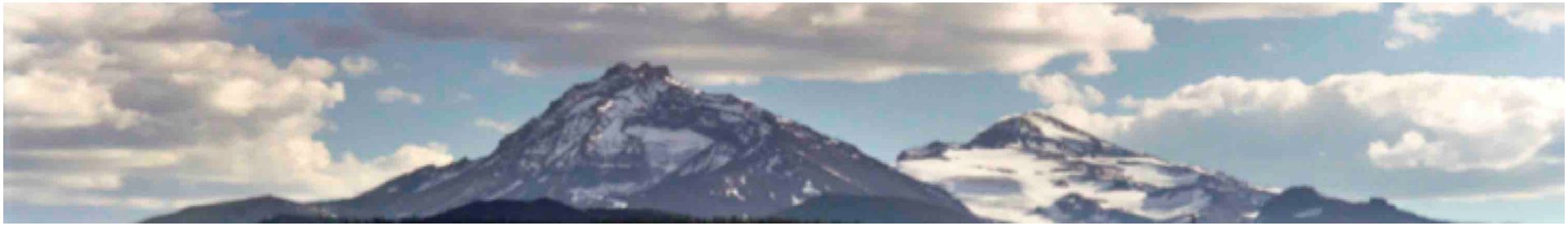


CS 415
Operating Systems
Multi-Resource Concurrency and Deadlocks

Prof. Jieyang Chen
Department of Computer and Information Science
Fall 2025

UNIVERSITY OF OREGON

Logistics

- Project 2 due this week
- Read Chapter 8

Outline

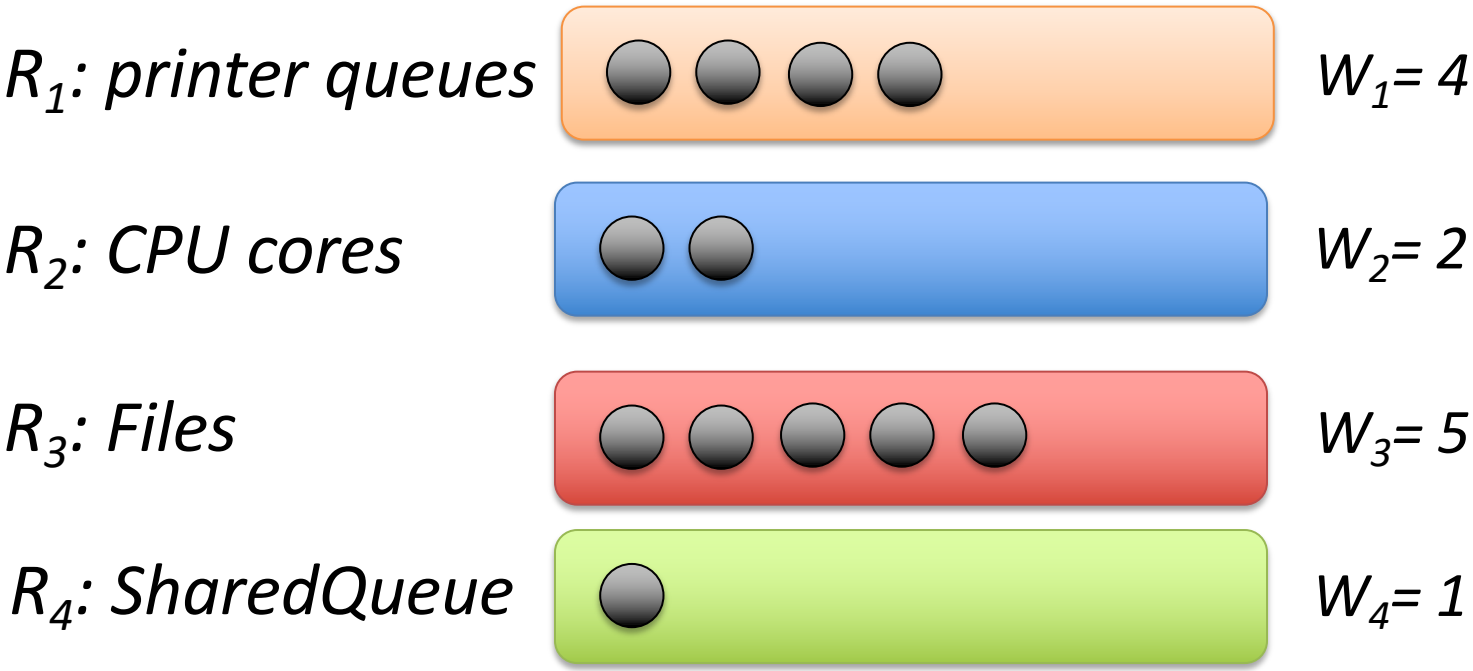
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

Objectives

- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system

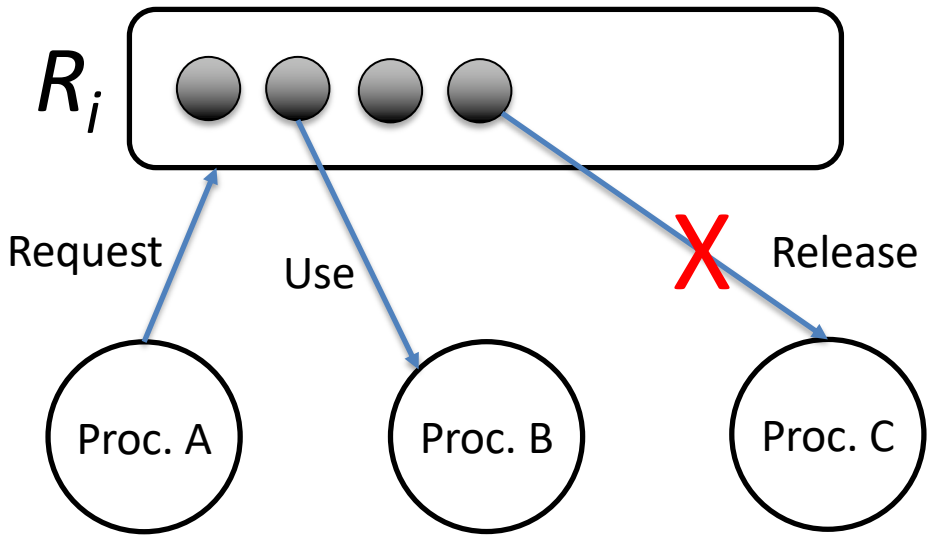
System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
- Each resource type R_i has W_i instances



System Model

- Each process utilizes a resource as follows:
  - Request
  - Use
  - Release



Side note: Deadlock problems affect both processes and threads. We use a process to illustrate for the rest of the slides, but remember everything applies to threads too!

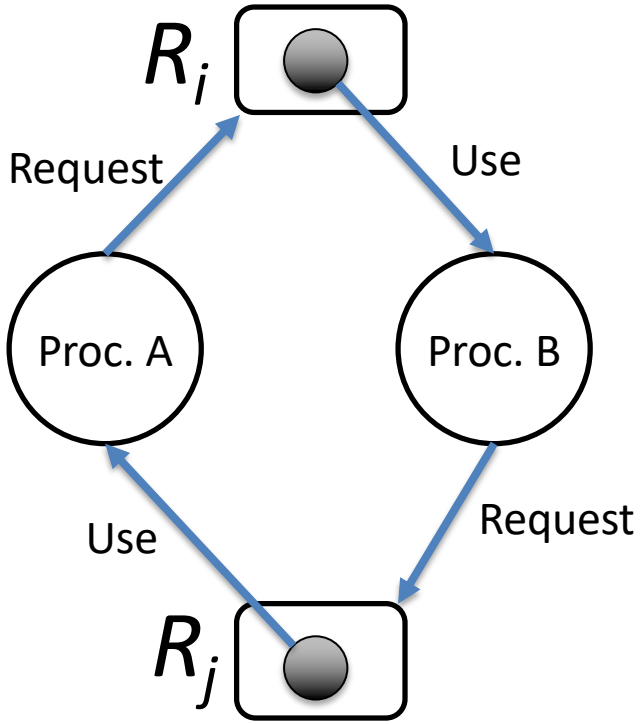
What is a deadlock?

- Given a set of processes running concurrently:

A ***deadlock*** is a condition where the computation can not make any progress because all processes are blocked while waiting for resources currently owned by other processes in the set.

Serious synchronization problem

- Caused by errors in synchronization logic
- What damage can a deadlock cause?

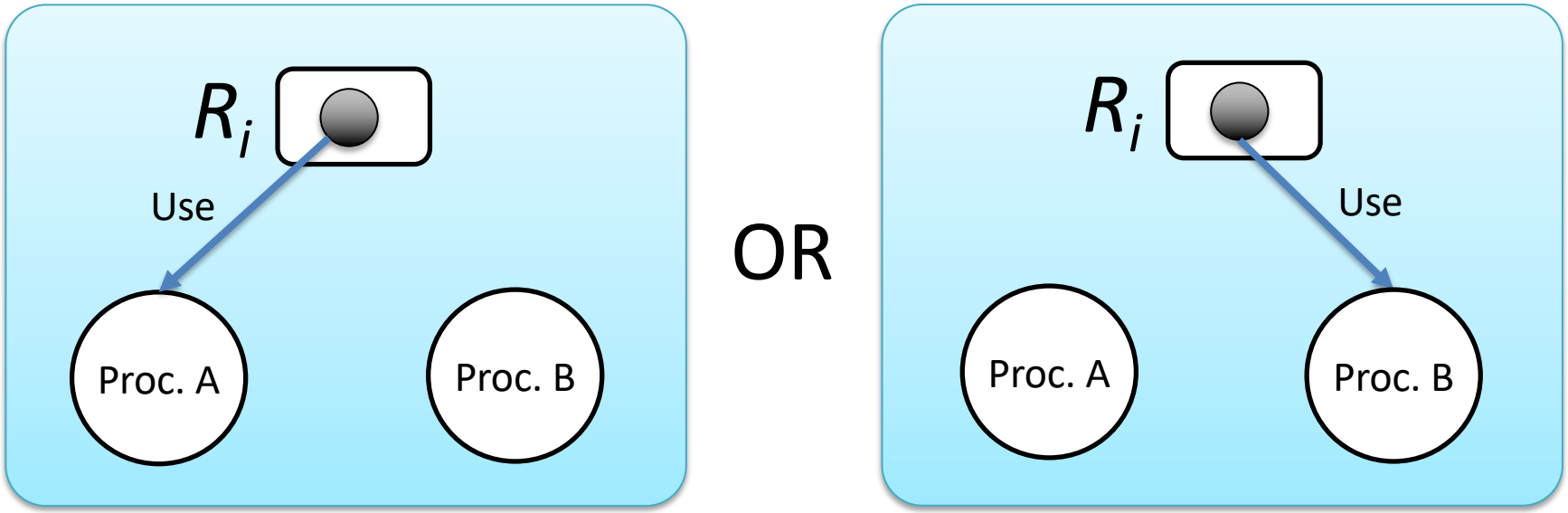


Deadlock Characterization

- Deadlock can arise if four conditions hold simultaneously:
  1. *Mutual exclusion*
  2. *Hold and wait*
  3. *No preemption*
  4. *Circular wait*

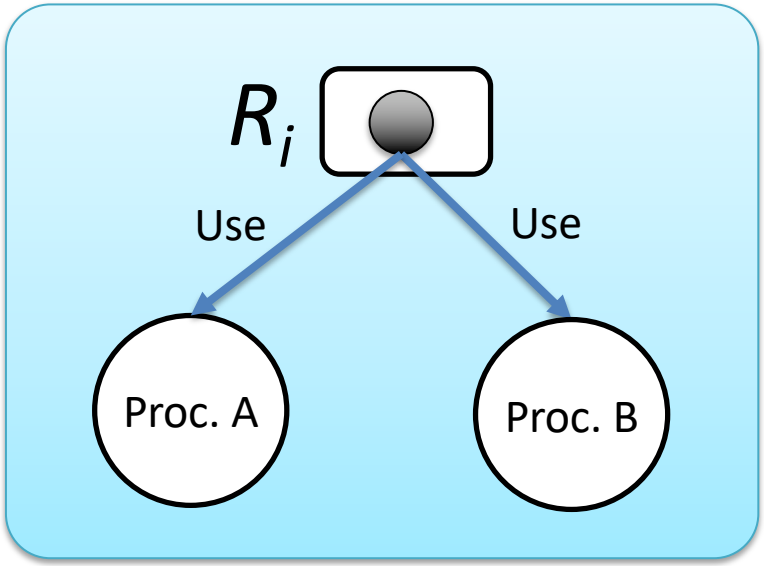
Deadlock Characterization

Mutual exclusion: only one process at a time can use a resource instance

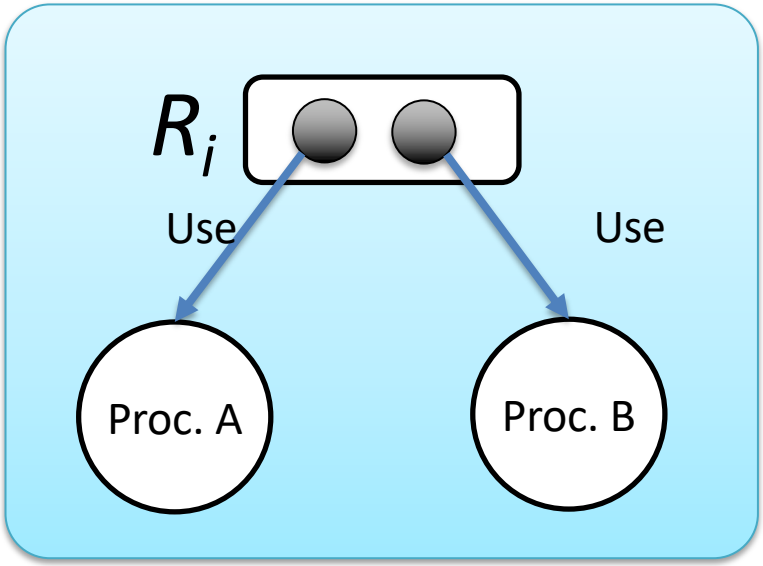


Deadlock Characterization

Mutual exclusion: only one process at a time can use a resource instance



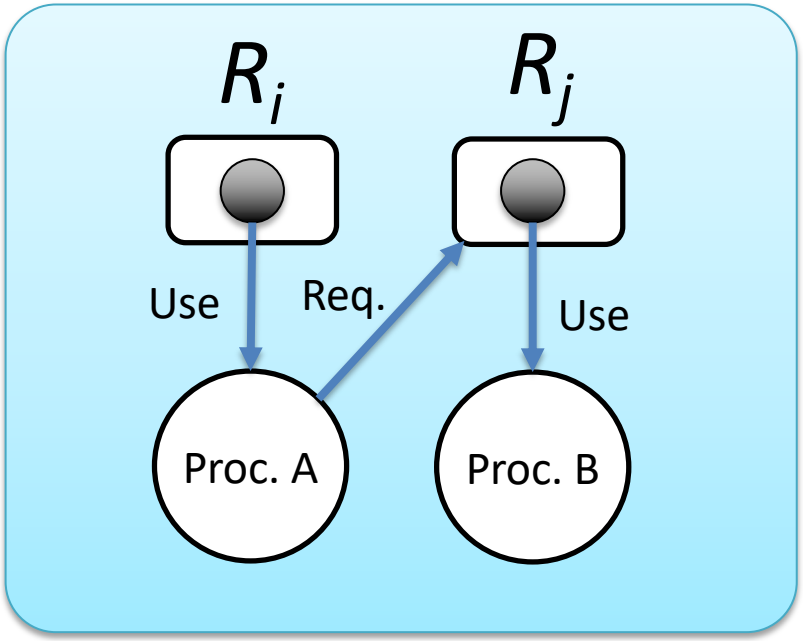
This is wrong!



This is fine.

Deadlock Characterization

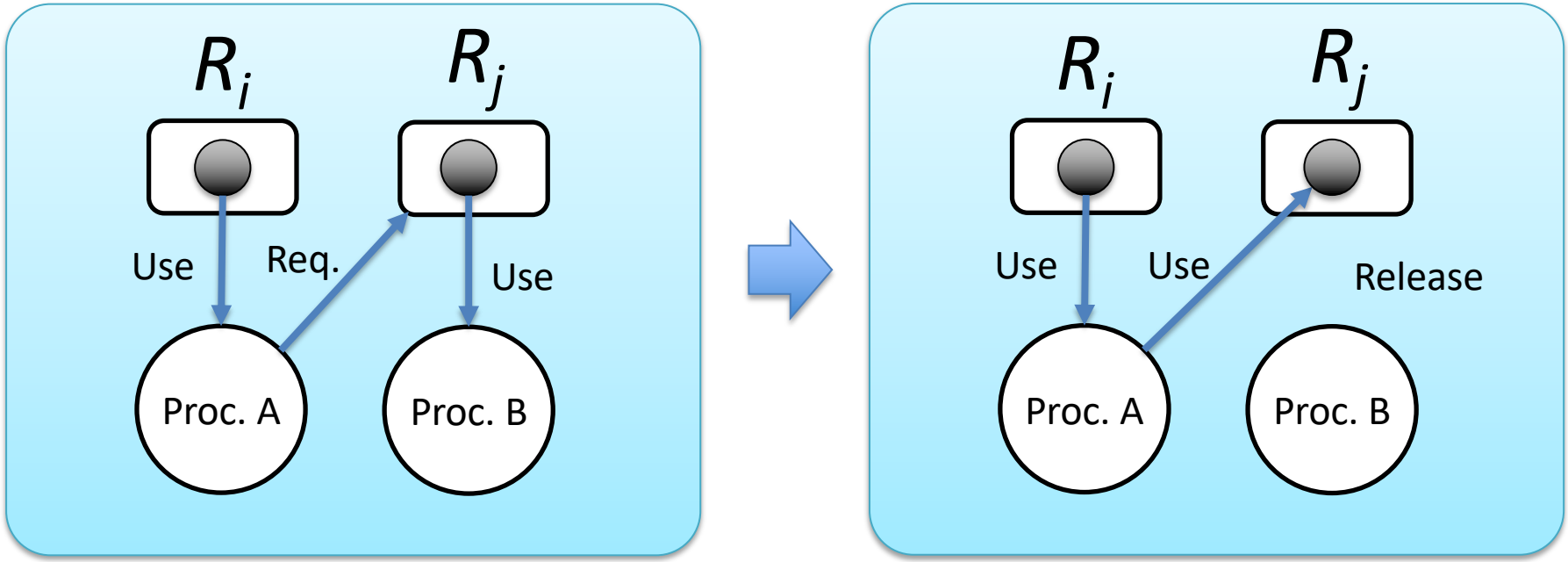
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes



Process A holds R_i and is waiting to acquire R_j, which is held by process B

Deadlock Characterization

No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task using the resource

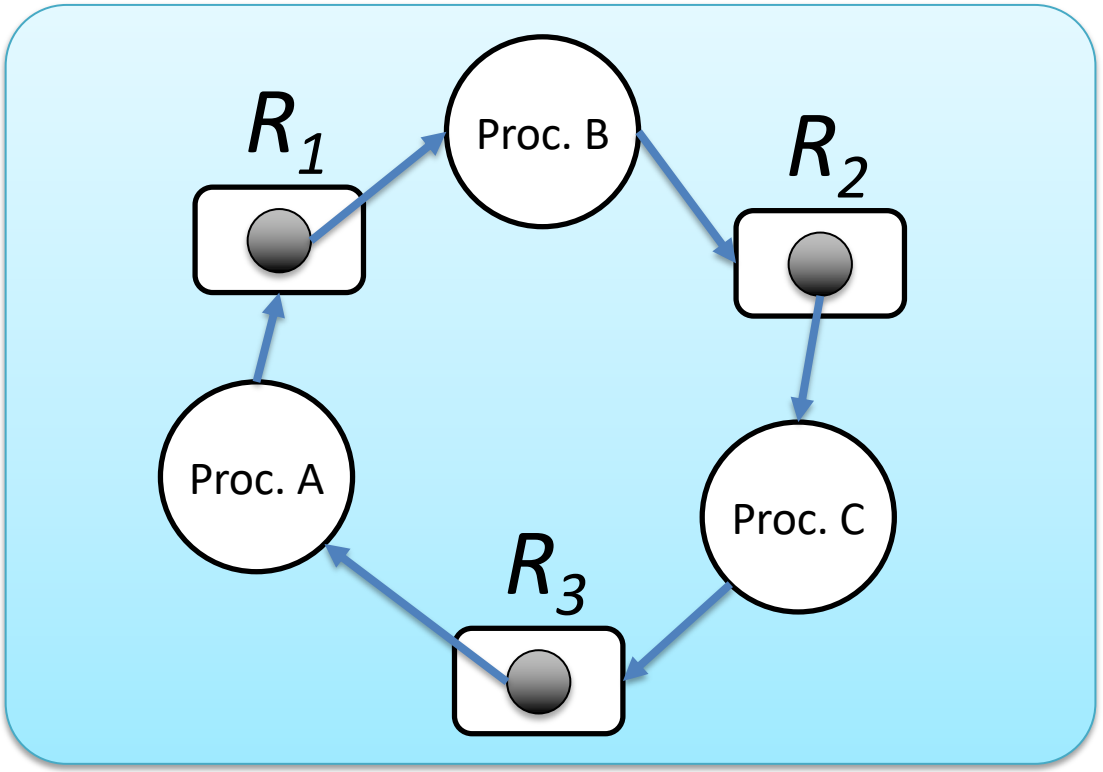


R*j* can only be released by its current user -- process B. It cannot be forcefully taken by process A while process B is still using it.

