Operating Systems And Concurrency: Deadlock

Dr Amanda Coles

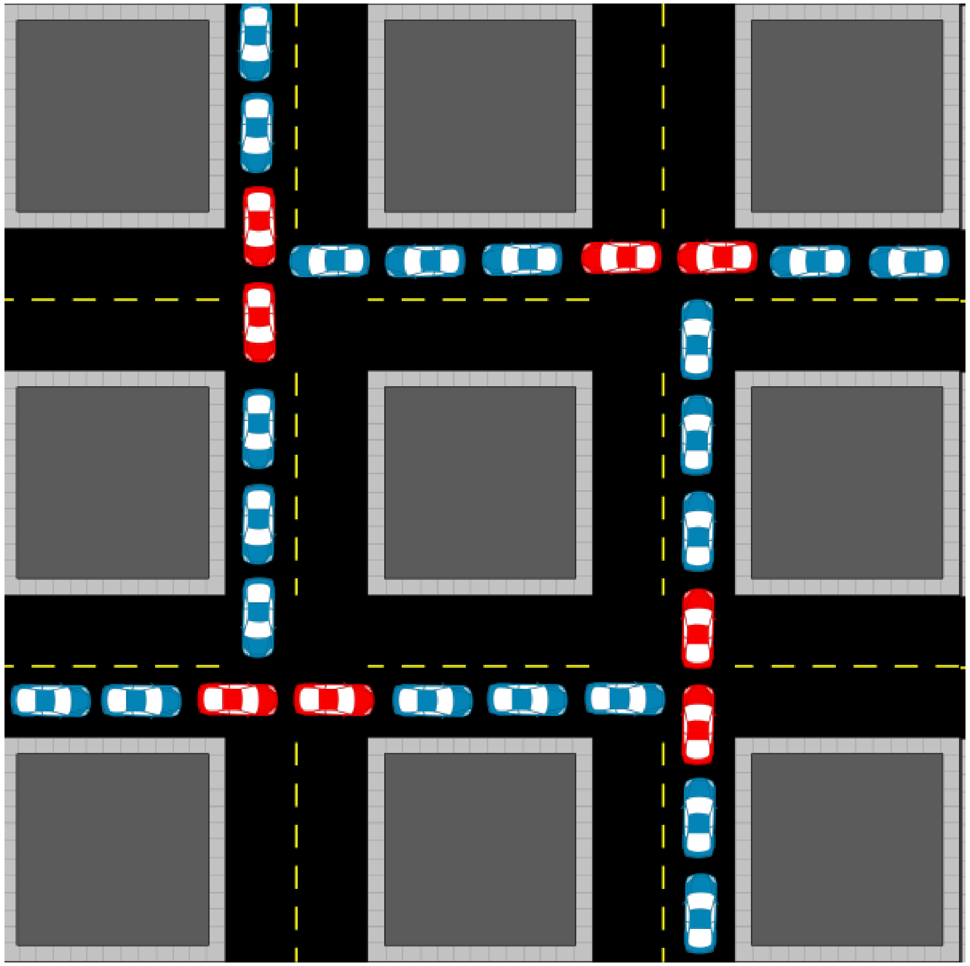
Chapter 7 of "Operating Systems Concepts" Silberschatz, Galvin, Gagne

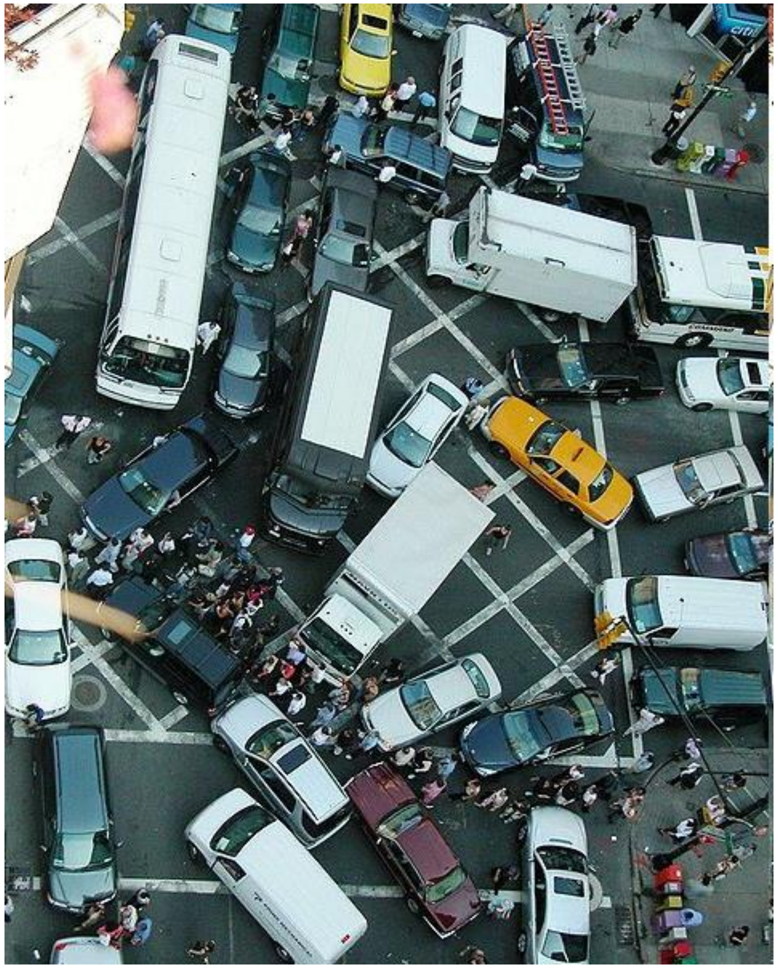
Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

