Deadlocks

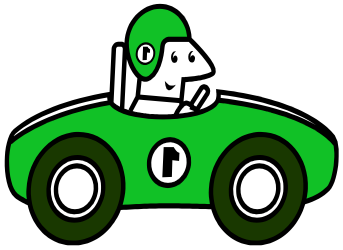
Outline

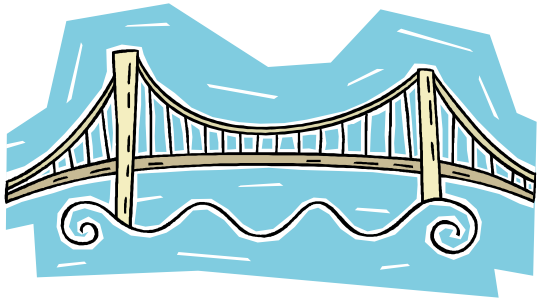
Part I: Prevention and Avoidance

- Discussion of Deadlocks
- Conditions for its occurrence
- Solutions for preventing/avoiding deadlocks

Deadlocks

- Complex multi-threaded programs contain several shared objects where we require multi-object synchronization with many synchronization primitives
- Definition: Deadlock exists among a set of processes if
  - Every process is waiting for an event
  - This event can be caused only by another process in the set
    - Event is the acquire of release of another resource





One-lane bridge

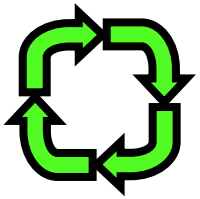
System Model

- There are non-sharable computer resources
  - Printers, Locks, Semaphores, DVD Drives, Tape drives, CPUs
    - Some of these can be made sharable with clever techniques
  - Maybe more than one instance of the resource

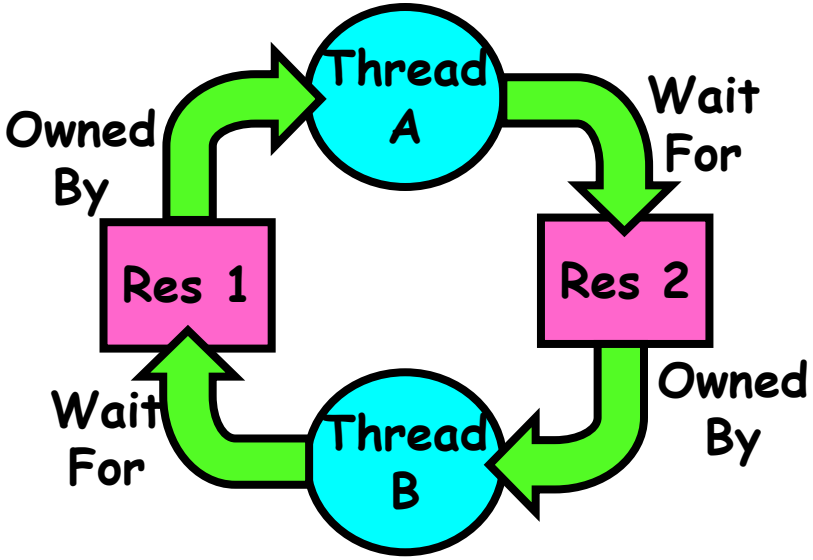
- Processes/threads need access to these resources
  - Acquire resource
    - If resource is available, access is granted
    - If not available, the process is blocked
  - Use resource
  - Release resource
- Undesirable scenario:
  - Process A acquires resource 1, and is waiting for resource 2
  - Process B acquires resource 2, and is waiting for resource 1

  $\Rightarrow$ Deadlock!

Starvation vs Deadlock



- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock $\Rightarrow$ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

For example: Semaphores

```
semaphore:      mutex1 = 1    /* protects resource 1 */
                mutex2 = 1    /* protects resource 2 */
```

```
Process A code:
 {
    /* initial compute */
   P(mutex1)
   P(mutex2)

  /* use both resources */

   V(mutex2)
   V(mutex1)
}
```

```
Process B code:
 {
    /* initial compute */
   P(mutex2)
   P(mutex1)

  /* use both resources */

   V(mutex1)
   V(mutex2)
}
```

Swapinng Ps?

Deadlock or starvation ? Always??

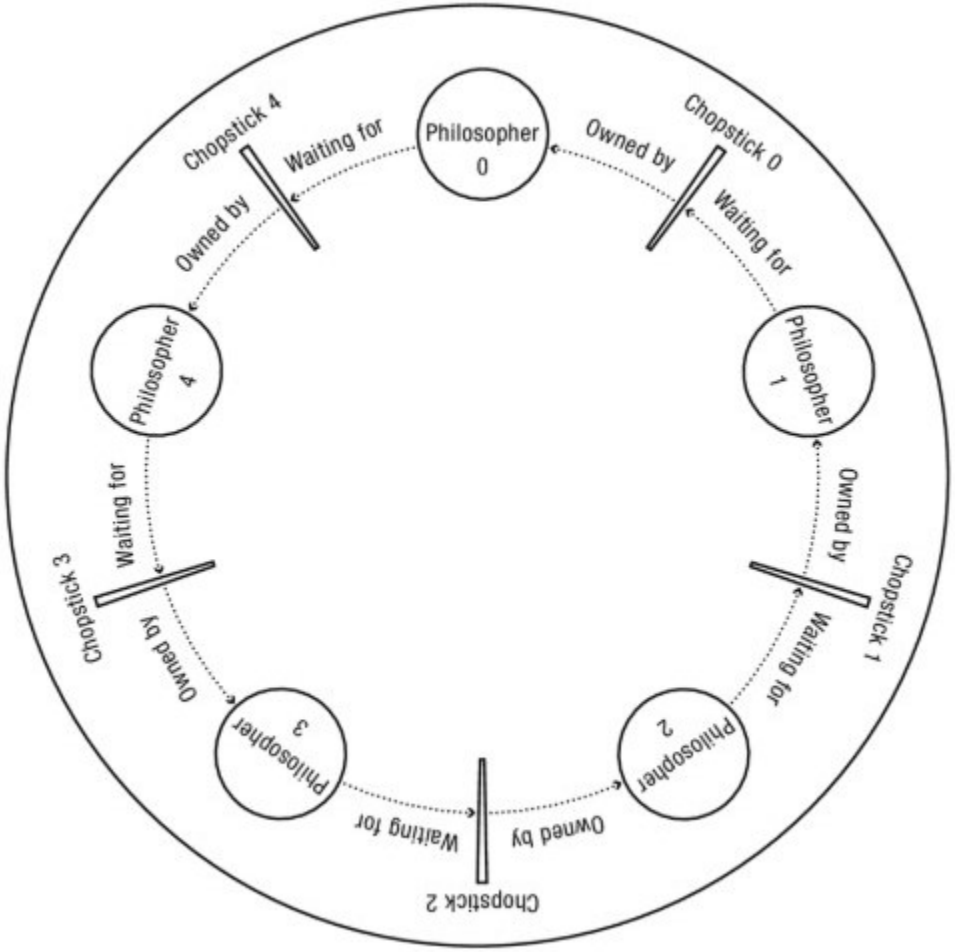
Since testing may not discover deadlocks, it's important to construct systems that are deadlock-free by design.

