CSE 451: Operating Systems
Winter 2022

Module 10
Deadlock

Gary Kimura

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.

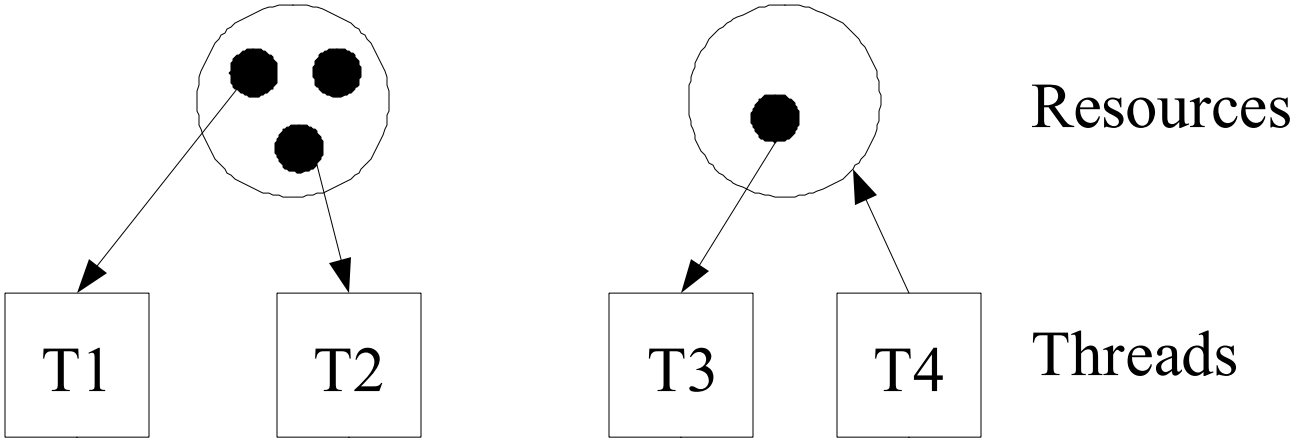
Four conditions must exist for deadlock to be possible

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

We'll see that deadlocks can be addressed by attacking any of these four conditions.