Deadlock Characterization

Circular wait: there exists a set $\{P_1, P_2, \ldots, P_n\}$ of waiting processes such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_1. Note: all processes in the set must be involved.

- Process A requests R1, which is held by process B
- Process B requests R2, which is held by process C
- Process C requests R3, which is held by process A



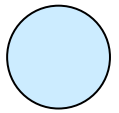
Resource Allocation Graph

- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, set of all processes
  - $R = \{R_1, R_2, \ldots, R_m\}$, set of all resource types
- *Request edge*
  - Directed edge $P_i \rightarrow R_j$
- *Assignment edge*
  - Directed edge $R_j \rightarrow P_i$
- A resource allocation graph is used to determine "state" of the resource system

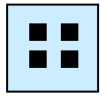
Resource-Allocation Graph Symbols

Graphic representation

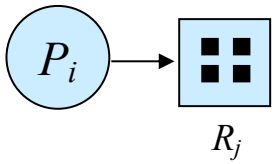
- Process



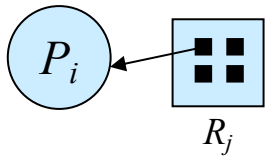
- Resource type with 4 instances



- P_i requests an instance of R_j



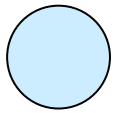
- P_i is holding an instance of R_j



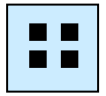
Resource-Allocation Graph Symbols

Graphic representation

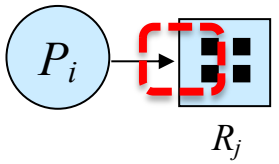
- Process



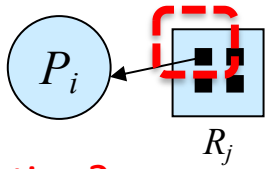
- Resource type with 4 instances



- P_i requests an instance of R_j



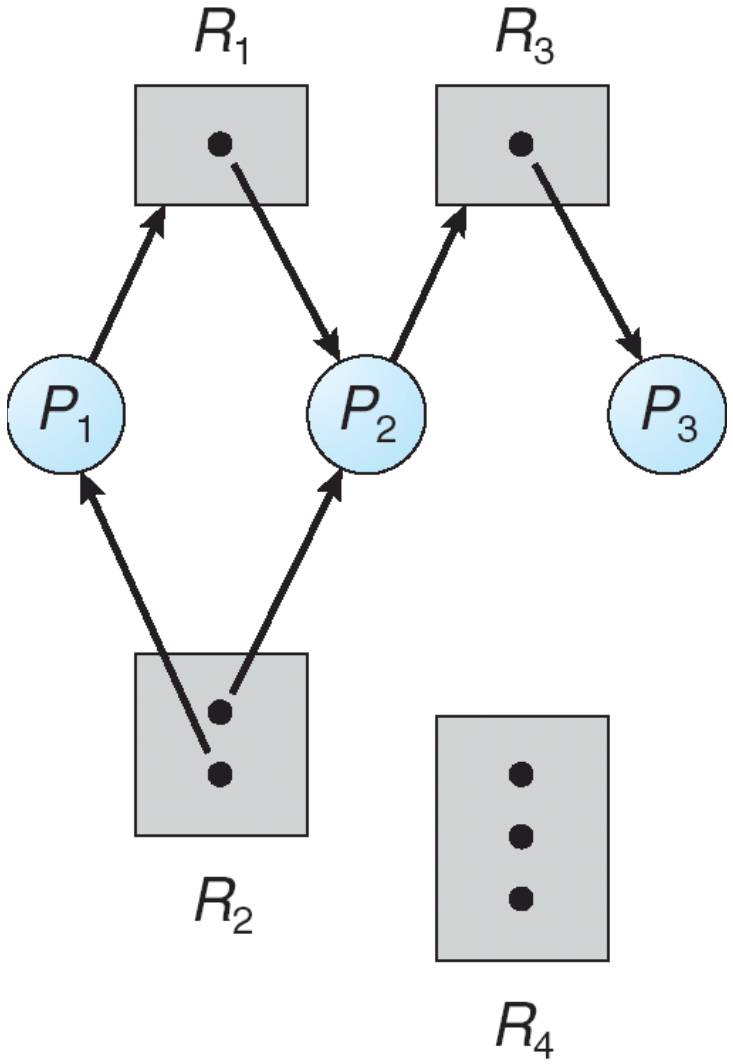
- P_i is holding an instance of R_j



Arrow direction?
Where are arrows connected to?

Example of a Resource Allocation Graph

- P_1 is requesting R_1 and holding an instance of R_2
- P_2 is holding an instance of R_1 and R_2 and is requesting R_3
- P_3 is holding an instance of R_3
- Resource allocation graph shows the "hold" and "wait" conditions

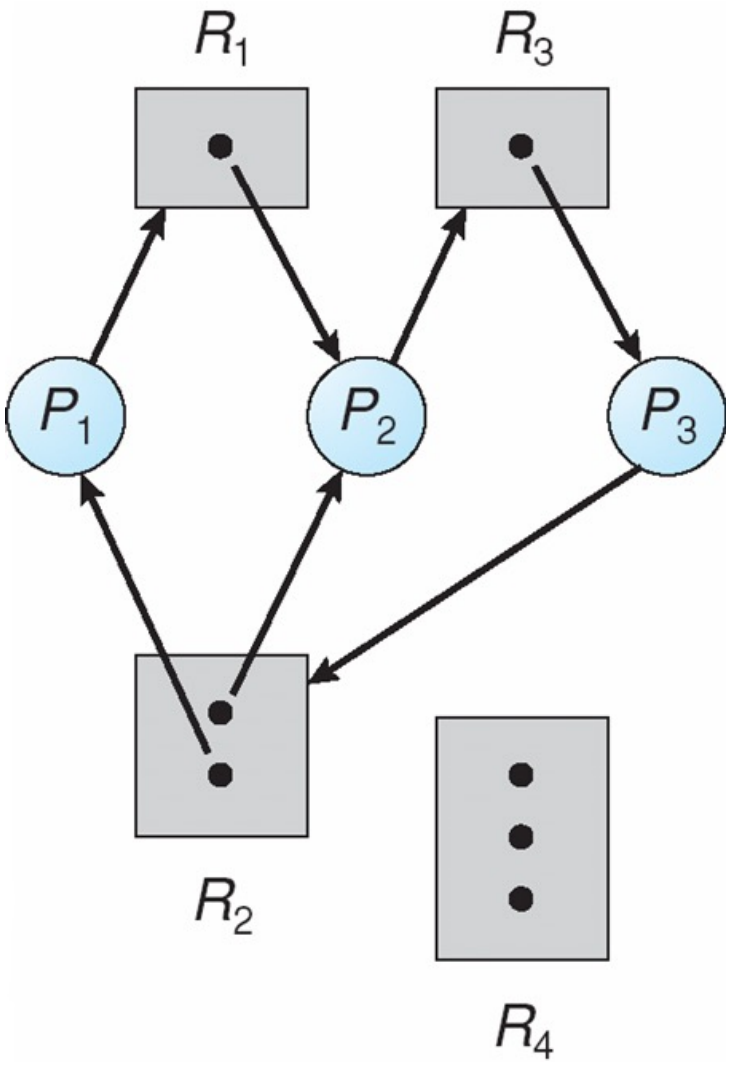


Is there a deadlock?

Example of a Resource Allocation Graph

- Circular waiting state
  - Every process is waiting on a resource that is being held by another process in the cycle

Is there a deadlock?

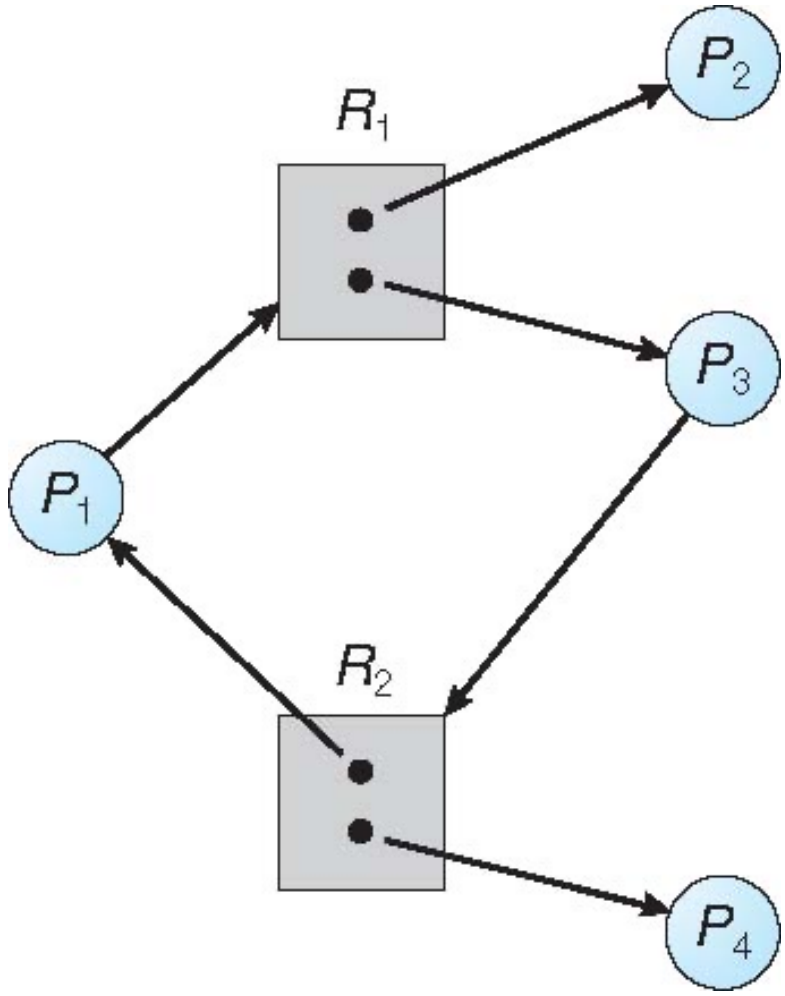


- Yes. If there is a circular wait state, then no process can proceed because they are all blocked
- Every process is waiting indefinitely because no resource will be released

Example of a Resource Allocation Graph

Does circular waiting state always indicate a deadlock?

- There exist a resource that is held by a process that is not in the cycle
- A resource can be released by a process that is not in the cycle, allowing for a process in the cycle to proceed

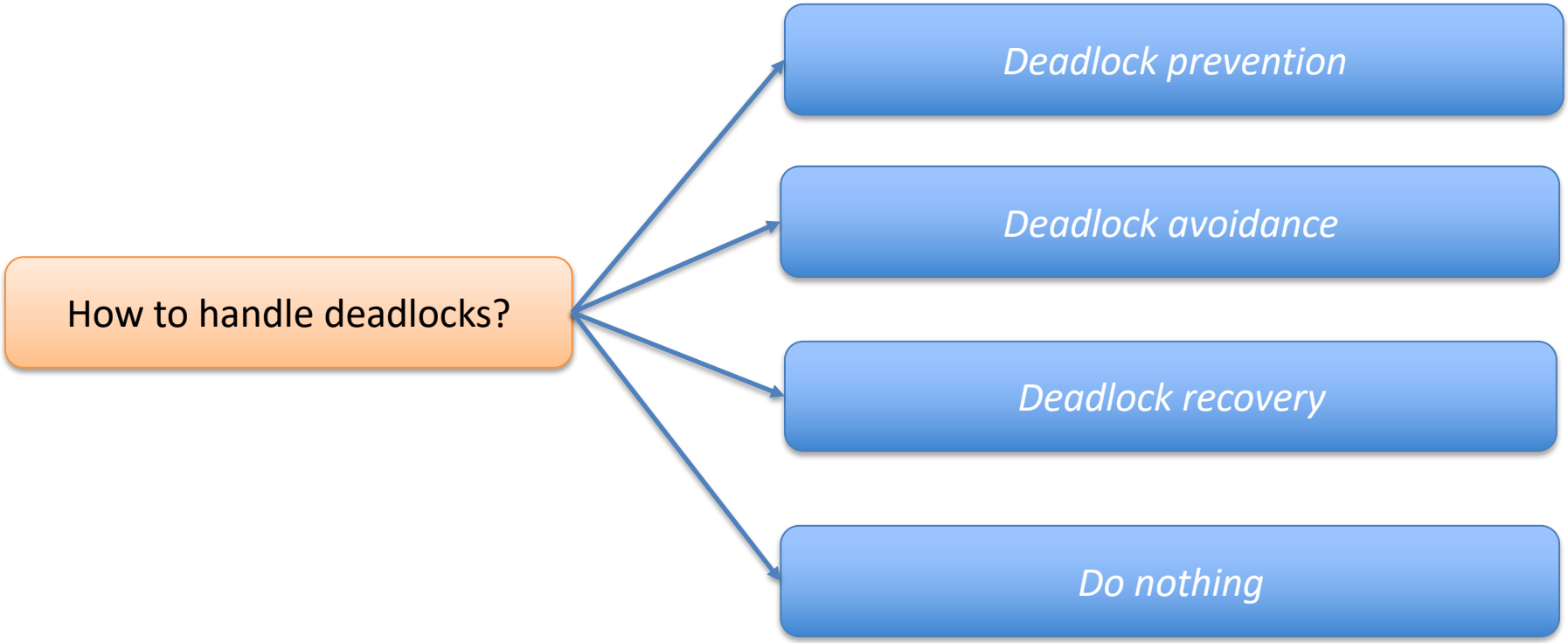


Basic Facts

- If resource allocation graph contains no cycles:
  - There can be no deadlock
- If resource allocation graph contains a cycle:
  - If there is only one instance per resource type
    ⇒ deadlock
  - If there are several instances per resource type
    ⇒ possibility of deadlock

Methods for Handling Deadlocks

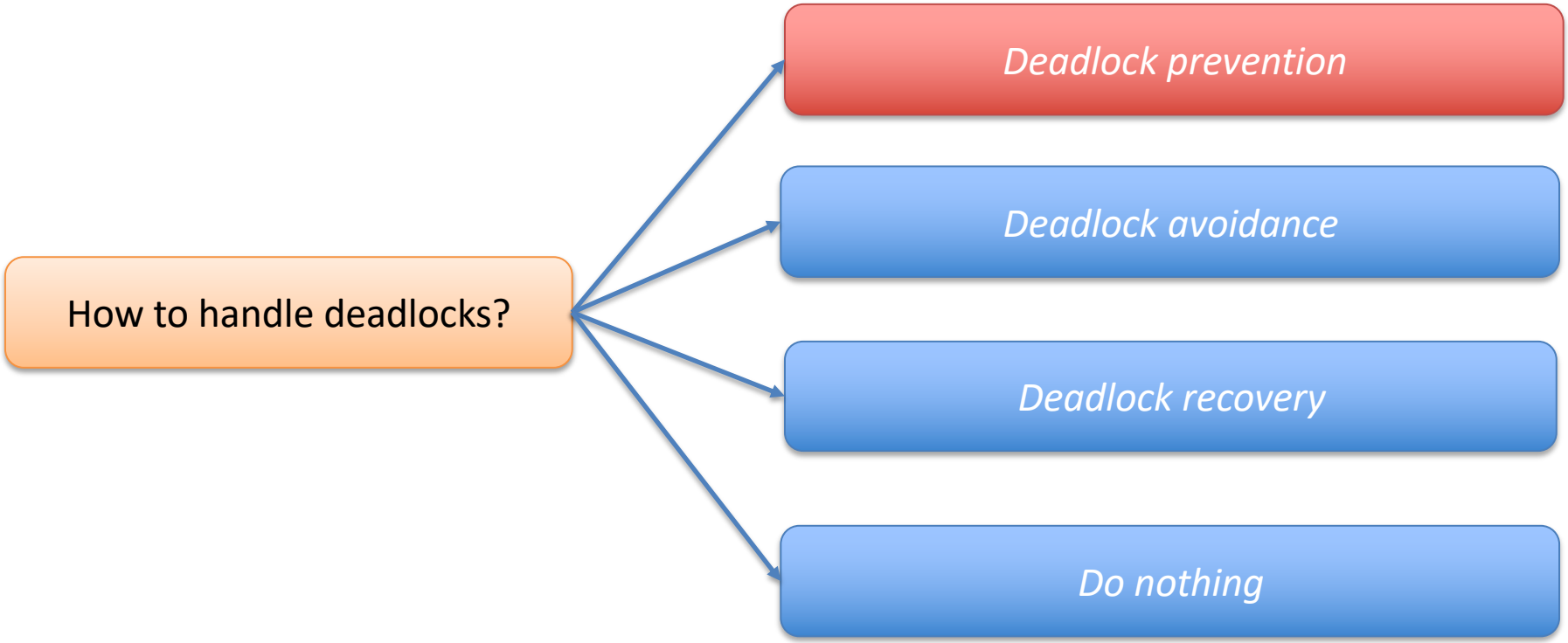
4 options to handle deadlocks



Note: these are NOT 4 steps.

Methods for Handling Deadlocks

4 options to handle deadlocks



Note: these are NOT 4 steps.

Deadlock Prevention

- Deadlock can arise if four conditions hold simultaneously:
  1. *Mutual exclusion*
  2. *Hold and wait*
  3. *No preemption*
  4. *Circular wait*

What happens when not all conditions can hold at the same time?

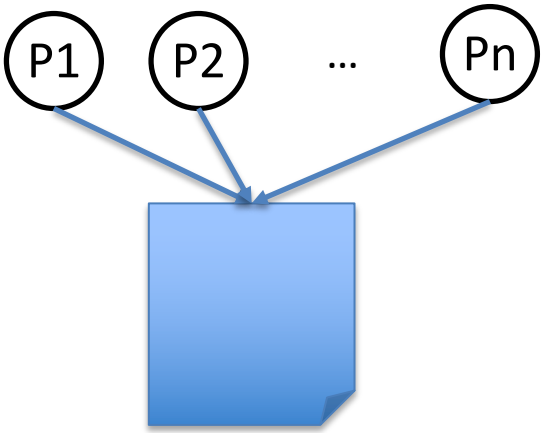
Deadlock prevention: ensuring that at least one of these conditions cannot hold, so we can prevent the occurrence of a deadlock.

Deadlock Prevention (1)

- Removing condition: *Mutual exclusion*
  - Some are hard to remove
  - Aim to remove unnecessary ones

Example: read-write-lock.c

What if some want to write?



If we know all processes will just read this file, is mutual exclusion still necessary?

