TDDB68 + TDDE47

Lecture 6:
Deadlocks

Klas Arvidsson

Slides by Adrian Pop and Mikael Asplund

Thanks to Simin Nadjm-Tehrani and Christoph Kessler for much of the material behind these slides.

General info

- Teams! Ask questions there!
- Swap lab partner?
  - Good to work with someone on the same level/ambition!
- Office availability
- Read lab pm for lab 3 (and skim 4) before lesson on friday!

Synch example

Synch a bounded buffer i C with

- lock_aquire, lock_release

and

  - sema_down, sema_up

  or

  - cond_wait, cond_signal

Reading guidelines

- Silberschatz et al.,
  - 9th edition: chapter 7 Deadlocks
  - 10th edition: chapter 8 Deadlocks

- Worth checking out:
  - https://deadlockempire.github.io/
  - https://github.com/angrave/SystemProgramming/wiki

Consider interleaving the following

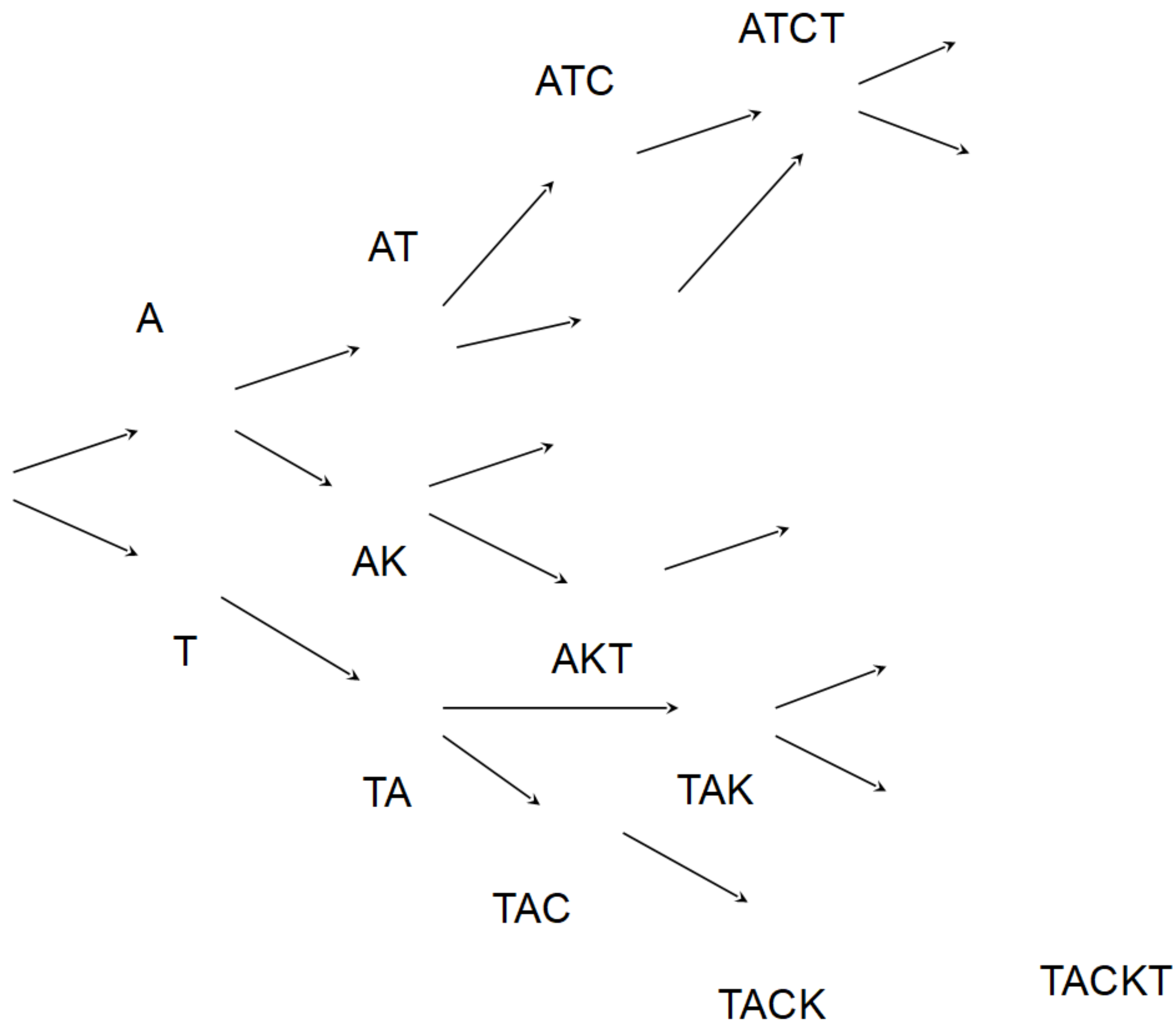
```
Process A

while true {
  print(A)
  print(K)
}
```

```
Process B

while true {
  print(T)
  print(C)
}
```