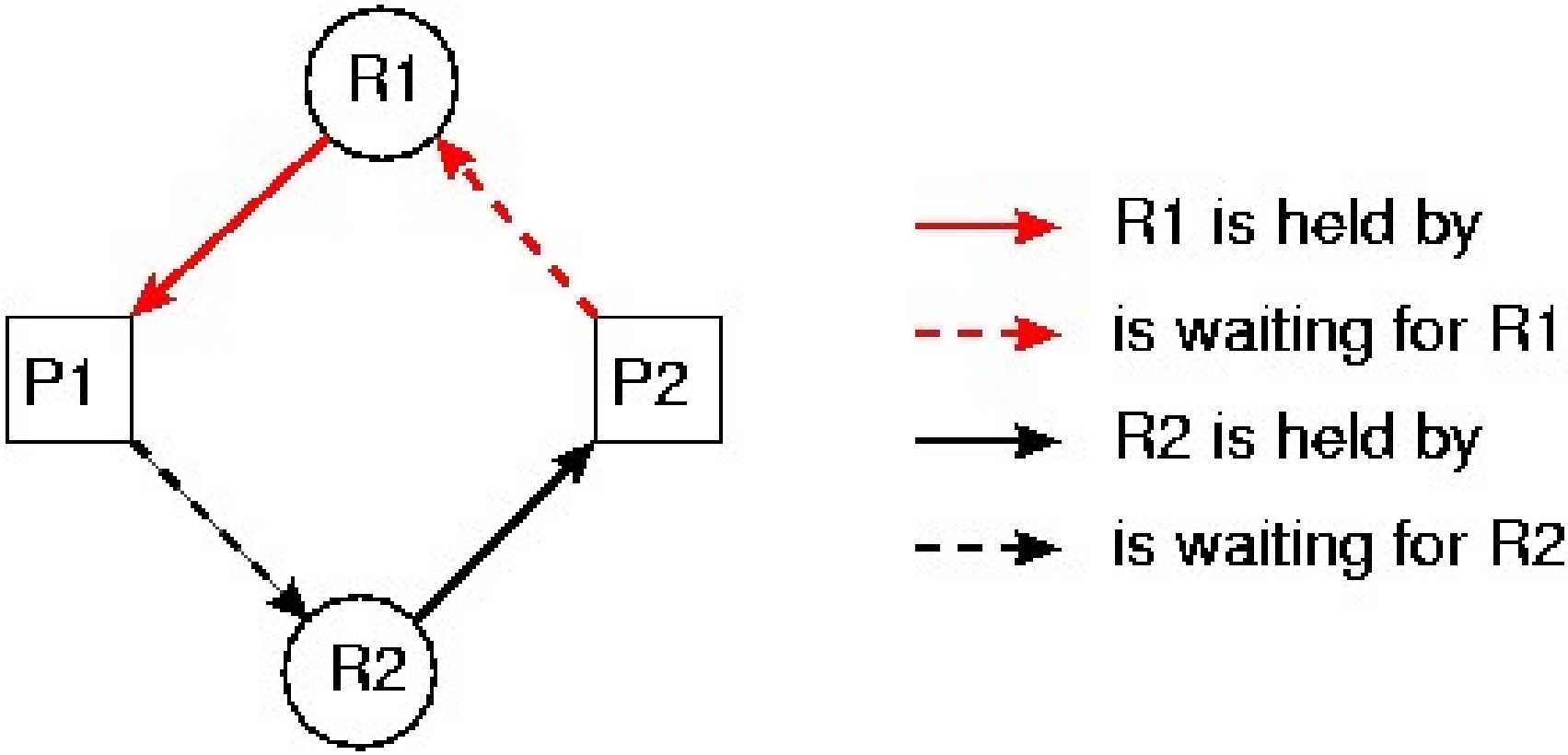
Resource Graphs

- Resource graphs are a way to visualize the (deadlock-related) state of the threads, and to reason about deadlock



- 1 or more identical units of a resource are available
- A thread may hold resources (arrows to threads)
- A thread may request resources (arrows from threads)