POSIX read-write lock

- Allow multiple threads to read a resource concurrently
- Mutual exclusion between writer-writer and writer-reader

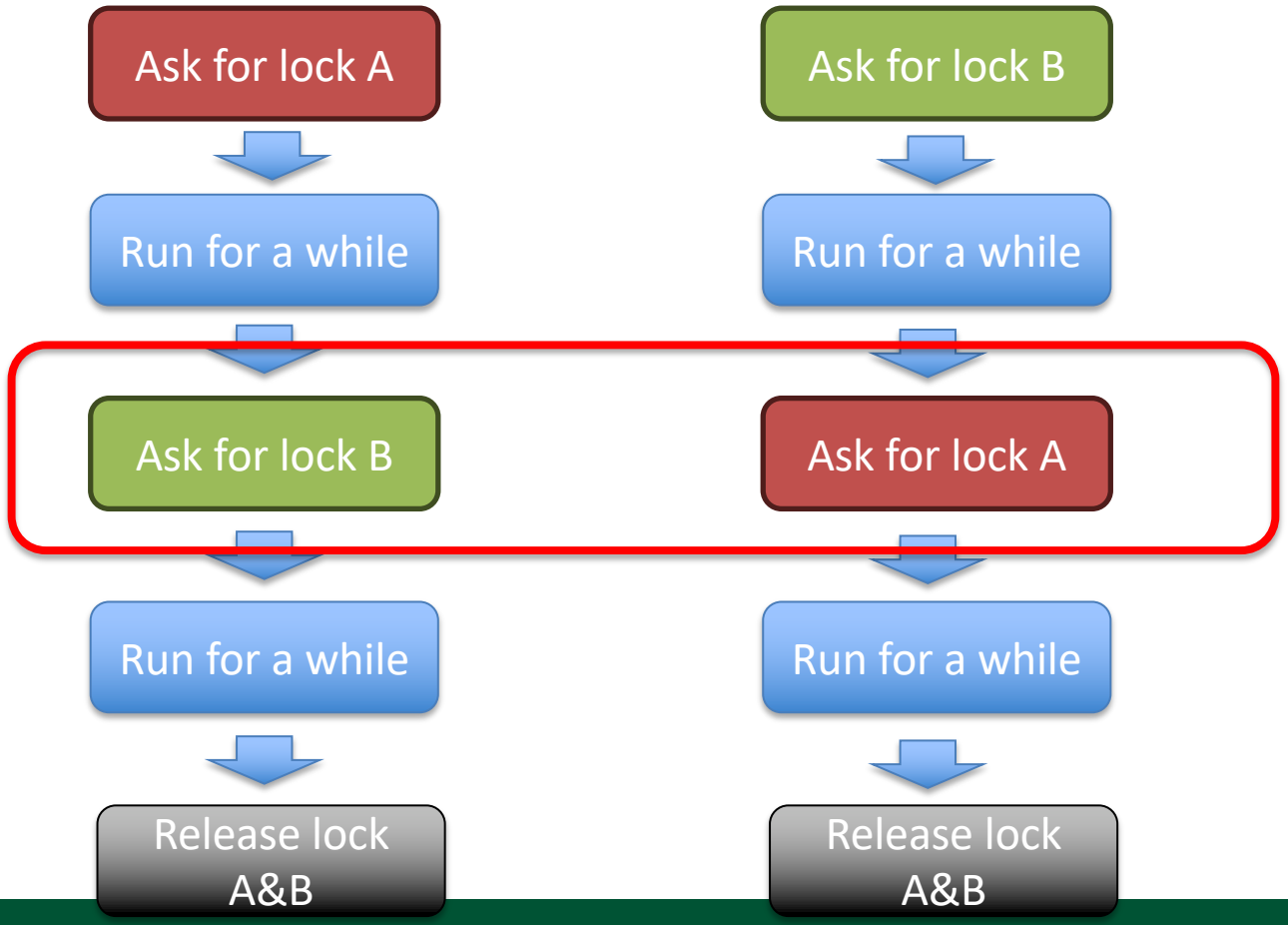
pthread_rwlock_rdlock() // read lock
pthread_rwlock_wrlock() // write lock

Deadlock Prevention

- Removing condition: *Hold and wait*
  - Require a process to request and be allocated ALL of its resources before it begins execution
  - If it failed to acquire any resource, release all and start over.
  - May lead to low resource utilization

Deadlock Prevention

- Removing condition: *Hold and wait*



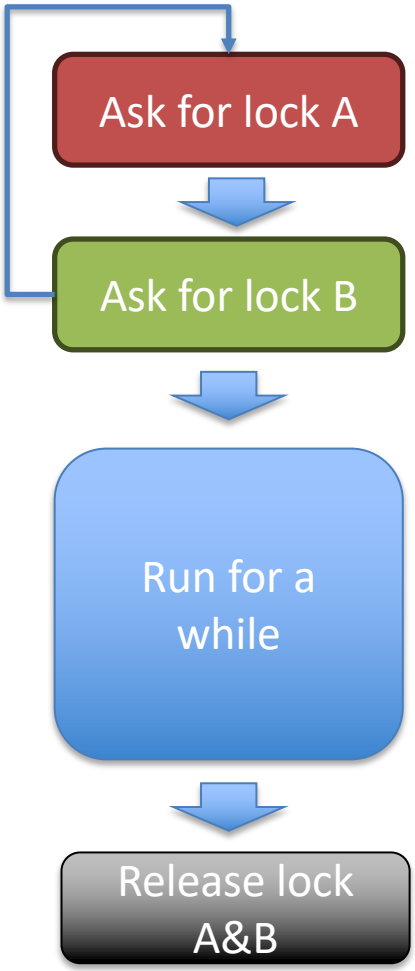
Hold and wait causes a deadlock

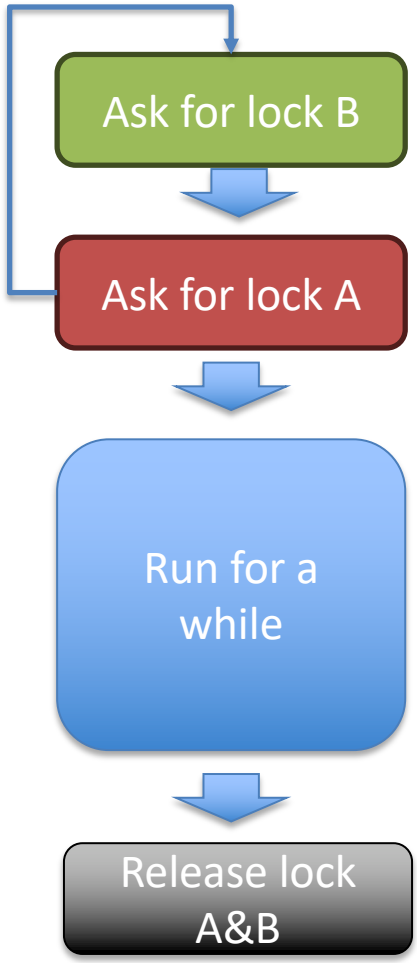
Deadlock Prevention

- Removing condition: *Hold and wait*

Example: disable-hold-wait.c

If it failed to acquire, release all already acquired





What potential issue does this have?

Deadlock Prevention– Requirements

- Removing condition: *No preemption*
  - Do normal lock-run-lock execution
  - Instead of blocking and waiting for locks, try first.
  - If it fails, preempt all acquired resources

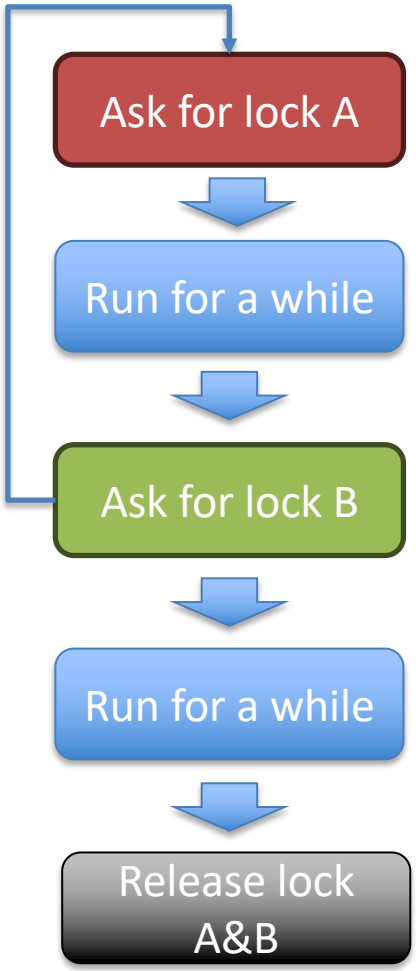
Deadlock Prevention– Requirements

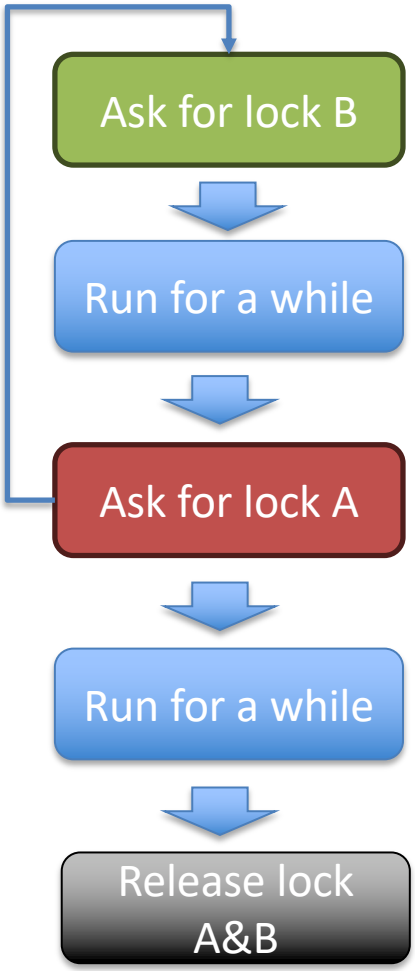
- Removing condition: *No preemption*

If it failed to acquire, release all already acquired

What potential issue does this have?

How is this different than removing 'hold-and-wait'?





Deadlock Prevention– Requirements

- Removing condition: *Hold and wait*
  - "Don't start unless you have everything you need."
- Removing condition: *No preemption*
  - "Don't need to get everything at the beginning, but if you get stuck at any time, give up what you have and start over."

Deadlock Prevention

- Deadlock can arise if four conditions hold simultaneously:
  1. *Mutual exclusion*
  2. *Hold and wait*
  3. *No preemption*
  4. *Circular wait*

The first three are hard to remove.

Deadlock Prevention

- Deadlock can arise if four conditions hold simultaneously:
  1. *Mutual exclusion*
  2. *Hold and wait*
  3. *No preemption*
  4. *Circular wait*

The first three are hard to remove.

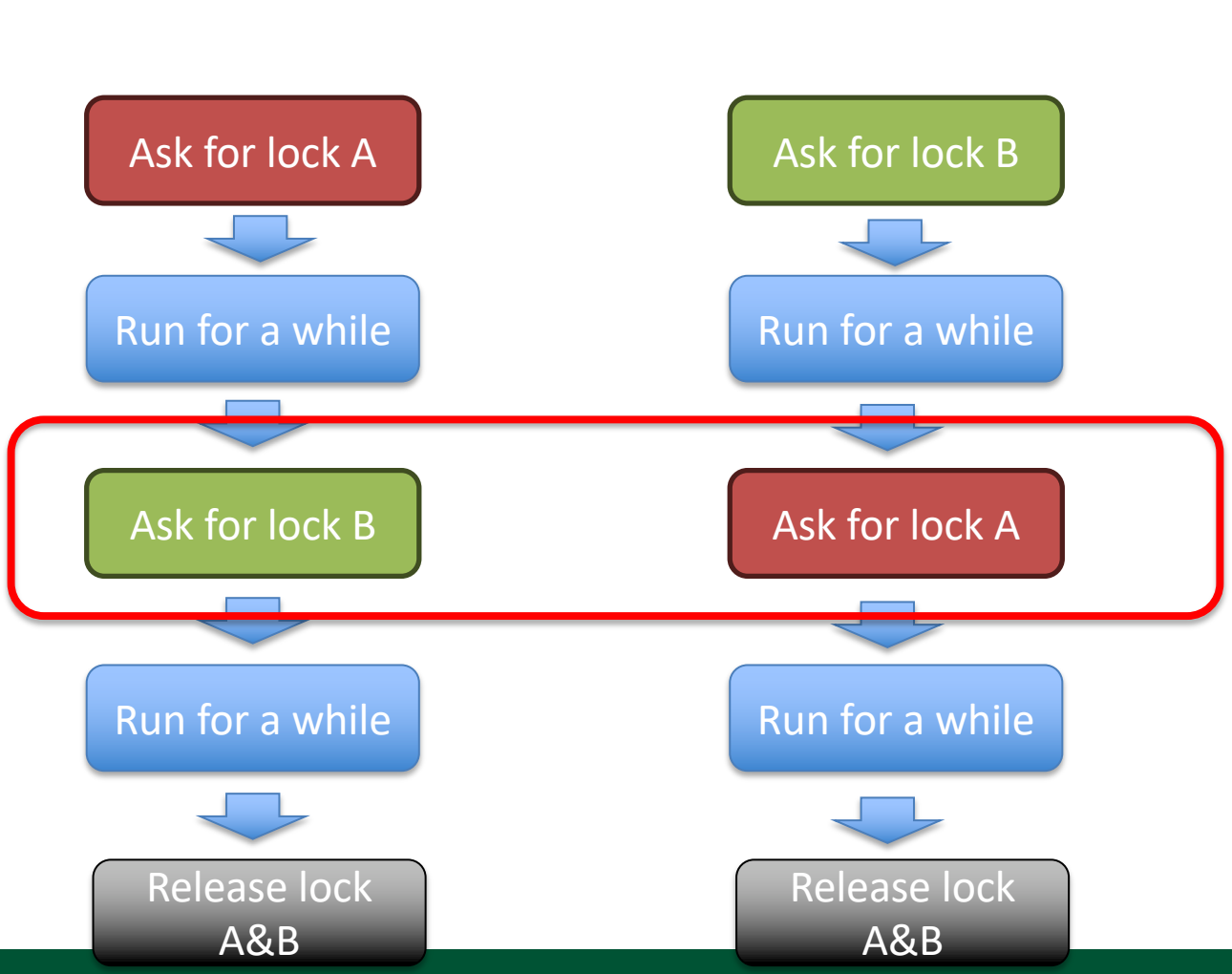
Deadlock Prevention– Requirements

- Removing condition: *Circular wait*
  - Assign each resource with a unique integer number
  - Require that each process request resources in an increasing order

Note: What recourse gets what number is not important. We care about enforcing the order of requests.

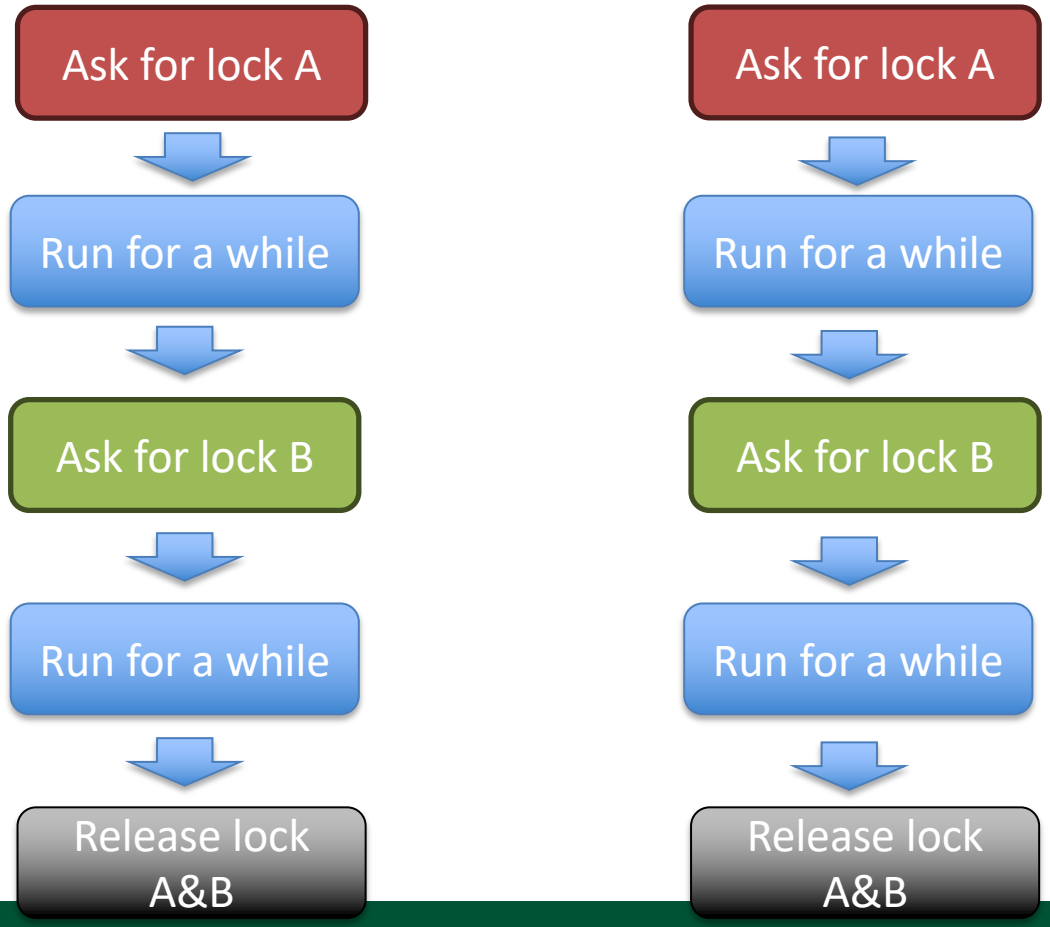
Deadlock Prevention

- This can be solved by enforcing the lock order



Deadlock Prevention

- This can be solved by enforcing the lock order



Lock A = 1
Lock B = 2

Enforce all process to request resources only in an increasing order

No deadlocks!

Deadlock Prevention– Requirements

- Removing condition: *Circular wait*
  - Assign each resource with a unique integer number
  - Require that each process request resources in an increasing order
  - Depends on application developers to write programs that follow the ordering

