CSE 410
Computer Systems

Hal Perkins

Spring 2010

Lecture 19 – Deadlock

Readings and References

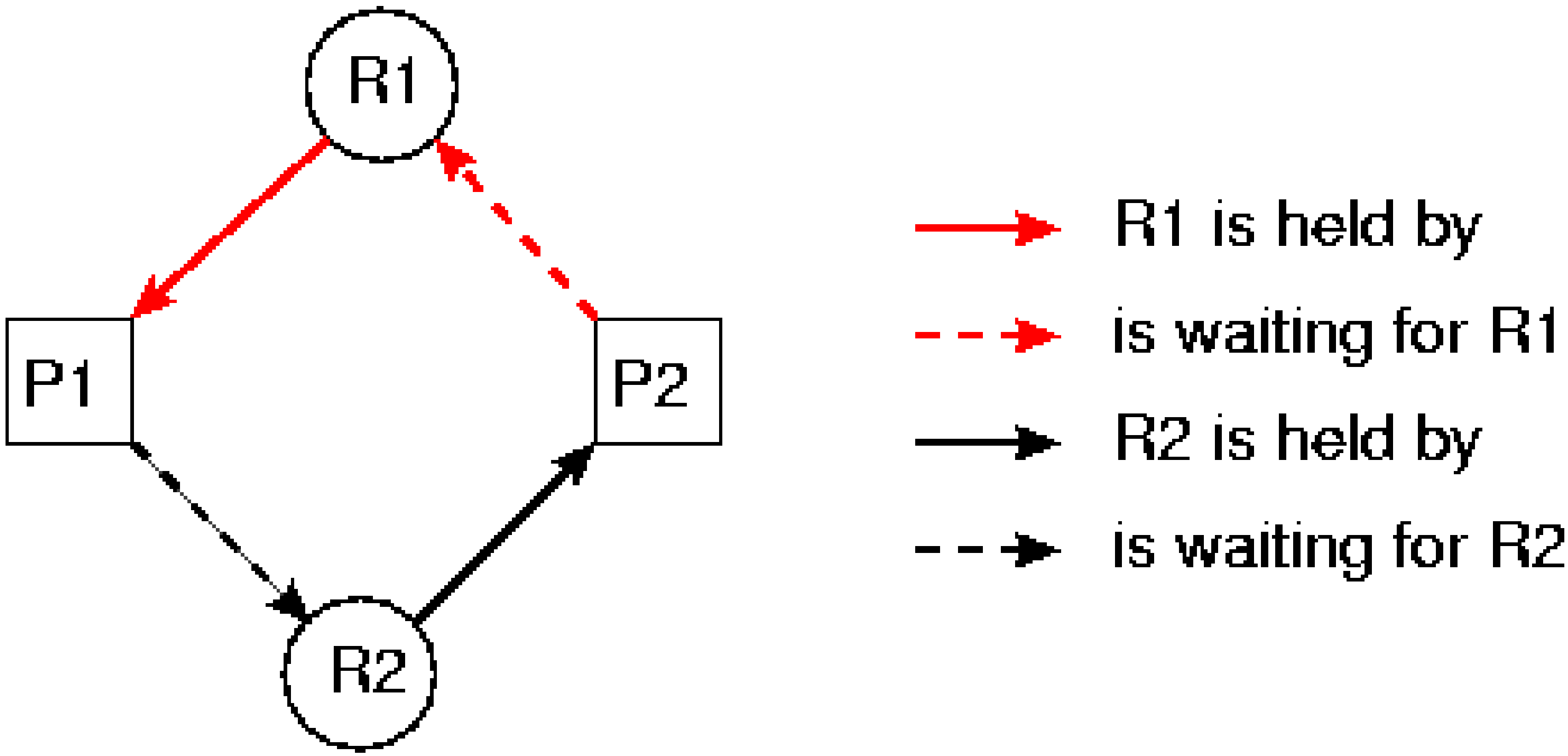
- Reading
  - Chapter 7, *Operating System Concepts*, Silberschatz, Galvin, and Gagne

(Is Google the greatest, or what?)

Definition

- A thread is deadlocked when it's waiting for an event that can never occur
  - I'm waiting for you to clear the intersection, so I can proceed
    - but you can't move until he moves, and he can't move until she moves, and she can't move until I move
  - thread A is in critical section 1, waiting for access to critical section 2; thread B is in critical section 2, waiting for access to critical section 1
  - I'm trying to book a vacation package to Tahiti – air transportation, ground transportation, hotel, side-trips. It's all-or-nothing – one high-level transaction – with the four databases locked in that order. You're trying to do the same thing in the opposite order.