Deadlock

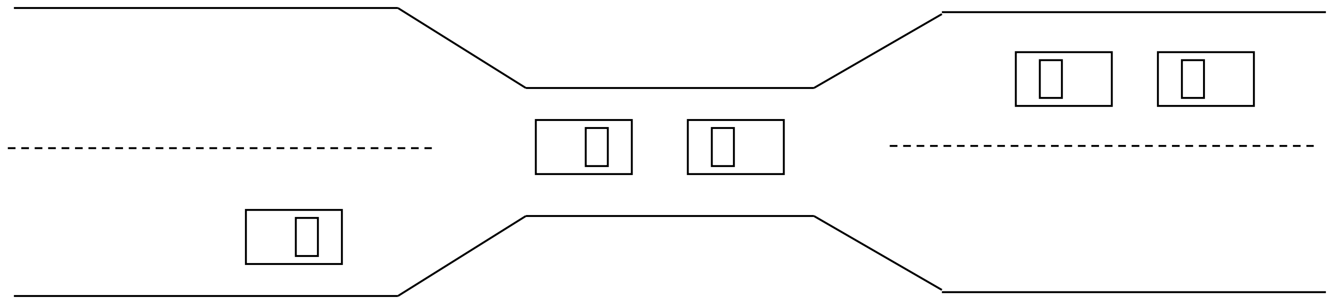
Images Wikipedia





World's worst traffic jam: on the Beijing-Zhangjiakou highway in China traffic congestion stretched more than 100 kilometres (62 mi) from 14th to 26th August 2010 including at least 11 days of total gridlock, it took some drivers 5 days to cross this stretch of highway.

Deadlock



- Single track bridge deadlock:
  - Two cars enter straight on;
  - We saw last time how hard we worked to avoid this.
  - If it happens can be resolved by one of the cars on the bridge reversing, but then, of course the cars behind it might also need to reverse.

In the Computer World

- Three processes want to copy a DVD:
  - Each one acquires the resource of a single DVD drive;
  - Each then waits for another to become available.
  - Of course it won’t do because no process can get one to write and complete it’s operation.
- Similar to the philosophers and their forks.

Third Attempt (Lecture 4)

boolean wantp ← false, wantq ← false	
p	q
`loop_forever`	`loop_forever`
`p1:  non-critical section`	`q1:  non-critical section`
`p2:  wantp ← true`	`q2:  wantq ← true`
`p3:  await wantq = false`	`q3:  await wantp = false`
`p4:  critical section`	`q4:  critical section`
`p5:  wantp ← false`	`q5:  wantq ← false`

- This interleaving gives rise to deadlock:
  - p1,q1,p2,q2
- Not all interleavings do, but if one that does exists then statistically the program will eventually deadlock.

Starvation vs Deadlock

Attempt 4 (Lecture 2)

boolean wantp ← false, wantq ← false	
p	**q**
`loop_forever`	`loop_forever`
`p1:   non-critical section`	`q1:   non-critical section`
`p2:   wantp ← true`	`q2:   wantq ← true`
`P3:   while wantq`	`q3:   while wantp`
`p4:     wantp ← false`	`q4:     wantq ← false`
`P5:     wantp ← true`	`q5:     wantq ← true`
`p6:   critical section`	`q6:   critical section`
`p7:   wantp ← false`	`q7:   wantq ← false`

- Starvation: *if* the scheduler always executes: p3,p4,p4, q3, q4, q5 both processes will starve. There exists a schedule that prevents both processes from entering their CS
- Deadlock: if the scheduler executes p3,p4,q4; q will enter its critical section therefore no deadlock.
- Starvation: there exists an interleaving in which a process doesn't enter its CS; deadlock: from a certain state there is no interleaving that will allow any process to enter the CS.