Program execution



Correctness properties

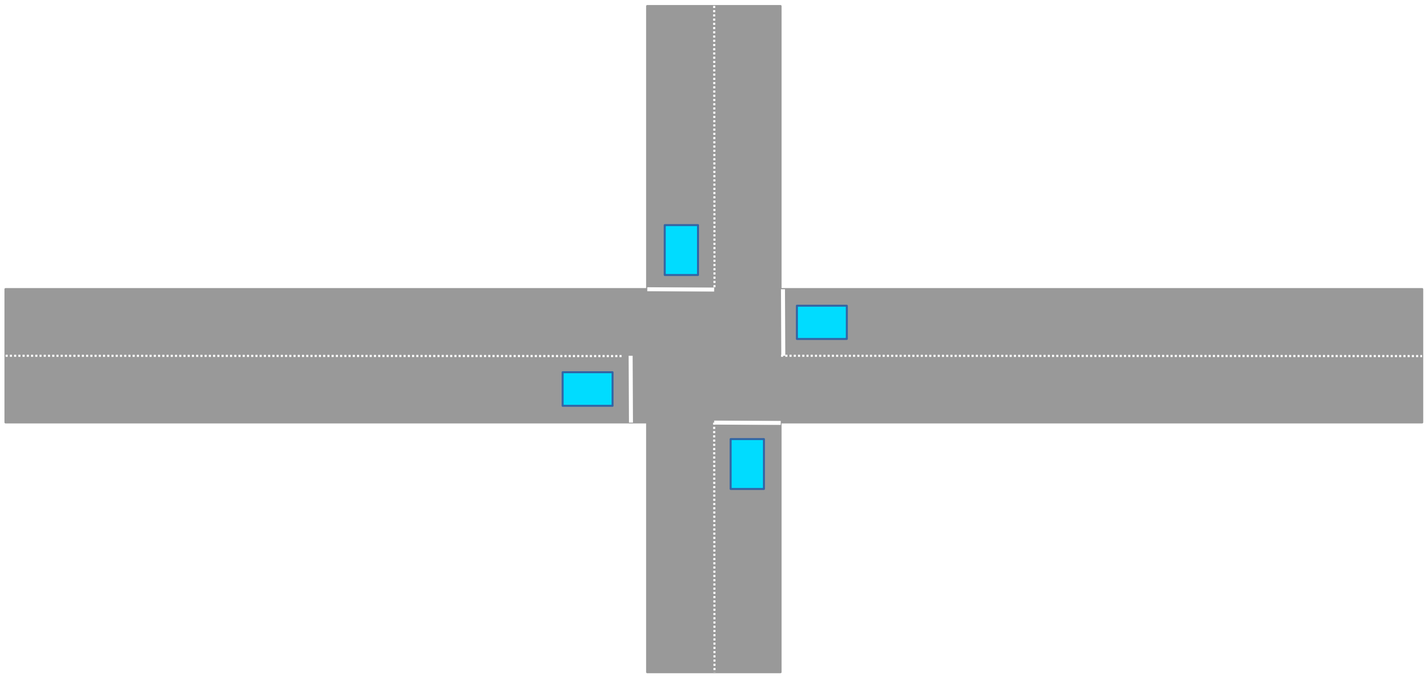
- Safety properties
  - Something **bad** will **not** happen
- Liveness properties
  - Something **good** will happen (eventually)
- More on this way of reasoning in the Software Verification course!

- Ability to understan and reason about code very important for concurrent programming.

Progress

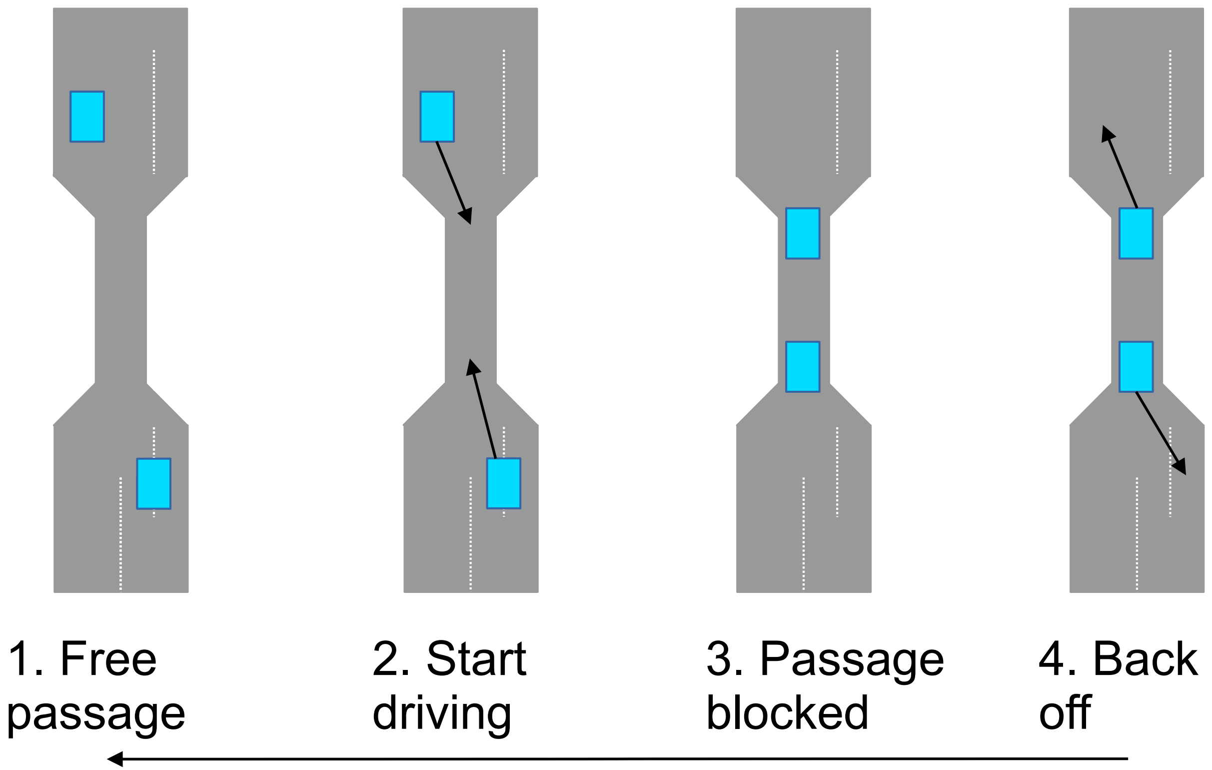
- A form of liveness
- Mathematically defined within a given system model
  - Can be defined on system or process level
  - Typically ensures that if system is in some state s, then it will reach some other state s' where some property P holds.
- Implies freedom from:
  - Deadlock
  - Livelock
  - (Starvation depending on the model)

Deadlock



Deadlock occurs when a group of processes are locked in a circular wait (more on this soon).

Livelock



Livelock occurs when a group of processes are stuck in a loop of actions where they stop each other from progressing

Deadlock-freedom

- Freedom from deadlock is fundamental to any concurrent system
- Necessary but not sufficient for progress!
- Topic for the rest of this lecture

Earlier

- Mutual exclusion and condition synchronization
  - Semaphores
  - Locks and condition variables
  - Concurrent data structures
- Worked well for single resource
- What about multiple resources?

