
 - (for sure, without maybe waiting)
- Any new sleep *could* form a cycle
 - “No, it's not safe, it's very dangerous, be careful.”
- What to do?
 - Don't grant the request (put the process to sleep *now*, *before* it gets that resource)

Avoidance - Multi-instance Resources

- Example
 - N interchangeable tape drives
 - Could represent by N tape-drive nodes
 - Needless computational expense
- Business credit-line model
 - Bank assigns maximum loan amount (“credit limit”)
 - Business pays interest on *current* borrowing amount

Avoiding “bank failure”

- Bank is “ok” when there is a *safe sequence*
- One company can
 - Borrow up to its credit limit
 - Do well
 - IPO
 - Pay back its full loan amount
- And then another company, etc.

No safe sequence?

- Company tries to borrow up to limit
 - Bank has no cash
 - Company C1 must wait for money C2 has
 - Maybe C2 must wait for money C1 has
- In real life
 - C1 cannot make payroll
 - C1 goes bankrupt
 - Loan never paid back in full
 - Can model as “infinite sleep”

Banker's Algorithm

```
int cash;
int limit[N]; /* credit limit */
int out[N] /* borrowed */;
boolean done[N]; /* global temp! */
int future; /* global temp! */

int progressor (int cash) {
    for (i = 0; i < N; ++i)
        if (!done[i])
            if (cash >= limit[i] - out[i])
                return (i);
    return(-1);
}
```

Banker's Algorithm

```
boolean is_safe(void) {  
    future = cash;  
    done[0..N] = false;  
  
    while ((p = progressor(future)) > 0) {  
        future += borrowed[p];  
        done[p] = true;  
    }  
    return (done[0..N] == true)  
}
```


Banker's Algorithm

- Can we loan more money to a company?
 - Pretend we did
 - update cash and out[i]
 - Is it safe?
 - Yes: lend more money
 - No: un-do to pre-pretending state, sleep
- Multi-resource Version
 - Generalizes easily to N independent resource types
 - See text

Avoidance - Summary

- Good news - *No deadlock*
 - + No static “laws” about resource requests
 - + Allocations flexible according to system state
- Bad news
 - Processes must pre-declare maximum usage
 - Avoidance is *conservative*
 - *Many* “unsafe” states are *almost* safe
 - System throughput reduced – extra sleeping
 - 3 processes, can allocate only 2 tape drives!?!?

Deadlock - What to do?

- Prevention
 - *Pass a law* against one of four ingredients
- Avoidance
 - Processes *pre-declare usage patterns*
 - Request manager avoids “unsafe states”
- *Detection/Recovery*
 - *Clean up only when trouble really happens*

Detection & Recovery - Approach

- Don't be paranoid
 - Don't refuse requests that *might* lead to trouble
 - (someday)
 - Most things work out ok in the end
- Even paranoids have enemies
 - Sometimes a deadlock *will* happen
 - Need a plan for noticing
 - Need a policy for reacting
 - Somebody must be told “try again later”

Detection - Key Ideas

- “Occasionally” scan for wait cycles
- Expensive
 - Must lock out all request/allocate/deallocate activity
 - Global mutex is the “global variable” of concurrency
 - Detecting cycles is an N^2 kind of thing

Scanning Policy

- Throughput balance
 - Scan too often - system becomes (very) slow
 - Scan before every sleep? Only in small systems
 - Scan too rarely - system becomes (extremely) slow
- Policy candidates
 - Scan every <interval>
 - Scan when CPU is “too idle”