This is not always easy

An Account Transaction Example

```
void transaction(Account from, Account to, double amount) {
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

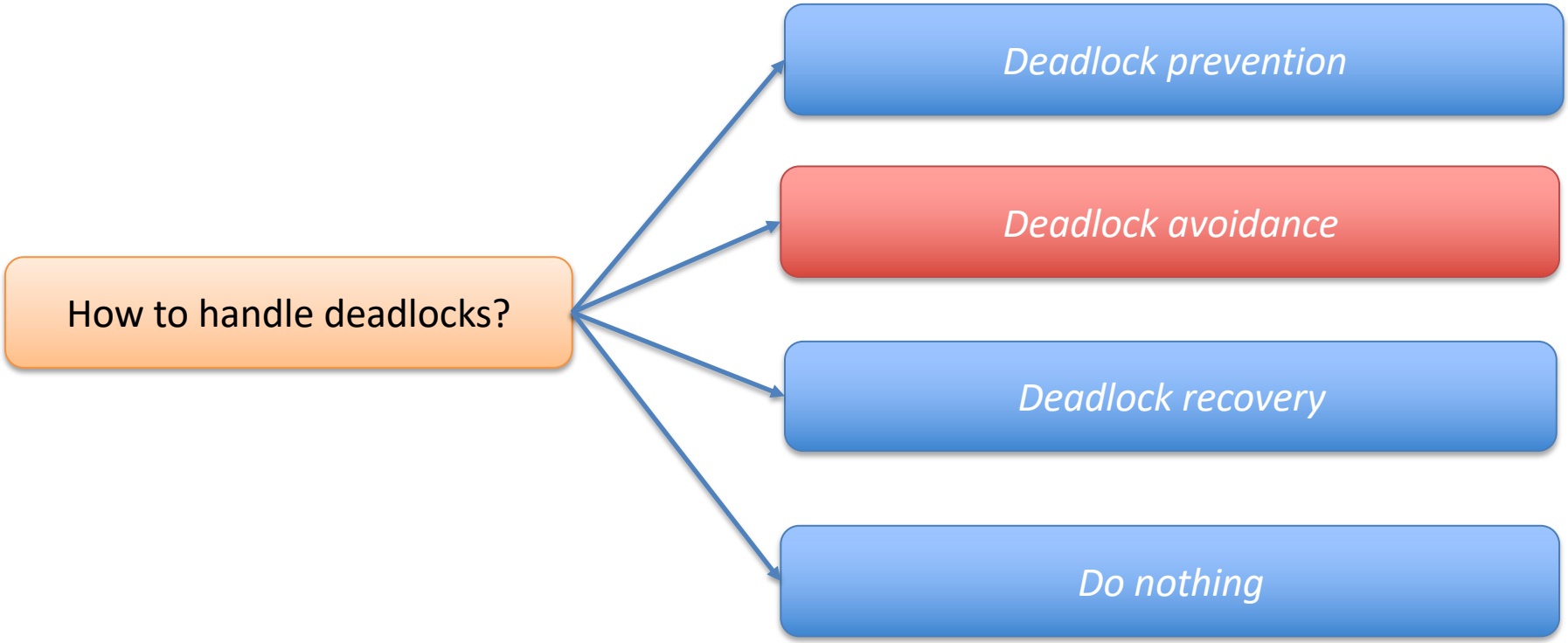
Does it work?

Think about calling this routine from 2 processes

- Transactions 1 and 2 execute concurrently
  - Transaction 1 transfers $25 from account A to account B
  - Transaction 2 transfers $50 from account B to account A

Methods for Handling Deadlocks

4 options to handle deadlocks



Note: these are NOT 4 steps.

Deadlock Avoidance

- More proactive than deadlock prevention
- Requires that the system has some additional *a priori* information available
- The simplest and most useful information: each process declares the *maximum number of resources* of each type that it may need

Deadlock Avoidance

Resource allocation state

Determined by

(1) number of available resource,

(2) number of allocated resources

(3) the *maximum* possible demands of the processes

- A deadlock-avoidance algorithm dynamically examines the **resource allocation state** to ensure that there can never be a circular-wait condition

Safe State

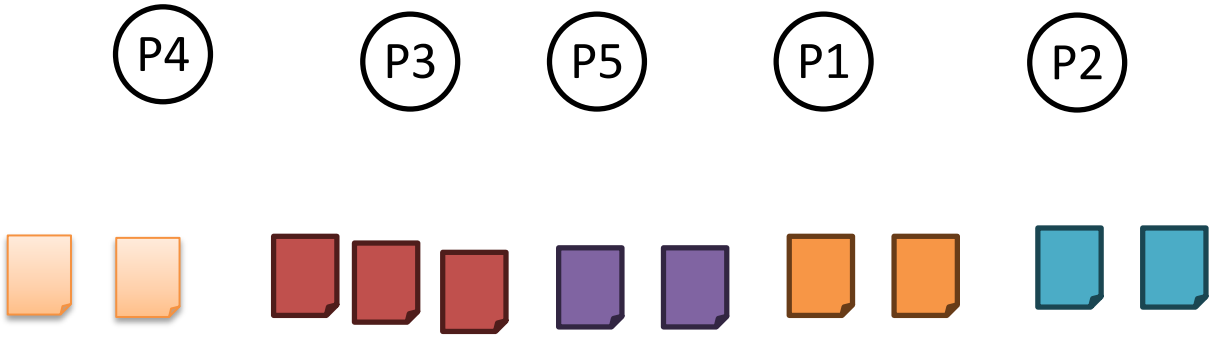
- A state is safe if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock.

Formal definition: Safe State

System is in a safe state if there exists a safe sequence: $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the system. Such that for each P_i : the resources P_i needs can be satisfied by:

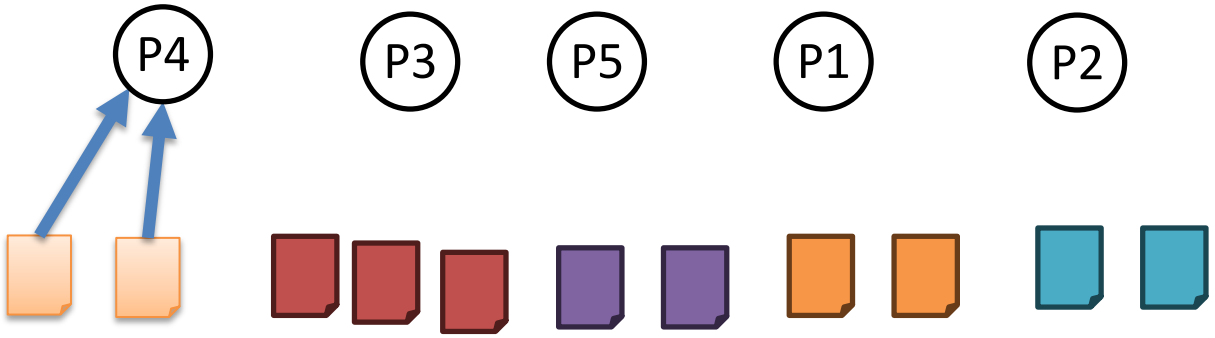
Currently available resources + resources held by all P_j with $j < i$

Safe State

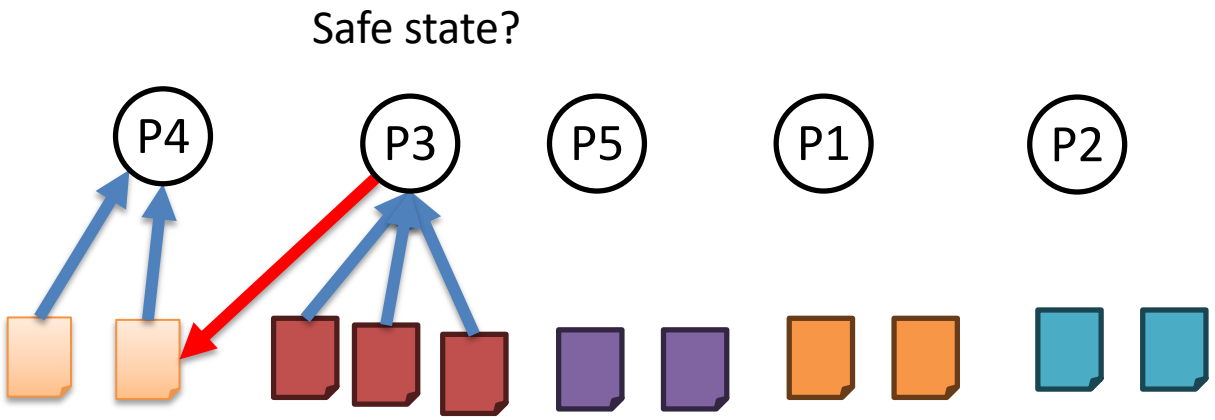


Safe State

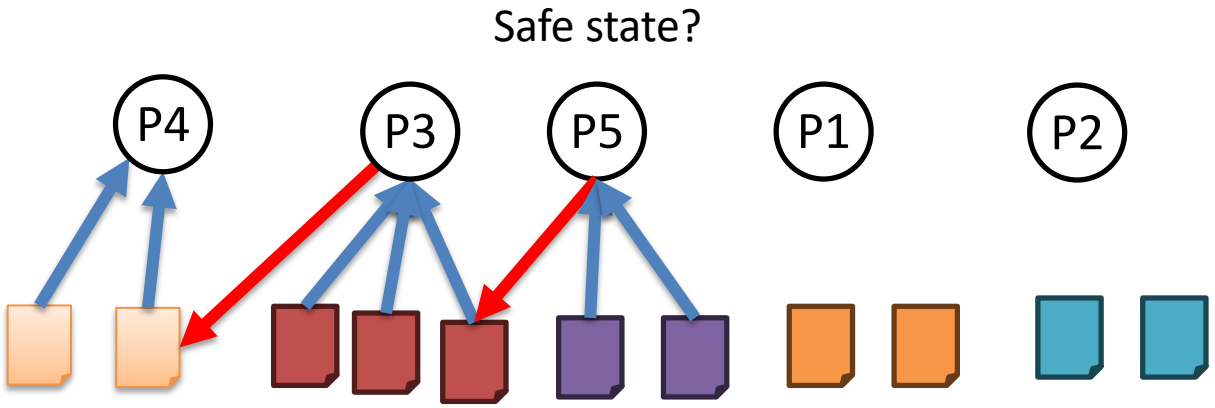
Safe state?



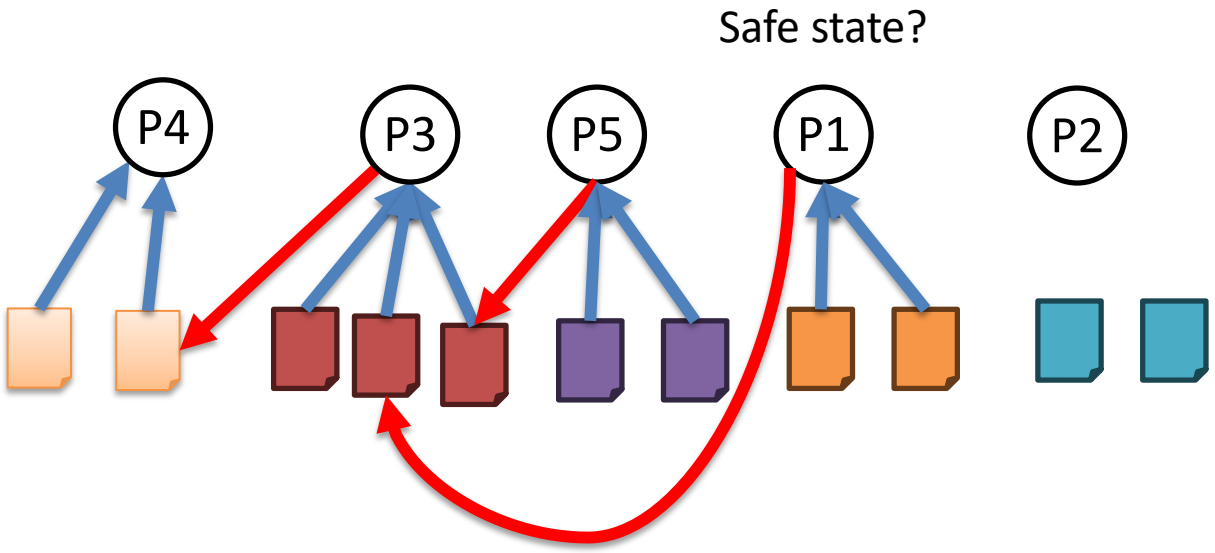
Safe State



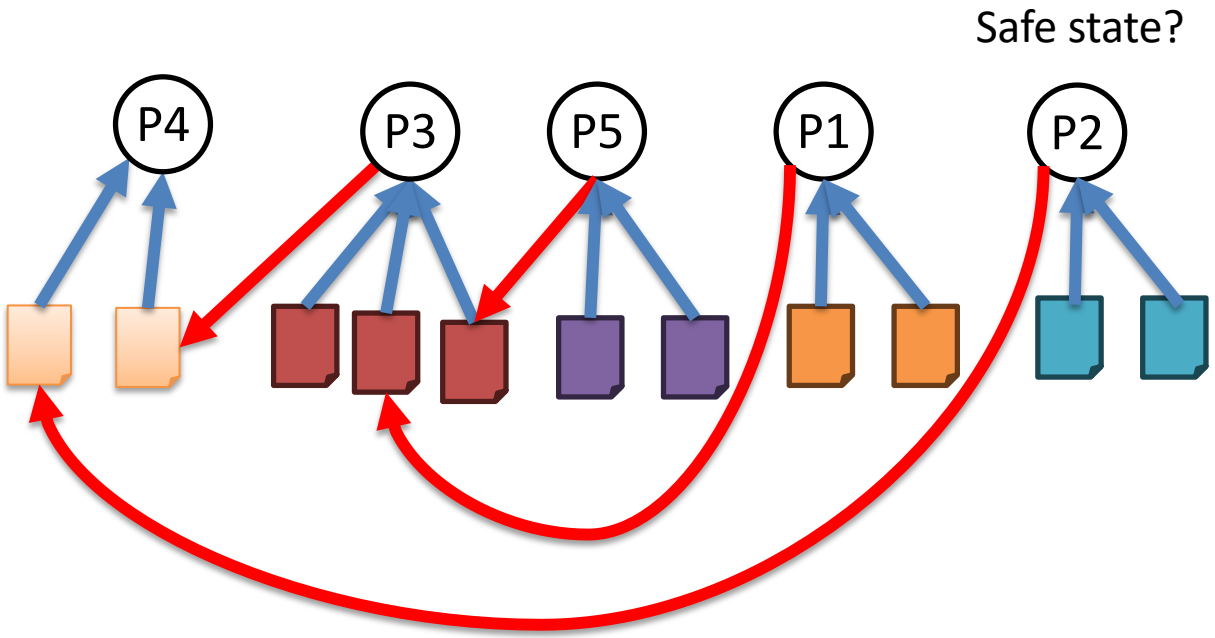
Safe State



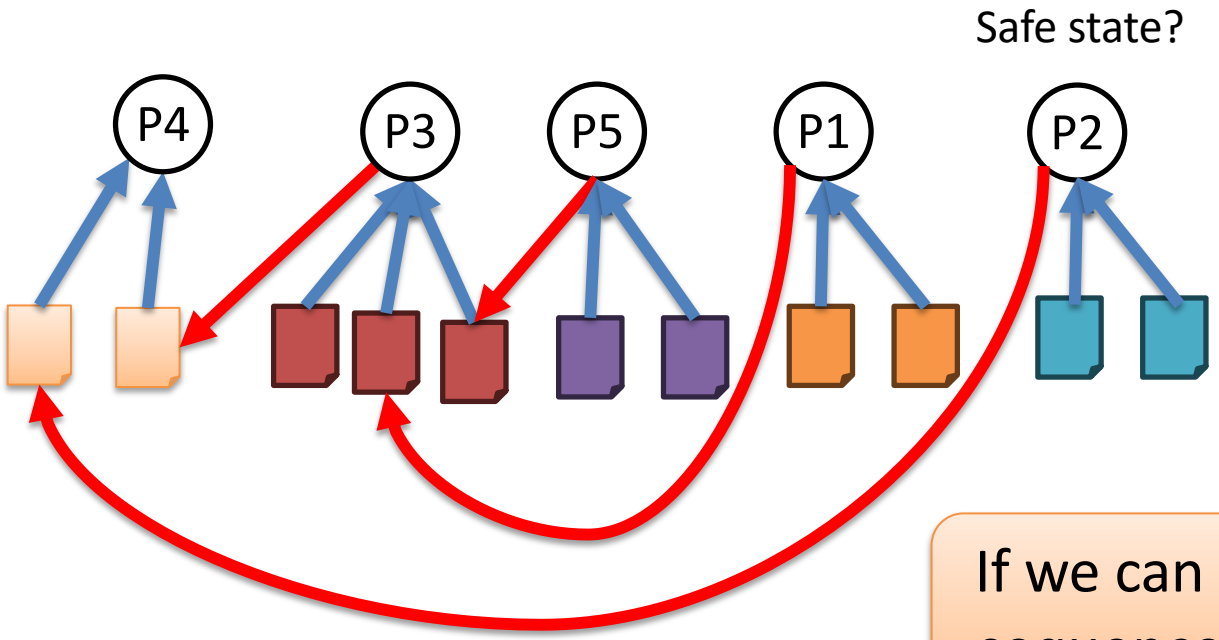
Safe State



Safe State



Safe State



If we can find at least one safe sequence, we are in safe state

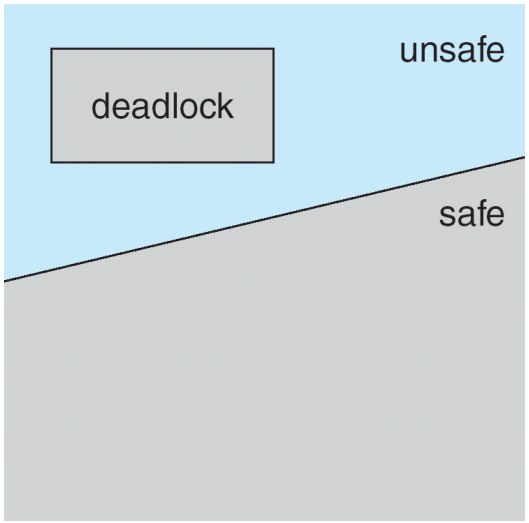
- For each process, all it needs is either available now or currently held ONLY by processes to its left (Pj).
  - Pi can wait until all Pj have finished.
- No process needs a resource held by processes to its right.
- When Pi terminates, Pi+1 can obtain its needed resources

Basic Facts

- Each resource allocation decision
  - Safe state → Unsafe state
  - Safe state → Safe state
  - Unsafe state ↛ Safe state (only possible if resources are released)
- We must check if we will still be in a *safe state* if we grant some resource allocation to a process
- Deadlock avoidance is achieved by ensuring that we never enter an unsafe state.

a safe state ⇒ no deadlocks
an unsafe state ⇒ possibility of deadlock

How to maintain a safe state?



Deadlock Avoidance Algorithms

- How to maintain a safe state?
  - Never allowing an unsafe state to be entered
- Only one instance of each resource type

Resource-Allocation-Graph Algorithm

- Multiple instances of each resource type

Banker's Algorithm

Deadlock Avoidance Algorithms

- How to maintain a safe state?
  - Never allowing an unsafe state to be entered
- Only one instance of each resource type

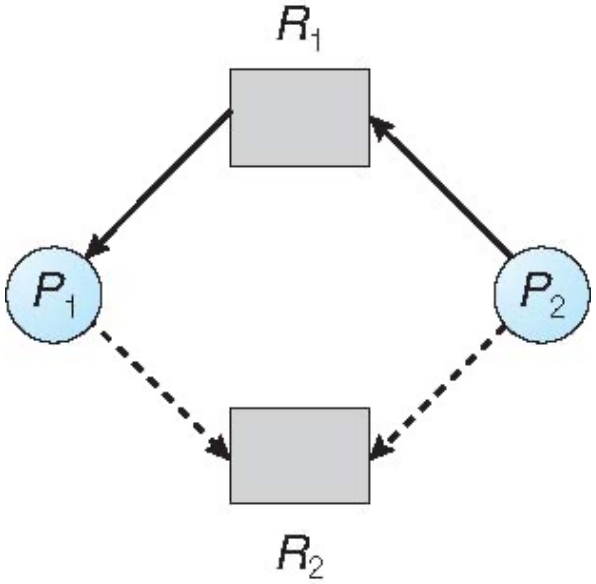
Resource-Allocation-Graph Algorithm

- Multiple instances of each resource type

Banker's Algorithm

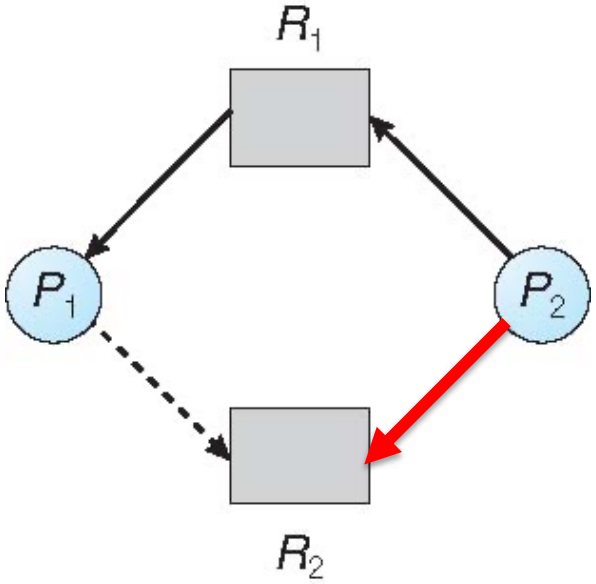
Resource Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - Represented by a dashed line



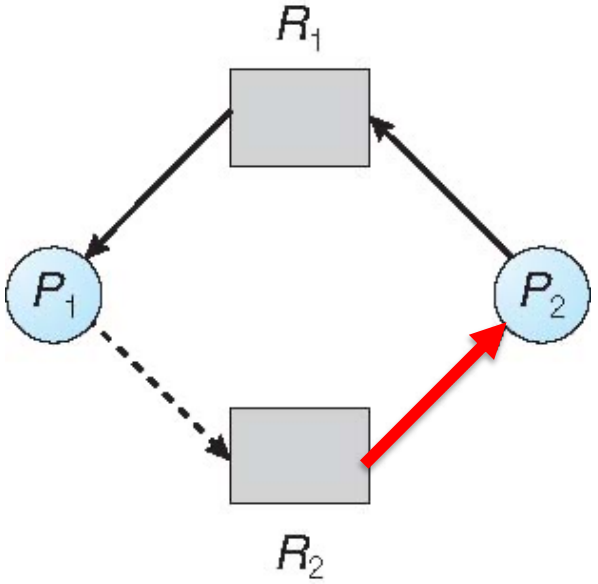
Resource Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - Represented by a dashed line
- Claim edge converts to a *request edge* when a process requests a resource



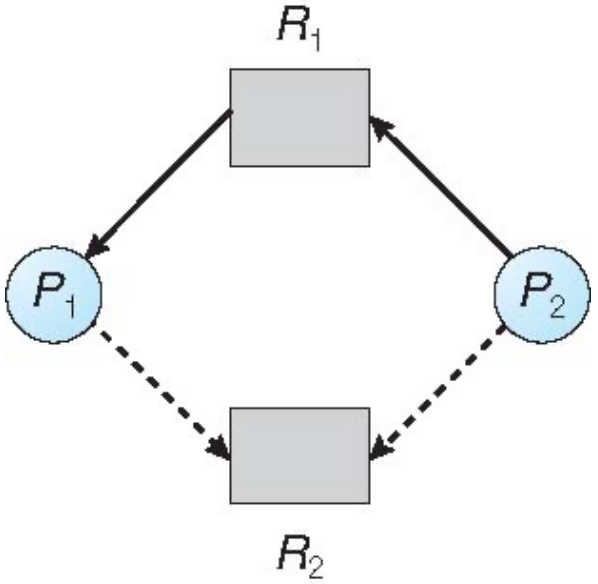