Simple deadlock situation

- Two semaphores
  - S1 for resource R1
  - S2 for resource R2

Process P1:	**Process P2:**
`wait(S2)`	`wait(S1)`
`wait(S1)`	`wait(S2)`
`...`	`...`
`signal(S1)`	`signal(S2)`
`signal(S2)`	`signal(S1)`

Coffman conditions

Four necessary conditions for deadlock:

1. Mutual exclusion

Access to a resource is limited to one (or a limited number of) process(es) at a time

2. Hold & wait

A process may hold a resource and wait for another resource at the same time

Coffman conditions continued

3. Voluntary release

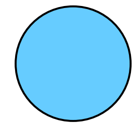
Resources can only be released by a process voluntarily

4. Circular wait

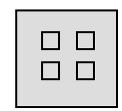
There is a chain of processes where each process holds a resource that is required by another process

Resource-Allocation Graph

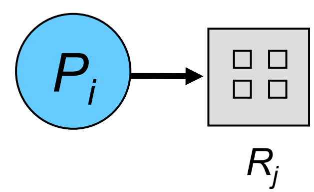
Process



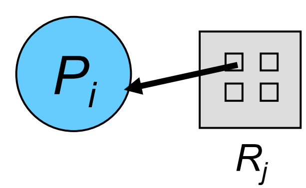
Resource type with 4 instances



P_i requests an *instance* of R_j

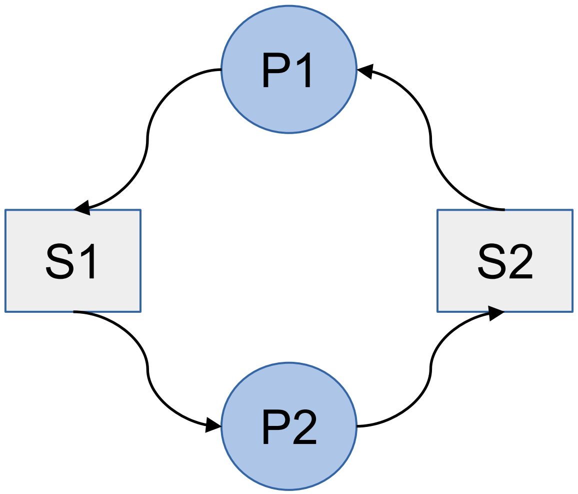


P_i is holding an *instance* of R_j



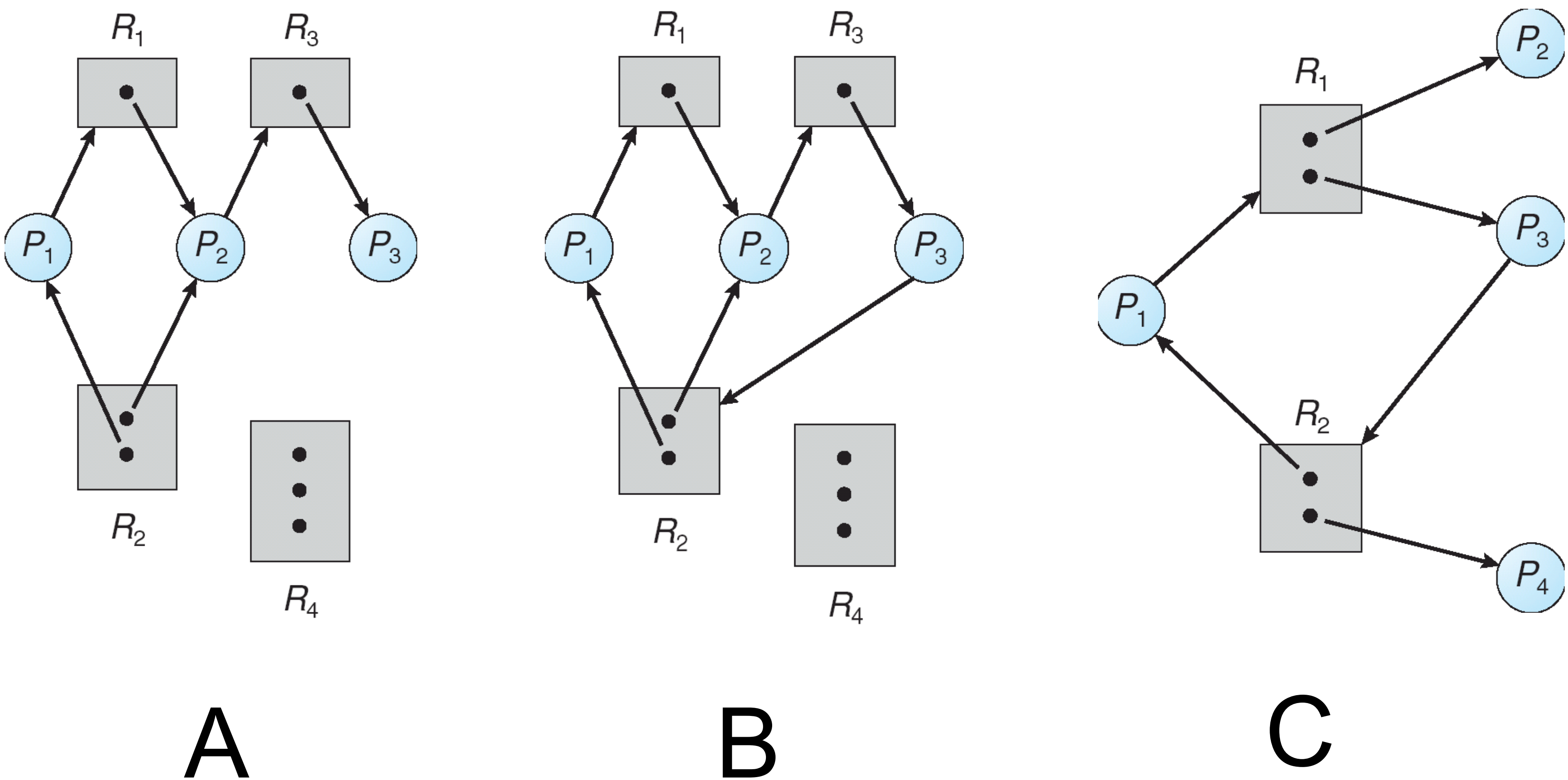
Example

```
Process P1:  Process P2:
wait(S2)     wait(S1)
wait(S1)     wait(S2)
...          ...
signal(S1)   signal(S2)
signal(S2)   signal(S1)
```



Which of these have a deadlock?

