Deadlock (2)

David A. Eckhardt
School of Computer Science
Carnegie Mellon University

de0u@andrew.cmu.edu

Status Rendezvous & Outline

Project 2

- Questions?
- Some people have started!
    - Good!

Outline

- Review: Prevention/Avoidance/Detection&Recovery
- Avoidance
- Detection & Recovery

Deadlock - What to do?

Prevention

- restrict behavior or resources
- violate one of the 4 conditions

Avoidance

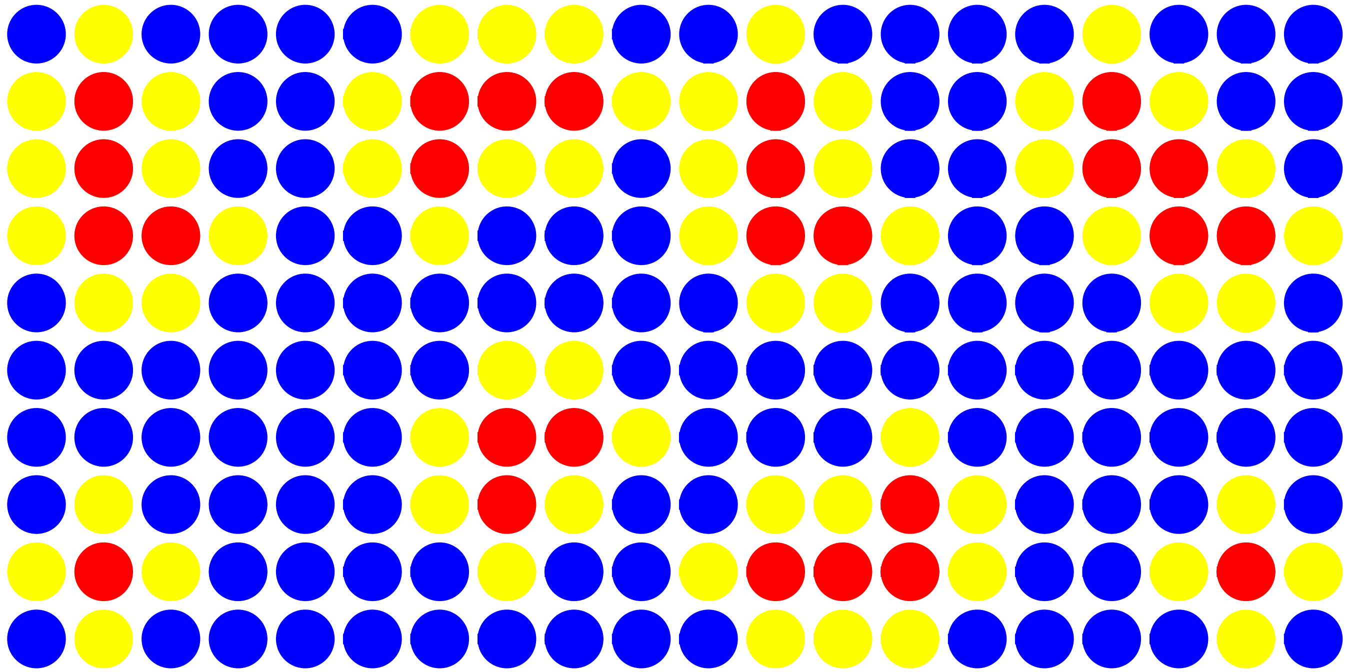
- dynamically examine requests
- keep system in "safe state"

Detection/Recovery

- maybe deadlock won't happen today
- gee, it seems quiet
- oops, here is a cycle
- abort some processes

Just reboot when it gets "too quiet"

State Space



Each node is a resource allocation *graph*

- Allocation/Deallocation moves system among nodes
- Islands of deadlock surrounded by “dangerous” states
  - Blocking for *some* requests will cause deadlock

Avoidance - Approach

Processes describe worst-case behavior

- Actual usage is always a subset

System rejects unsafe states

- Each request is evaluated for potential trouble
- Imagine granting request
  - Could any request from that state cause deadlock?