Deadlock

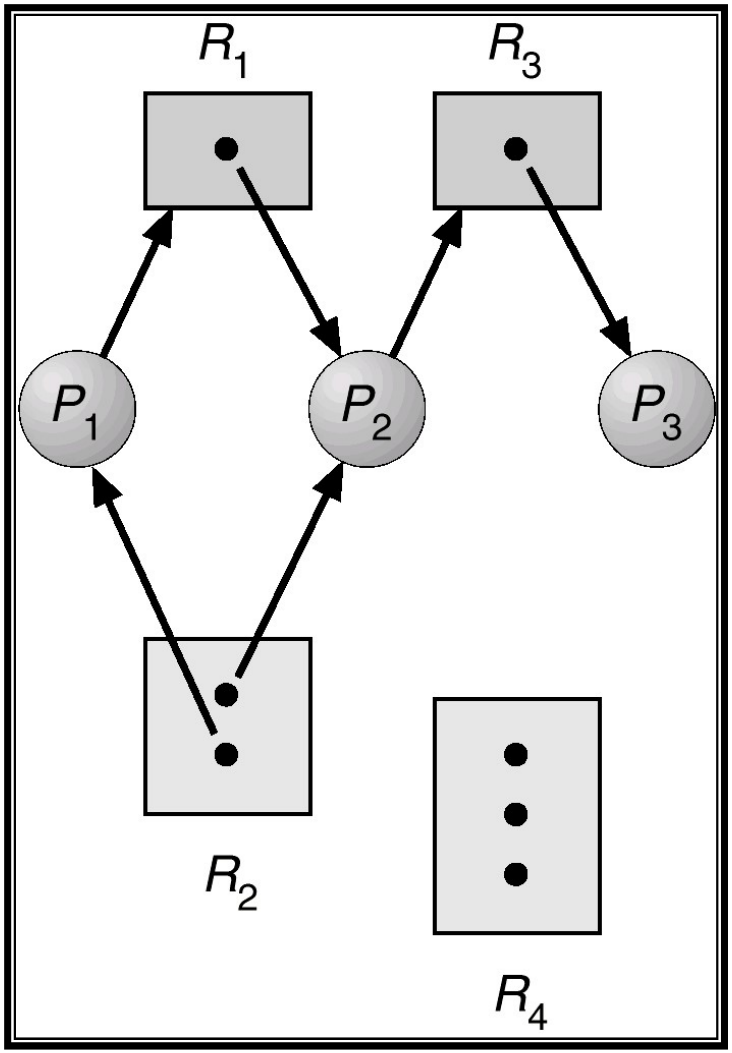


- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

Graph reduction

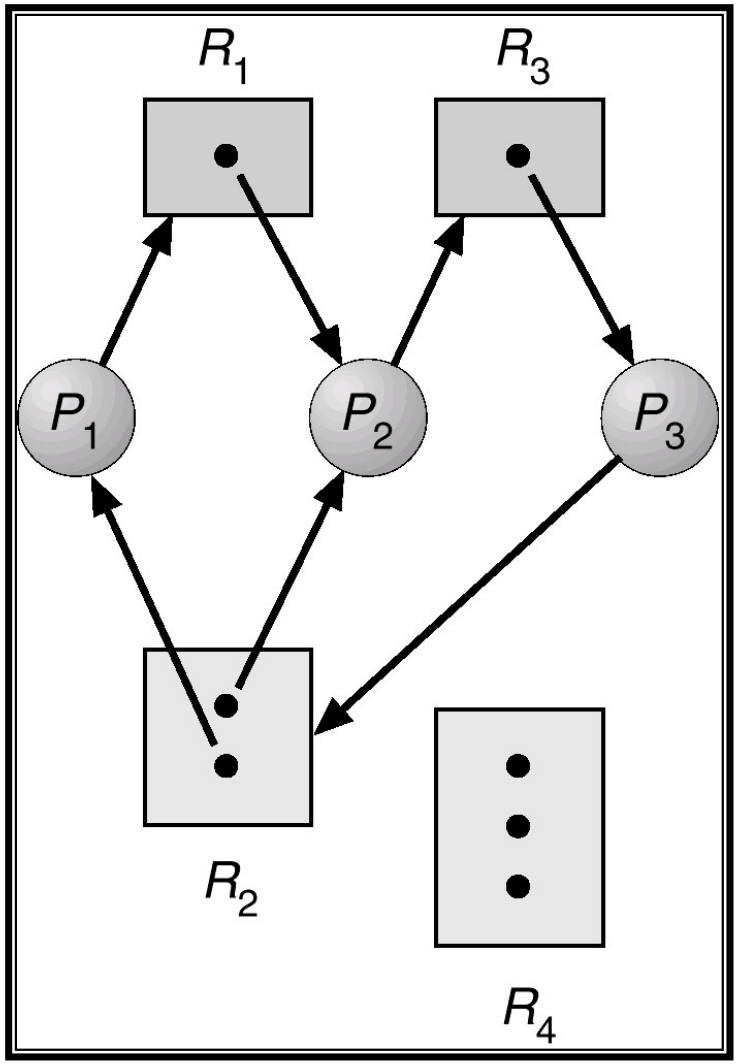
- A graph can be *reduced* by a thread if all of that thread's requests can be granted
  - in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to/from it in the graph are deleted
- Miscellaneous theorems (Holt, Havender):
  - There are no deadlocked threads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