URL: www.menti.com
Code: 5135 6077



Basic Facts

- Graph contains no cycles $\Rightarrow$ no deadlock.
- Graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, *possibility* of deadlock.

Deadlock elimination

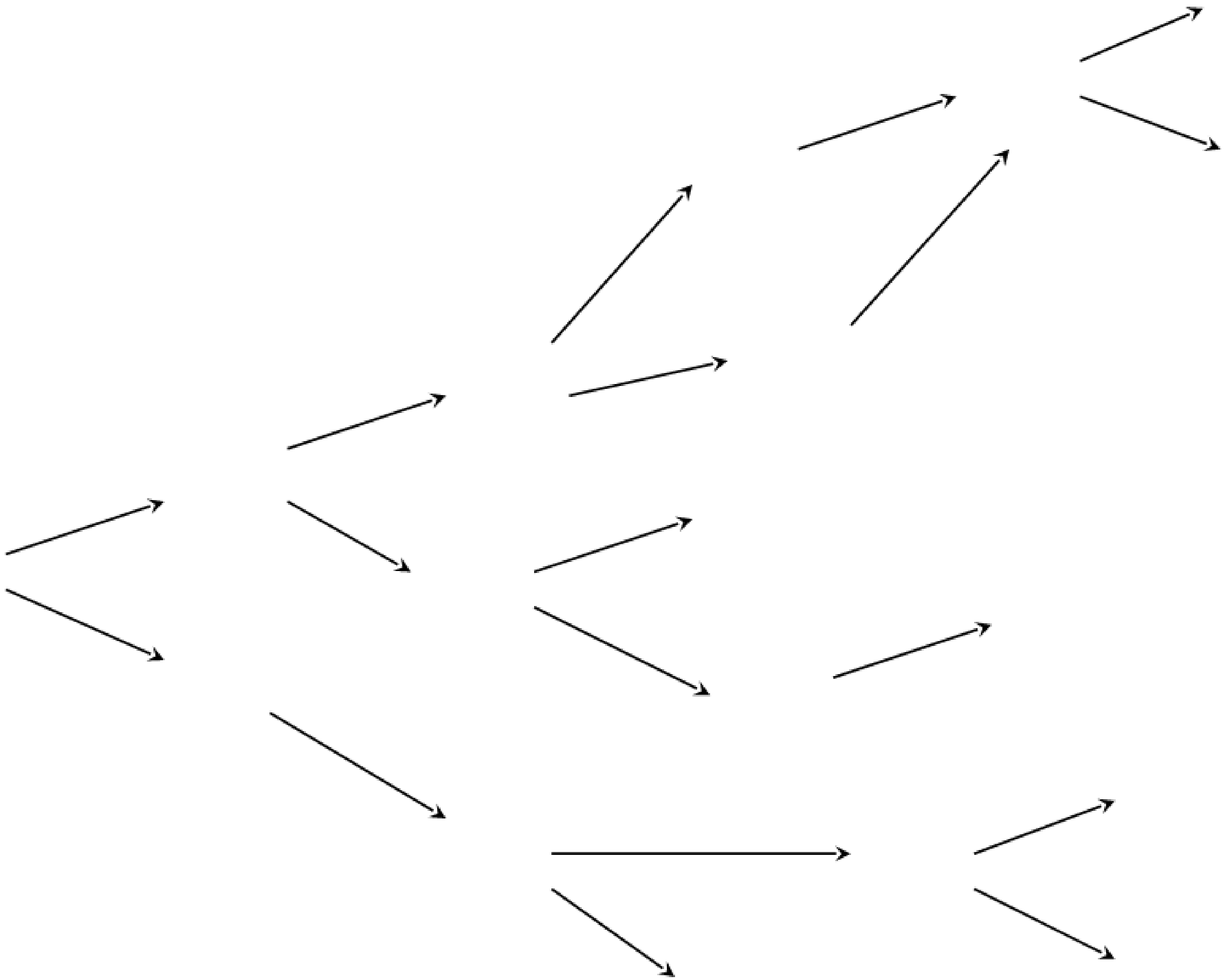
Four approaches:

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and treatment
- Ignore the problem

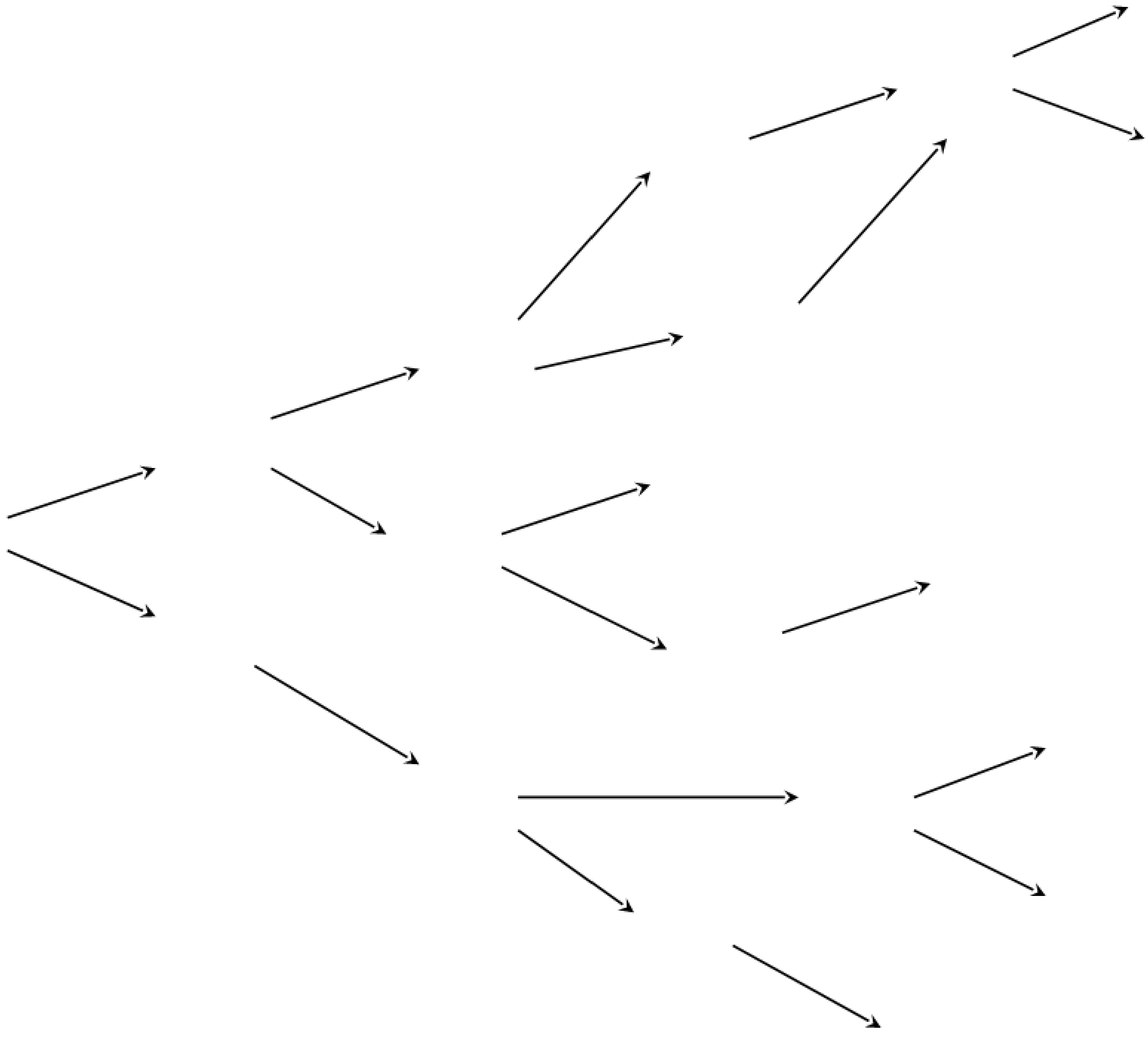
State transition
(in terms of resources)