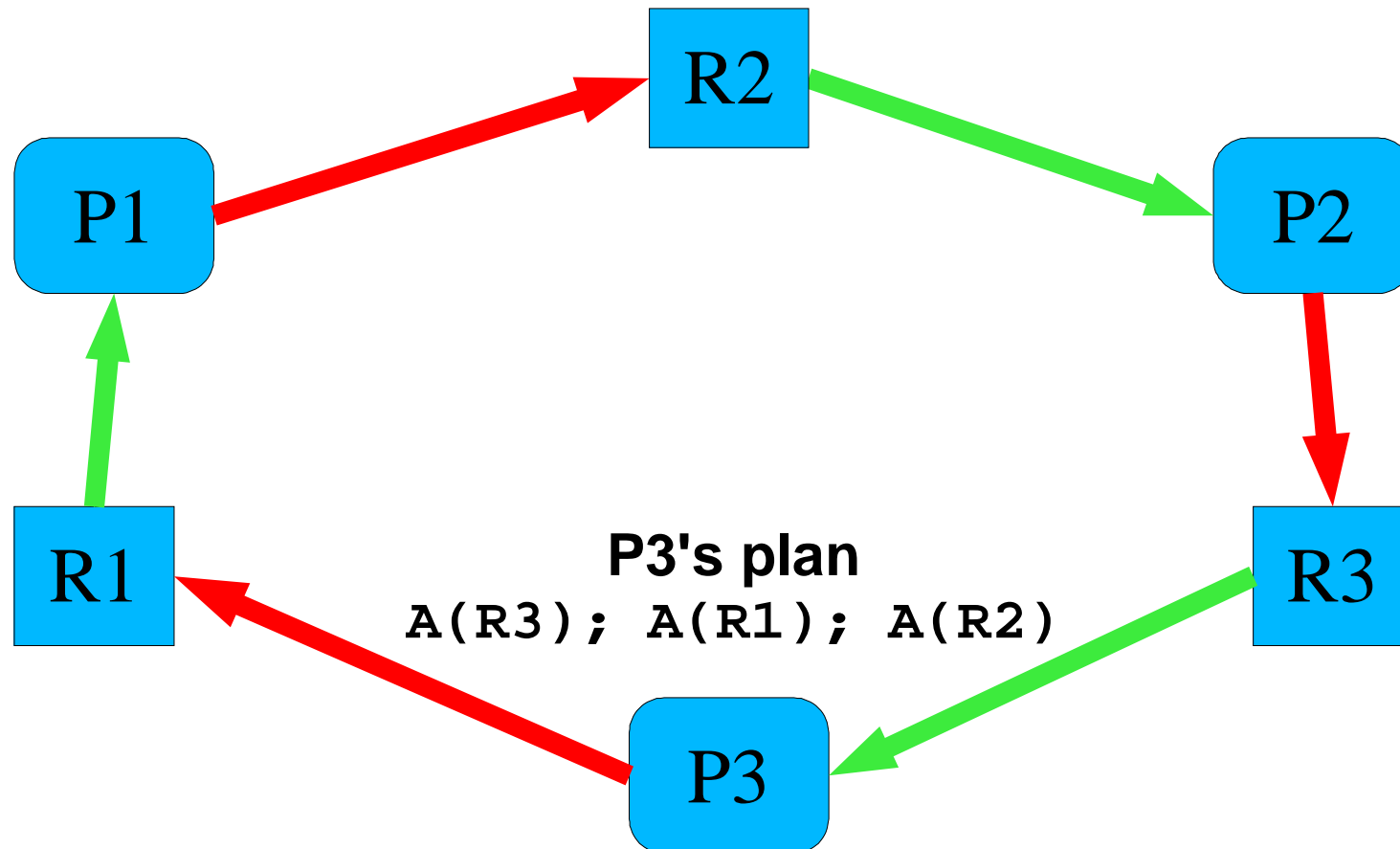
Detection - Algorithms

- Detection: Unique Resources
 - Search for cycles in resource graph
 - (see above)
- Detection: Multi-instance Resources
 - Slight variation on Banker's Algorithm
 - (see text)
- Find a deadlock? Now what?
 - Abort
 - Preempt