Four Conditions for Deadlock

- Coffman et. al. 1971
- Necessary conditions for deadlock to exist:
  - **Mutual Exclusion**
    - At least one resource must be held is in non-sharable mode by a process
  - **Hold and wait**
    - There exists a process holding at least one resource, and waiting for additional resources
  - **No preemption**
    - Resources cannot be preempted
  - **Circular wait**
    - There exists a set of processes {P_1, P_2, … P_N}, such that
      - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur

Dining Philosophers meet ALL Four Conditions for Deadlock



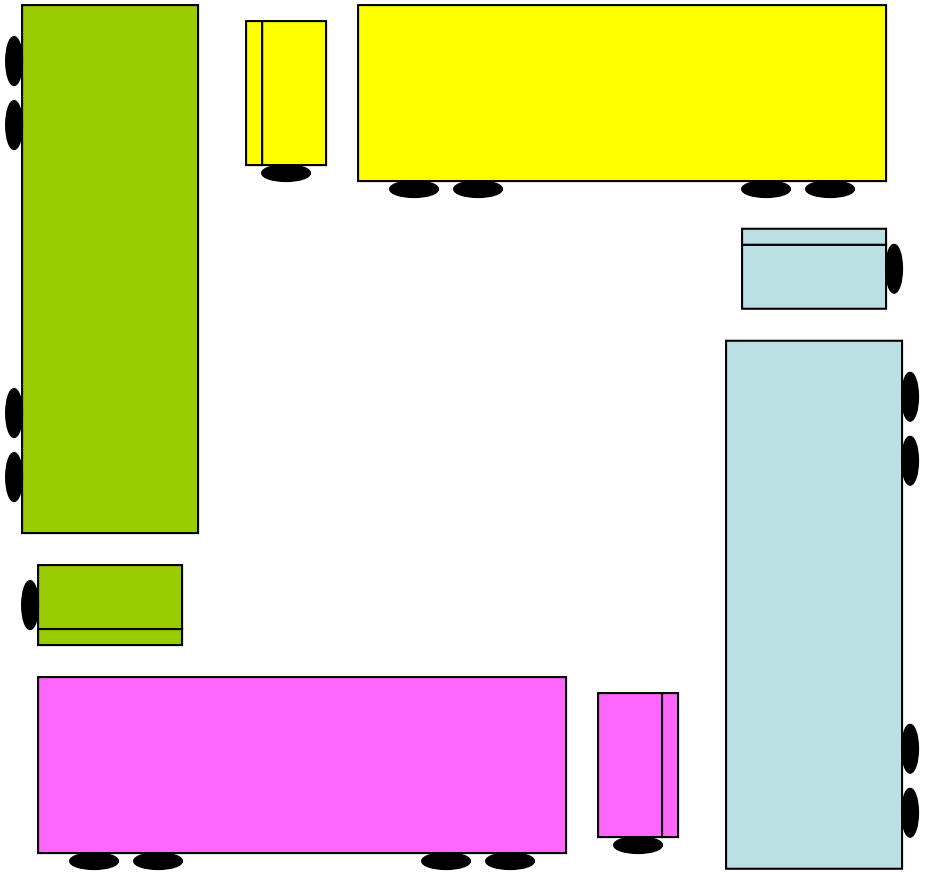
Real World Deadlocks?

- Truck A has to wait for truck B to move

- Not deadlocked

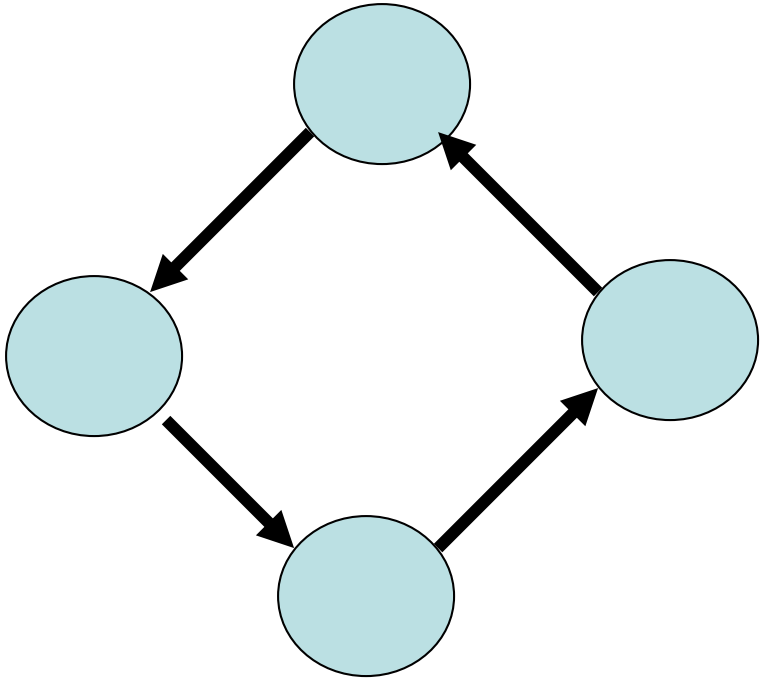
Real World Deadlocks?

- Gridlock



Real World Deadlocks?

- Gridlock



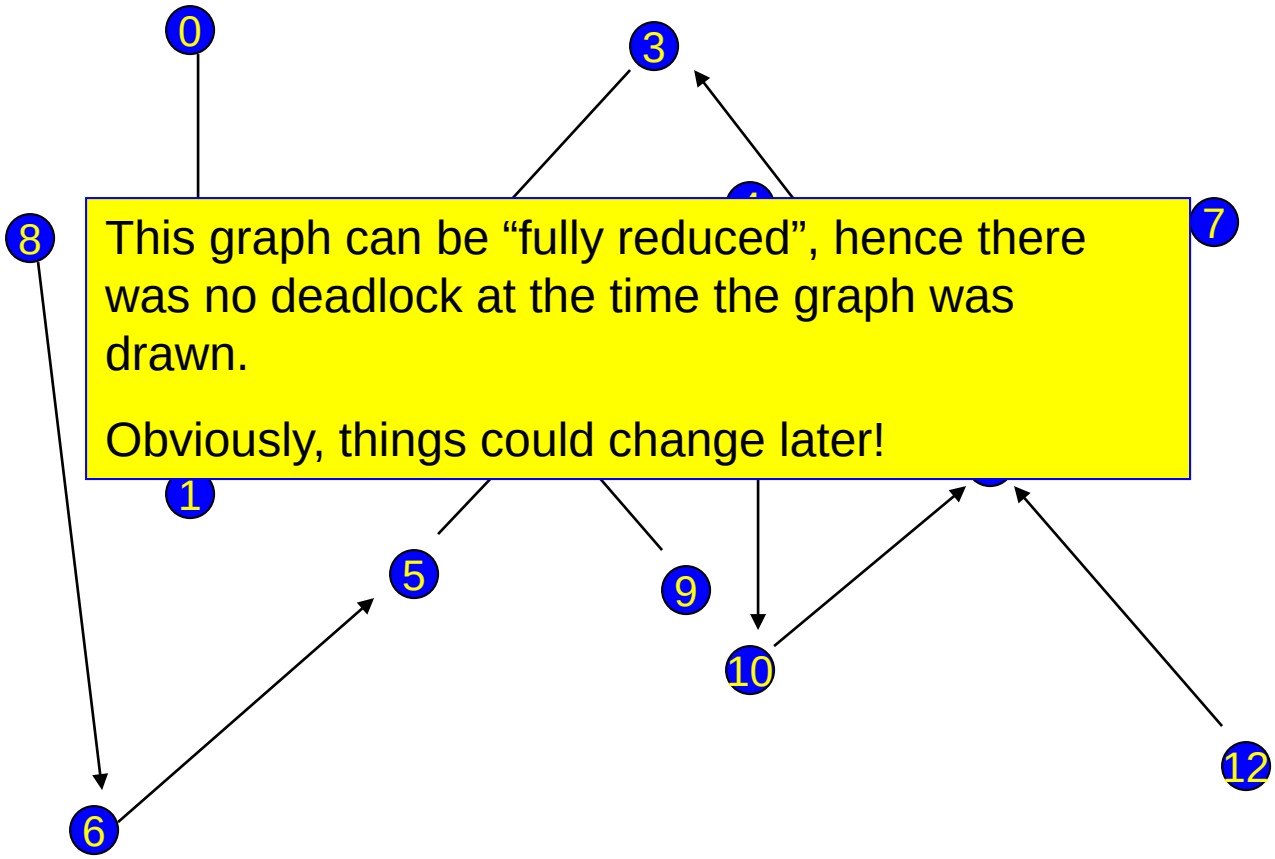
Testing for deadlock

- Steps
  - Collect “process state” and use it to build a graph
    - Ask each process “are you waiting for anything” held by others?
    - Put an edge in the graph between the processes if so
  - We need to do this in a single instant of time, not while things might be changing
- Now need a way to test for cycles in our graph