Resource is acquired or released

Program execution with deadlock



Deadlock prevention



Deadlock prevention:
Ensure that at least one of the Coffman conditions can never occur

No execution path *can* lead to deadlock!

Prevent mutual exclusion (ME)

- ME is needed only for *limited* shared resources

- Example: Read-only-file access by arbitrarily many readers
  - Readers-writer lock

Prevent Hold & Wait

- Whenever a process requests a resource, it cannot hold any other resources.
- Request all resources at once
  - Dining philosopher solution
- Low resource utilization; starvation possible; not flexible.

Prevent Voluntary release

- Ensure preemption
- Force another process to release its resources
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

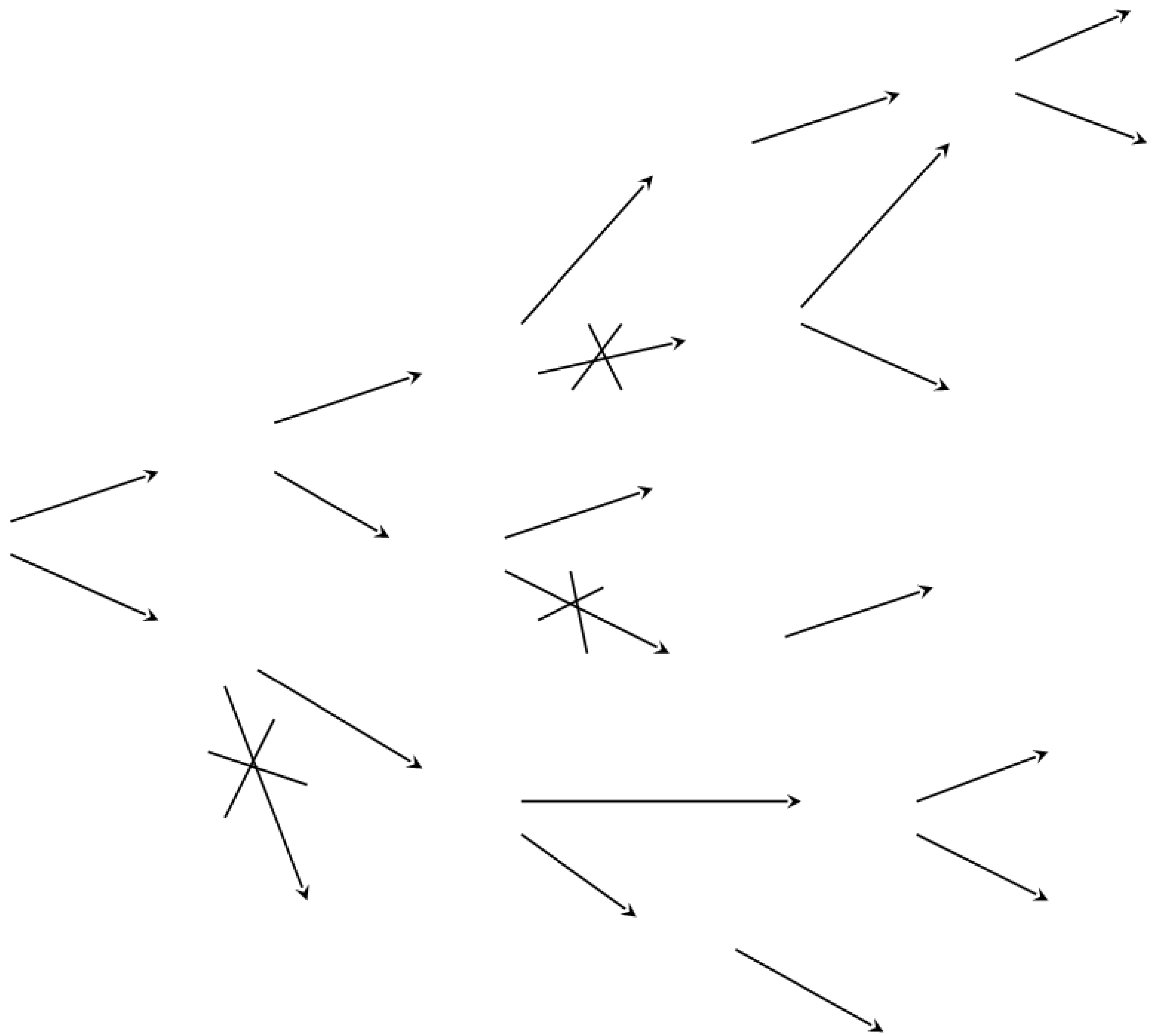
Prevent circular wait

- Impose a *total ordering* of all resources
  - requests must be performed in this order.
- Priorities of processes and resources
  - e.g., Immediate Ceiling Protocol in Real-time scheduling

