CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Spring 2022 L13

Deadlocks

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

FAQ

- Producer-consumer with bounded buffer
  - Should the production and consumption rates be a perfect match?
  - Why circular buffer? Can buffer be full?
- Readers-Writers Problem
  - Allow multiple readers to read at the same time
  - Semaphores for mutual exclusion (mutex) and counting
- Why do synchronization among processes/threads?
  - Machine instructions ⟹semaphores ⟹monitor
- Monitor: Implements
    - **mutual exclusion**: only one process may be active at a time
    - **Conditions** with associated queues where processes *wait* until *notified*
  - Our Monitor discussion is generic. Self Exercise 5 for a **Java** example.

Course Notes

- HW4 Due 3/10
  - Plan: diagram/pseudocode
  - Must have a working program 2-3 days earlier.
- Project D1: in
- Midterm: Tues March 8
  - On-campus: in class Respondus lockdown browser on laptop
  - Online: Local: with on-campus class, others: Honorlock
- D2 progress report: 4/7/22

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable thus can be used to create a monitor
- Non-portable extensions include:
  - read-write locks
  - Spinlocks
- A simple example

Colorado State University

CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Deadlocks

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Chapter 8: Deadlocks

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

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  Resource may be CPU cycles, memory space, I/O devices, critical sections
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock **can** arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use 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 $\{P_0, P_1, \ldots, P_n\}$ 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_n is waiting for a resource that is held by P_0.

Deadlock with Mutex Locks

- Deadlocks can occur via system calls, locking, etc.
- See example
  - Dining Philosophers: each get the right chopstick first
  - we saw this example earlier

Let *S* and *Q* be two semaphores initialized to 1

P_0

```
wait(S);
wait(Q);
 ...
signal(Q);
signal(S);
```

P_1

```
wait(Q);
wait(S);
 ...
signal(S);
signal(Q);
```

P0 executes wait(s), P1 executes wait(Q)

P0 must wait till P1 executes signal(Q)

P1 must wait till P0 executes signal(S)   Deadlock!

Colorado State University

Resource-Allocation Graph

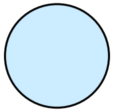
A set of vertices *V* and a set of edges *E*.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

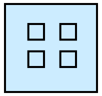
Colorado State University

Resource-Allocation Graph (Cont.)

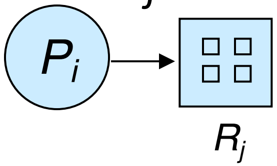
- Process



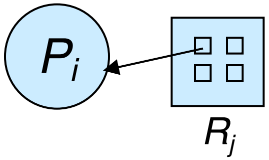
- Resource Type with 4 instances



- P_i requests instance of R_j

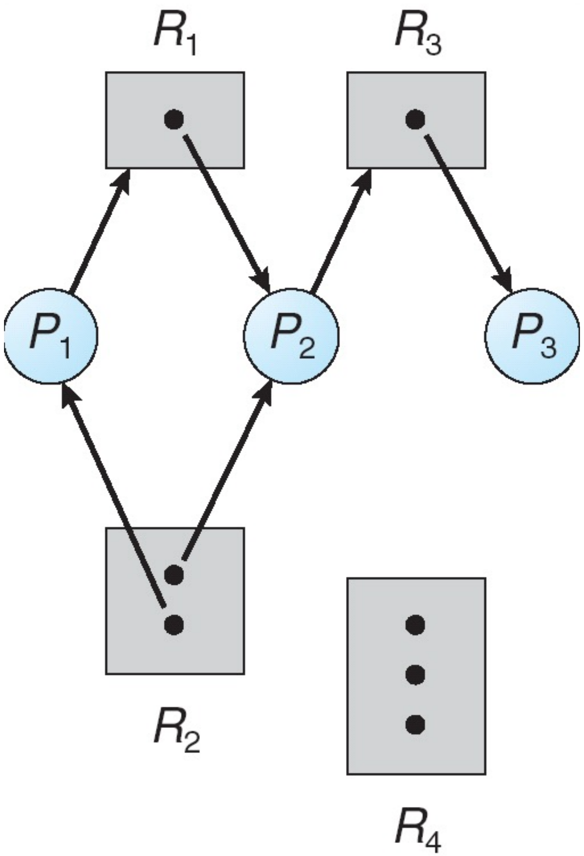


- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

P1 **holds** an instance of R2 and is **requesting** R1 ..