Resource graph



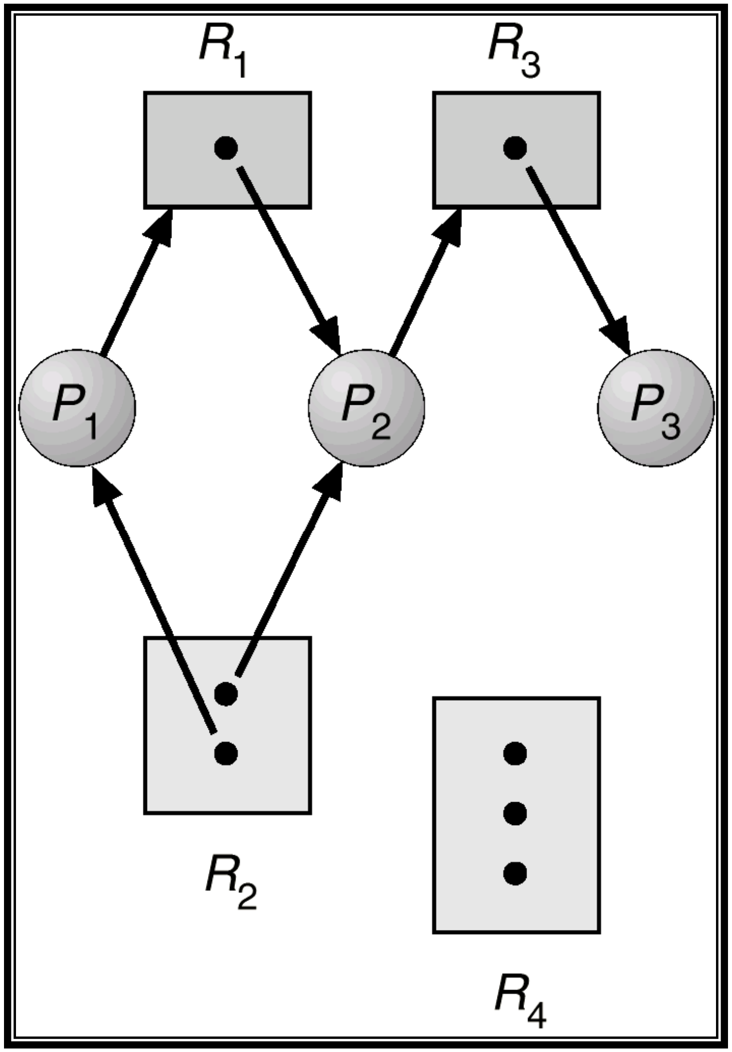
- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

Graph reduction

- A graph can be *reduced* by a thread if all of that thread's requests can be granted
  - in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to it in the graph are deleted
- Miscellaneous theorems (Holt, Havender):
  - There are no deadlocked threads iff the graph is completely reducible
  - The order of reductions is irrelevant

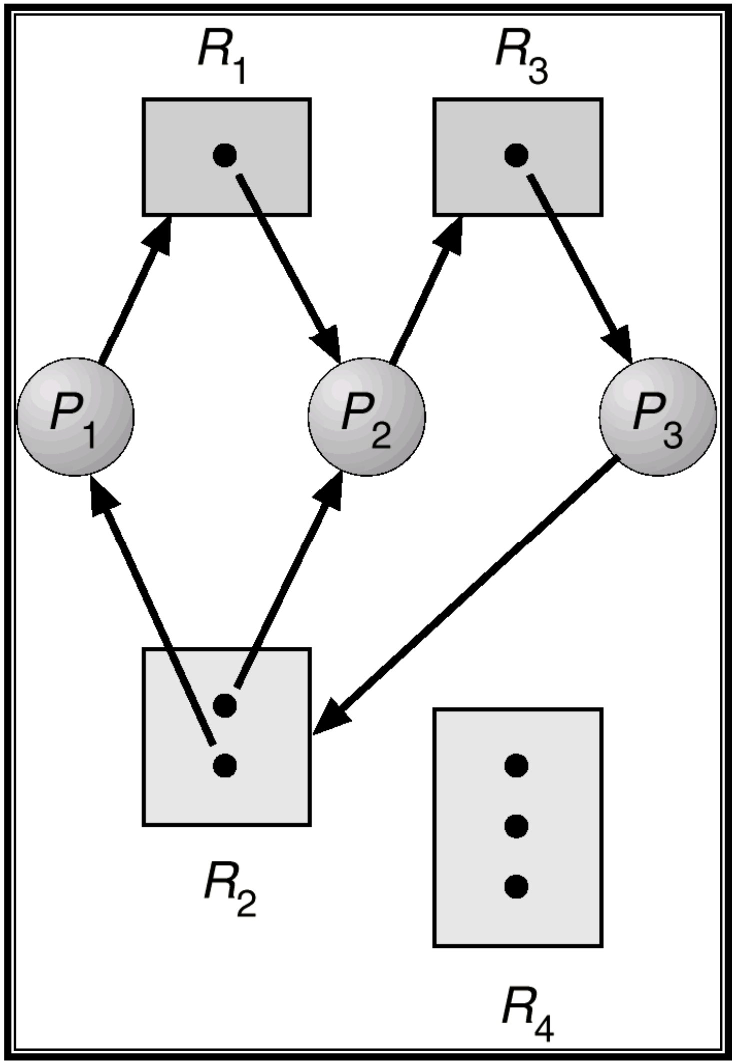
- (Detail: resources with multiple units)