Tools to eliminate circular wait

- Windows driver verifier
- Linux lockdep tool
- Static analysis tools
  - Cbmc for pthreads (http://www.cprover.org/deadlock-detection/)

Deadlock avoidance



Deadlock avoidance

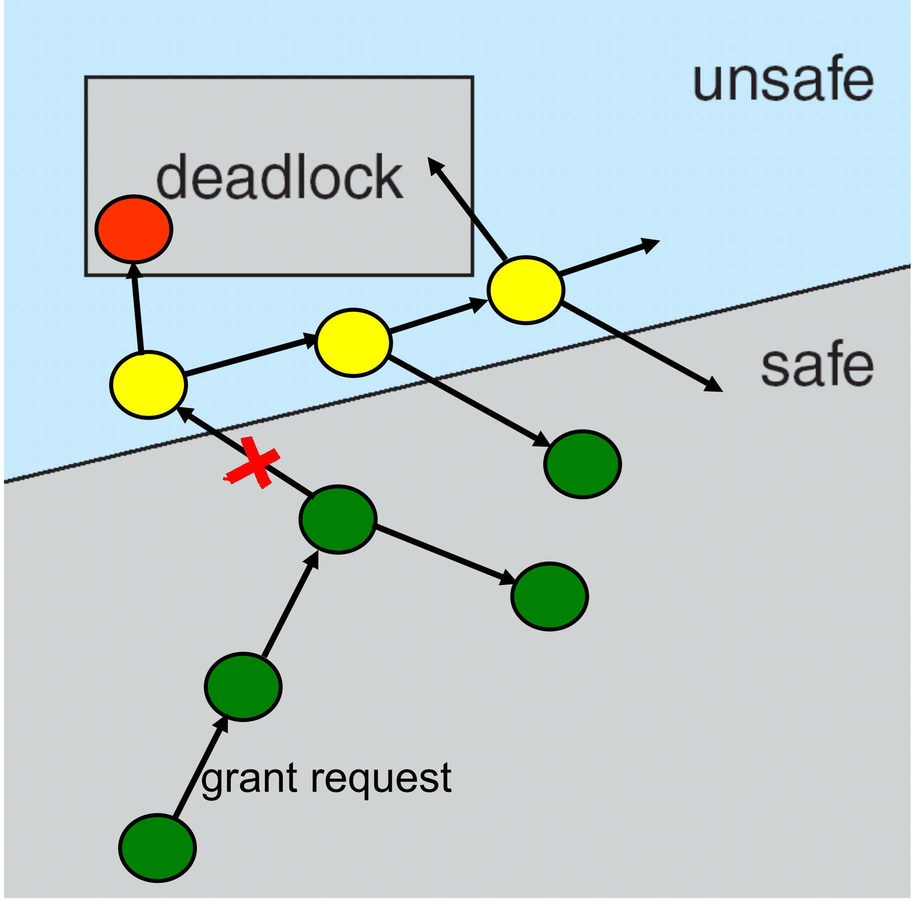
- We allow for some execution paths that can lead to deadlock.
- We stay clear of all paths that looks the least suspisios. (Even some that could turn up alright)

Safe state

System is in **safe state** if there *exists* a **safe sequence** (i.e., completion sequence) of *all* processes.

Safe states and deadlocks

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ *possibility* of deadlock.
- Avoidance:
  ensure that a system will never enter an unsafe state.



Assumptions

- Requires **a priori** knowledge of needed resources

- Assume that each process declare the amount of resources needed

Deadlock Avoidance Algorithms

Avoidance Algorithms for 2 Cases:

- Case 1: All resource types have 1 instance only
  - Resource Allocation Graph Algorithm
- Case 2: Multiple instances per resource type
  - Banker's Algorithm

Banker’s algorithm

- Multiple instances of each resource
- Upon each process request
  - Check that the request is within the maximum limit for that process
  - Check that the new state is safe

Rejecting a request

- When allocating a request does not lead to a new “safe” state:
  - Refuse to grant

- The request can be repeated in some future state and get granted

Inputs and outputs of Banker's

- **Input:**
  - Matrix **Max**
  - Vector **Available**
  - Matrix **Allocation**
  - **Request**[i] for some process i

- **Output**:
  - Yes + new state, or
  - No + unchanged state (Request[i] can not be allocated now)

Data structures

Let n = number of processes, and m = number of resources types.

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

Max*: n x m* matrix. If *Max* [*i,j*] = k, then process i may request at most k instances of resource type R_j, *Max[i]* denotes the *i'th* row.

Allocation*:* n x m matrix. If *Allocation[i,j]* = k then i is currently allocated k instances of R_j, *Allocation[i]* denotes the *i'th* row.

Need*:* n x m matrix. If *Need*[*i,j*] = k, then i may need k more instances of R_j to complete its task, *Need[i]* denotes the *i'th* row.

Banker’s algorithm

1. Need := Max – Allocation
   Check that Request[i] <= Need[i]
2. Check whether Request[i] <= Available
   if not, return ”No”
3. Pretend that resources in Request are to be allocated, compute new state:
   Allocation’ := Allocation + Request
   Need’ := Need – Request
   Available’ := Available – Request[i]
4. Test whether the new state is deadlock-avoiding (denoted safe), in which case return ”Yes”.
   Otherwise, return ”No” - roll back to the old state.

Testing for safe state

- Start with a given **Allocation’** and check if it is safe (avoids future deadlocks) according to the 3-step algorithm.

Safety algorithm data structures

Finish: n vector with Boolean values (initially false)

Work : m vector denotes the changing resource set as the processes become ready and release resources (initially **Work** := **Available'**)

Safety algorithm

1. Check if there is some process i for which **Finish**[i] = false and for which **Need'**[i] <= **Work**.
   If there is no such process i, go to step 3.
2. Free the resources that i has used to get finished:
3. **Work** := **Work** + **Allocation'**[i]
   Finish[i] := true
   continue from step 1.
4. If **Finish**[i] = true for all i then the initial state is deadlock-avoiding, otherwise it is not.