Deadlock with Semaphores

Semaphore S1 ← (1, Ø), Semaphore S2 ← (1, Ø)	
p	q
`loop_forever`	`loop_forever`
`p1:  non-critical section`	`q1:  non-critical section`
`p2:  wait(S1)`	`q2:  wait(S2)`
`p2:  wait(S2)`	`q2:  wait(S1)`
`p3:  critical section`	`q3:  critical section`
`p4:  signal(S2)`	`q4:  signal(S1)`
`p4:  signal(S1)`	`q4:  signal(S2)`

- This interleaving gives rise to deadlock:
  - p1,q1,p2,q2
- General rule: request semaphores in the same order in every thread.

Java Deadlock

- Taken from Oracle’s Java Tutorials:

- *“Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time.”*

```
public class Friend extends Thread {
    private String name; LinkedList<Friend> friends;
    public Friend(String nameIn, LinkedList<Friend> allFriends) {
      name = nameIn; friends = allFriends;
    }
    public String getFriendName(){
      return name;
    }
    public synchronized void bow(Friend bower) {
      System.out.println(name + ": " + bower.getFriendName()
                          + " has bowed to me!");
       bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
      System.out.println(bower.getFriendName() + ": " + name
                         + " has bowed back to me!");
    }
    public void run(){
        for(Friend f : friends){
            if(!this.equals(f)){
                bow(f);
            }
        }
    }}
```

```
public class Deadlock {
  public static void main(String[] args) {
      friends = new LinkedList();
      Friend alphonse = new Friend("Alphonse");
      Friend gaston = new Friend("Gaston");
      friends.add(alphonse);
      friends.add(gaston);
      alphonse.start();
      gaston.start();
  }
}
```

- Alphonse: `System.out.println(name + ": " + bower.getFriendName() + " has bowed to me!");`
- Gatson: `System.out.println(name + ": " + bower.getFriendName() + " has bowed to me!");`
- Alphonse: `bower.bowBack(this);`
- Gatson: `bower.bowBack(this);`
- Alphonse holds the lock for alphonse, and is waiting for the lock for Gatson; Gatson holds the lock for gatson and is waiting for the lock for Alphonse.

Program Concurrently with Care

- The risk of deadlock is high when writing concurrent programs.
- Deadlock occurs in interaction between threads, so the cause is not always visible in a single part of the code.
- You may have to run your code for some time before it deadlocks if the deadlock is rather improbable.
- The longer your program will be left to run for the more risk there is it will deadlock if there is potential for it to do so.

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

System Model

- We will consider a number of processes (or threads if you prefer);
- A number of resources:
  - Physical resources: e.g. Disk
  - Semaphores/locks.
- Each process requests resources as it needs to use them;
- And releases them when it has finished with them;
- Some tasks may need more than one resource;
- We may have multiple instances of each resource (e.g. DVD writer).

Necessary Conditions for Deadlock

Coffman conditions: all must hold for deadlock to be possible:

- **Mutual exclusion:** only one process at a time can use a(n instance of a) resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set {P_0, P_1, …, P_0} of waiting processes such that:
  - P_0 is waiting for a resource that is held by P_1,
  - P_1 is waiting for a resource that is held by P_2,
  - …,
  - P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

What do Operating Systems Do?

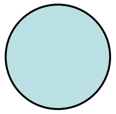
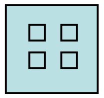
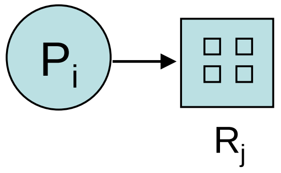
- *“Ostrich”* Algorithm.
- Ignore deadlock.
- Assume that programmers will write correct programs and therefore it won’t be a problem.

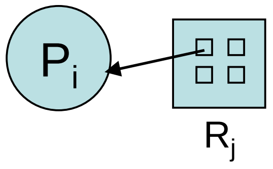
- Why?
  - Deadlock detection and prevention are expensive (time and resource, not money…).
  - Deadlock is rare: if deadlock occurs quickly enough in a given program the programmer will usually notice it and fix it.

Resource Allocation Graphs

- A graph where there is a vertex representing each process, and one representing each resource.
- ***Request edge:*** An edge from a process to a resource means that process has requested an instance of that resource;
- ***Assignment edge:*** An edge from a resource to a process means that process currently holds an instance of that resource.
- When a process gets a resource it has requested the request edge is removed and the assignment edge is added.
- Convention: represent resources with rectangles and processes with circles.

Graph Components

- A process 
- Resource with 4 instances 
- P1 requests an instance of R1. 