Testing for deadlock

- One way to find cycles
  - Look for a node with no outgoing edges i.e., a process which is not waiting for resources held by others
  - Erase this node, and also erase any edges coming into it
    - Idea: This was a process others might have been waiting for, but it wasn't waiting for anything else to finish off
  - If (and only if) the graph has no cycles, we'll eventually be able to erase the whole graph!
- This is called a graph reduction algorithm

Graph reduction example

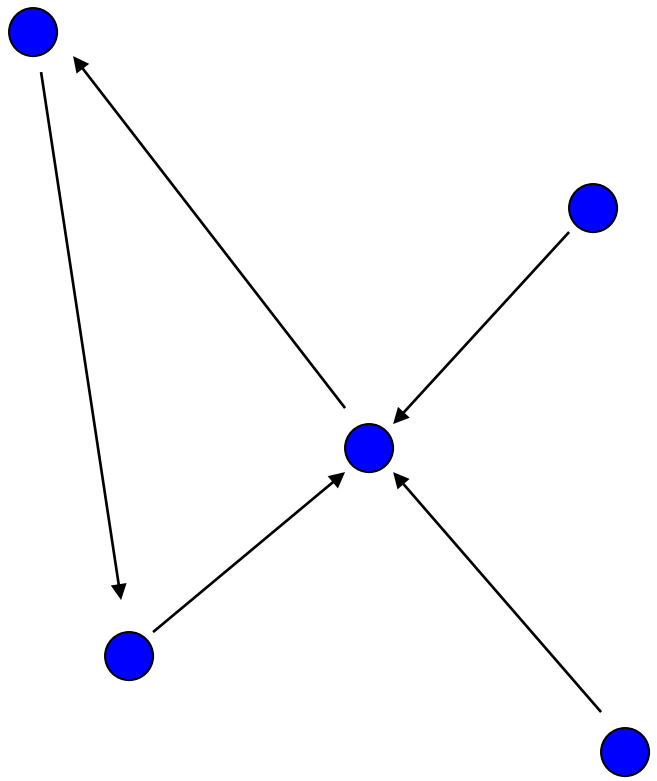


This graph can be “fully reduced”, hence there was no deadlock at the time the graph was drawn.

Obviously, things could change later!

Graph reduction example

- This is an example of an “irreducible” graph
- It contains a cycle and represents a deadlock, although only some processes are in the cycle
- Once deadlock occurs, no of processes in the deadlocked state grows over the time…

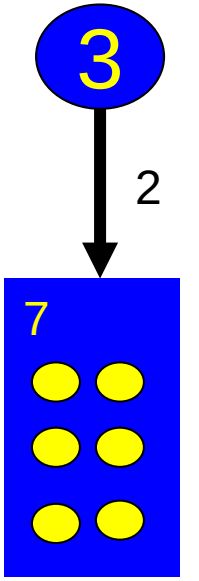


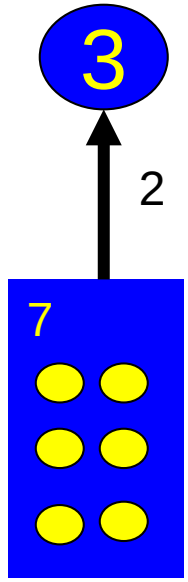
What about “resource” waits?

- When dining philosophers wait for one-another, they don’t do so directly
- Instead, they wait for resources
- Can we extend our graphs to represent resource wait?

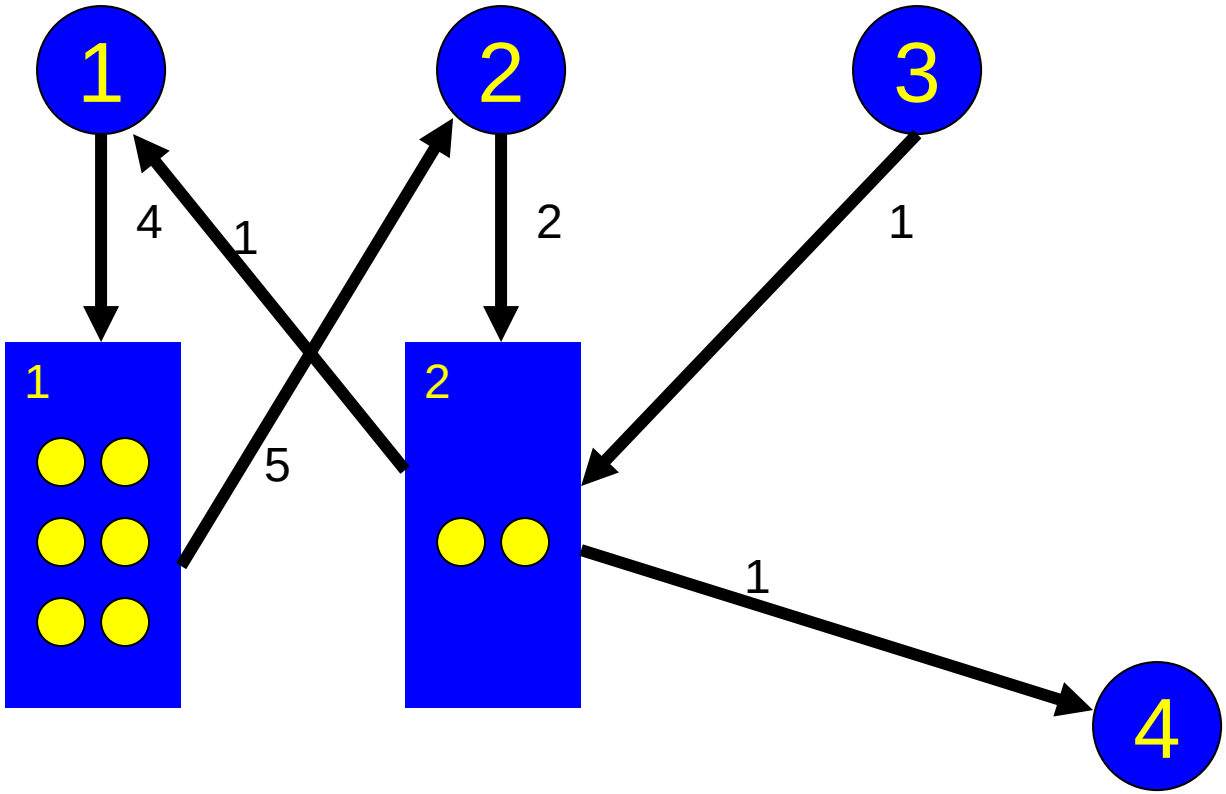
Resource-wait graphs

- We'll use two kinds of nodes
- A process: P_3 will be represented as circle:
- A resource: R_7 will be represented as rectangle:
  - A resource often has multiple identical units, such as "blocks of memory"
  - Represent these as circles in the box
- Arrow from a process to a resource: "I want *k* units of this resource."
  - P_3 wants 2 units of R_7
- Arrow to a process: "I hold *k* units of this resource"
  - P_3 holds 2 units of R_7