Remember

- Banker’s algorithm:
  - 4 step algorithm
  - 4th step is a 3-step iterative safety algorithm

Example problem

Consider the following resource allocation problem in a system with 3 resources (R1-R3), and 4 processes (P1-P4). The table indicates the currently allocated resources and in parenthesis the maximum possible demand.

	R1	R2	R3
P1	1 (5)	3 (6)	0 (0)
P2	1 (3)	0 (0)	0 (0)
P3	3 (6)	0 (0)	1 (1)
P4	2 (4)	0 (1)	3 (4)

The currently available resources are: [2, 4, 1].
Use Banker's algorithm to determine if the request [1, 0, 0] from Process P4 should be granted.

Running Banker's algorithm (1-3)

	R1	R2	R3
P1	1 (5)	3 (6)	0 (0)
P2	1 (3)	0 (0)	0 (0)
P3	3 (6)	0 (0)	1 (1)
P4	2 (4)	0 (1)	3 (4)

Available: [2, 4, 1]   Request[P4]: [1, 0, 0]

1. **Need** := Max – Allocation =

   [[4 3 0] [2 0 0] [3 0 0] [2 1 1]]

   Check that Request[P4] <= Need[P4]

– OK!

2. Check Request[P4] <= Available

   if not, return "No"

- OK!

3. Pretend that resources in Request are to be allocated, compute new state:

   Allocation' := Allocation + Request = [[1 3 0] [1 0 0] [3 0 1] [3 0 3]]

   Need' := Need – Request = [[4 3 0] [2 0 0] [3 0 0] [1 1 1]]

   Available' := Available – Request[P4] = [1 4 1]

Running Safety algorithm (step 4)

Initial finish vector = [False, False, False. False]
Initial work vector = Available’ = [1, 4, 1]

Can finish process: P4
Work vector: [4 4 4]
Finish vector: [False, False, False, True]

Could finish process: P1
Work vector: [5 7 4]
Finish vector: [True, False, False, True]

Could finish process: P2
Work vector: [6 7 4]
Finish vector: [True, True, False, True]

Could finish process: P3
Work vector: [9 7 5]
Finish vector: [True, True, True, True]

1. Check if there is some process i for which **Finish**[i] = false and for which **Need’**[i] <= **Work**. If there is no such process i, go to step 3.
2. Free the resources that i has used to get finished:
   Work := **Work** + **Allocation’**[i]
   Finish[i] := true
   continue from step 1.
3. If **Finish**[i] = true for all i then the initial state is deadlock-avoiding, otherwise it is not

Allocation’= [[1 3 0] [1 0 0] [3 0 1] [3 0 3]]

Need’ = [[4 3 0] [2 0 0] [3 0 0] [1 1 1]]

Available’ = [1 4 1]

Running Bankers algorithm (result)

- The outcome of the Safety algorithm is that the new state is safe

- **Result:** The request [1, 0, 0] from P4 can be granted!

Weaknesses of Banker's algorithm?

Weaknesses of the Banker's Algorithm

- Assumes a fixed number of resources
  - not realistic – number of resources can vary over time
- Assumes a fixed population of processes
  - not realistic for interactive systems
- Assumes that processes state maximum needs in advance
  - often not known
    (depend e.g. on input data or user commands)
- Waiting for completion of one or several processes may take very long / unpredictable time before a request is granted

Deadlock Detection and Recovery

- Allow system to enter deadlock state
- Detection algorithm
  - Single instance of each resource type
  - Multiple instances
- Recovery scheme

Exam question...

Which of the following statements are true about deadlocks?:

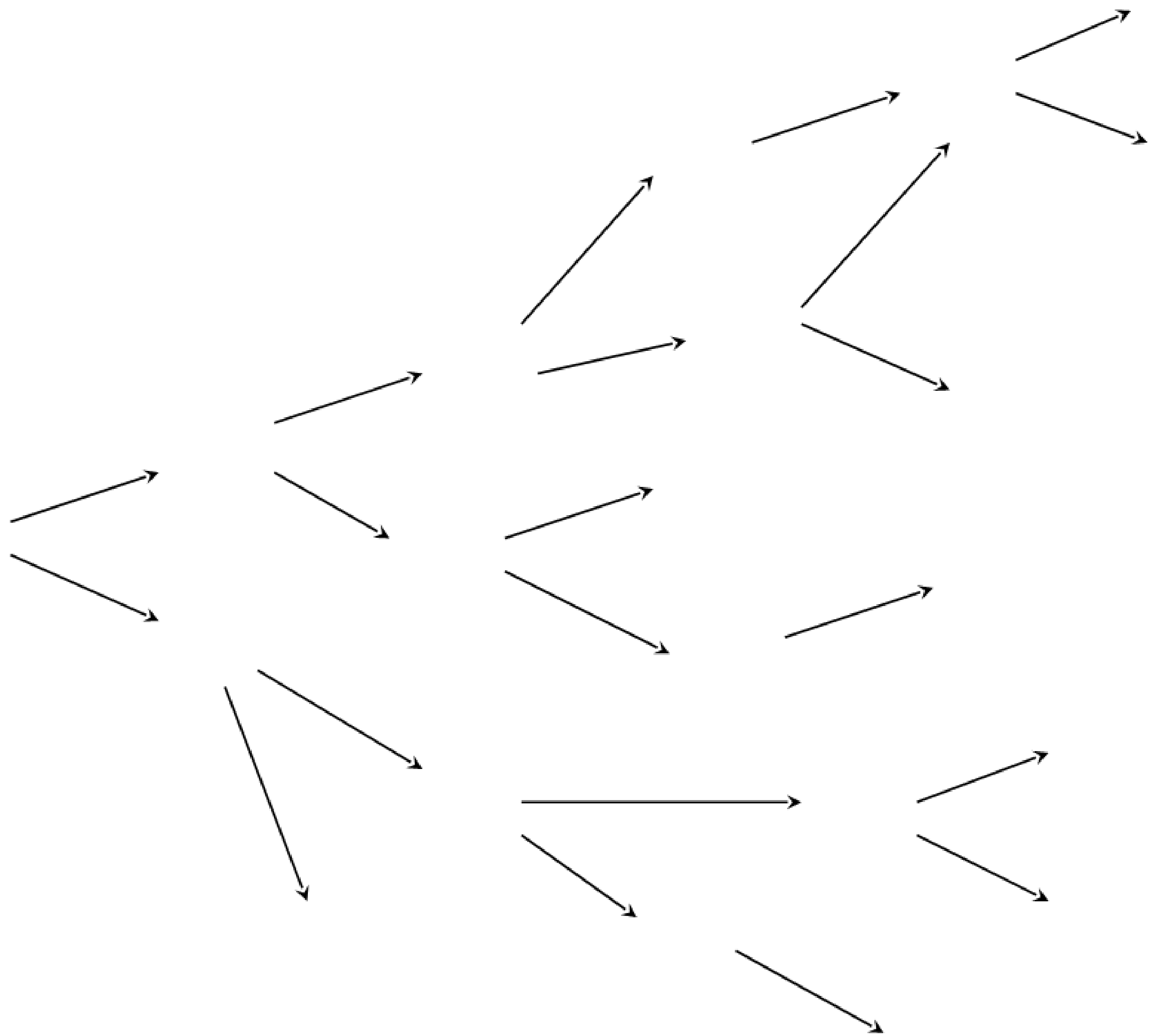
A. If there is only a single instance of every resource, a cycle in the resource allocation graph means that there is a deadlock.