Resource Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - Represented by a dashed line
- Claim edge converts to a *request edge* when a process requests a resource
- Request edge converted to an *assignment edge* when the resource is allocated to the process



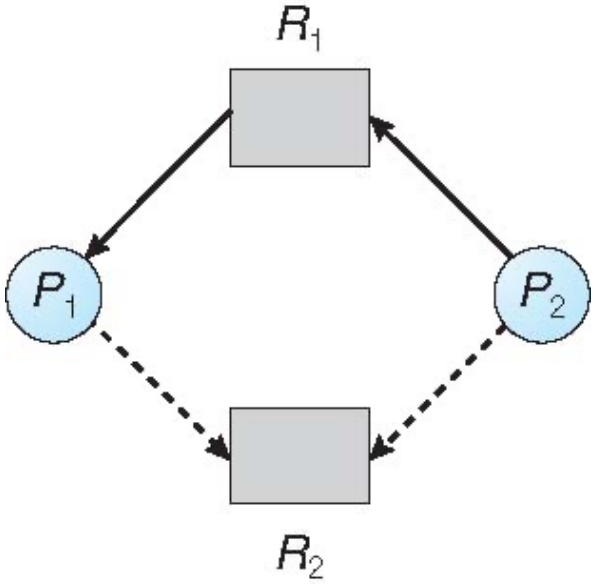
Resource Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - Represented by a dashed line
- Claim edge converts to a *request edge* when a process requests a resource
- Request edge converted to an *assignment edge* when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge



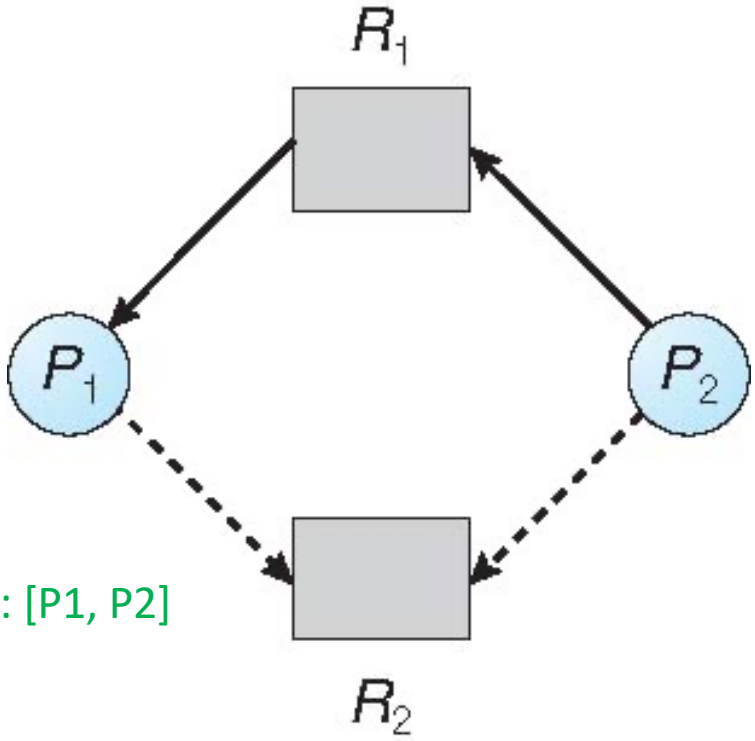
Resource Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j
  - Represented by a dashed line
- Claim edge converts to a *request edge* when a process requests a resource
- Request edge converted to an *assignment edge* when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system (maximum possible claim per resource)



Resource Allocation Graph

- Is this in a safe state?
- Can you find a sequence such that all processes can get what the resources they need to finish?



Both P1 and P2 need R1 and R2 to finish

1. Grant P1's request on R2
2. P1 finishes and releases all
3. Grant P2's request on R2&R1
4. P2 finishes and releases all

Safe sequence: [P1, P2]

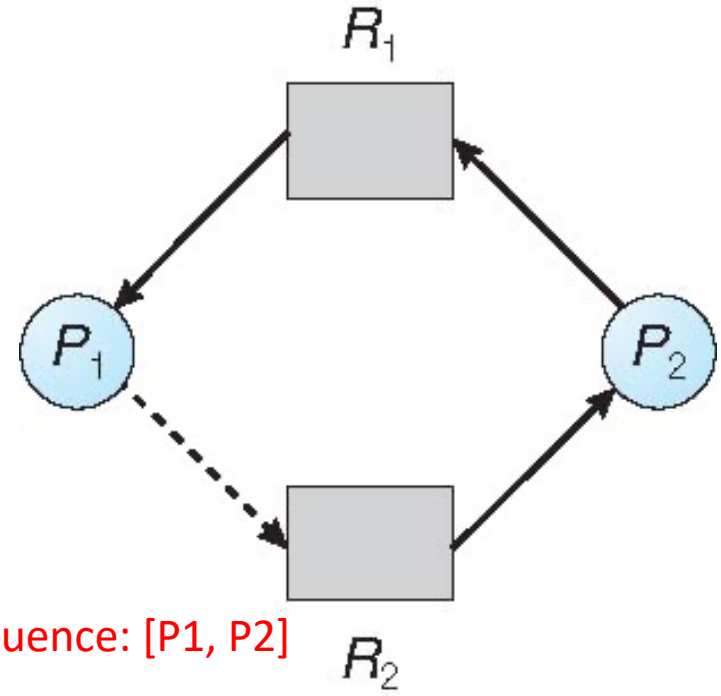
1. Grant P2's request on R2
2. P2 cannot finish as it cannot get R1
3. P1 cannot finish as it cannot get R2

Unsafe sequence: [P2, P1]

If we can find at least one safe sequence, we are in a safe state

Unsafe State In Resource Allocation Graph

- What about this?
- Is this in a safe state?
- If it is unsafe, why?



Both P1 and P2 need R1 and R2 to finish

1. Cannot grant P1's request on R2
2. P1 cannot finish as it cannot get R2
3. P2 cannot finish as it cannot get R1

Unsafe sequence: [P1, P2]

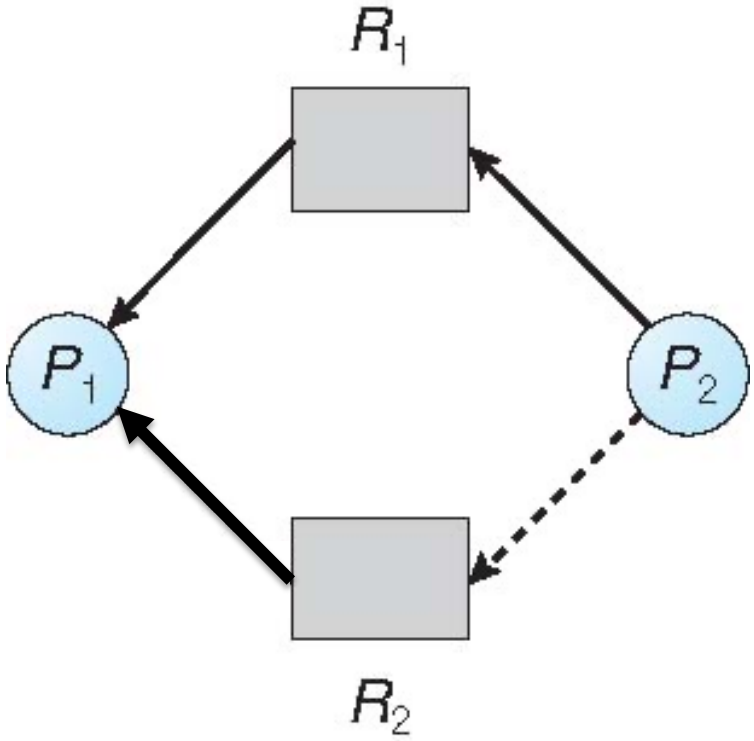
1. Grant P2's request on R2
2. P2 cannot finish as it cannot get R1
3. P1 cannot finish as it cannot get R2

Unsafe sequence: [P2, P1]

If we cannot find at least one safe sequence, we are in an unsafe state

Resource Allocation Graph

- Is this in a safe state?
- Can you find a sequence such that all processes can get what the resources they need to finish?



Both P1 and P2 need R1 and R2 to finish

1. P1 finishes and releases all
2. Grant P2's request on R2&R1
3. P2 finishes and releases all

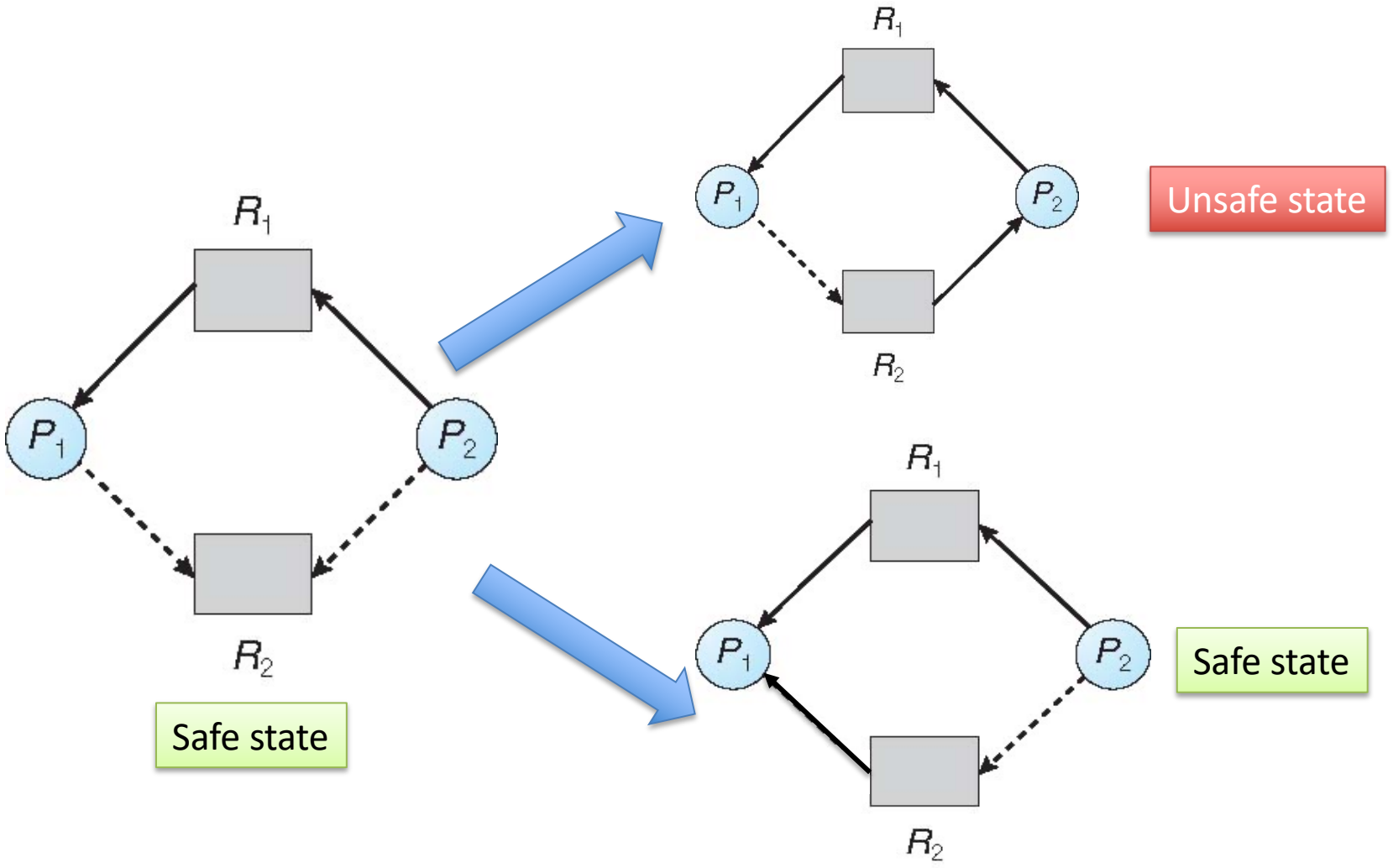
} Safe sequence:

1. Cannot grant P2's request on R2
2. P2 cannot finish as it cannot get R1
3. P1 can finish but before P2

} Unsafe sequence: [P2, P1]

If we can find at least one safe sequence, we are in a safe state

Resource Allocation Graph

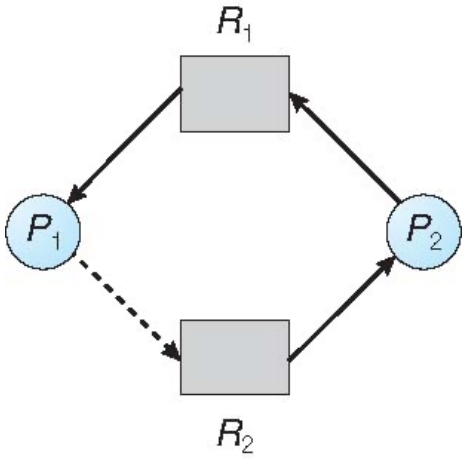


Resource Allocation Graph Algorithm

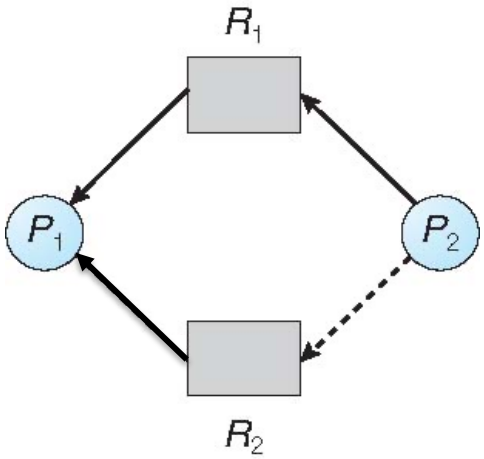
- How to use a graph to avoid an unsafe state?

Resource Allocation Graph Algorithm

1. Suppose that process P_i requests a resource R_j
2. The request can be granted only if converting to an assignment edge does not result in a cycle in the resource allocation graph
3. Cycles are evaluated using all types of edges, including claim edges



Unsafe state: form a cycle



Safe state: no cycle

Deadlock Avoidance Algorithms

- How to maintain a safe state?
  - Never allowing an unsafe state to be entered
- Only one instance of each resource type

Resource-Allocation-Graph Algorithm

- Multiple instances of each resource type

Banker's Algorithm

Banker's Algorithm

- Suppose we have multiple instances
- Requirements:
  - Each process must *a priori* claim maximum use
  - When a process requests a resource it may have to wait
  - When a process gets all its resources it must return them in a finite amount of time