Resource allocation graph with no cycle

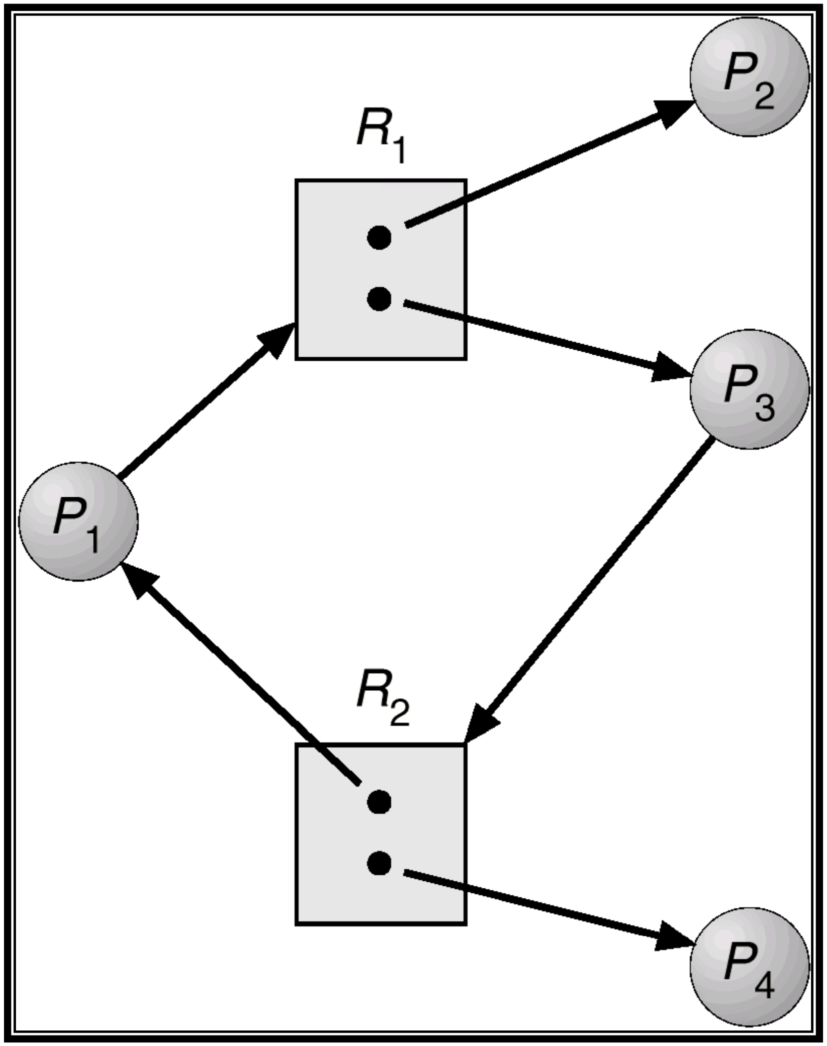


What would cause a deadlock?