B. All four Coffman conditions must be met for there to be a deadlock.

C. Banker's algorithm is used to detect and remove deadlocks.

D. Banker's algorithm guarantees freedom from starvation.

Deadlock detection



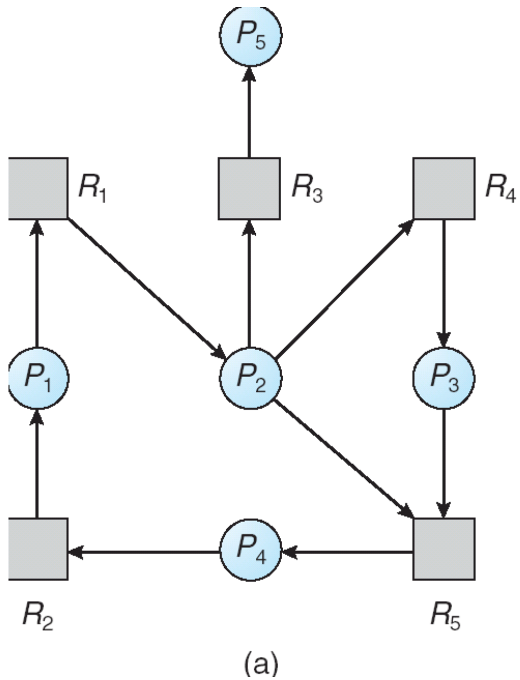
Deadlock detection with **single** instance resources

Search for cycle in wait-for graph

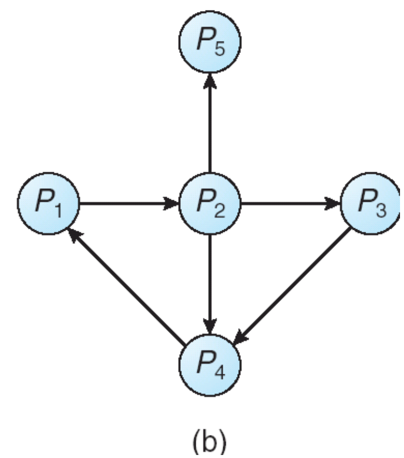
- Maintain ***wait-for* graph**
  - Nodes are processes.
  - $P_i \rightarrow P_j$
    iff P_i is waiting for P_j.

- Periodically invoke an algorithm
  that searches for a cycle in the graph.

Transformation RAG-WFG



Resource-Allocation Graph



Corresponding wait-for graph

Deadlock detection with **multiple** instance resources

Deadlock Detection Algorithm

- *Available:* vector of length m
  indicates the number of available resources of each type.

- *Allocation:* n x m matrix
  defines the number of resources of each type currently allocated to each process.

- *Request:* n x m matrix
  indicates the currently pending requests of each process.
  Request [i, j] = k iff P_i is requesting k more instances of R_j.

Detection Algorithm [Coffman et al. 1971]

1. Vectors ***Work***[1..*m*], ***Finish***[1..*n*] initialized by:

   Work = *Available*

   for *i* = 1,2, …, *n*, **if** $Allocation_i \neq 0$ **then** *Finish*[i] = false
   otherwise *Finish*[i] = *true*

2. Find an index *i* such that both:

   (a) *Finish*[*i*] == *false*

   (b) $Request_i \leq Work$

   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*, $1 \leq i \leq n$,
   then the system is in deadlock state.
   Specifically, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Difference to Banker's algorithm

- What is a safe state?
  - Consider the actual request (optimistically), not the maximum needs

- Reason: We compute if there is a deadlock **now**, not if one may happen later.

Detection-Algorithm Usage

- 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
- Invocation at every resource request?
  - Too much overhead
- Occasional invocation?

  (e.g., once per hour, or whenever CPU utilization below 40%)

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 the process needs to complete.
  - How many processes will need to be terminated.

Recovery from Deadlock: Resource Preemption

- Selecting a victim
  - minimize cost
- Rollback
  - return to some safe state, restart process for that state.
- Starvation
  - same process may always be picked as victim, include number of rollbacks in cost factor.

Summary

- Deadlock characterization
  - 4 necessary conditions (Coffman)
  - Resource allocation graph
- Deadlock prevention
  - Prohibit one of the four necessary conditions
- Deadlock avoidance
  - 1 instance-resources: Resource allocation graph algorithm
  - Banker’s algorithm (state safety, request granting)
- Deadlock detection and recovery
  - 1 instance-resources: Find cycles in Wait-for graph
  - Several instances: Deadlock detection algorithm
- Do nothing – **lift the problem to the user / programmer**

Next

- Lesson 2: Lab 2-4
- Lecture 7: Memory management I
  Ch. 9, 10.1-1.3