Safe state

- Informally - at least 1 state away from deadlock
- Formally - "safe sequence" must exist

Avoidance - Safe Sequence

Assumptions

- Every process will ask for everything it declared
- But will eventually finish work & exit

Safe sequence <P_1, P_2, ... P_n>

- System can satisfy P_1's growth to max
  - with currently-free resources
- When P_1 exits, system can satisfy P_2's growth to max
  - with current-free + P_1-growth
- When P_2 exits, system can satisfy P_3's growth to max
  - with current-free + P_1-growth + P_2-growth

Avoidance - Key Ideas

Safe state

- “Some safe sequence exists”
- Prove it by finding one

Unsafe state

- No safe sequence exists
  - some P_w could legally ask for “too much”
  - enough that P_x would need to wait
  - enough that P_y would need to wait
- Deadlock could result

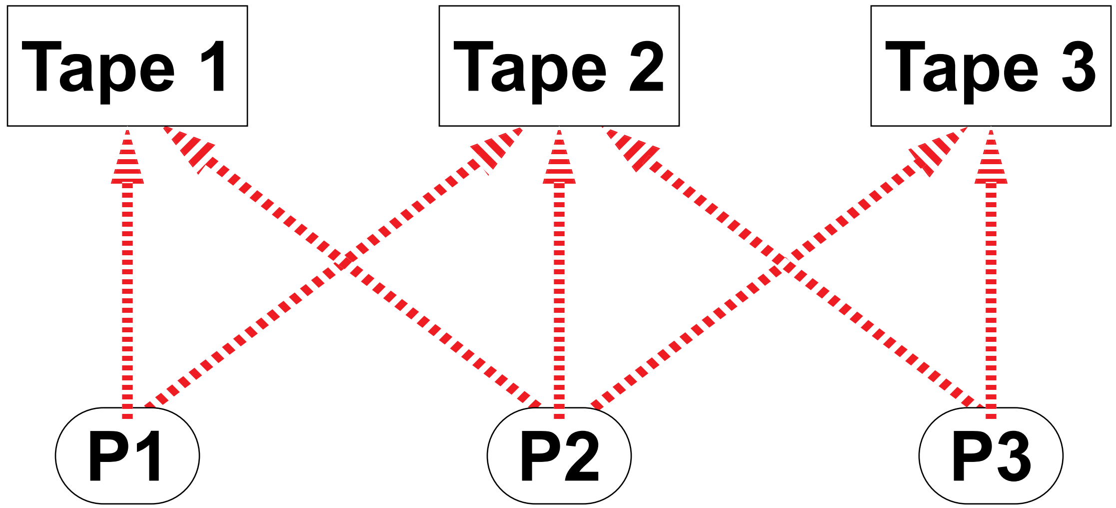
Unsafe may not be fatal

- Processes might exit early
- Processes might not use max resources today

System efficiency reduced

- Lots of unsafe states
- Many would not actually deadlock (today)