- P1 holds an instance of R1 

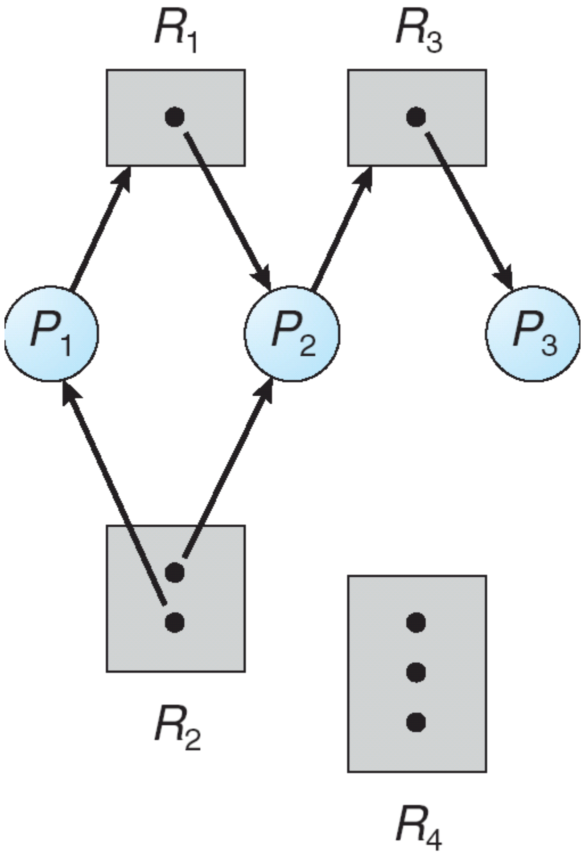
Resource Allocation Graph

Defined by three sets:

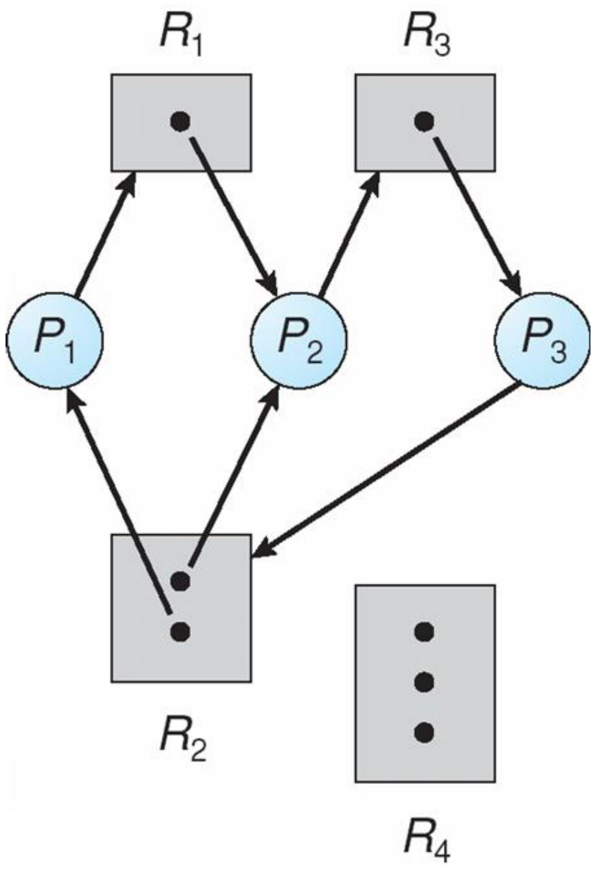
- $P = \{P_1, P_2, P_3\}$,
- $R = \{R_1, R_2, R_3, R_4\}$,
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

And the number of instances of each resource:

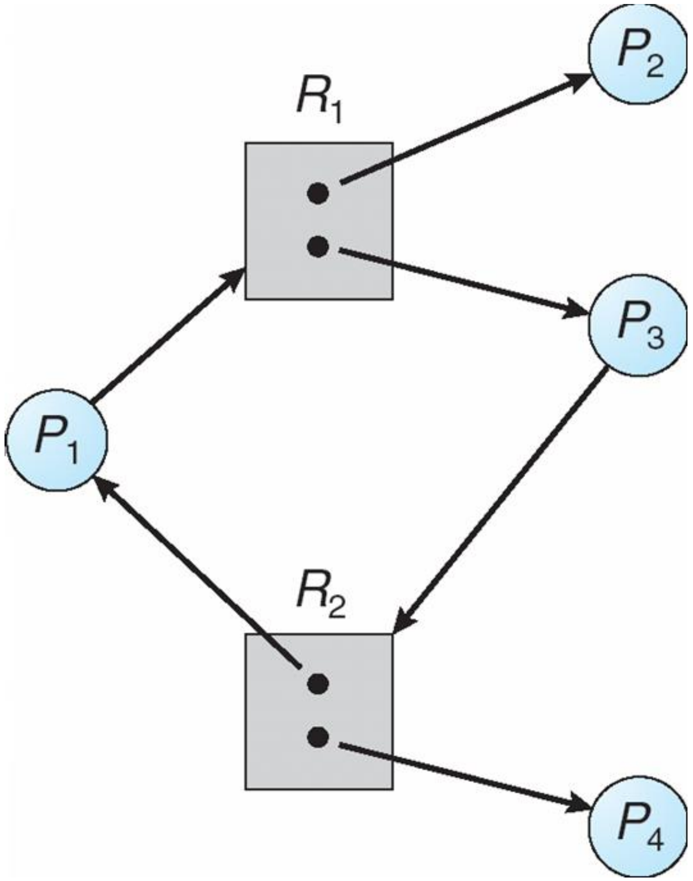
- R_1:1,R_2:2,R:1,R_4:3



Resource Allocation Graph with Deadlock



Resource Allocation Graph with a Cycle but no Deadlock



Cycles in Resource Allocation Graphs

- If there are no cycles in the graph then there is no deadlock.
- If there are cycles in the graph:
  – If there is only one instance of each resource then there is deadlock.
  – If there are multiple instances of resources then there is a possibility of deadlock.

b. The following table shows the resource allocation and request status of a system on which there are four processes (P1, P2, P3 and P4) running sharing three resources (R1, R2 and R3), there is only one instance of each resource and no other processes are running on the system. Empty cells indicate no resources are held/requested by that process as appropriate.

Process	Requested	Holds
P1	R2	R1
P2	R1,R2	
P3		R2,R3
P4	R2, R3	

i. Draw a resource allocation graph for this system.

[8 marks]

ii. Use your graph to determine whether the system is deadlocked, explain how you did this.

[4 marks]

iii. What happens if P3 requests an instance of R1?

[3 marks]

iv. Would your answer to part 2.b.iii change if there were three instances of R1? Explain why or why not.

[4 marks]

May 2015 Question 2b (50/150 marks)

Handling Deadlocks

- Prevention: ensure the system cannot enter a deadlock state.
- Cure: allow deadlock to occur and then recover.
- Ignorance: ostrich approach, ignore deadlock.

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

Deadlock Prevention

- Break ***one*** of the conditions required for deadlock.
- ***Mutual Exclusion:*** not easy since if a resource needs mutual exclusion that must be respected; however we can reduce the risk of a problem by only enforcing mutual exclusion or resources that absolutely require it, rather than on all resources if it is not necessary.
- ***Hold and Wait:*** Guarantee that whenever a process requests a resource it does not hold any others:
  - Process must request and hold all its resources before it can begin;
  - Or process can only request resources when it has none.
  - Problems: low resource utilisation; possible starvation.

Deadlock Prevention II

- ***Pre-emption:***
  - if a process requests a resource that cannot be allocated immediately, then it must immediately release all resources it holds (pre-emption).
  - A list of which resources the process is waiting for is maintained (including the pre-empted ones);
  - The process is only restarted when all the resources it is waiting for are available.

- ***Circular wait:***
  - Impose a total ordering on resource types and make all processes request resources in increasing order.

Deadlock Avoidance

- To avoid deadlock extra information about processes is needed:
  - e.g. max number of resources of each type a process will need.
  - Is it realistic to assume that we know this?

  - We can use the **resource allocation state** of a system to enforce the condition that we will never permit a circular wait.
    - Single instance of each resource, we can check this using a resource allocation graph;
    - Multiple instances of each resource, we must use the Banker's algorithm.

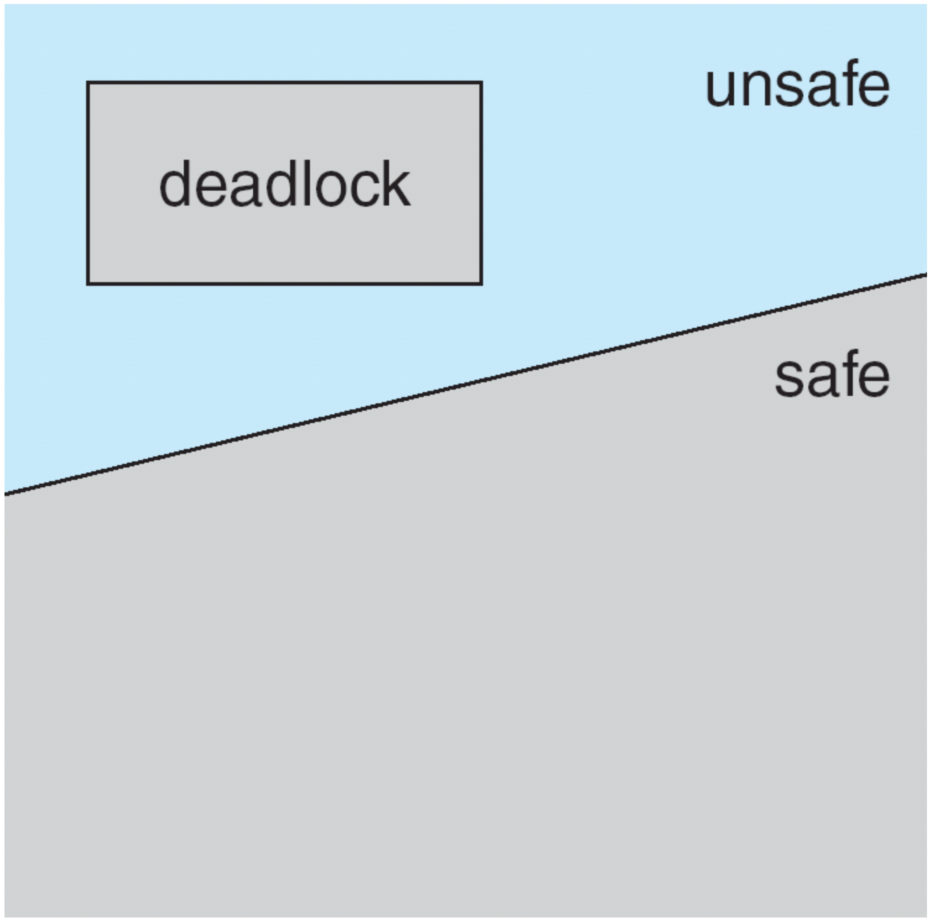
  - **Resource allocation state** is defined by the number of available and allocated resources along with the maximum number of resources each process can need.