Does a deadlock exist here?

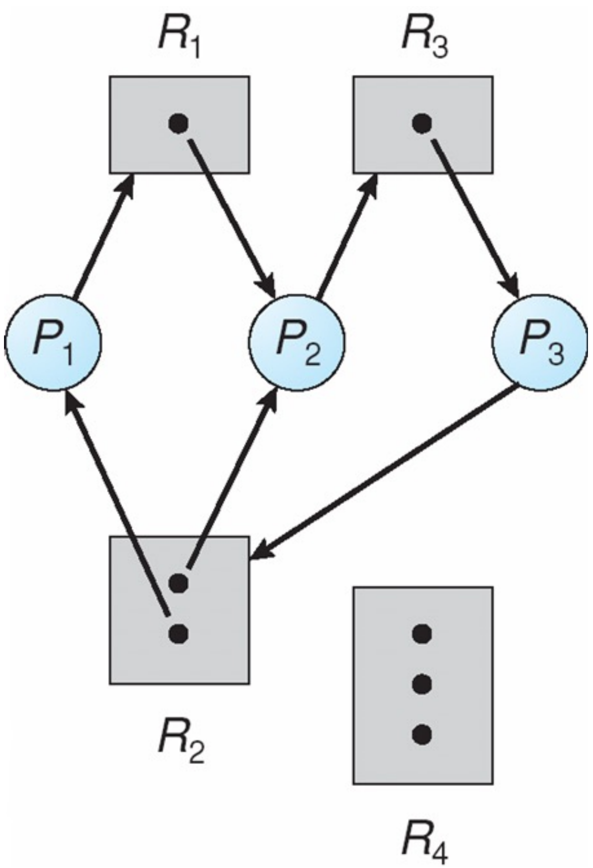
P3 will eventually be done with R3, letting P2 use it.

Thus, P2 will be eventually done, releasing R1. …
Answer: No.

Observation: If the graph contains no cycles, then no process in the system is deadlocked.
If the graph does contain a cycle, then a deadlock *may* exist.

Colorado State University

Resource Allocation Graph With A Deadlock



Does a deadlock exist?

At this point, two minimal cycles exist in the system:

P1→ R1→ P2→ R3→ P3→ R2→ P1

P2→ R3→ P3→ R2→ P2

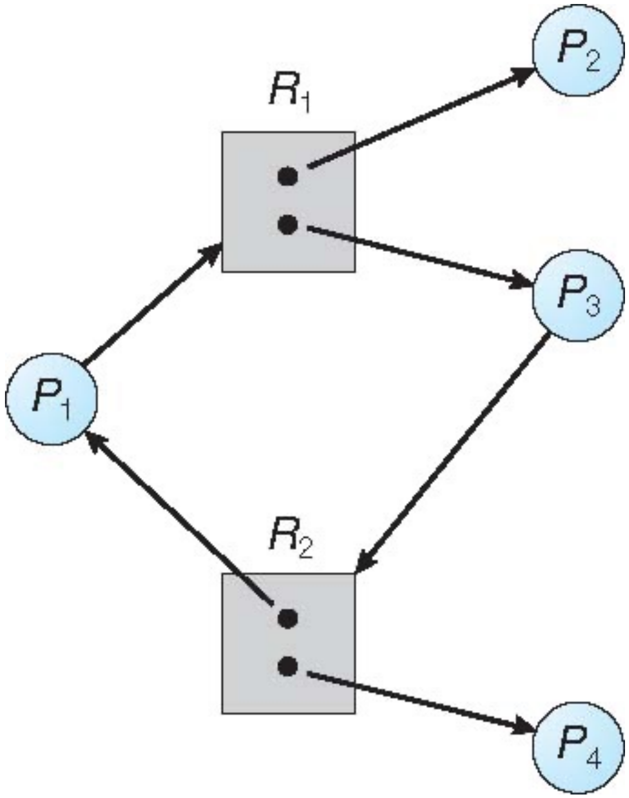
Processes *P1, P2,* and *P3* are deadlocked.