- Banker's algorithms is a bookkeeping method for tracking and assigning resources

Data Structures for Banker's Algorithm

- 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**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

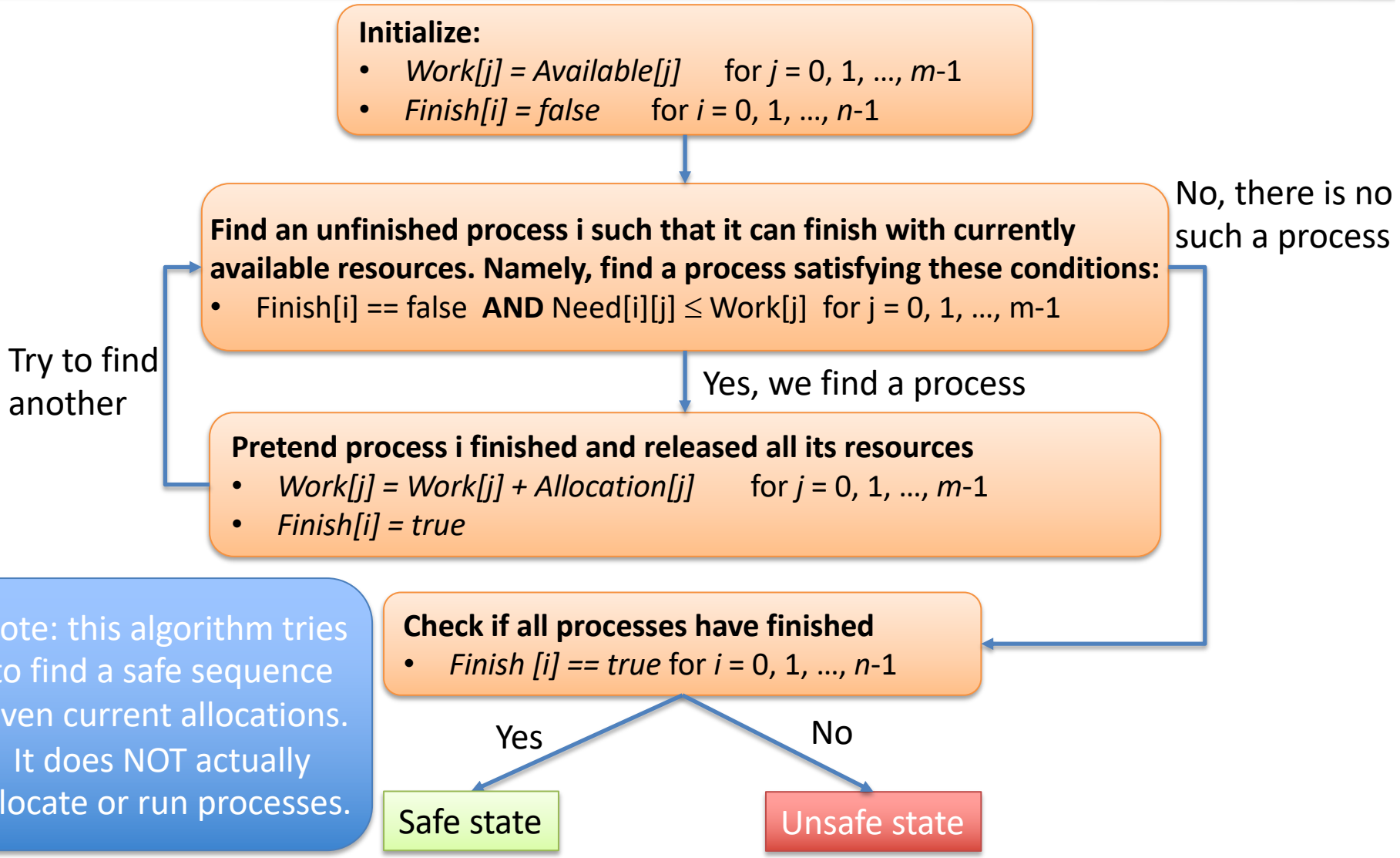
  $$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

- **Request**: n x m matrix. If $Request[i,j] = k$, then P_i may is requesting k more instances of R_j now (not necessarily can finish)

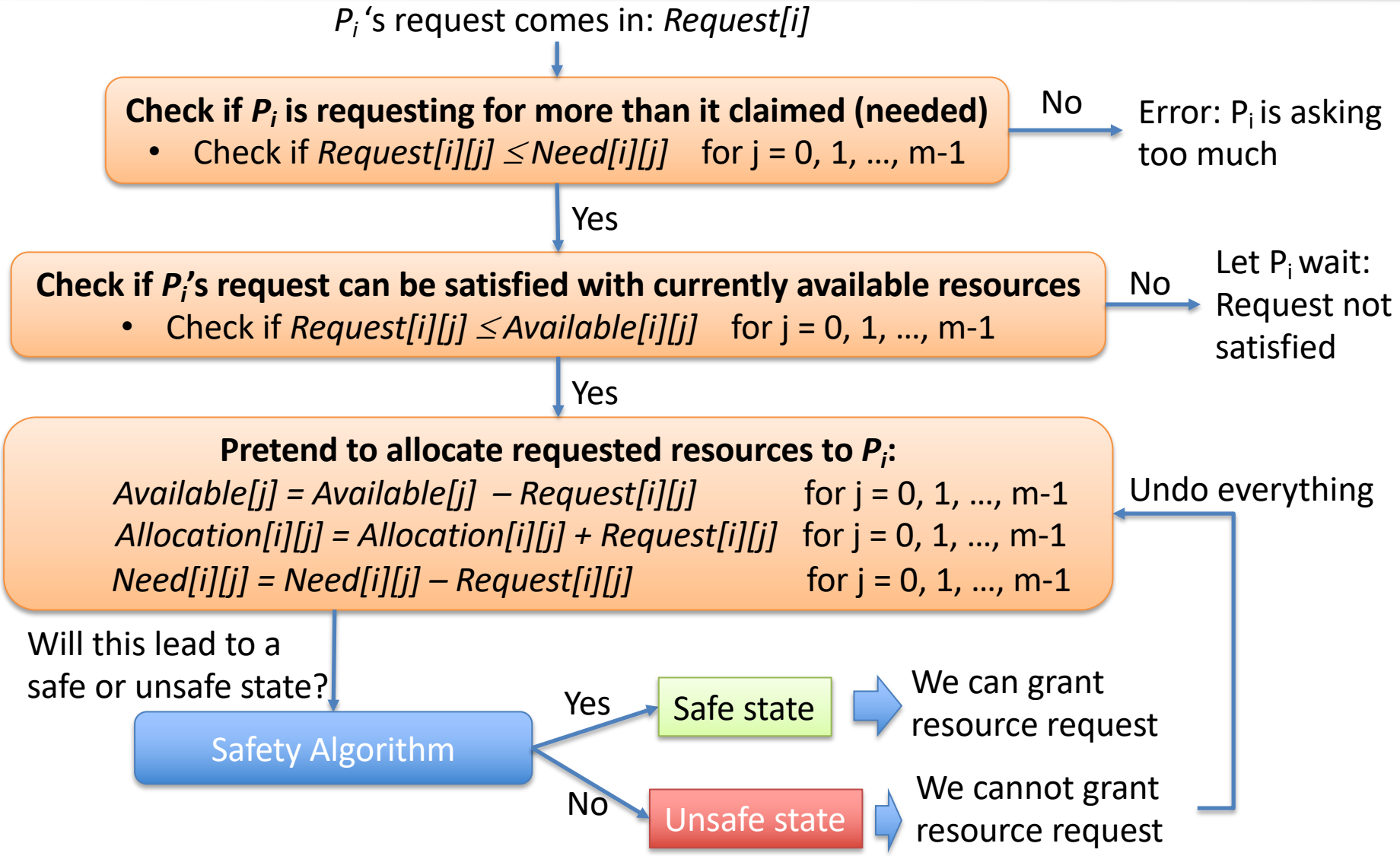
Safety Algorithm

Resource-Request Algorithm

Safety Algorithm: check if we are in a safe state or not



Resource-Request Algorithm: Check if we can grant the resource request for Process P_i



Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types:
  - A (10 instances), B (5 instances), C (7 instances)
- Snapshot at time T_k:

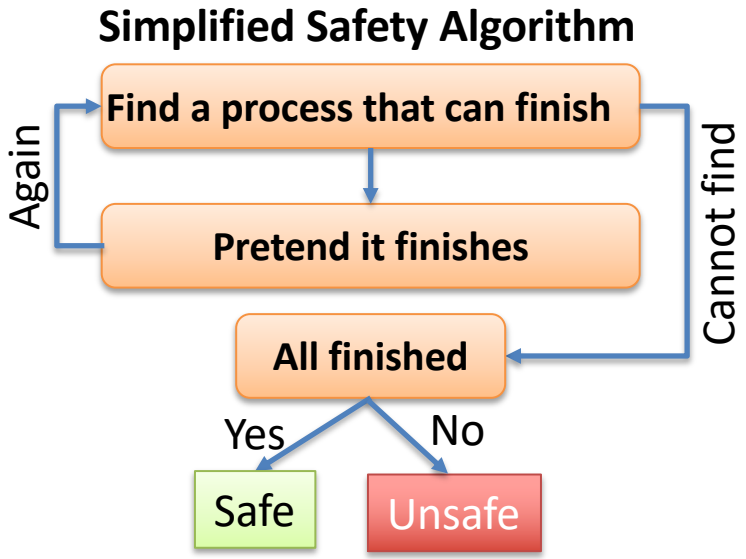
Process	*Allocation*	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Is it in a safe state?

Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	3 3 2	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*No*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*No*
P_4	0 0 2	4 3 3		4 3 1	*No*

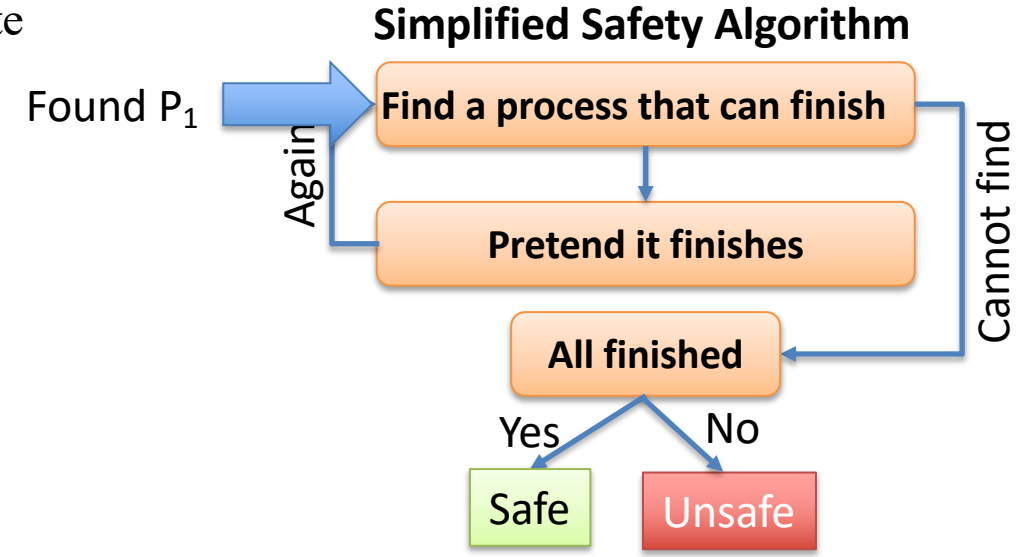
Find a sequence for processes to complete (one after the other)



Check for Safety

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	3 3 2	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*No*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*No*
P_4	0 0 2	4 3 3		4 3 1	*No*

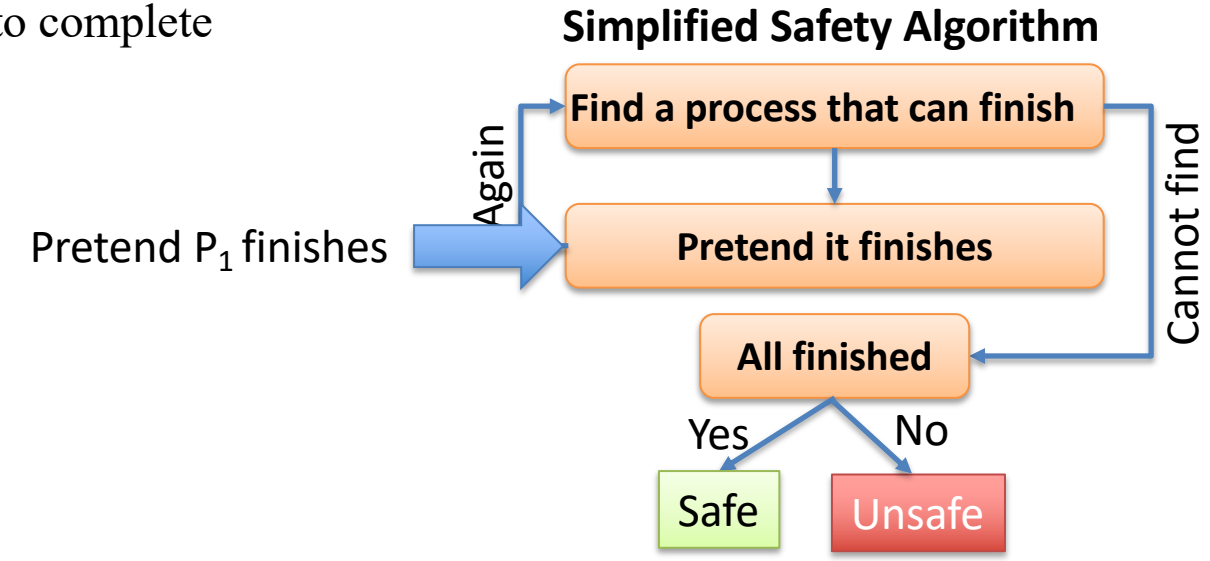
Find a sequence for processes to complete (one after the other)



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	5 3 2	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*No*
P_4	0 0 2	4 3 3		4 3 1	*No*

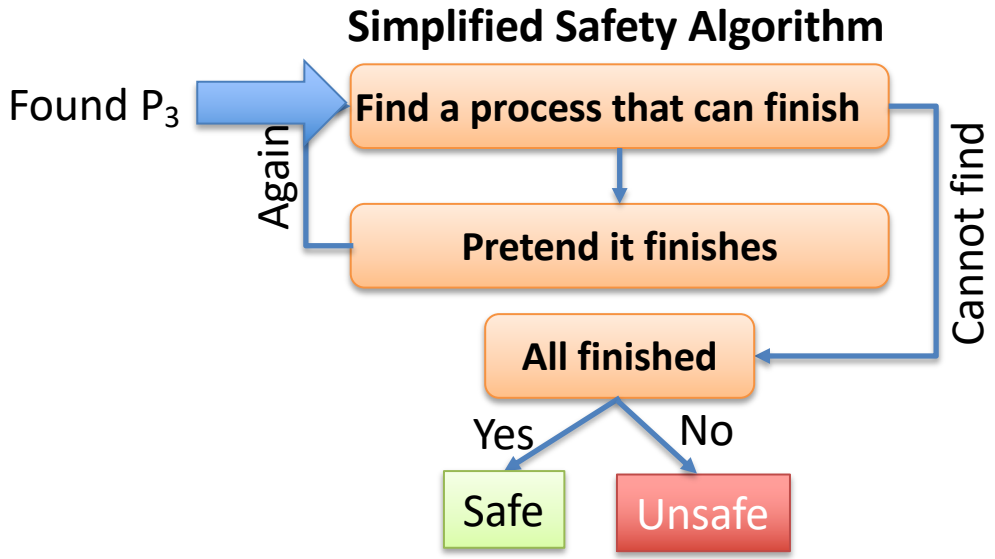
Find a sequence for processes to complete (one after the other): P_1



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	5 3 2	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*No*
P_4	0 0 2	4 3 3		4 3 1	*No*

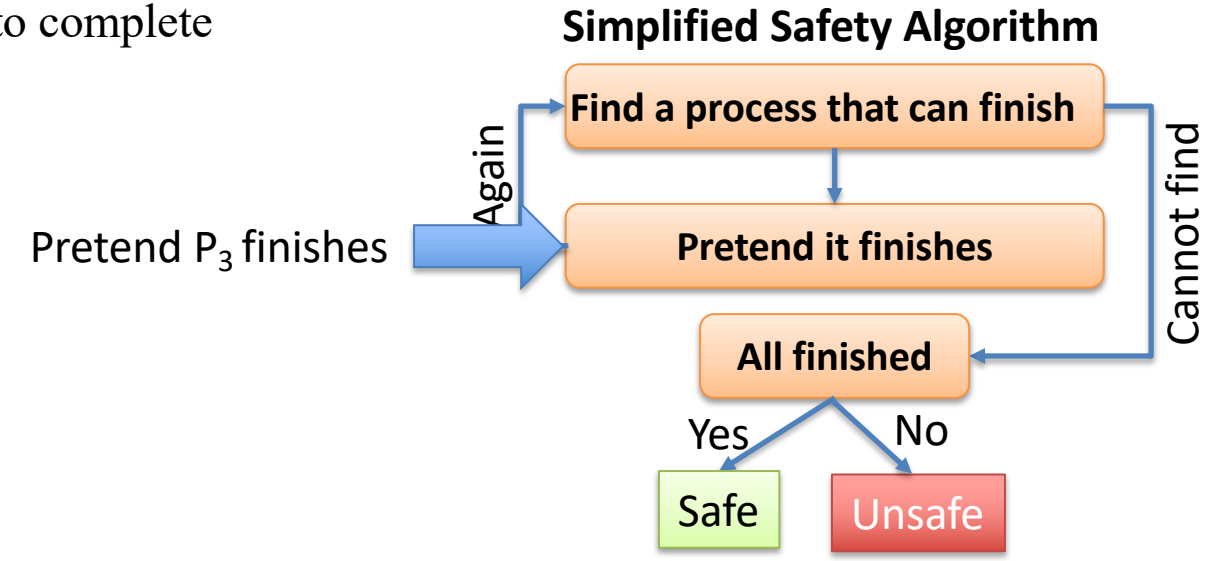
Find a sequence for processes to complete (one after the other): P_1



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	7 4 3	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*No*

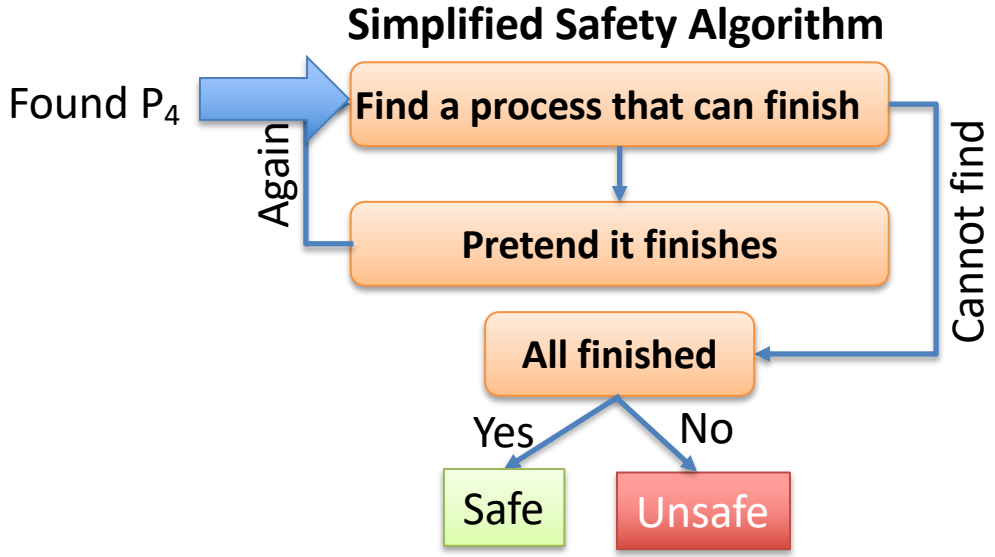
Find a sequence for processes to complete (one after the other): P_1 P_3



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	7 4 3	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*No*

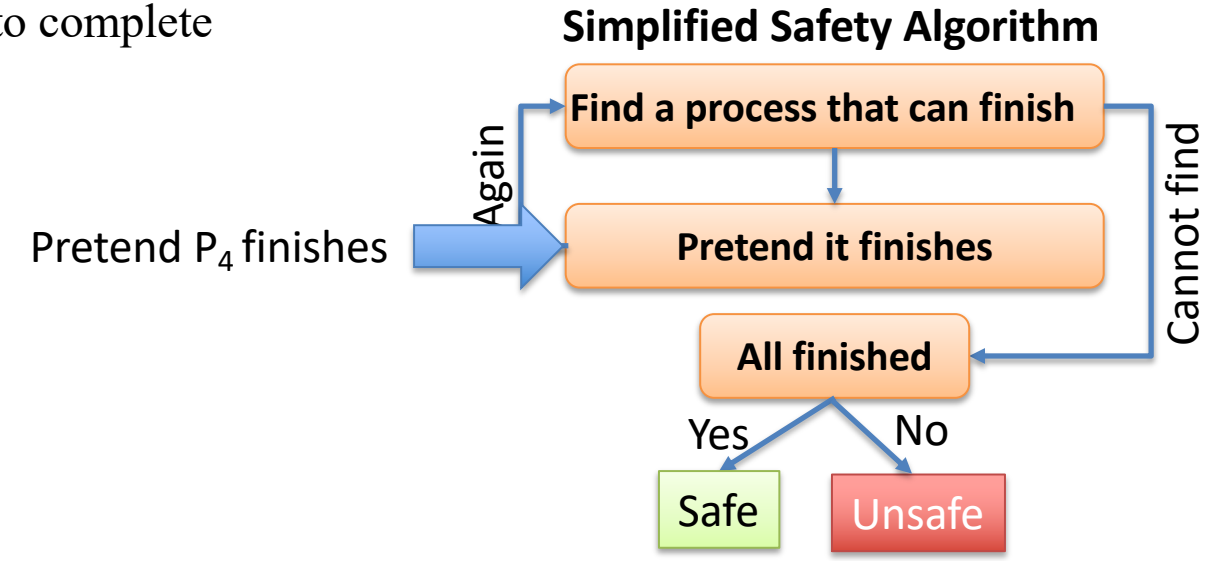
Find a sequence for processes to complete (one after the other): P_1 P_3



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	7 4 5	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

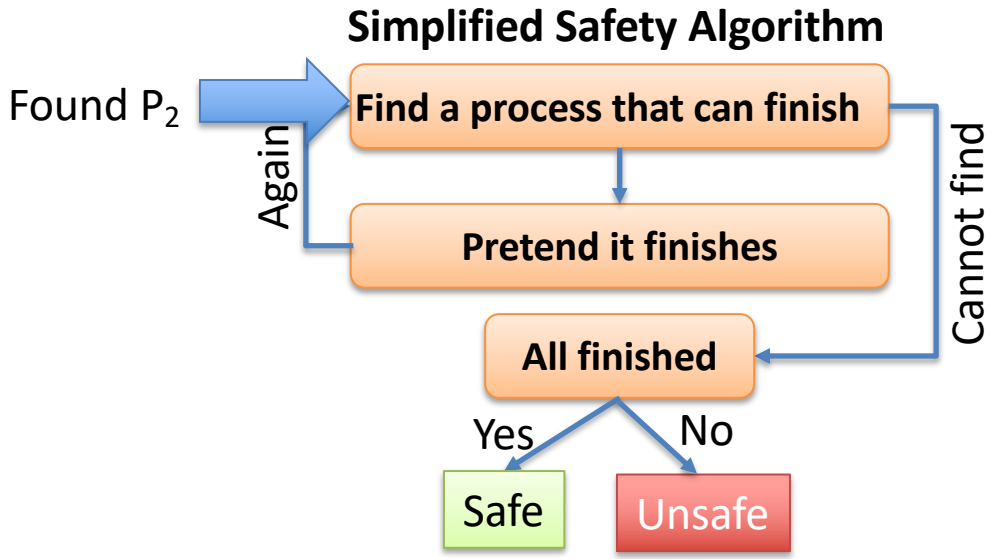
Find a sequence for processes to complete (one after the other): P_1 P_3 P_4



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	7 4 5	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

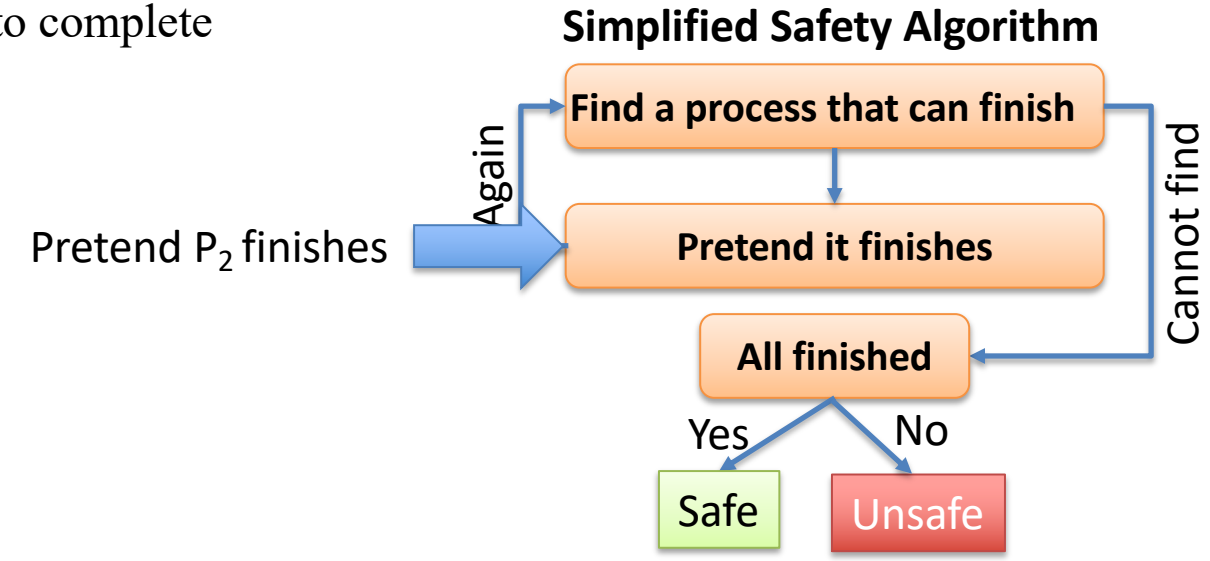
Find a sequence for processes to complete (one after the other): P_1 P_3 P_4



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	10 4 7	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*Yes*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

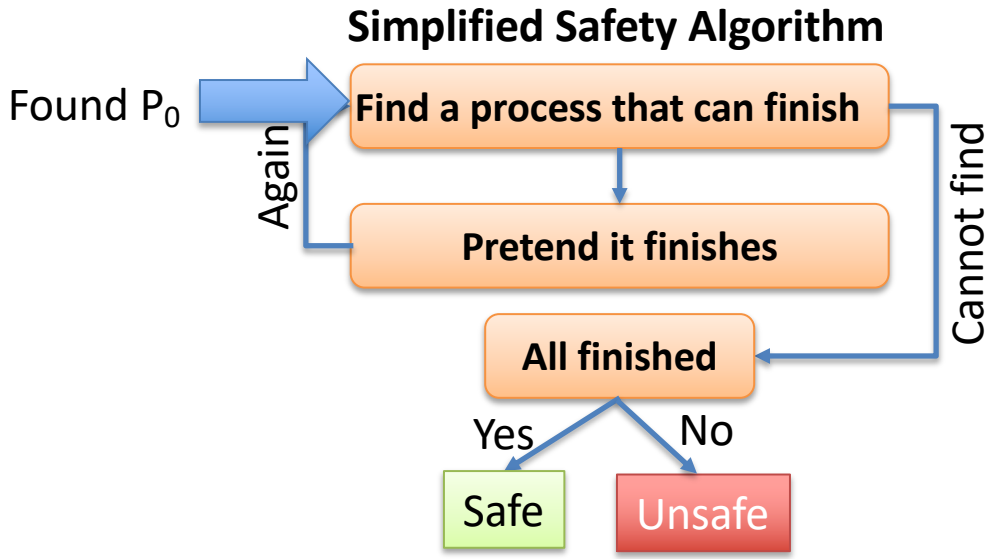
Find a sequence for processes to complete (one after the other): $P_1\ P_3\ P_4\ P_2$



Check for Safety

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	10 4 7 ←	7 4 3	*No*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*Yes*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

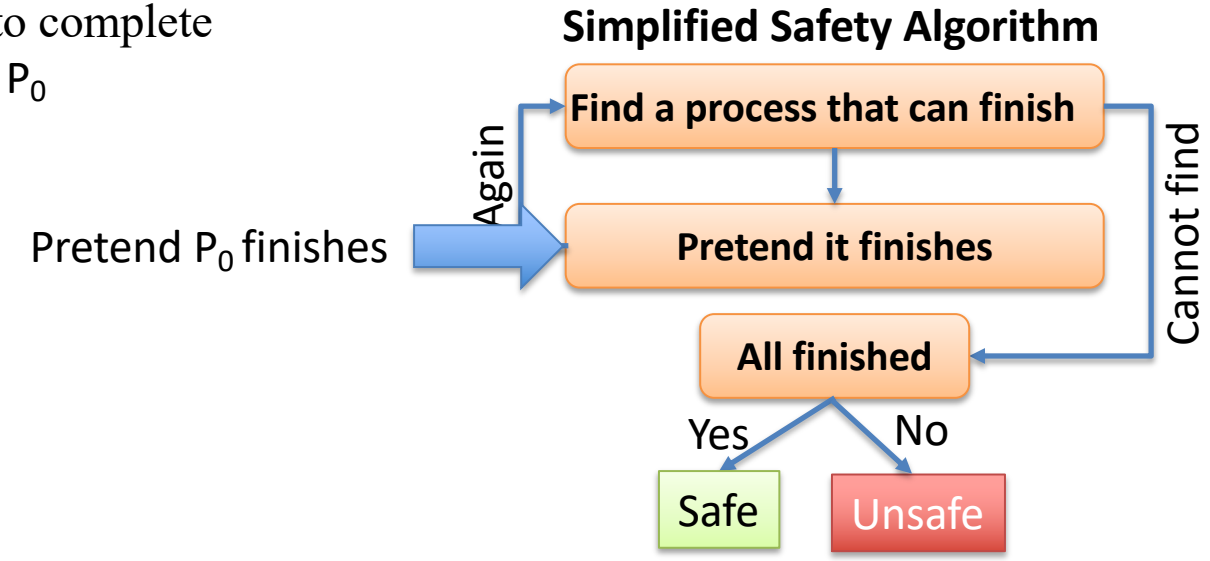
Find a sequence for processes to complete (one after the other): $P_1\ P_3\ P_4\ P_2$