Resource allocation graph with no cycle

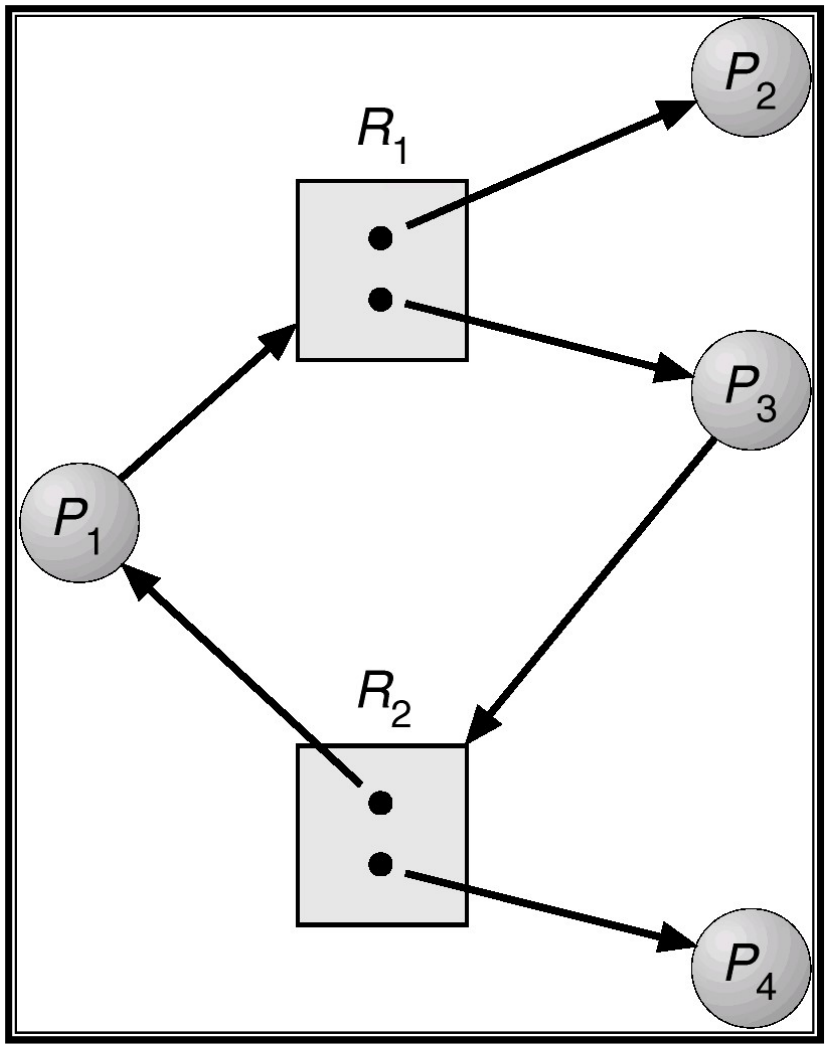


What would cause a deadlock?

Resource allocation graph with a deadlock



Resource allocation graph with a cycle but no deadlock



Handling Deadlock

- Eliminate one of the four required conditions
  - Mutual Exclusion
    - Clearly we're not going to eliminate this one!
  - Hold and Wait
  - No Preemption
  - Circular Wait
- Broadly classified as:
  - Prevention, or
  - Avoidance, or
  - Detection (and recovery)

Prevention

Applications must conform to behaviors guaranteed not to deadlock

- Eliminating hold and wait
  - each thread obtains all resources at the beginning
  - blocks until all are available
    - drawback?
- Eliminating circular wait
  - resources are numbered
  - each thread obtains resources in sequence order (which could require acquiring some before they are actually needed)
    - why does this work?
    - pros and cons?

Avoidance

Less severe restrictions on program behavior

- Eliminating circular wait
  - each thread states its maximum claim for every resource type.
  - system runs the Banker's Algorithm at each allocation request
    - Banker $\Rightarrow$ incredibly conservative
    - if I were to allocate you that resource, and then everyone were to request their maximum claim for every resource, could I find a way to allocate remaining resources so that everyone finished?
  - More on this in a moment…

Detect and recover

- Every once in a while, check to see if there's a deadlock
  - how?
    - Identify stuck threads
    - Look for cycles
    - Don't get spoofed
- If so, eliminate it
  - how?
    - Reboot?
    - Choose a victim to restart

Avoidance: Banker's Algorithm example

- Background
  - The set of controlled resources is known to the system
  - The number of units of each resource is known to the system
  - Each application must declare its maximum possible requirement of each resource type
- Then, the system can do the following:
  - When a request is made
    - pretend you granted it
    - pretend all other legal requests were made
    - can the graph be reduced?
      - if so, allocate the requested resource
      - if not, block the thread until some thread releases resources, and then try pretending again

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.”

- Windows internals (Linux no different)
  - “The Windows 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/or detection)
- In practice, you'll encounter lock ordering, periodic deadlock detection/correction, and minefields
- Lock granularity can make life easier or harder.

Debugging deadlocks
What's in our favor

- Once the system is deadlock, it doesn't go away. That is, you can slowly and painfully walk through all the locks on the system and all the threads on the system and see each thread owns and what it is waiting on.
- This does require the ability to identify the owner(s) of a lock. Having their return address when the acquire the lock also helps.
- Once you draw the graph. You have the deadlock.
- Often the harder part is figuring out how to avoid the deadlock.

Debugging Deadlocks
What didn't work well

- Mutex levels. In theory they avoided deadlocks but in practice they were too cumbersome to use, and deadlocks were still possible when mixed with other kinds of locks.

Debugging Deadlocks
Summary

- In the Windows Kernel deadlock avoidance was the strategy taken.
- Slowly, most deadlocks have been eliminated, but there are probably still some unusual situations where deadlocks can still occur.
- Don’t confuse starvation with deadlocks.
- Using Monitors and Condition Variables does not prevent deadlocks.