Graph With A Cycle But No Deadlock

Is there a deadlock?

P4 will release its instance of resource type *R2* . That resource can then be allocated to *P3* , breaking the cycle. Thus, there is no deadlock.

If a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state.
If there is a cycle, then the system may or may not be in a deadlocked state.



Basic Facts

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

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
    - ensuring that at least one of the 4 conditions cannot hold
  - Deadlock avoidance
    - Dynamically examines the resource-allocation state to ensure that it will never enter an unsafe state, and thus there can never be a circular-wait condition
- Allow the system to enter a deadlock state
  - Detection: detect and then recover. Hope is that it happens rarely.
- Ignore the problem and pretend that deadlocks never occur in the system; used by *most* operating systems, including UNIX. However..

Methods for Handling Deadlocks

- Deterministic: Ensure that the system will ***never*** enter a deadlock state at any cost
- Probabilistic view: Hope it happens rarely. Handle if it happens: Allow the system to enter a deadlock state and then recover.

Methods for Handling Deadlocks

Approach	Resource allocation policy	Scheme	Notes
Prevention	Conservative, undercommits resources	Requesting all resources at once	Good for processes with a single burst of activity
		Preemption	Good when preemption cost is small
		Resource ordering	Compile time enforcement possible
Avoidance	midway	Find at least one safe path (dynamic)	Future max requirement must be known
Detection	Liberal	Invoked periodically	Preemption may be needed

Ostrich algorithm

Ostrich algorithm: Stick your head in the sand; pretend there is no problem at all .

Advantages:

- Cheaper, rarely needed anyway
- Prevention, avoidance, detection and recovery
  - Need to run constantly

Disadvantages:

- Resources held by processes that cannot run
- More and more processes enter deadlocked state
  - When they request more resources
- Deterioration in system performance
  - Requires restart

To be fair to the ostriches, let me say that …

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

- **Mutual exclusion:** only one process at a time can use 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 $\{P_0, P_1, \ldots, P_n\}$ of waiting processes that are circularly waiting.

Deadlock Prevention: **Limit Mutual Exclusion**

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.

Restrain the ways request can be made:

- **Limit Mutual Exclusion** –
  - not required for sharable resources (e.g., read-only files)
  - (Mutual Exclusion must hold for non-sharable resources)

Colorado State University

Deadlock Prevention: Limit Hold and Wait

- **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

  1. Require process to request and be allocated all its resources before it begins execution

  2. Allow a process to request resources when it is holding none.

  Ex: Copy data from DVD, sort file, and print

  - First request DVD and disk file
  - Then request file and printer,
  - then start
- Disadvantage: starvation possible

Colorado State University

Deadlock Prevention: Limit No Preemption

- **Limit No Preemption** –
  - If a process that is holding some resources, requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - *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

Deadlock Prevention: Limit Circular Wait

- **Limit Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
- Assign each resource a unique number
  - Disk drive: 1
  - Printer: 2 …
  - Request resources in increasing order
    - *Example soon*

Dining philosophers problem: Necessary conditions **for** deadlock

Relax conditions to avoid deadlock

- Mutual exclusion
  - 2 philosophers *cannot share* the same chopstick
- Hold-and-wait
  - A philosopher *picks up one* chopstick at a time
  - Will not let go of the first while it *waits for the second* one
- No preemption
  - A philosopher *does not snatch chopsticks* held by some other philosopher
- Circular wait
  - Could happen if each philosopher *picks chopstick with the same hand* first

Deadlock Example: numbering

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Allows deadlock. Redesign to avoid.