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	10 5 7	7 4 3	*Yes*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*Yes*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

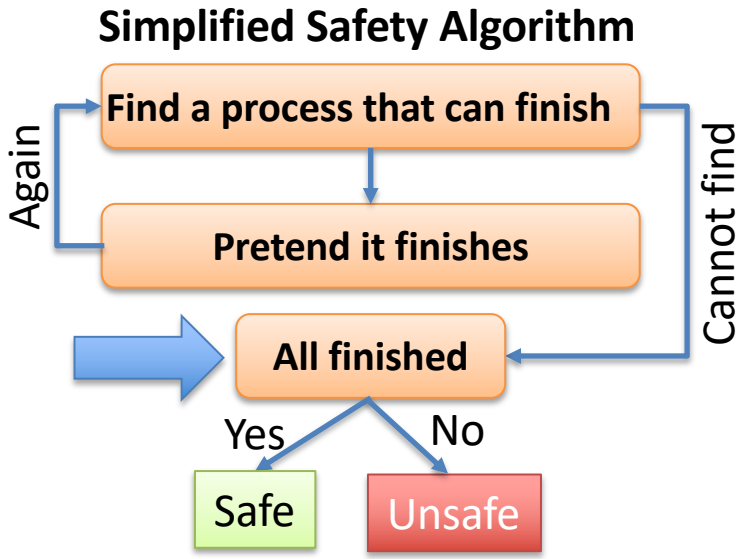
Find a sequence for processes to complete (one after the other): $P_1\ P_3\ P_4\ P_2\ P_0$



Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	10 5 7	7 4 3	*Yes*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*Yes*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

Find a sequence for processes to complete (one after the other): $P_1\ P_3\ P_4\ P_2\ P_0$



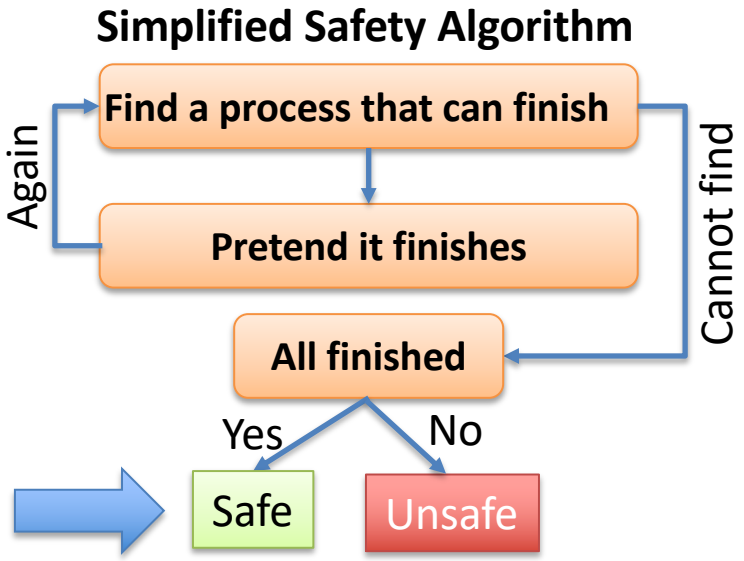
Check for Safety

Process	*Allocation*	*Max*	*Work*	*Need*	Finish
	A B C	*A B C*	*A B C*	*A B C*	
P_0	0 1 0	7 5 3	10 5 7	7 4 3	*Yes*
P_1	2 0 0	3 2 2		1 2 2	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*Yes*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*Yes*

Find a sequence for processes to complete (one after the other): $P_1\ P_3\ P_4\ P_2\ P_0$

Safe sequence

Is the safe sequence unique?
What is the runtime of finding a sequence?

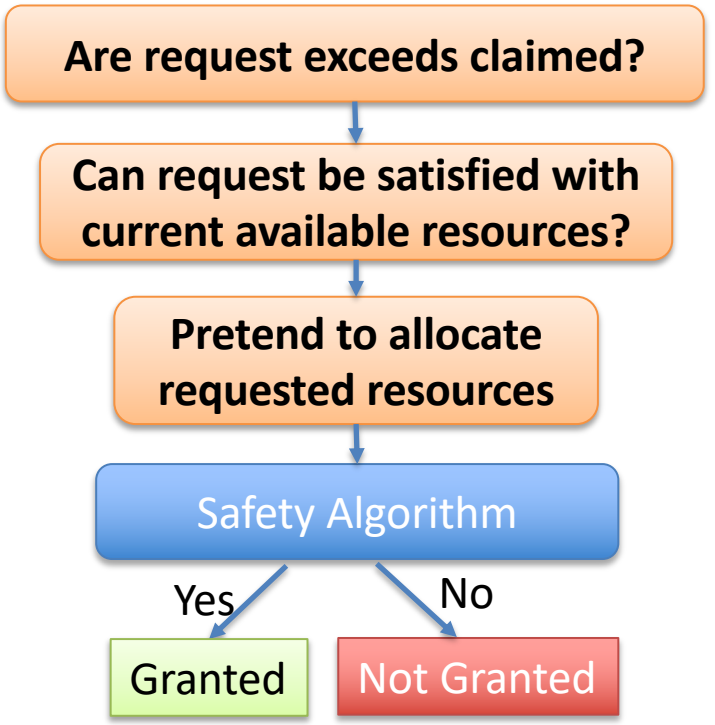


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

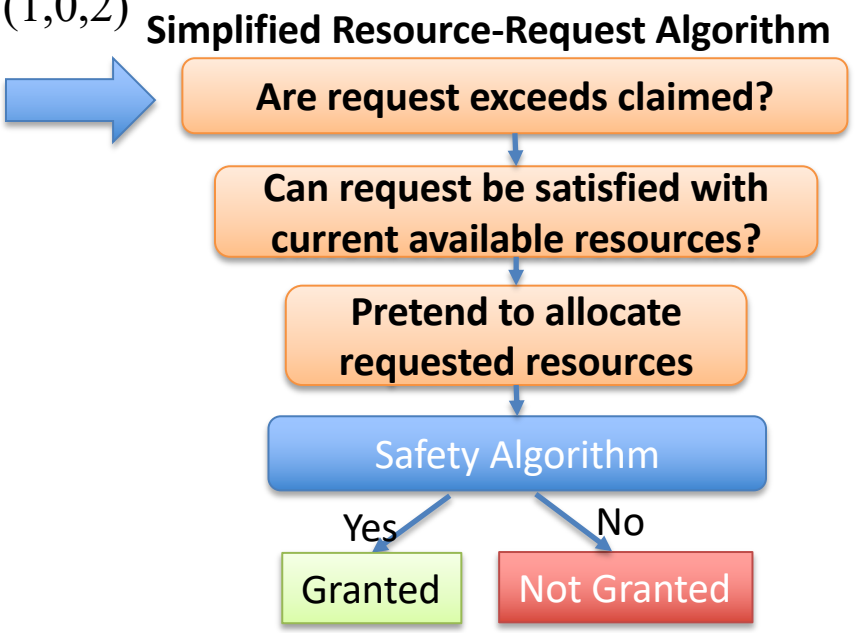
Simplified Resource-Request Algorithm



P_1 makes a requests (1,0,2)

Process	Allocation	Max	Available	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	3 3 2	7 4 3	No
P_1	2 0 0	3 2 2		1 2 2	No
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	No
P_4	0 0 2	4 3 3		4 3 1	No

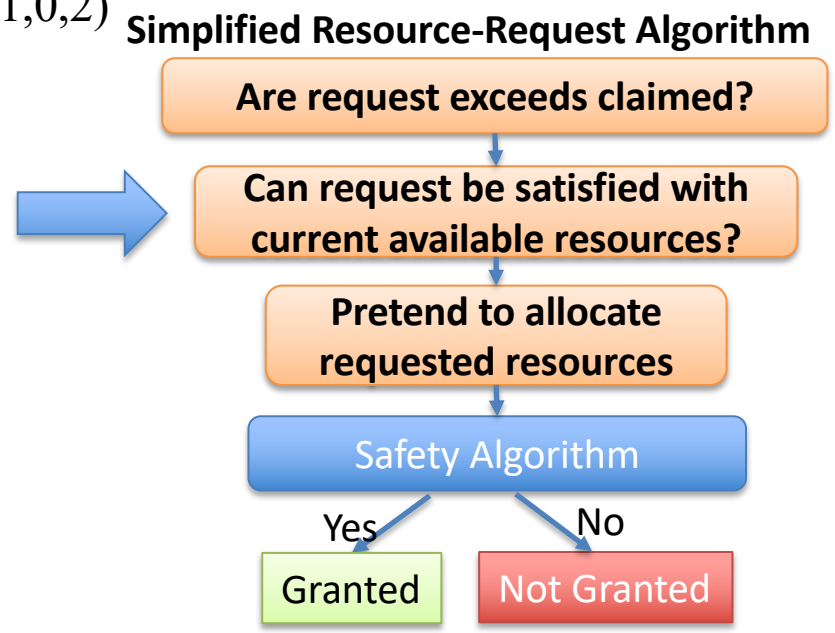
- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?



P_1 makes a requests (1,0,2)

Process	Allocation	Max	Available	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	3 3 2	7 4 3	No
P_1	2 0 0	3 2 2		1 2 2	No
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	No
P_4	0 0 2	4 3 3		4 3 1	No

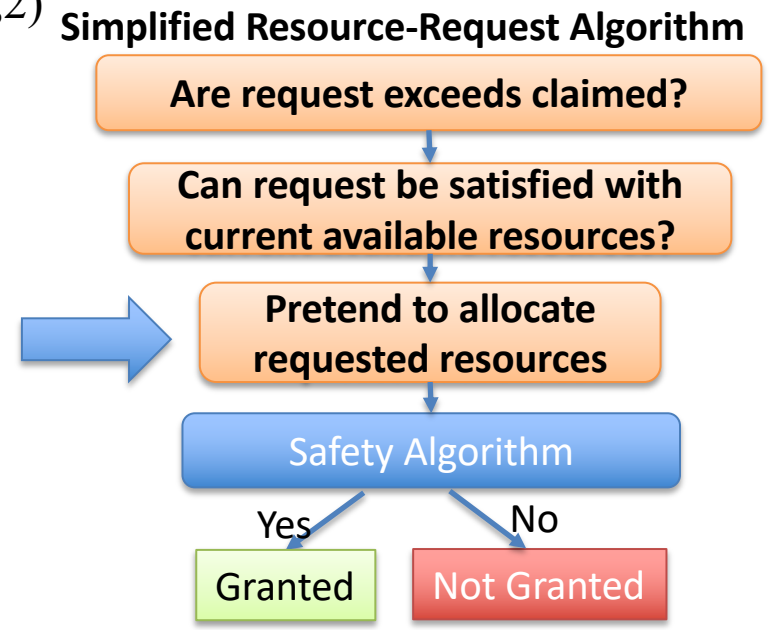
- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?



P_1 makes a requests (1,0,2)

Process	Allocation	Max	Available	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	2 3 0	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	No
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	No
P_4	0 0 2	4 3 3		4 3 1	No

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

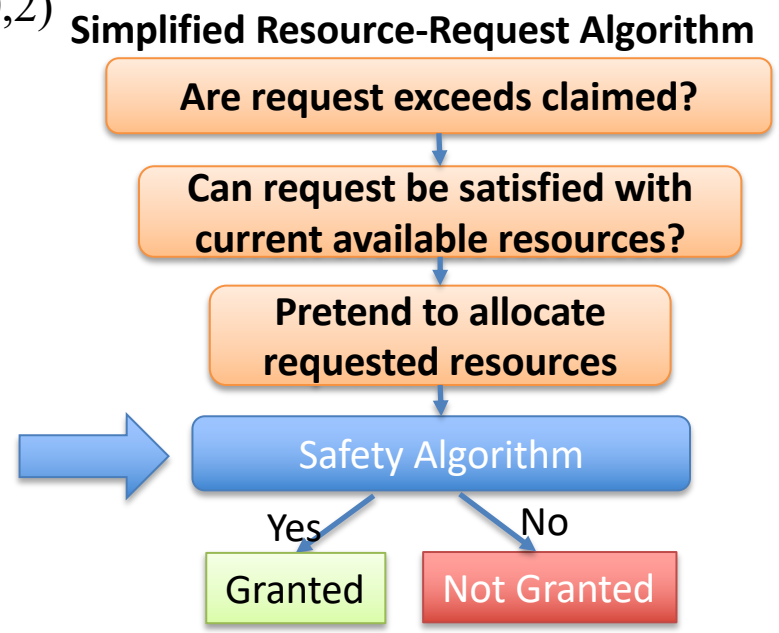


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	2 3 0	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	No
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	No
P_4	0 0 2	4 3 3		4 3 1	No

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: P_1

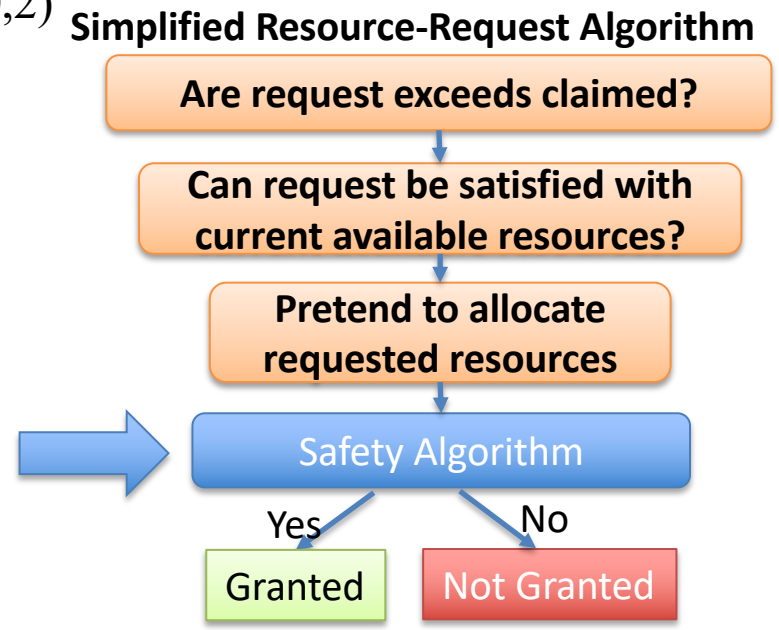


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	5 3 2	7 4 3	*No*
P_1	3 0 2	3 2 2		0 2 0	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*No*
P_4	0 0 2	4 3 3		4 3 1	*No*

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: P_1

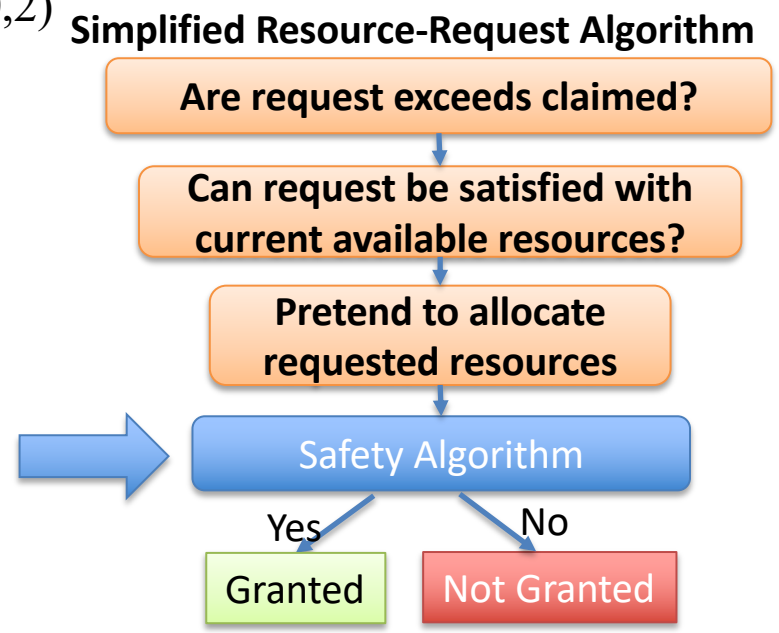


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	5 3 2	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	No
P_4	0 0 2	4 3 3		4 3 1	No

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: P_1 P_3

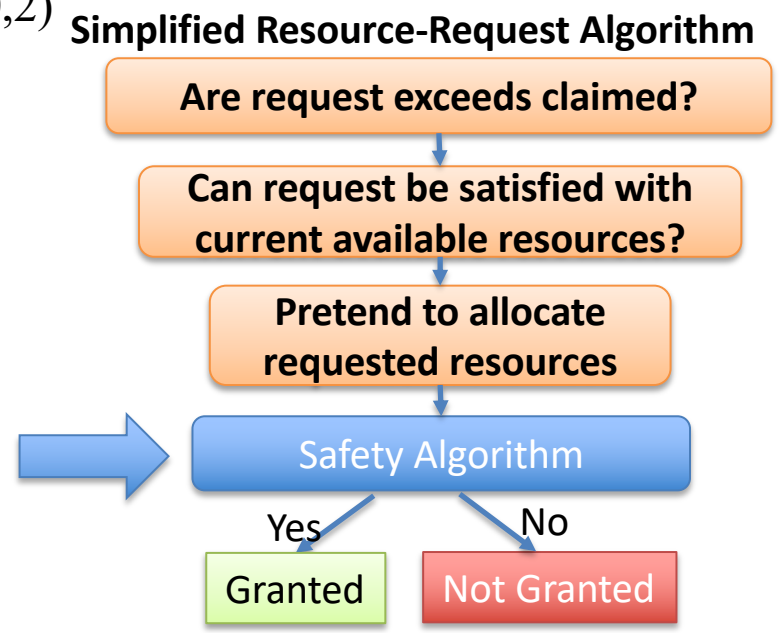


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 4 3	7 4 3	*No*
P_1	3 0 2	3 2 2		0 2 0	*Yes*
P_2	3 0 2	9 0 2		6 0 0	*No*
P_3	2 1 1	2 2 2		0 1 1	*Yes*
P_4	0 0 2	4 3 3		4 3 1	*No*

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: P_1 P_3

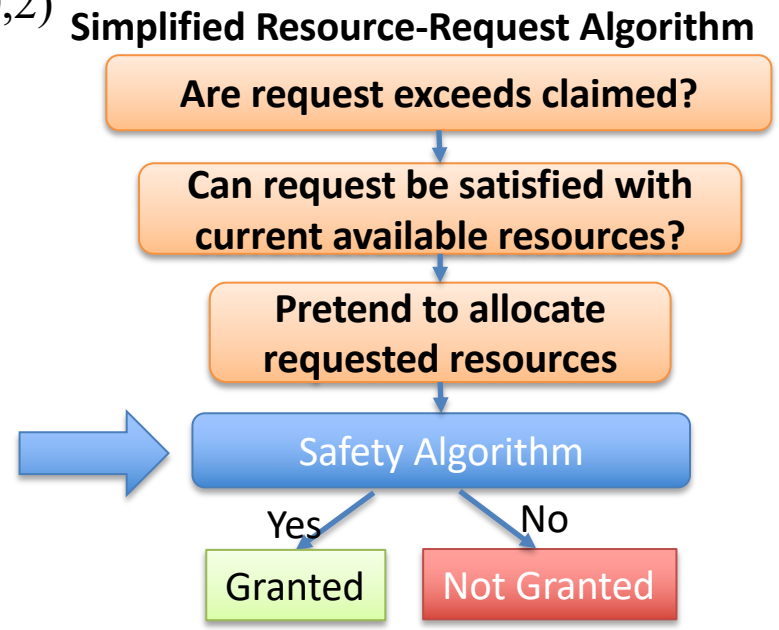


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 4 3	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	No

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4$

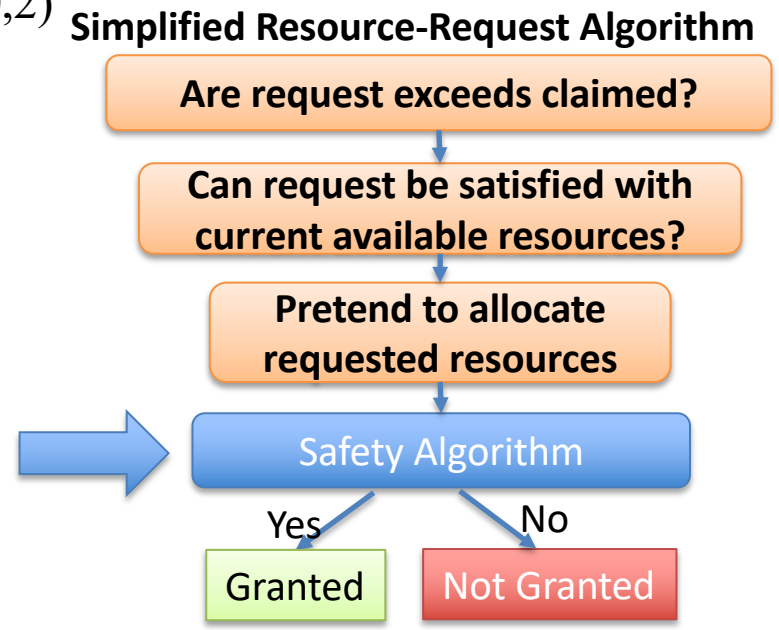


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 4 5	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4$