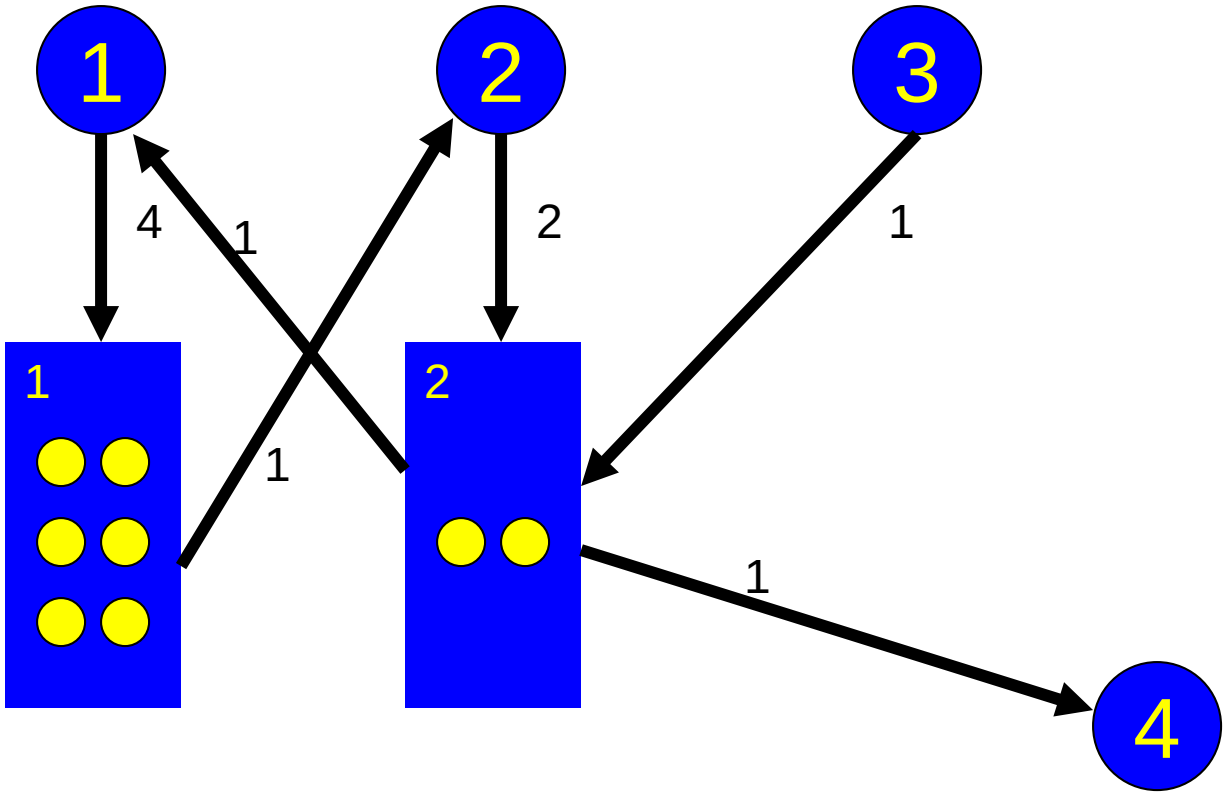
Resource-wait graphs



Reduction rules?

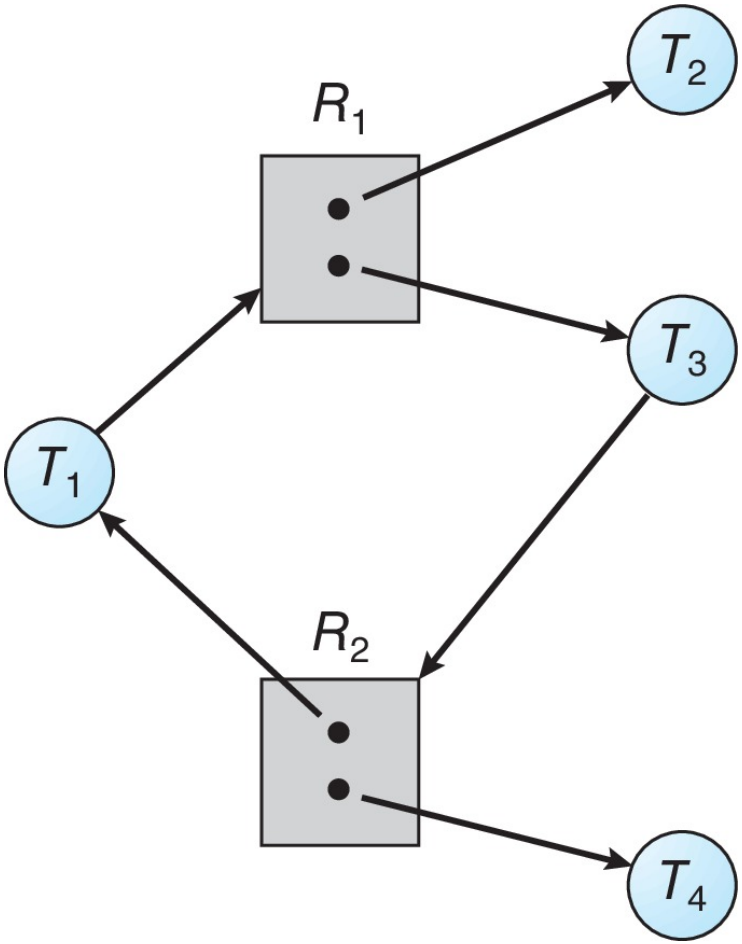
- Find a process that can have all its current requests satisfied (e.g. the “available amount” of any resource it wants is at least enough to satisfy the request)
- Erase that process (in effect: grant the request, let it run, and eventually it will release the resource)
- Continue until we either erase the graph or have an irreducible component. In the latter case we might have identified a deadlock.