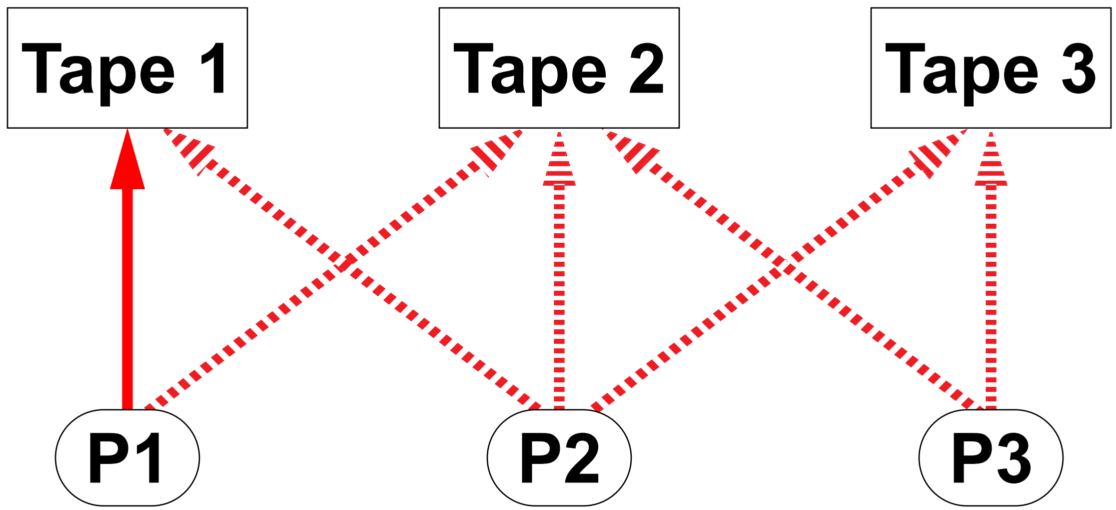
Avoidance - Unique Resources



Edges

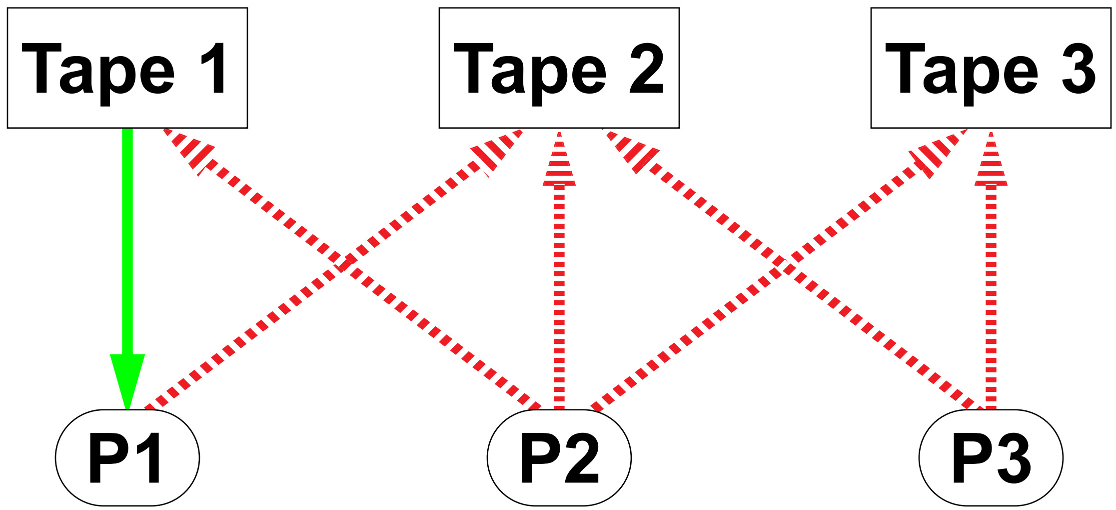
- Claim (future request)
- Request
- Assign