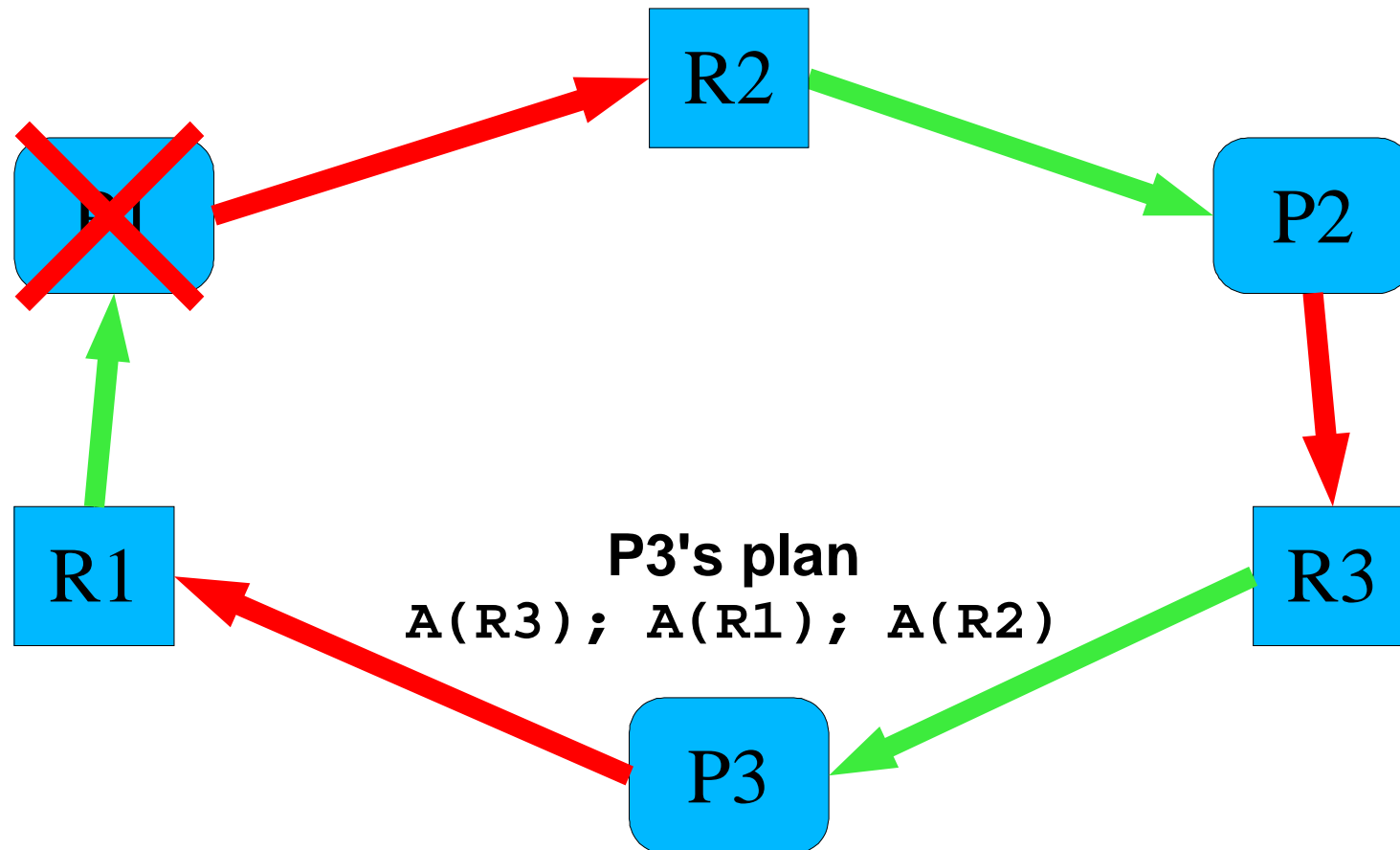
Recovery - Abort

- Evict processes from the system
- All processes in the cycle?
 - Simple & blame-free policy
 - Lots of re-execution work later
- *Just one* process in the cycle?
 - *Which* one?
 - Priority? Work remaining? Work to clean up?
 - Often immediately creates a smaller cycle – re-scan?