This graph is reducible: The system is not deadlocked

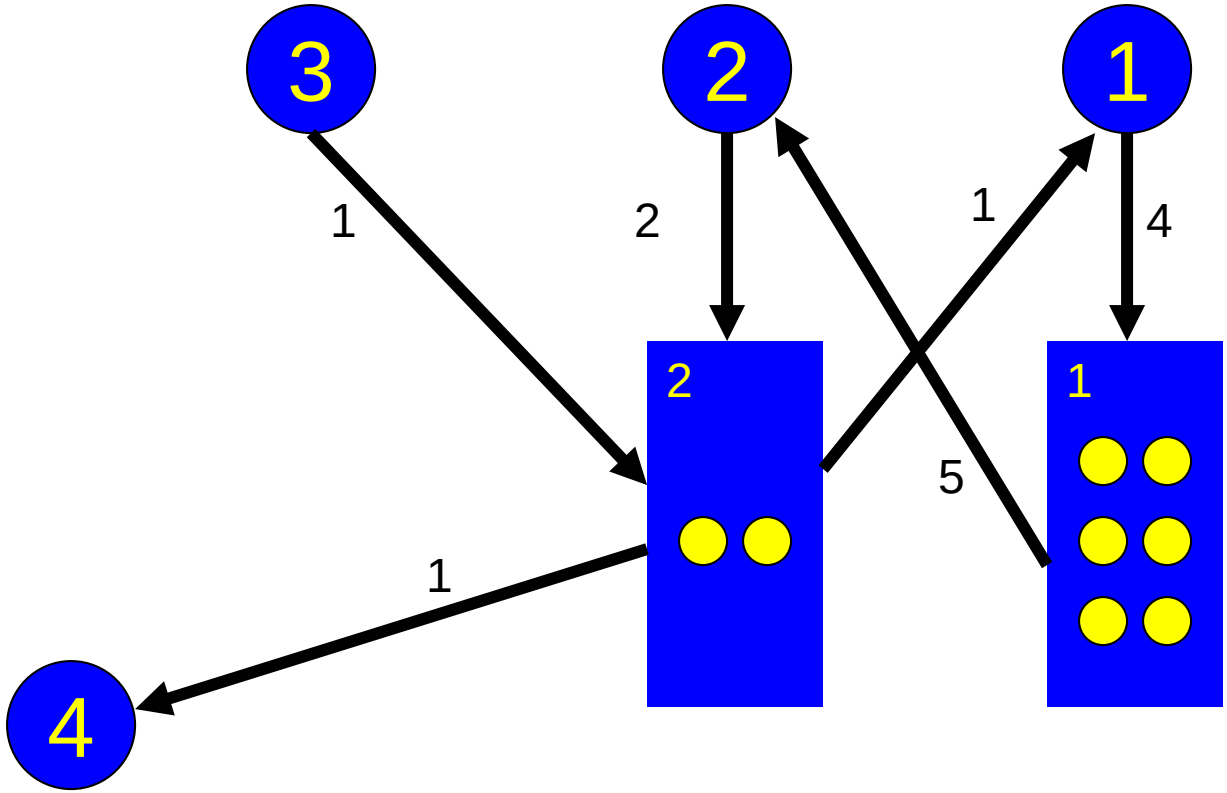


(P4, P3, P1, P2) or (P1, P4, P3, P2) or …

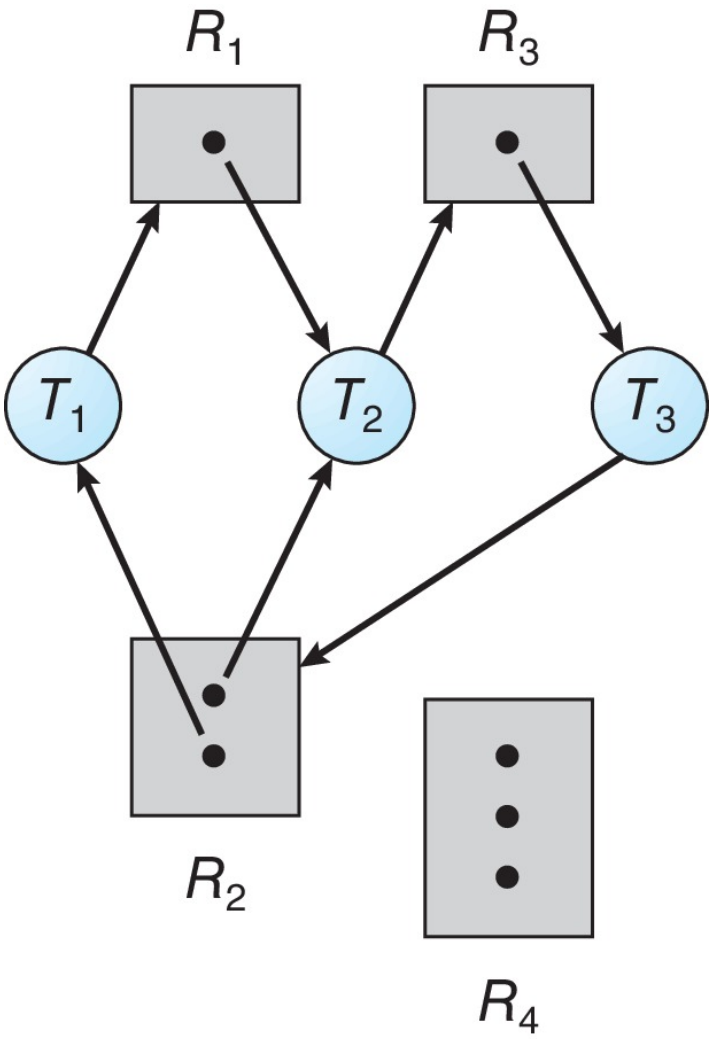
Another Graph With A Cycle But No Deadlock!



This graph is not reducible: The system is deadlocked



This graph is not reducible: The system is deadlocked



Comments

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock
- It isn't common for systems to actually implement this kind of test
- However, we'll later use a version of the resource reduction graph as part of an algorithm called the "Banker's Algorithm"
- Idea is to schedule the granting of resources so as to avoid potentially deadlock states

Some questions you might ask

- Does the order in which we do the reduction matter?
  - Answer: No. The reason is that if a node is a candidate for reduction at step i, and we don't pick it, it remains a candidate for reduction at step i+1
  - Thus eventually, no matter what order we do it in, we'll reduce by every node where reduction is feasible

Some questions you might ask

- If a system is deadlocked, could this go away?
  - No, unless someone kills one of the threads or something causes a process to release a resource
  - Many real systems put time limits on “waiting” precisely for this reason.
    - When a process gets a timeout exception, it gives up waiting and this also can eliminate the deadlock
  - But that process may be forced to terminate itself because often, if a process can’t get what it needs, there are no other options available!

Some questions you might ask

- Suppose a system isn't deadlocked at time T.
- Can we assume it will still be free of deadlock at time T+1?
  - No, because the very next thing it might do is to run some process that will request a resource…
    - … establishing a cyclic wait
    - … and causing deadlock

Methods for Handling Deadlocks

1. Reactive Approaches:
   - Periodically check for evidence of deadlock
     - For example, using a graph reduction algorithm
   - Then need a way to recover
     - Could blue screen and reboot the computer
     - Could pick a “victim” and terminate that thread
       - But this is only possible in certain kinds of applications
       - Basically, thread needs a way to clean up if it gets terminated and has to exit in a hurry
       - Often thread would then “retry” from scratch
   - Despite drawbacks, database systems do this

Dealing with Deadlocks

2. Proactive Approaches:
   - Deadlock Prevention
     - OS ensures that one of the 4 necessary conditions can't hold
     - …. This will prevent deadlock from occurring by constraining how requests for resources are made
   - Deadlock Avoidance
     - OS carefully allocates resources based on future knowledge of resource requests by processes
     - So, deadlocks are avoided

Dealing with Deadlocks

3. Ignore the problem
   - OS pretends deadlocks will never occur
   - Cheapest solution
   - Ostrich approach… but surprisingly common!
   - Linux and Windows do the same!!
   - Application programs then need to handle the deadlocks

Deadlock Prevention

Deadlock Prevention

- Can the OS prevent deadlocks?
- Prevention: Negate one of necessary conditions
  - Mutual exclusion:
    - Make resources sharable
    - Not always possible (spooling, locks?)
  - Hold and wait
    - Do not hold resources when waiting for another

    $\Rightarrow$ Request all resources before beginning the execution
    - Processes do not know what all they will need
    - Starvation (if waiting on many popular resources)
    - Low utilization (Need resource only for a bit)
    - Alternative: Release all resources before requesting anything new
      - Still has the last two problems

Deadlock Prevention

- Prevention: Negate one of necessary conditions
  - No preemption:
    - Make resources preemptable (2 approaches)
    - 1: Preempt requesting processes' current resources if all new resources are not available and let it re-acquire old and newly requesting resources
    - 2: Preempt resources of already waiting processes with the resources to satisfy request of current process
    - Good when easy to save and restore state of resource
      - CPU registers, memory space
      - Not good for printers, tape drives, locks
    - Bad if in middle of critical section and resource is a lock
  - Circular wait:
    - Impose partial ordering on resources, request them in order

Breaking Circular Wait

- Order resources (R1, R2, …) based on 1-to-1 F: R→N
- Acquire resources in strictly increasing/decreasing order of enumeration
- Protocol for increasing scenario:
  - A process holding resource Ri can request Rj iff F(Rj) > F(Ri)
  - Or a process requesting Rj must have released all Ris such that F(Ri) >= F(Rj)
  - When requests to multiple instances of same resource, make the request a single operation

Breaking Circular Wait

- Ordering not always possible
- Low resource utilization
- It is up to application developers to write programs that follow the ordering!
- Witness s/w tool in FreeBSD works as lock-order verifier and gives warnings when locks are acquired out-of-order
  - Example: T1 acquires lock1 and lock2 in that order. Then Witness records this order and gives warning if T2 tries to acquires lock2 first and then lock1 next
- Note that imposing lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically
  - Refer Figure 7.5 in Galvin textbook (9th ed) for an example program: Deadlock example with lock ordering for funds transfer between two accounts