Safe State

- Each time a process requests a resource the system must decide if allocation of that resource leaves the system in a ***safe state***.
- System is in ***safe state*** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all processes in the system such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i needs a resource held by P_j ($j > i$) then P_i will not be allowed to acquire resources, instead it has to wait until P_j has finished.
  - Therefore a process can never be holding a resource, but waiting for a process with a higher number to release a resource.
  - This means circular wait cannot occur: P_j is allowed to hold resources and wait for P_i but P_i can't hold resources and wait for P_j.
  - Note this is not a fixed number for each process, but rather at any point processes must be able to be assigned numbers such that the condition holds.

Safe States and Deadlock

- If a system is in a safe state it cannot deadlock;
- If a system is in an unsafe state it may deadlock;
- We can avoid deadlock by preventing unsafe states.



Resource Allocation Graph Method
only works with single resource instances

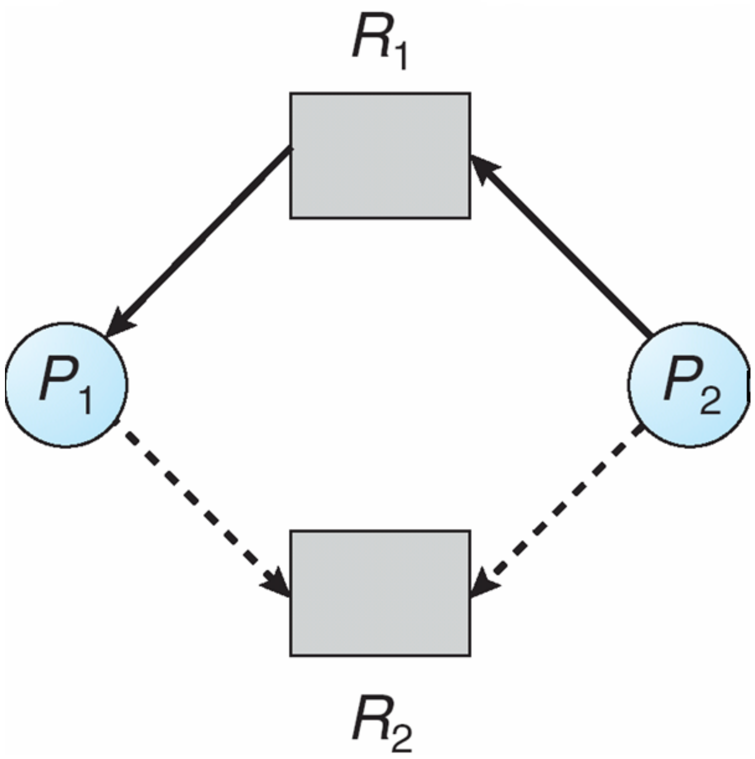
- Add a new type of edge to the RAG: a ***claim*** edge.
- Claim edges $P_i \rightarrow R_j$ indicate that a process P_i can request a resource R_i . That is, the system specified *apriori* that that process may require that resource.
- Illustrated with a dashed line.
- Claim edge becomes an assignment edge when R_i is allocated to P_i.
- Assignment edge reverts to a claim edge when P_i releases R_i.
- All resources must be requested *apriori* i.e. all claim edges are known and put in the graph initially.