Avoidance - Unique Resources



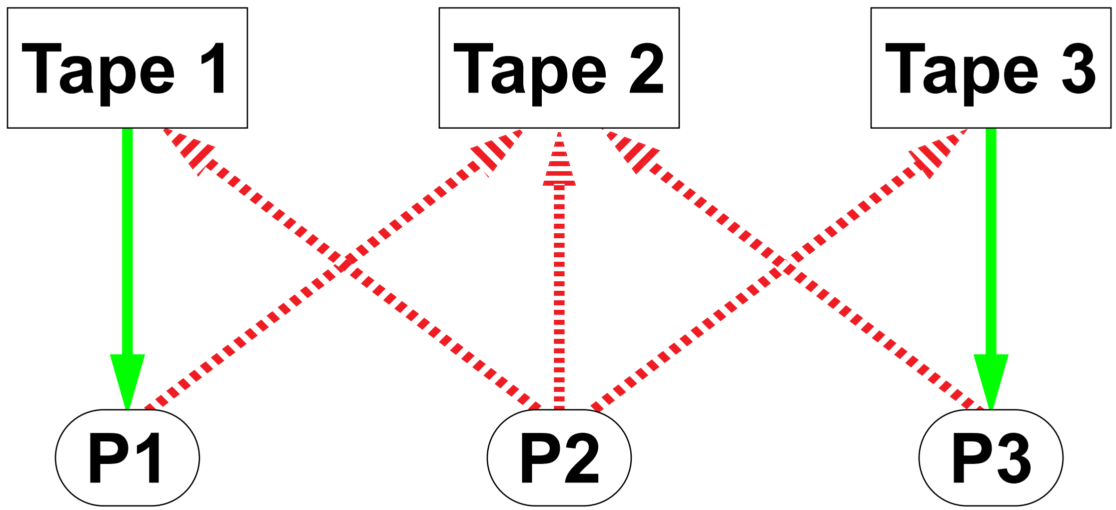
Claim -> Request

Avoidance - Unique Resources



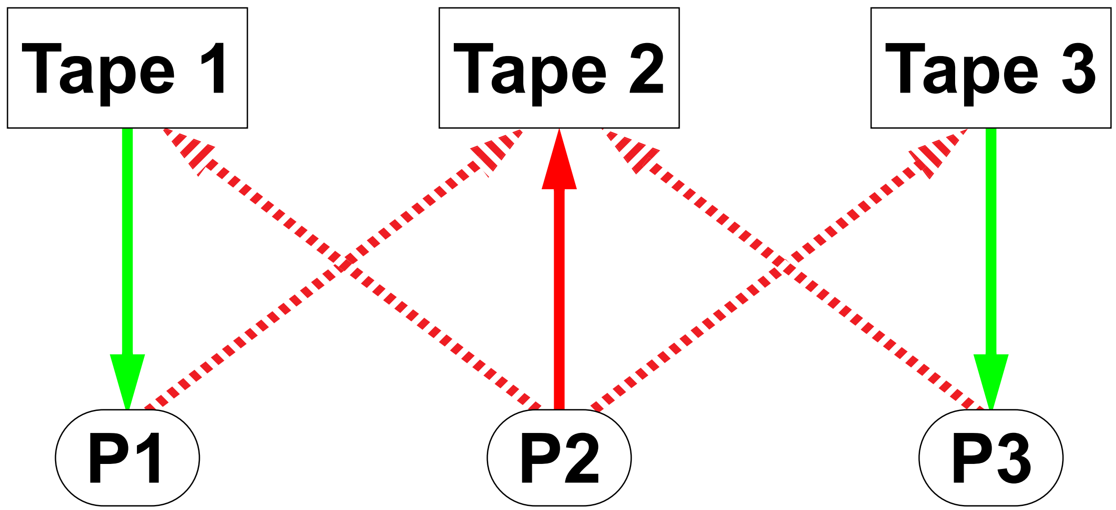
Request -> Assignment

Avoidance - Unique Resources



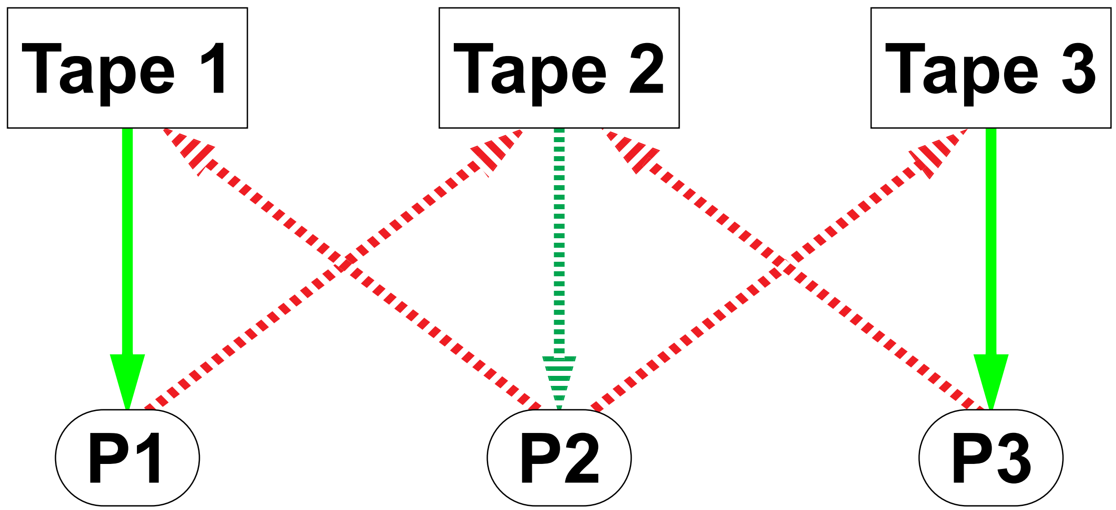
Non-cycle-forming requests are ok

Avoidance - Unique Resources



A request we should not grant

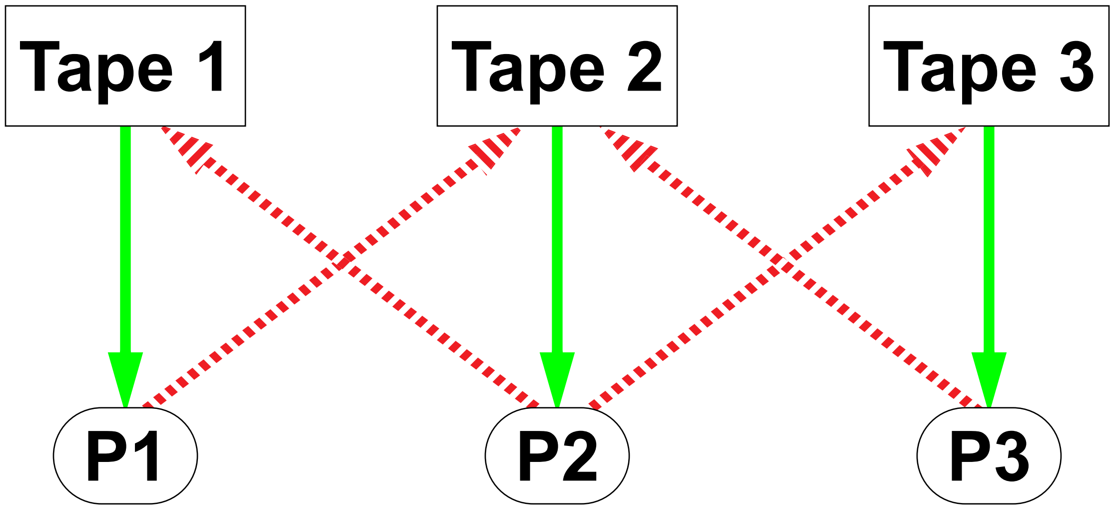
Avoidance - Unique Resources



Pretend to grant it

- Would you have a cycle?
  - Lots!
- So what!? Everything looks fine...

Avoidance - Unique Resources



No safe sequence

- *No* process can, without waiting
  - Acquire maximum-declared set of resources
  - Complete & release resources

Anybody going to sleep *might* never wake up

- So we can’t grant this (seemingly ok) request

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
- 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 must wait (and the next, and the next...)

In real life

- Company cannot make payroll
- Company goes bankrupt
- Loan not paid back

Banker's Algorithm

```
int cash;
int credit_limit[N];
int borrowed[N];
int could_borrow[N]; /*credit_limit-borrowed*/

boolean is_safe(void)
  int future = cash;
  boolean done[N] = { false };

  while (find debtor d:
   !done[d] && could_borrow[d] < future)
    future += borrowed[d];
    done[d] = true;

  if (FORALL(d) done[d])
   return (true);
  else
   return (false);
```

Banker's Algorithm

Can we loan more money to a company?

- Pretend we did
  - update cash, borrowed[], and could_borrow[]
- Is it safe?
  - Yes: ok!
  - No: un-do to pre-pretending state, say “not at this time”

Multi-resource Version

- Generalizes easily to N independent resource types (see text)

Avoidance - Summary

Good news

- No static “laws” about resource requests
- Processes can pre-declare *any* set of resources