  T1 → transaction(checking account, savings account, 10);

  T2→ transaction(savings account, checking account, 20);

Two phase locking

- *Acquire all* resources before doing any work. If any is locked, *release all*, wait a little while, and retry

```
print_file {
        do{
        release all Resources;
        tryLock(file);
        tryAcquire(printer);
        tryAcquire(disk);
        } while(all not acquired);
        read file from disk;
        print file on printer;
        release all Resources
}
```

- Pro: dynamic, simple, flexible
- Con:
  - Could "spin" endlessly
  - How does cost grow with number of resources?
  - Hard to know what locks are needed a priori

Deadlock Avoidance

Deadlock Avoidance

- The key here is that underlying 4 conditions (Mutex, hold & wait, no preemption, and circular wait) are still there, but you do something more clever to avoid getting into a deadlock
- It requires knowing future information
  - Max resource requirement of each process before they execute
- Can we guarantee that deadlocks will never occur?
- Avoidance Approach:
  - Before granting resource to a process, check if system state is **safe**
  - If the state is safe $\Rightarrow$ no deadlock & grant the requested resource to the process

Safe State

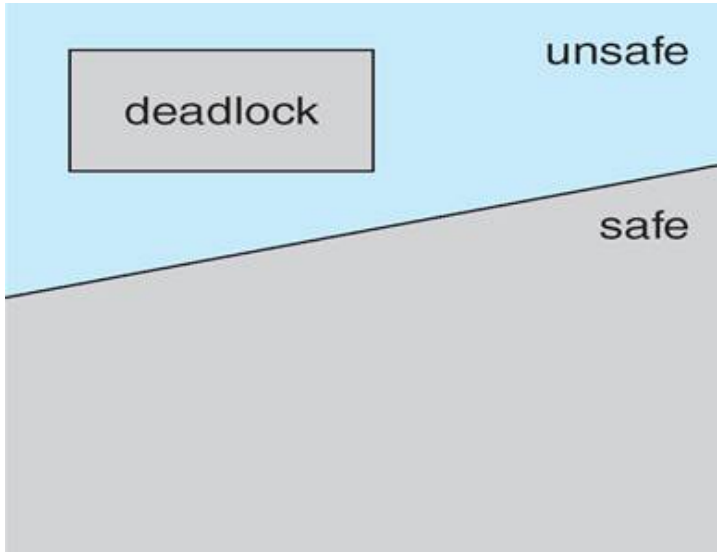
- A state is said to be **safe**, if it has a sequence

  $\{P_1, P_2, \ldots, P_n\}$ of ALL processes, such that for each P_i,

  the resources that P_i can still request can be satisfied by the currently available resources + the resources held by all P_j, where $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- State is safe because OS can definitely avoid deadlock
  - by blocking any new requests until safe order is executed
- This avoids circular wait condition
  - Process waits until safe state is guaranteed

Safe State

- If a system is in **safe state → no deadlocks**
  - For any possible sequence of resource requests, there exists at least one safe sequence of processing the requests that would succeed in granting all pending and future requests
- If a system is in **unsafe state → possibility of deadlock**
  - There is at least one sequence of future requests that leads to deadlock no matter what processing order is tried
- In a deadlocked state, the system has at least one deadlock.
- **Avoidance** ➔ ensure that the system will never enter an **unsafe state**
  - **Conservative approach**



Safe State Example

- Suppose there are 12 tape drives in a system

	max need	current usage	could ask for
P0	10	5	5
P1	4	2	2
P2	9	2	7

3 drives available for granting

- Current state is safe because a safe sequence exists: <p1,p0,p2>
  - p1 can complete with currently available resources (2 of 3)
  - p0 can complete with current (5) + available (5) after p1
  - p2 can complete with current (2) + available: 10 (after p1, p0)

- If p2 requests 1 drive
  - then it must wait to avoid unsafe state.

Deadlock Avoidance Algorithms

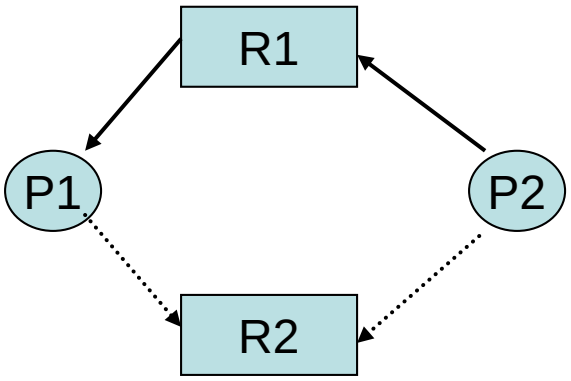
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the Banker's Algorithm

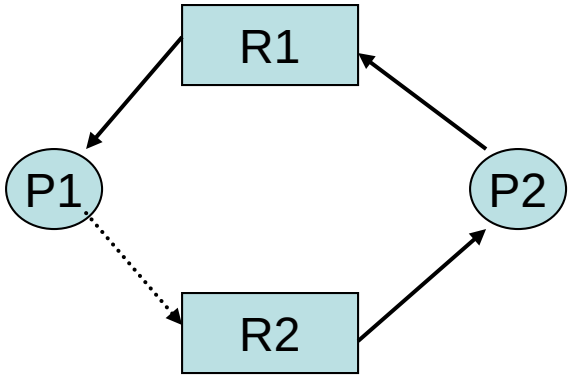
These are less conservative than Acquire-all/Release-all approach seen earlier in case of Two Phase Locking

Res. Alloc. Graph Algorithm

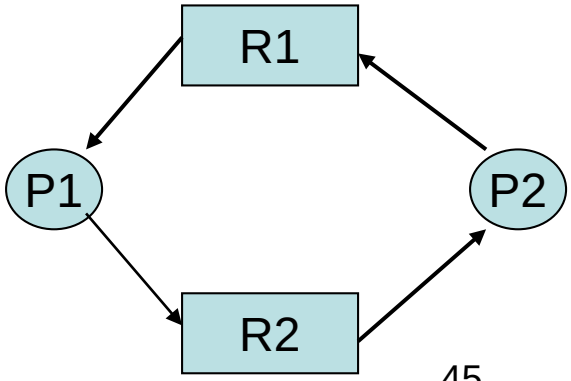
- Deadlock can be described using a *resource allocation graph, RAG*
- Works if only **one** instance of each resource type
- Algorithm:
  - Add a **claim edge**, $P_i \rightarrow R_j$ if P_i can request R_j in the future
    - Represented by a dashed line in graph
  - A request $P_i \rightarrow R_j$ can be granted only if:
    - Adding an **assignment edge** $R_j \rightarrow P_i$ does not introduce cycles
      - Since cycles imply unsafe state