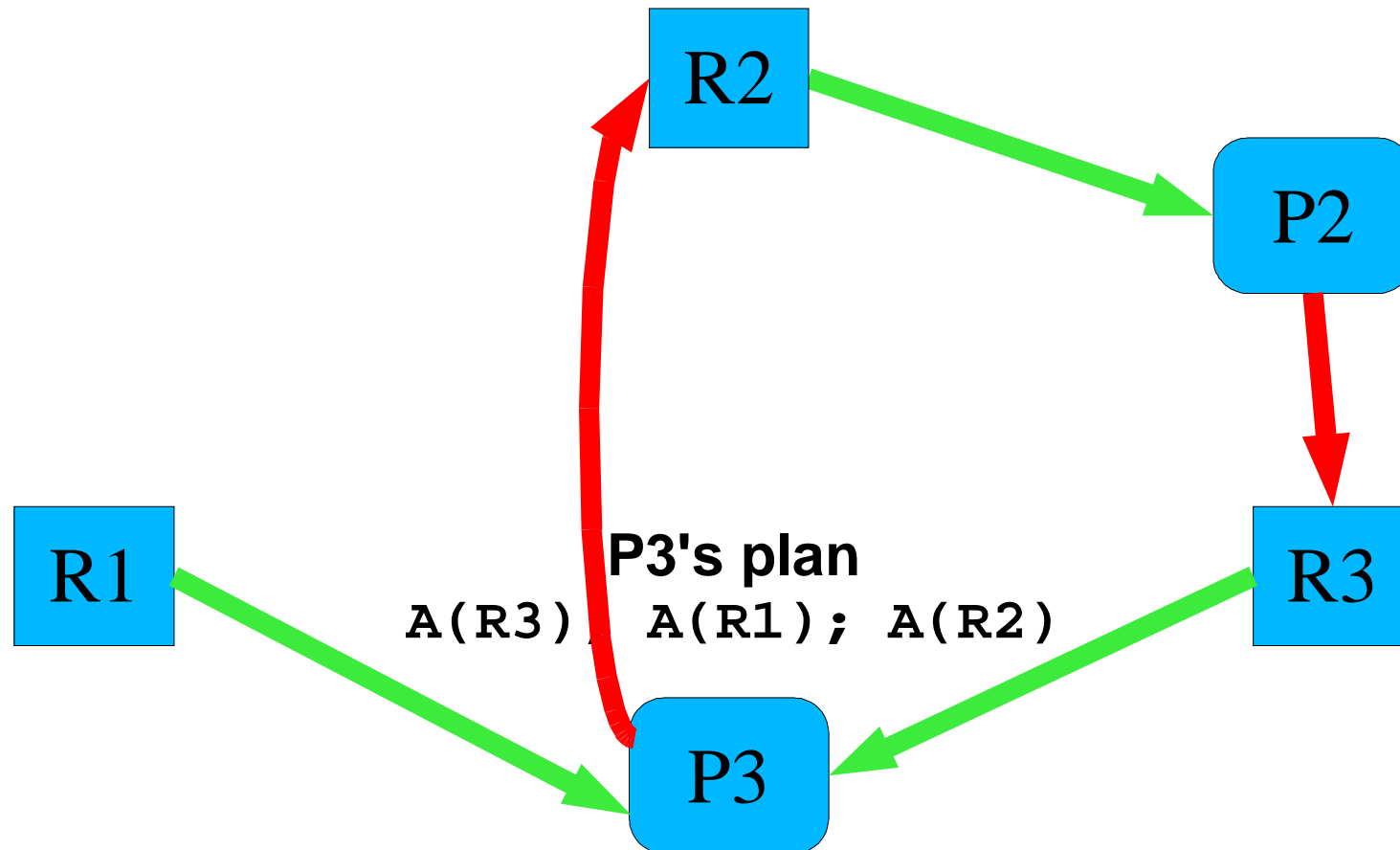
Recovery – Abort Just One?



Recovery – Abort Just One?



Recovery – Abort Just One?



Recovery – Can we do better?

- Aborting processes is undesirable
 - Re-running processes is *expensive*
 - Long-running tasks may *never* complete
 - Starvation

Recovery - Resource Preemption

- Tell some process(es)
- `lock(R346) ⇒ “EDEADLOCK”`
- Policy question: which process loses?
- Lowest-numbered? ⇒ *starvation!*
- What does “EDEADLOCK” mean?
 - *Can't* just retry the request (make sure you see this)
 - Must release *other* resources you hold, try later
 - Forced release may require “rollback” (yuck)

Summary - Deadlock

- Deadlock is...
 - Set of processes
 - Each one waiting for something held by another
- Four “ingredients”
- Three approaches
 - (aside from “Hmmm...<reboot>”)

Deadlock - Approaches

- Prevention - Pass a law against one of:
 - Mutual exclusion (unlikely!)
 - Hold & wait (maybe, but...)
 - No preemption (maybe?)
 - Circular wait (sometimes)

Deadlock - Approaches

- Avoidance - “Stay out of danger”
 - Requires pre-declaration of usage patterns
 - Not all “danger” turns into *trouble*
- Detection & Recovery
 - Scan frequency: delicate balance
 - Preemption is hard, messy
- Rebooting
 - Was it *really* hung?

Summary - Starvation

- Starvation is a ubiquitous danger
- Deadlock Prevention is one extreme
 - Need something “illegal”?
 - “Illegal” = *Eternal* starvation!
- Detection & Recovery
 - Less structural starvation
 - Still must make good choices