Resource Allocation Graph Algorithm

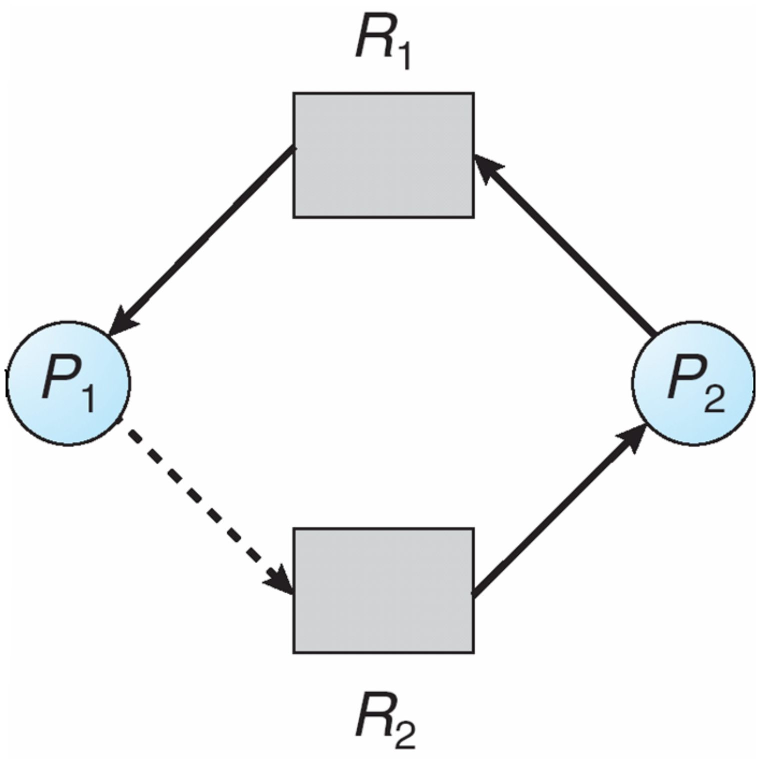
If a process *Pi* requests a resource *Rj.* The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Resource Allocation Graphs

Safe State



Unsafe State



The Banker’s Algorithm

- Developed by Dijkstra.
- Works with multiple instances of resources;
- When a process requests resources it may have to wait.
- Assumption: finite time to exit critical section (i.e. release resources). Same assumption as we had earlier.
- The name comes from an analogy with the way that bankers account for liquidity constraints.
- Not cheap: $O(mn^2)$, n processes, m resources.

Helpful YouTube Clip: https://www.youtube.com/watch?v=oIXj9FfSyxY

Input Data Structures for n Processes and m Resources

Allocation: *n* x *m* matrix. If Allocation[*i,j*] = *k* then *Pi* is currently allocated *k* instances of *Rj*

Max: *n x m* matrix. If *Max* [*i,j*] = *k*, then process *Pi* may request at most *k* instances of resource type *Rj*

Available: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type *Rj* available

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

R_0	R_1	R_2
3	3	2

Basic Idea: Safety Check

1. Find a process for which there are sufficient available resources now to meet its maximum need.
2. Simulate executing the process i.e. release all its resources (make them to available);
3. Go back to 1;
4. If all processes have completed execution the system is safe.
5. If you cannot find such a process, but there are still processes that haven’t executed, then the system is unsafe.

Abuse of Syntax

- Array1 >= Array2 (both same length) means:
  - Array1[i] > Array2[i] for all i.
- Array1 += Array2 means:
  - Array1[i] += Array2[i] for all i;

A Little More Algorithmic

1. Let *Work* and *Finish* be arrays of length *m* and *n*, respectively. Initialize:
   - *Work = Available*
   - *Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1
2. Find and *i* such that both:
   - *Finish* [*i*] = *false and Max[i]-Allocation[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*] == true for all *i*, then the system is in a safe state otherwise it is unsafe.

Example

Allocation

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
3	3	2

Safe Order

Finished

P_0	P_1	P_2	P_3	P_4
F	F	F	F	F

Example

Allocation

	R_0	R_1	R_2
P_0	0	1	0
~~P_1~~	~~2~~	~~0~~	~~0~~
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
5	3	2

Safe Order

P_1

Finished

P_0	P_1	P_2	P_3	P_4
F	T	F	F	F

Example

Allocation

	R_0	R_1	R_2
P_0	0	1	0
~~P_1~~	~~2~~	~~0~~	~~0~~
P_2	3	0	2
~~P_3~~	~~2~~	~~1~~	~~1~~
P_4	0	0	2

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
7	4	3

Safe Order

P_1 P_3

Finished

P_0	P_1	P_2	P_3	P_4
F	T	F	T	F

Example

Allocation

	R_0	R_1	R_2
~~P_0~~	~~0~~	~~1~~	~~0~~
~~P_1~~	~~2~~	~~0~~	~~0~~
P_2	3	0	2
~~P_3~~	~~2~~	~~1~~	~~1~~
P_4	0	0	2

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
7	5	3

Safe Order

P_1 P_3 P_0

Finished

P_0	P_1	P_2	P_3	P_4
T	T	F	T	F

Example

Allocation

	R_0	R_1	R_2
~~P_0~~	~~0~~	~~1~~	~~0~~
~~P_1~~	~~2~~	~~0~~	~~0~~
~~P_2~~	~~3~~	~~0~~	~~2~~
~~P_3~~	~~2~~	~~1~~	~~1~~
P_4	0	0	2