Safe State



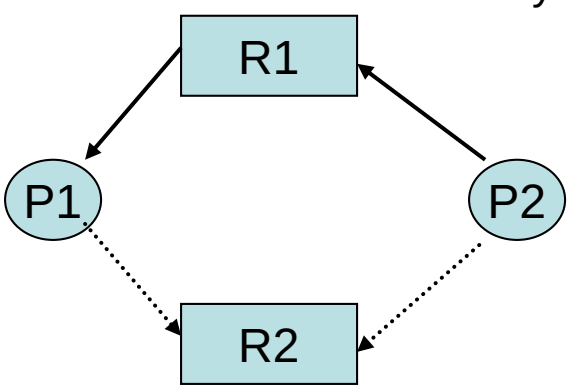
Unsafe State



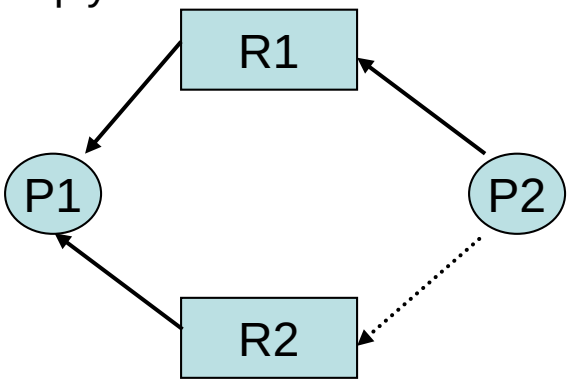
Deadlock State

Res. Alloc. Graph Algorithm

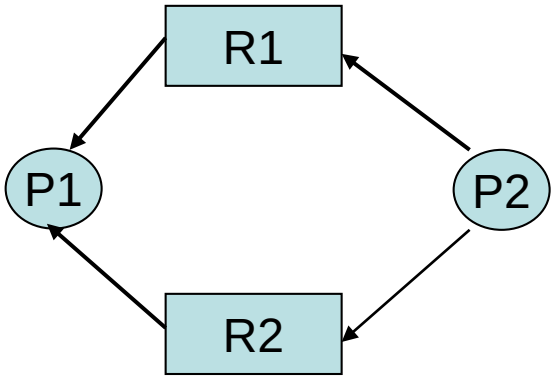
- Deadlock can be described using a *resource allocation graph, RAG*
- Works if only **one** instance of each resource type
- Algorithm:
  - Add a **claim edge**, $P_i \rightarrow R_j$ if P_i can request R_j in the future
    - Represented by a dashed line in graph
  - A request $P_i \rightarrow R_j$ can be granted only if:
    - Adding an **assignment edge** $R_j \rightarrow P_i$ does not introduce cycles
      - Since cycles imply unsafe state



Safe State



Safe State



Safe State

Banker's Algorithm

- Dijkstra developed it to improve on the performance of Acquire-all approach

- Suppose we know the "worst case" aka MAX resource needs of processes in advance
  - A bit like knowing the credit limit on your credit cards. (This is why they call it the Banker's Algorithm)
  - Ensures that bank never allocates its cash reserves in such a way that it could no longer satisfy the needs of all customers.

- Observation: Suppose we just give some process ALL the resources it could need…
  - Then it will execute to completion.
  - After which it will give back the resources.

Banker's Algorithm

- So…
  - A process pre-declares its worst-case (MAX) resource needs
  - Then it asks for what it "really" needs, a little at a time and it may release some of the resources acquired
  - The algorithm decides when to grant requests by assigning resources
- It delays a request to avoid deadlocks unless:
  - It can find a sequence of processes…
  - …. such that it could grant their outstanding need…
  - … so they would terminate…
  - … letting it collect their resources…
  - … and in this way it can execute everything to completion!

Banker's Algorithm

- How will it really do this?
  - The algorithm will just implement the graph reduction method for resource graphs
  - Graph reduction is "like" finding a sequence of processes that can be executed to completion
- So: given a request
  - Build a resource graph
  - See if it is reducible, only grant request if so
  - Else must delay the request until someone releases some resources, at which point can test again

Banker's Algorithm

- Toward right idea:
  - State maximum resource needs in advance, but each job may not require all of these Max no. of resources at a time to proceed
- Banker's algorithm (less conservative):
  - Allocate resources dynamically as and when a job needs
    - Evaluate each request and grant if some ordering of processes is still deadlock free afterward
    - Technique: pretend each request is granted, then run safety algorithm,

      - Check for ([Max_{node}]-[$Alloc_{node}$] ≤ [Avail])
      - Grant request if result is deadlock free (conservative!)
    - Keeps system in a "SAFE" state, i.e. there exists at least one safe sequence {P_1, P_2, … P_n} with P_1 requesting all remaining resources, finishing & releasing its resources, then P_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current processes to be greater than the total resources

Banker's Algorithm

- Decides whether to grant a resource request.
- Data structures:

n: integer	# of processes/threads
m: integer	# of resource types
available[1..m]	available[i] is # of avail resources of type i
max[1..n,1..m]	max demand of each Pi for each Ri
allocation[1..n,1..m]	current allocation of resource Rj to Pi
need[1..n,1..m]	max # resource Rj that Pi may still request

$need_i = max_i - allocation_i$

let request[i] be vector of # of resources of type j (1<=j<=m) process Pi wants

Basic Algorithm

1. If request[i] > need[i] then

   error (asked for too much)

2. If request[i] > available[i] then

   wait (can't supply it now)

3. Resources are available to satisfy the request

   Let's assume that we satisfy the request. Then we would have:

   available = available - request[i]

   allocation[i] = allocation [i] + request[i]

   need[i] = need [i] - request [i]

   Now, check if this would leave system in a safe state.

   if yes, grant the request for resources,

   if no, then leave the state as is and cause process to wait.

Safety Check Algorithm

free[1..m] = available /* how many resources are available */
finish[1..n] = false (for all processes i) /* none finished yet */

Step 1: Find a process i such that finish[i]=false and need[i] <= free
/* find a process that can complete its request now */
if no such i exists, go to step 3 /* we're done */

Step 2: Found a process i:
finish [i] = true /* done with this process */
free = free + allocation [i]
/* assume this process is finished and add its allocation back to the available list */
go to step 1

Step 3: If finish[i] = true for all i, the system is safe. Else Not

Example: Memory Page Allocation

- Consider a system having 8 memory pages
- Three processes
  - A requires MAX of 4 pages
  - B requires MAX of 5 pages
  - C requires MAX of 5 pages

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

this is a safe state: safe sequence <P1, P3, P4, P2, P0>

Suppose that P1 requests (1,0,2)

- add it to P1's allocation and subtract it from Available

Banker's Algorithm: Example

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is still safe: safe seq <P1, P3, P4, P0, P2>

In this new state,
P4 requests (3,3,0)

not enough available resources

P0 requests (0,2,0)

let's check resulting state

Banker's Algorithm: Example

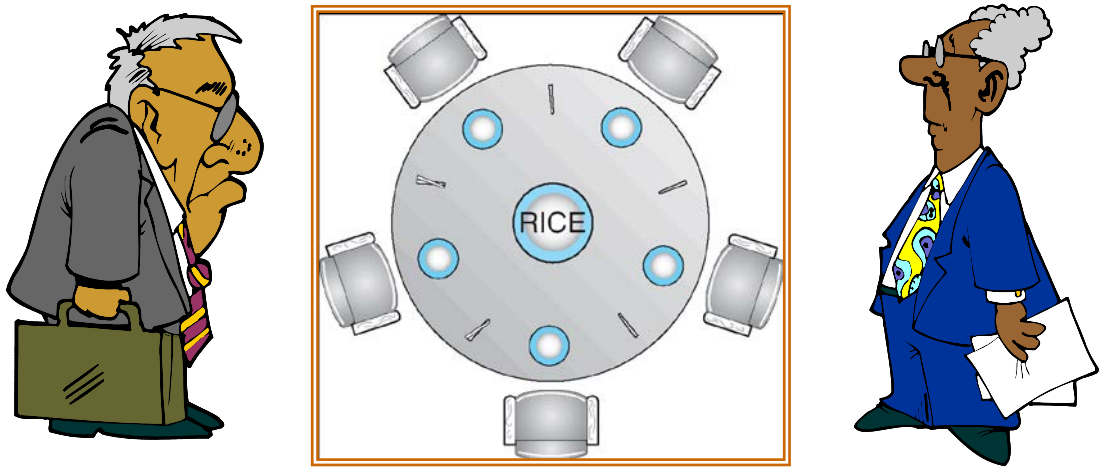
	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	3	0	7	5	3	2	1	0
P1	3	0	2	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

This is unsafe state (why?)