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex.

Solution: **Lock-order verifier** "**Witness**" records the relationship that first mutex must be acquired before second mutex. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Colorado State University

Deadlock may happen *even* with Lock Ordering

```
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);
}
```

Lock ordering:
First *from* lock, then *to* lock

Ex: Transactions 1 and 2 execute concurrently.

Transaction 1 transfers $25 from account A to account B, and

Transaction 2 transfers $50 from account B to account A.

Deadlock is possible, even with lock ordering.

Deadlock Avoidance

Manage resource allocation to ensure the system never enters an unsafe state.

Colorado State University

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Deadlock Avoidance

- Require additional information about how resources are to be requested
- Knowledge about sequence of requests and releases for processes
  - Allows us to decide if resource allocation could cause a future deadlock
    - Process P: Tape drive, then printer
    - Process Q: Printer, then tape drive

Deadlock Avoidance: Handling resource requests

- For each resource request:
  - Decide whether or not process should wait
    - To avoid possible future deadlock
- Predicated on:
  1. Currently available resources
  2. Currently allocated resources
  3. *Future requests and releases of each process*

Avoidance: amount and type of information needed

- **Resource allocation state**
  - Number of available and allocated resources
  - Maximum demands of processes
- *Dynamically* examine resource allocation state
  - Ensure circular-wait cannot exist
- Simplest model:
  - Declare maximum number of resources for each type
  - Use information to avoid deadlock

Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

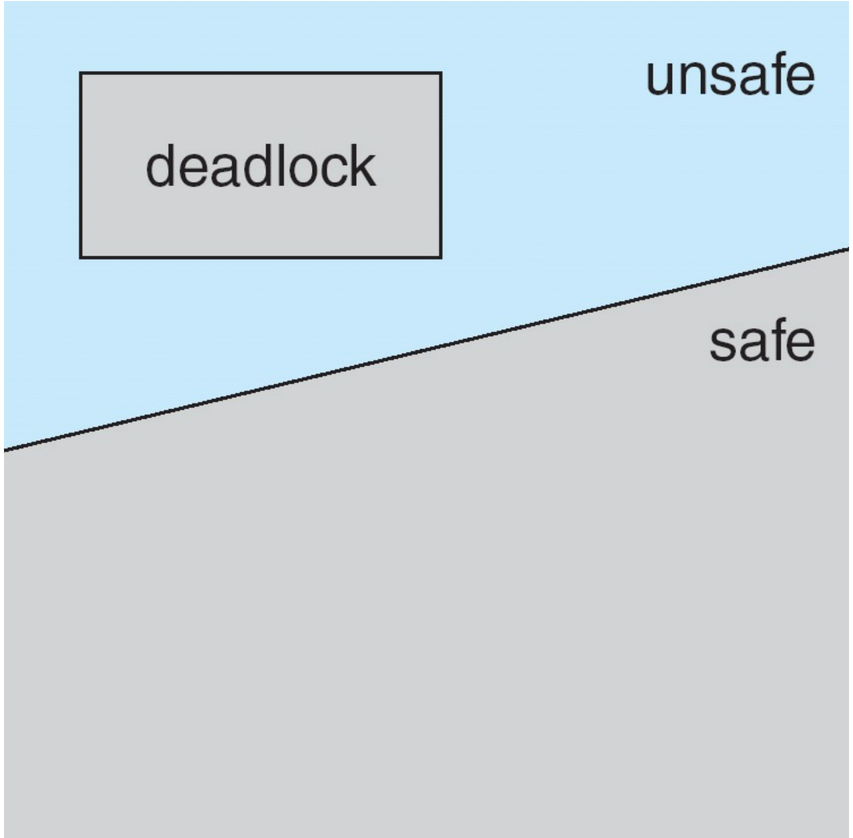
System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes such that

- 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 resource needs are not immediately available, then P_i can wait until all P_j have finished and released resources
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If no such sequence exists: system state is **unsafe**