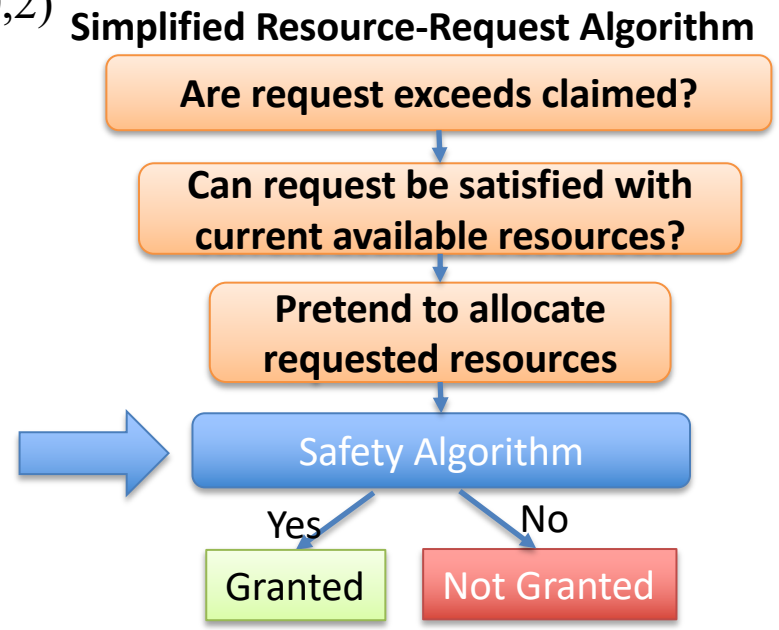


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 4 5 →	7 4 3	No
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4\ P_0$

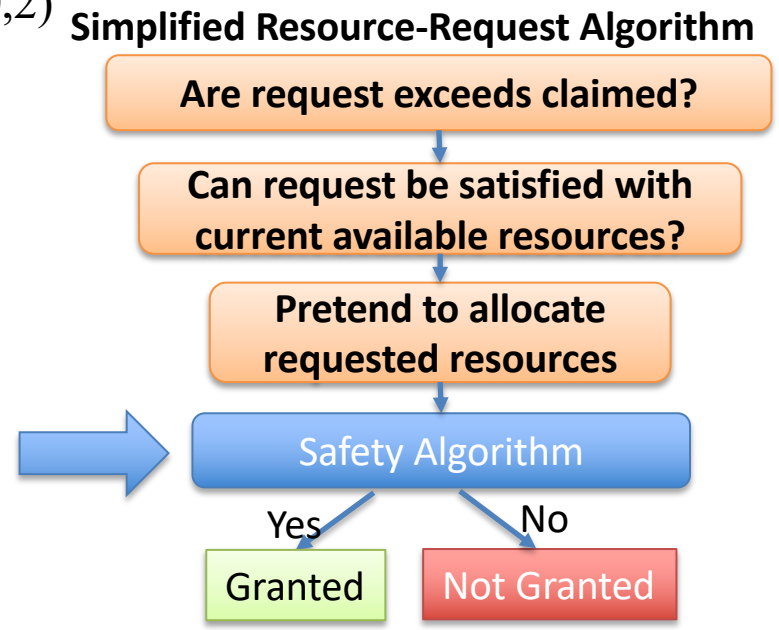


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 5 5	7 4 3	Yes
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4\ P_0$

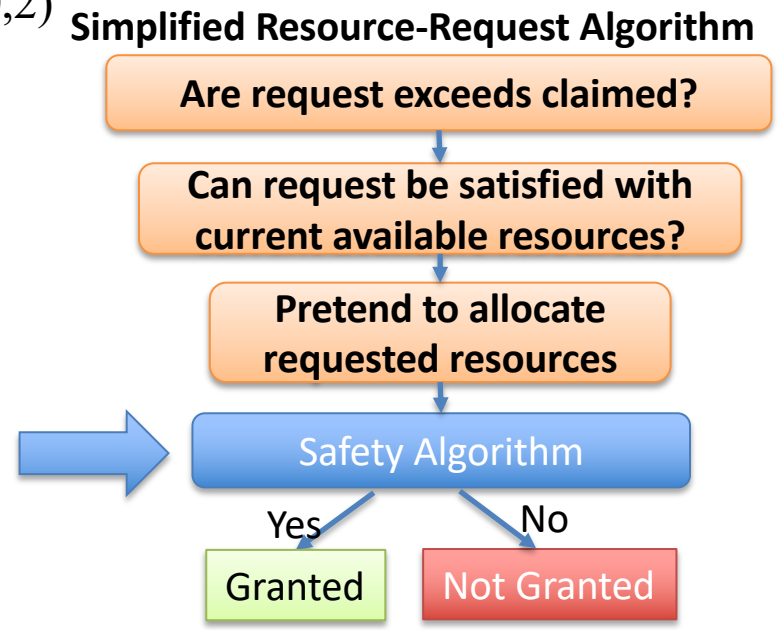


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	7 5 5	7 4 3	Yes
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	No
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4\ P_0\ P_2$

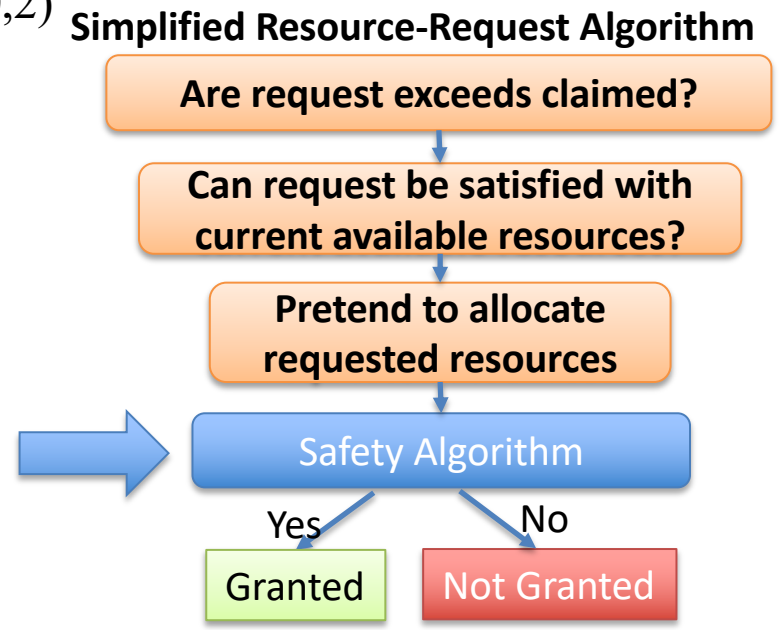


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	10 5 7	7 4 3	Yes
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	Yes
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4\ P_0\ P_2$

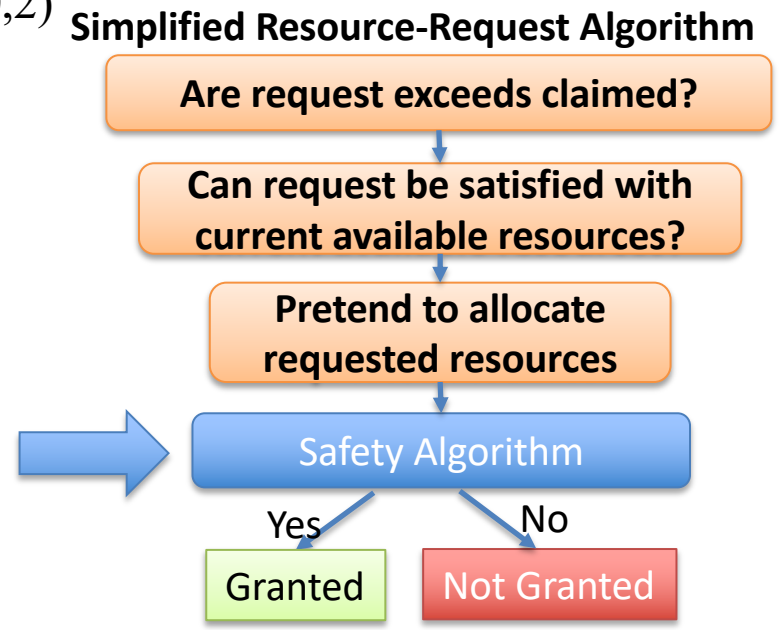


P_1 makes a requests (1,0,2)

Process	Allocation	Max	Work	Need	Finish
	A B C	A B C	A B C	A B C	
P_0	0 1 0	7 5 3	10 5 7	7 4 3	Yes
P_1	3 0 2	3 2 2		0 2 0	Yes
P_2	3 0 2	9 0 2		6 0 0	Yes
P_3	2 1 1	2 2 2		0 1 1	Yes
P_4	0 0 2	4 3 3		4 3 1	Yes

- Let's advance the system with P_1 making request (1,0,2)
- Can we grant it?

Safe sequence: $P_1\ P_3\ P_4\ P_0\ P_2$

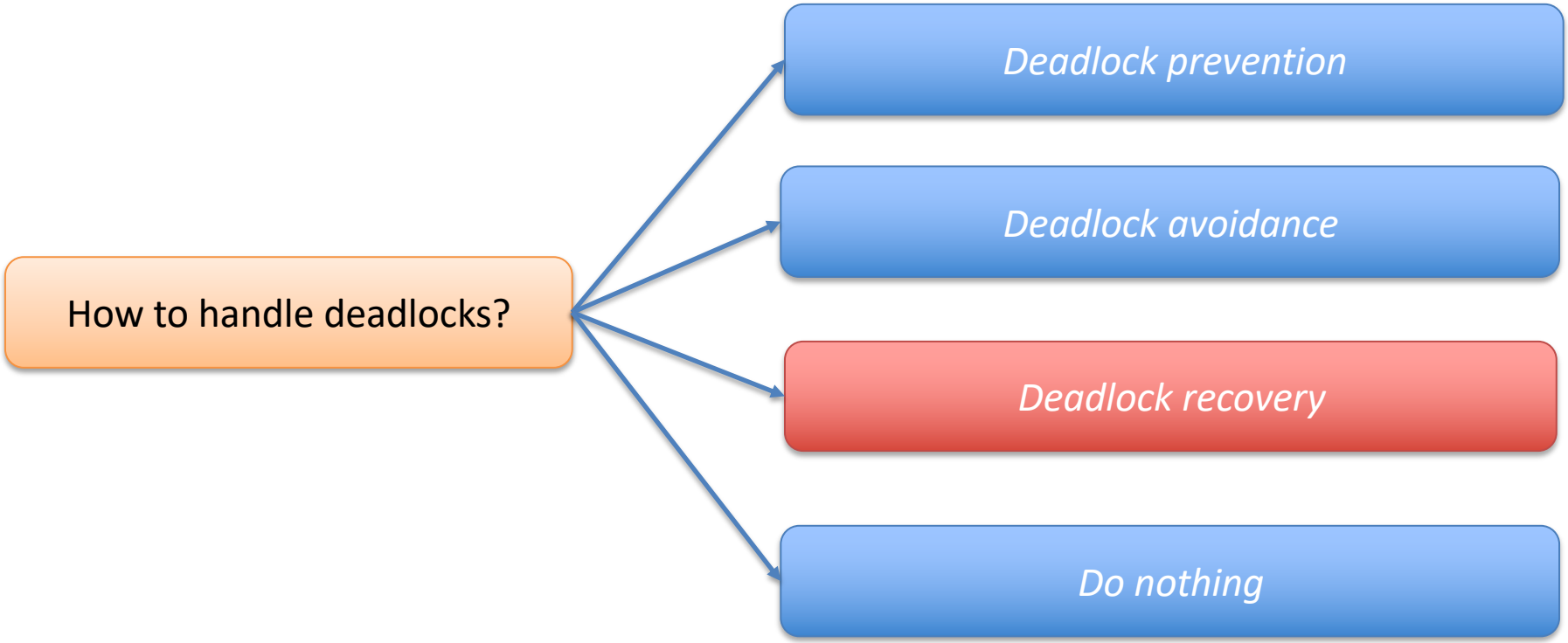


More questions

- Can request for (3,3,0) by P_4 be granted?
- If so, can request for (0,2,0) by P_0 then be granted?

Methods for Handling Deadlocks

4 options to handle deadlocks



Note: these are NOT 4 steps.

Deadlock Detection and Recovery

- What's wrong with deadlock avoidance?

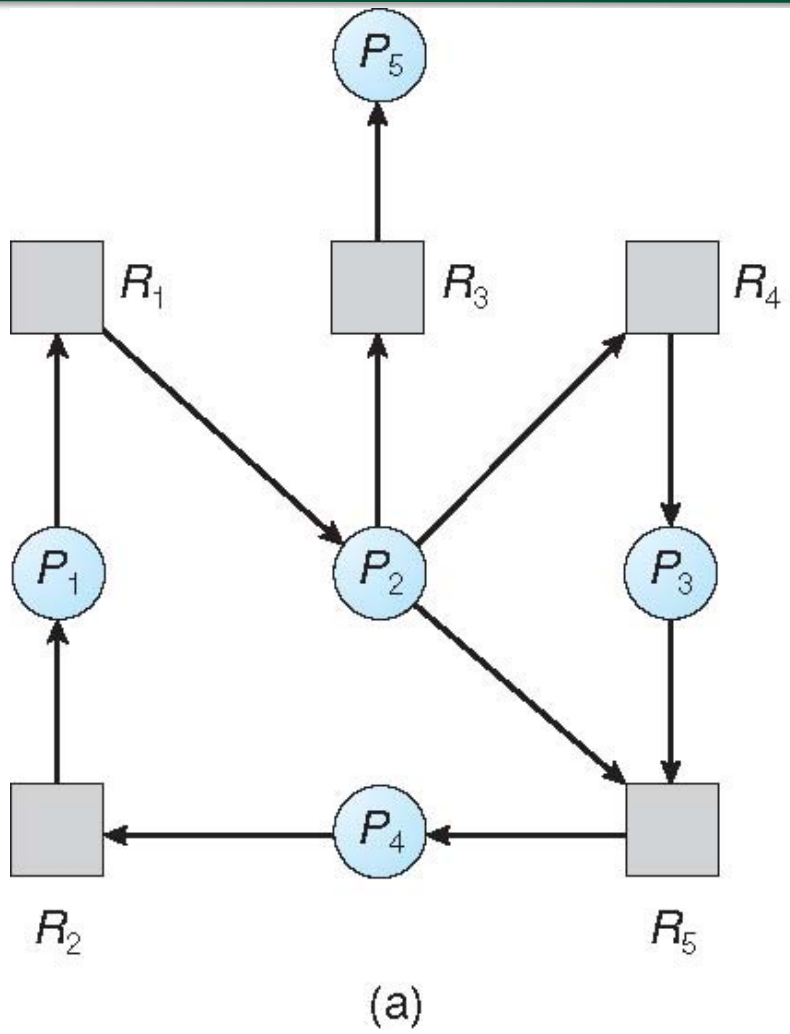
- Suppose we allow system to enter a deadlock state
  - How do we know?
- Deadlock detection
  - Single instance of each resource type → Wait-for Graph
  - Multiple instances of each resource type → Modified Safety Algorithm
- Deadlock recovery

Single Instance of Each Resource Type

- Need to maintain a *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j

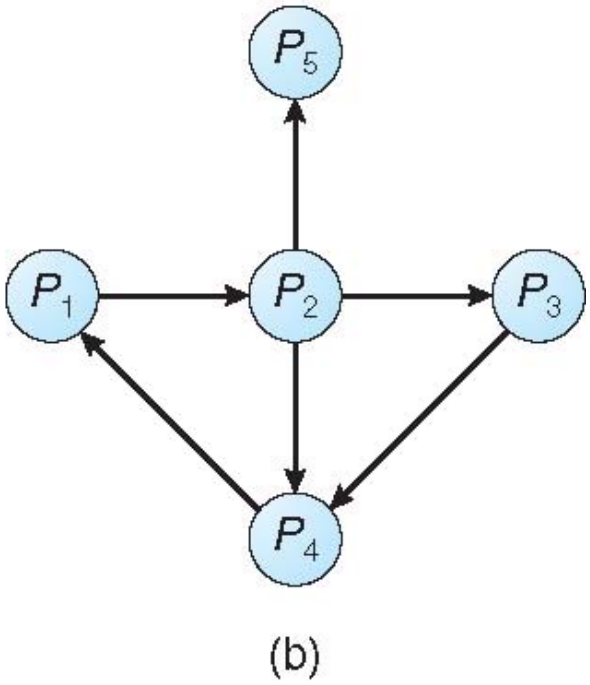
- Periodically invoke an algorithm that searches for a cycle in the wait-for graph
  - If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation and Wait-for Graph



Resource-Allocation Graph

If there is a single instance of every resource, then can just draw the processes

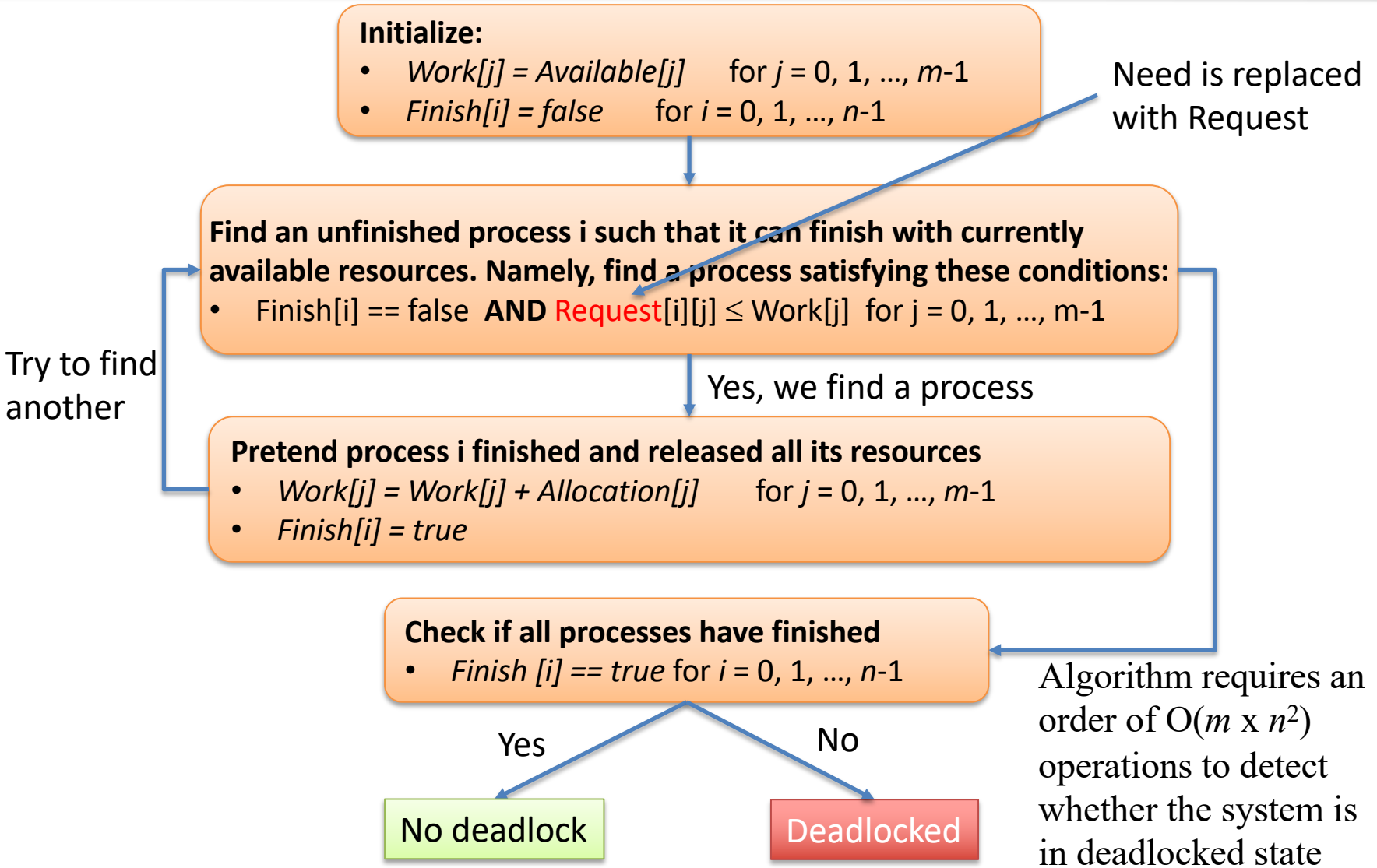


Corresponding wait-for graph

Several Instances of a Resource Type

- Consider several instances of resource types
- ***Available***: A vector of length *m* indicates the number of available resources of each type
- ***Allocation***: An *n x m* matrix defines the number of resources of each type currently allocated to each process
- ***Request***: An *n x m* matrix indicates the current request of each process
  - If *Request[i][j] = k*, then process P_i is requesting *k* more instances of resource type R_j

Detection Algorithm (modified safety algorithm)



Example of Detection Algorithm

- Five processes P_0 through P_4
- Three resource types
  - A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T0:

Process	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> gives *Finish[i] = true* for all *i*

Deadlock?

- P_2 requests an additional instance of type C

Process	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

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

- 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

Deadlock Recovery – Process Termination

- Aborting 1 or more processes will release their resources
- Different schemes
  - 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?

Deadlock Recovery – Resource Preemption

- Selecting a victim
  - Attempt to 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 rollback in cost factor