So P0's request will be denied

Problems with Banker's Algorithm?

Banker's Algorithm Example



- Banker's algorithm with dining philosophers where a philosopher can pick any two of available chopsticks from the Center
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - Not last chopstick or
    - Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don't allow if:
    - It's the last one, and no one would have k
    - It's 2nd to last, and no one would have k-1
    - It's 3rd to last, and no one would have k-2
    - …

Deadlocks: Part II
Detection and Recovery

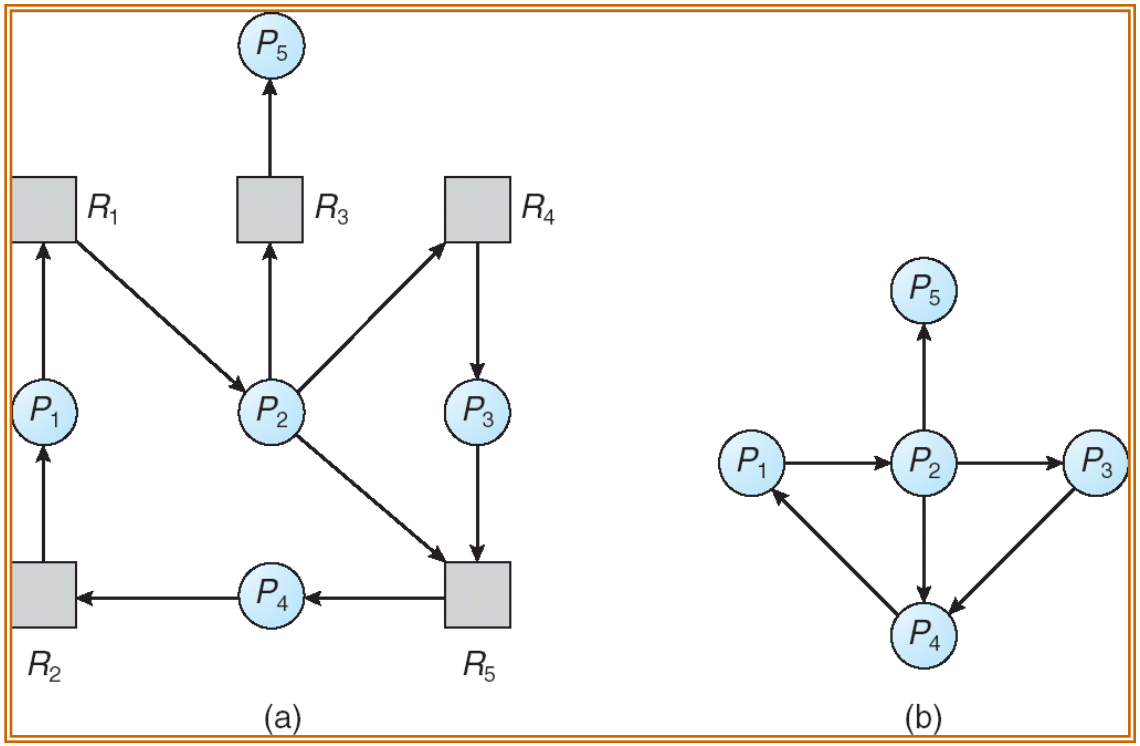
Deadlock Detection & Recovery

- If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - Detection: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - Recovery: untangle the mess.
- Expensive to detect, as well as to recover

Using the RAG Algorithm to detect deadlocks

- Suppose there is only one instance of each resource
- 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
- Use a **wait-for graph:**
  - Collapse resources
  - An edge $P_i \rightarrow P_k$ exists only if RAG has $P_i \rightarrow R_j$ & $R_j \rightarrow P_k$
  - Cycle in wait-for graph $\Rightarrow$ deadlock!

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

2nd Detection Algorithm

- What if there are multiple resource instances?
- Data structures:

```
n: integer    # of processes
m: integer    # of resources

available[1..m]          available[i] is # of avail resources of type i
request[1..n,1..m]       max current demand of each Pi for each Ri
allocation[1..n,1..m]    current allocation of resource Rj to Pi
finish[1..n]              true if Pi's request can be satisfied

let request[i] be vector of # instances of each resource Pi wants
```

2nd Detection Algorithm

1. work[]=available[]
   for all i < n, if allocation[i] ≠ 0 //not yet finished
   then finish[i]=false else finish[i]=true
2. find an index i such that:
   finish[i]=false;
   request[i]<=work
   if no such i exists, go to Step 4.
3. work=work+allocation[i]
   finish[i] = true, go to Step 2
4. if finish[i] = false for some i,
   then system is deadlocked with Pi in deadlock

Example

Finished = {F, F, F, F};

Work = Available = (0, 0, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3	0	0	1
P_4	1	1	1

Request

Example

Finished = {F, F, T, F};

Work = (1, 1, 1);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4	1	1	1

Request

Example

Finished = {F, F, T, T};

Work = (2, 2, 2);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2	2	2	1
P_3			
P_4			

Request

Example

Finished = {F, T, T, T};

Work = (4, 3, 4);

	R_1	R_2	R_3
P_1	1	1	1
P_2	2	1	2
P_3	1	1	0
P_4	1	1	1

Allocation

	R_1	R_2	R_3
P_1	3	2	1
P_2			
P_3			
P_4			

Request

Example

Finished = {T, T, T, T};

Work = (4, 3, 4);

	R_1	R_2	R_3
P_1			
P_2			
P_3			
P_4			

Allocation

	R_1	R_2	R_3
P_1			
P_2			
P_3			
P_4			

Request

When to run Detection Algorithm?

- For every resource request?
- For every request that cannot be immediately satisfied?
- Once every hour?
- When CPU utilization drops below 40%?

Deadlock Recovery

- Killing one/all deadlocked processes
  - Crude, but effective
  - Keep killing processes, until deadlock broken
  - Repeat the entire computation
- Preempt resource/processes until deadlock broken
  - Selecting a victim (# resources held, how long executed)
  - Transactions: Rollback (partial or total) and Retry
  - Starvation (prevent a process from being executed)
- Proceed without a resource!

Java Concurrency Tools

- java.lang.management.ThreadMXBean.findDeadlockedThreads()
  - Part of the JVM management API
  - Can be used to detect deadlocks: returns you the threadIDs of threads currently blocking waiting to enter an object (and “ownable synchronizers”)
  - “It might be an expensive operation” - so when would you run it?
- java.util.concurrent

  .Semaphore (and thus mutex)

  .CountDownLatch & .CountDownLatch

  .Exchanger (Nice - similar in flavor to co-routines from e.g. Scheme/Lisp)
- java.util.concurrent.atomic
  - A toolkit of classes that support lock-free programming (given H/W support):

    AtomicInteger aint = new AtomicInteger(42);

    aint.compareAndSet(whatIThinkItIs, whatIWantItToBe); //No lock needed..
- ..and more. Check out the following:
  - http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
    http://sourceforge.net/projects/javaconcurrenta/
  - https://www.ibm.com/developerworks/java/library/j-jtp10264/

Summary

- Dining Philosophers Problem
  - Highlights need to multiplex resources
  - Context to discuss starvation, deadlock, livelock

- Four conditions for deadlocks
  - Mutual exclusion
    - Only one thread at a time can use a resource
  - Hold and wait
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - Resources are released only voluntarily by the threads
  - Circular wait
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock prevention
  - Prevent one of four necessary conditions for deadlock
- Deadlock avoidance
  - Assesses, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this
- Deadlock detection and recover
  - Attempts to assess whether waiting graph can ever make progress
  - Recover if not

Reading Assignment

- Chapter 6 from OSPP by Anderson et al 2nd Edition
- Chapter 7 from OSC by Galvin et al 9th Edition
- Chapter 2 from MOS by Tanenbaum et al