Deadlock avoidance: Safe states

- If the system can:
  - Allocate resources to each process in some order
    - Up to the maximum for the process
  - Still avoid deadlock
  - Then it is in a **safe state**
- A system is safe ONLY IF there is a safe sequence
- A safe state is not a deadlocked state
  - Deadlocked state is an unsafe state
  - Not all unsafe states are deadlock

Safe, Unsafe, Deadlock State



Examples of safe and unsafe states in next 3 slides

Example A: Assume 12 Units in the system

	Max need	Current holding
av		3
P0	10	5
P1	4	2
P2	9	2

At time T0 (shown):
9 units allocated
3 (12-9) units available

A unit could be a drive, a block of memory etc.

- Is the system at time **T0** in a safe state?
  - Try sequence <P1, P0 , P2>
  - P1 can be given 2 units
  - When P1 releases its resources; there are now 5 available units
  - P0 uses 5 and subsequently releases them (10 available now)
  - P2 can then proceed.
- Thus <P1, P0 , P2> is a safe sequence, and at T0 system was in a safe state

More detailed look

Colorado State University

Example A: Assume 12 Units in the system (timing)

Is the state at T0 safe? Detailed look for instants T0, T1, T2, etc..

Time →

	Max need	Current holding	+2 allo to P1	P1 releases all	..	..	..
		T0	T1	T2	T3	T4	T5
av		**3**	1	5	0	10	3
P0	10	**5**	5	5	10 done	0	0
P1	4	**2**	4 done	0	0	0	0
P2	9	**2**	2	2	2	2	9 done

Thus the state at T0 is safe.

Example B: 12 Units initially available in the system

	Max need	T0	T1 safe?
Av		3	2
P0	10	5	5
P1	4	2	2
P2	9	2	3 Is that OK?

Before T1:
3 units available

At T1:
2 units available

- At time **T1,** P2 is allocated 1 more units. Is that a good decision?
  - Now only P1 can proceed (already has 2, and given be given 2 more)
  - When P1 releases its resources; there are 4 units
  - P0 needs 5 more, P2 needs 6 more. Deadlock.
    - **Mistake** in granting P2 the additional unit.
- The state at **T1** is not a safe state. Wasn't a good decision.

Avoidance Algorithms

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

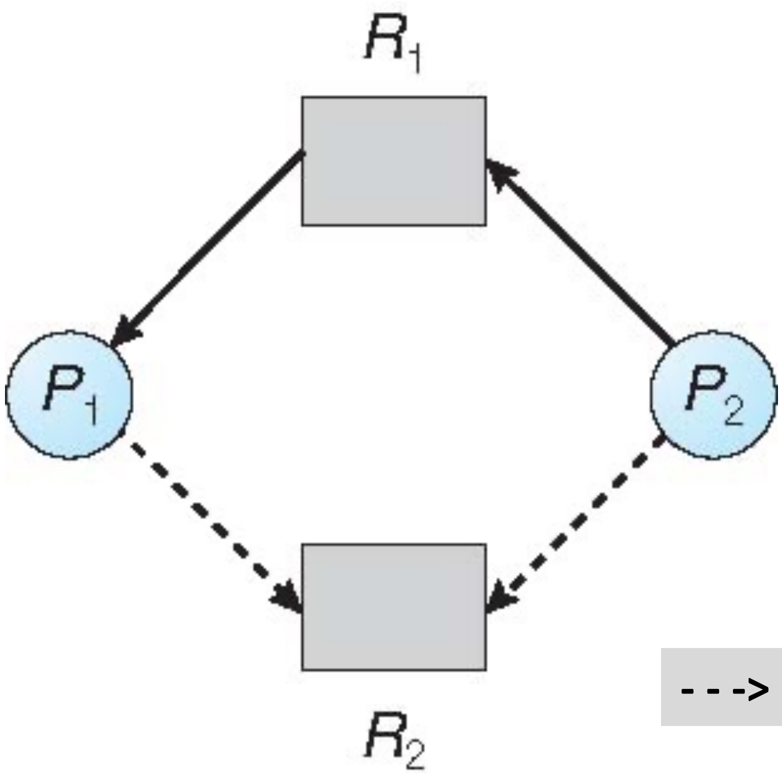
- Multiple instances of a resource type
  - Use the banker's algorithm (Dijkstra)