Resource allocation graph with a deadlock

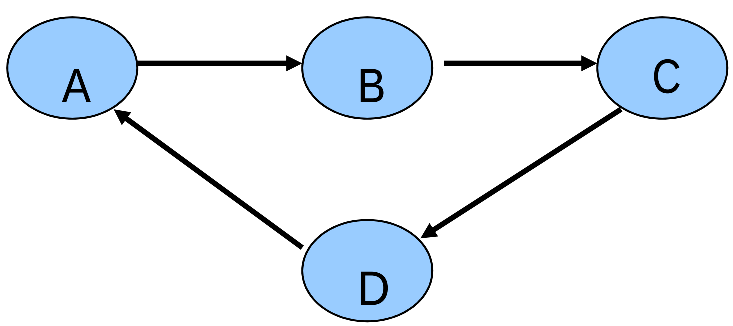


Resource allocation graph with a cycle but no deadlock



Necessary Conditions for Deadlock

- Mutual Exclusion
  - The resource can't be shared
- Hold and Wait
  - Task holds one resource while waiting for another
- No Preemption
  - If a task has a resource, it cannot be forced to give it up
- Circular Wait
  - A waits for B, B for C, C for D, D for A



Dealing with Deadlock

- Deadlock Prevention
  - Ensure statically that deadlock is impossible
- Deadlock Avoidance
  - Ensure dynamically that deadlock is impossible
- Deadlock Detection and Recovery
  - Allow deadlock to occur, but notice when it does and try to recover
- Ignore the Problem
  - Let the operator untangle it, that's what they're paid for

Deadlock Prevention

- There are four necessary conditions for deadlock
- Take any one of them away and deadlock is impossible
- Let’s attack deadlock by
  - examining each of the conditions
  - considering what would happen if we threw it out

Condition: Mutual Exclusion

- Usually can't eliminate this condition
  - some resources are intrinsically non-sharable
- Examples include printer, write access to a file or record, entry into a section of code
- However, you can often mitigate this by adding a layer of abstraction
  - For example, write to a queue of jobs for a shared resource instead of locking the resource to write

Condition: Hold and Wait

- Eliminate partial acquisition of resources
- Task must acquire all the resources it needs before it does anything
  – if it can't get them all, then it gets none
- Issue: Resource utilization may be low
  – If you need P for a long time and Q only at the end, you still have to hold Q's lock the whole time
- Issue: Starvation prone
  – May have to wait indefinitely before popular resources are all available at the same time

Condition: No Preemption

- Allow preemption
  - If a process asks for a resource not currently available, block it and take away all of its other resources
  - Add the preempted resources to the list of resources the process is waiting for
- This strategy works for some resources:
  - CPU state (contents of registers can be spilled to memory)
  - memory (can be spilled to disk)
- But not for others:
  - printer – let everyone print and sort the pages later?

Condition: Circular Wait

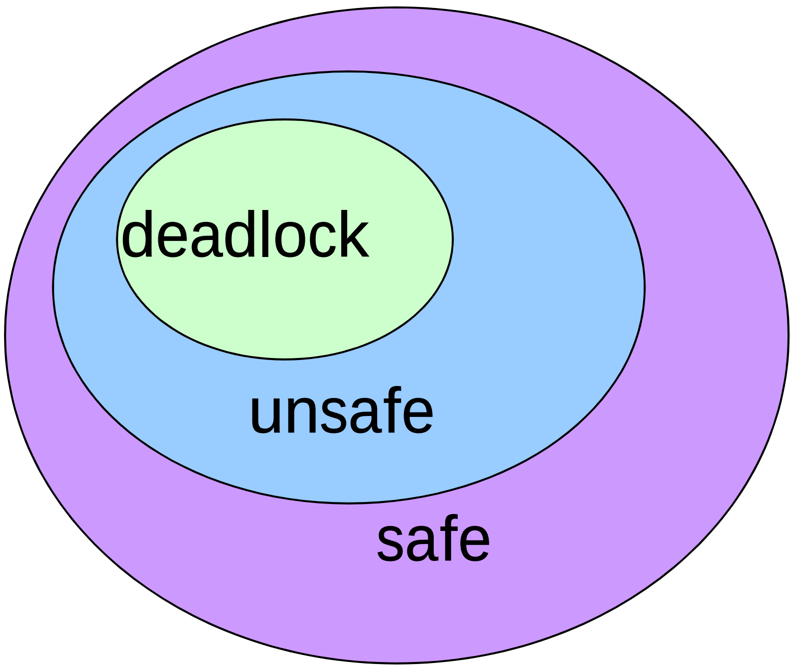
- To attack the circular wait condition:
  - Assign each resource a priority
  - Make processes acquire resources in priority order
- Two processes need the printer and the scanner, both must acquire the printer (higher priority) before the scanner
- This is a common form of deadlock prevention
- A problem: sometimes forced to relinquish a resource that you thought you had locked up
- A problem: sometimes (often?) impossible to assign a global, total order on resources/priorities