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
10	5	5

Safe Order

P_1 P_3 P_0 P_2

Finished

P_0	P_1	P_2	P_3	P_4
T	T	T	T	F

Example

Allocation

	R_0	R_1	R_2
~~P_0~~	~~0~~	~~1~~	~~0~~
~~P_1~~	~~2~~	~~0~~	~~0~~
~~P_2~~	~~3~~	~~0~~	~~2~~
~~P_3~~	~~2~~	~~1~~	~~1~~
~~P_4~~	~~0~~	~~0~~	~~2~~

Max

	R_0	R_1	R_2
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Work

R_0	R_1	R_2
10	5	7

Safe Order

P_1 P_3 P_0 P_2 P_4

Finished

P_0	P_1	P_2	P_3	P_4
T	T	T	T	T

How do we use This?

- If a process *Pi* requests a resource of type *Rj* :
  - Update allocated as if *Pi* had been allocated the resource.
  - Run the safety check algorithm.
  - If the algorithm says that the state is safe, Pi can be allocated that resource;
  - Otherwise undo the update and do not allocate the resource: *Pi* must wait.

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

Deadlock Detection

- Instead of preventing deadlock we could allow deadlock to happen and then detect and resolve it.
- If there is a single instance of each resource then we can look for cycles in the resource allocation graph;
- If there are multiple instances of each resource, we can use a slight variation of the Banker's algorithm:
  - Instead of using the maximum number of resources a process might need (max), use the resources a process is currently requesting (request).

Banker's Algorithm Deadlock Detection

1. Let *Work* and *Finish* be arrays of length *m* and *n*, respectively. Initialize:
   - *Work = Available*
   - *Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1
   - If Pi has no resources currently allocated Finish [i] = true.
2. Find and *i* such that both:
   - *Finish*[*i*] = *false and 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*] == true for all *i*, then the system is in a safe state otherwise if Finish[i] == false then p_i is deadlocked.

Optimistic Assumption

2. Find and *i* such that both:
   - *Finish*[*i*] = *false and Request[i]* ≤ *Work*
   - If no such *i* exists, go to step 4
3. *Work = Work + Allocation[i]*
   Finish[*i*] = *true*
   go to step 2

- We assume that if Pi has sufficient resources to continue now it will complete.
- But actually it may require to request more resources in the future.
- That is okay, however, since we will detect that deadlock when those resources are requested.

Example

Allocation

	R_0	R_1	R_2
P_0	0	1	0
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Requested

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Work

R_0	R_1	R_2
0	0	0

Finished

P_0	P_1	P_2	P_3	P_4
F	F	F	F	F

Example

Allocation

	R_0	R_1	R_2
~~P_0~~	~~0~~	~~1~~	~~0~~
P_1	2	0	0
P_2	3	0	3
P_3	2	1	1
P_4	0	0	2

Requested

	R_0	R_1	R_2
P_0	0	0	0
P_1	2	0	1
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Work

R_0	R_1	R_2
0	1	0

Finished

P_0	P_1	P_2	P_3	P_4
T	F	F	F	F

c. This part of the question is about the Banker's Algorithm.

i. Use the Banker's algorithm to determine whether the following system is in a safe or unsafe state. If multiple processes are able to run at once, break ties by choosing the one with the lowest ID. If the system is safe, give the safe order in which the processes would execute, if the system is in an unsafe state list the processes that cannot run. The current resource availability in the system is 1,1,1 and the following processes are running:

- P0 max requirement 1,2,3 and allocation 0,0,0.
- P1 max requirement 3,1,3 and allocation 1,0,2.
- P2 max requirement 2,0,0 and allocation 1,0,0.
- P3 max requirement 0,4,2 and allocation 0,3,0.
- P4 max requirement 0,2,0 and allocation 0,0,0.

[6 marks]

ii. The Banker's algorithm can show that a system is in either a safe or an unsafe state. If a system is in a safe state does this mean it cannot deadlock? If a system is in an unsafe state does this mean it will definitely deadlock?

[2 marks]

When to Call Detection Algorithm

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

- Now we have detected a deadlock how can we recover?

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.

Recovery From Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Pre-emption

- Select a process and take resources away from it to break the deadlock.

- Rollback – return to some safe state, restart process from that state

- Selecting a victim – minimize cost

- Starvation – same process may always be picked as victim, include number of rollbacks in cost factor

Overview

- What is Deadlock?
- Necessary Conditions for Deadlock;
- Handling Deadlock:
  - Prevention;
  - Detection;
  - Recovery.