Colorado State University

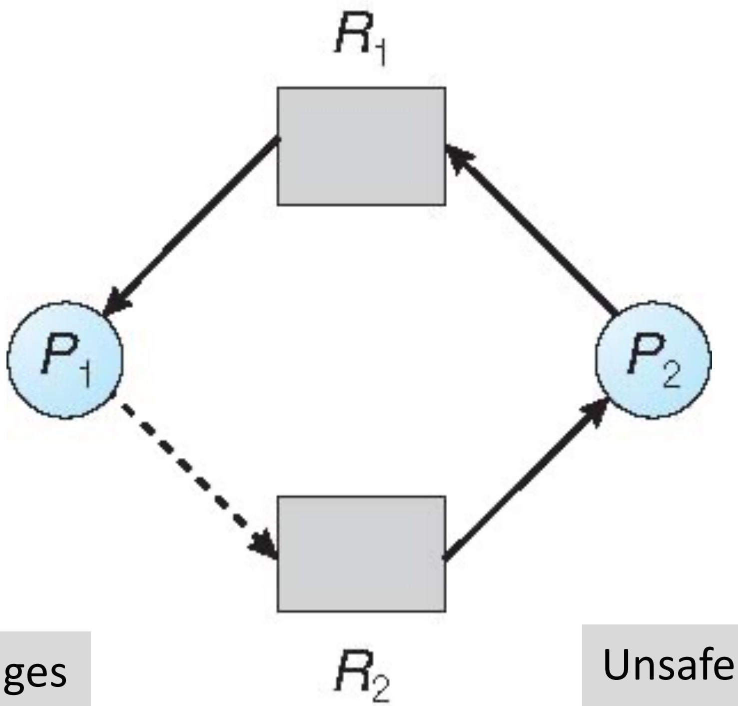
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line. This is new.
- Claim edge converts to 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
- Requirement: Resources must be claimed *a priori* in the system

Resource-Allocation Graph



- - -> Claim edges



Unsafe state

Suppose *P2* requests *R2.* Can R2 be allocated to P2?

Although *R2* is currently free, we cannot allocate it to *P2,* since this action will create a cycle getting system in an unsafe state. If *P1* requests *R2,* and *P2* requests *R1,* then a deadlock will occur.

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- 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

Banker's Algorithm: examining a request

- Multiple instances of resources.
- Each process must a priori claim maximum use
- When a process requests a resource,
  - it may have to wait until the resource becomes available (resource request algorithm)
  - Request should not be granted if the resulting system state is unsafe (safety algorithm)
- When a process gets all its resources it must return them in a finite amount of time
- Modeled after a banker in a small-town making loans.

Data Structures for the 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

Processes vs resources:

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

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

Safety Algorithm: Is this a safe state?

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = Initially *Available resources

   ***Finish* [*i*] = *initially false* for *i* = 0, 1, …, *n*- 1** (processes done)

2. Find a process ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$

   Finish*[*i*] = *true

   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

n = number of processes,

m = number of resources types

$\mathbf{Need_i}$: **additional** res needed

Work: res currently free

$\mathbf{Finish_i}$: processes finished

$\mathbf{Allocation_i}$: allocated to i

Colorado State University

Resource-Request Algorithm for Process P_i

Notation: ***Request**$_i$* = request vector for process P_i.
If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

Algorithm: Should the allocation request be granted?

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. **Is allocation safe?:** Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is preserved.

Use safety algorithm here

Colorado State University

Example 1A: Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types: *A* (10 instances), *B* (5 instances), and *C* (7 instances)
- Is it a safe state?

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
Currently available				3	3	2			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

The Need matrix is redundant