Deadlock Avoidance

- Deadlock prevention is often too strict
  - low device utilization
  - reduced system throughput
- If the OS had more information, it could do more sophisticated things to avoid deadlock and keep the system in a safe state
  - "If" is a little word, but it packs a big punch
  - predicting all needed resources a priori is hard

The Banker's Algorithm

- Idea: know what each process *might* ask for
- Only make allocations that leave the system in a *safe* state
- Inefficient



Resource allocation state space

Deadlock Detection

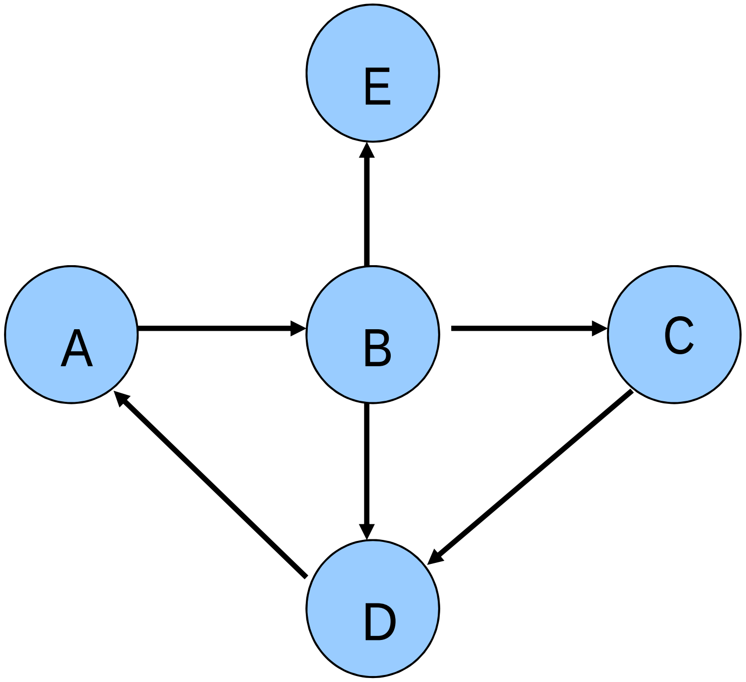
- Build a *wait-for* graph and periodically look for cycles, to find the circular wait condition
- The wait-for graph contains:
  - nodes, corresponding to tasks
  - directed edges, corresponding to a resource held by one task and desired by the other

A waits for **B**

B waits for **D**

D waits for **A**

deadlock!



Deadlock Recovery

- Once you've discovered deadlock, what next?
- Terminate one of the tasks to stop circular wait?
  - Task will likely have to start over from scratch
  - Which task should you choose?
- Take a resource away from a task?
  - Again, which task should you choose?
  - How can you roll back the task to the state before it had the coveted resource?
  - Make sure you don't keep on preempting from the same task: avoid starvation

Ignoring Deadlock

- Not always a bad policy for operating systems
- The mechanisms outlined previously for handling deadlock may be expensive
  - if the alternative is to have a forced reboot once a year, that might be acceptable
- However, for thread deadlocks, your users may not be quite so tolerant
  - “the program only locks up once in a while”

Current practice

- Microsoft SQL Server
  - “The SQL Server Database Engine automatically detects deadlock cycles within SQL Server. The Database Engine chooses one of the sessions as a deadlock victim and the current transaction is terminated with an error to break the deadlock.”
- Oracle
  - As Microsoft SQL Server, plus “Multitable deadlocks can usually be avoided if transactions accessing the same tables lock those tables in the same order... For example, all application developers might follow the rule that when both a master and detail table are updated, the master table is locked first and then the detail table.”

More current practice….

- Windows internals (Linux no different)
  - "… the NT kernel architecture is a deadlock minefield. With the multi-threaded re-entrant kernel there is plenty of deadlock potential."
  - "Lock ordering is great in theory, and NT was originally designed with mutex levels, but they had to be abandoned. Inside the NT kernel there is a lot of interaction between memory management, the cache manager, and the file systems, and plenty of situations where memory management (maybe under the guise of its modified page writer) acquires its lock and then calls the cache manager. This happens while the file system calls the cache manager to fill the cache which in turn goes through the memory manager to fault in its page. And the list goes on."

Summary

- Deadlock is bad!
- We can deal with it either statically (prevention) or dynamically (avoidance and detection)
- In practice, you'll encounter lock ordering, periodic deadlock detection/correction, and minefields