- Allocation decisions flexible according to other processes

Bad news

- Avoidance bans *many* states with *many* positive scenarios
- Many totally ok paths through state space unavailable
  - System throughput reduced
    - 3 processes, can allocate only 2 tape drives!?!?

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-squared kind of thing

Throughput balance

- Too often - system becomes (very) slow
  - Before every sleep? Only in small systems
- 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 graph (see above)

Detection: Multi-instance Resources

- Slight variation on Banker's Algorithm

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?

- Should re-scan for immediate creation of shorter cycle
- Policy question: which one?
  - Priority?
  - Work remaining?
  - Work to clean up?

Recovery - Resource Preemption

Re-running processes is expensive

- Long-running tasks may *never* complete
    - Starvation

Tell one/some/all waiting processes "No"

- Policy question: which one?
    - Always choose lowest-numbered?
        - Starvation!

What does "no" mean?

- Can't retry the request!
- Must release other resources, "walk away", "come back"
- "State rollback" can be messy

Summary - Overall

Deadlock is...

- Set of processes
- Each one waiting for something held by another

Approaches

- Prevention - Pass a law against one of:
  - Mutual exclusion (right!)
  - Hold & wait (maybe...)
  - No preemption (maybe?)
  - Circular wait (sometimes)
- Avoidance - “Stay out of danger”
  - Not all “danger” turns into “trouble”
- Detection & Recovery
  - Frequency: delicate balance
  - Preemption is hard
- Rebooting

Summary - Starvation

Starvation is a ubiquitous danger

Prevention is one extreme

- Need something “illegal”? Starve for sure!

Detection & Recovery

- Less *structural* starvation
- Silll must make good choices