Deadlocks

What is a *deadlock?*

- **Deadlock** -- *permanent* blocking of a set of processes that compete for system resources
- **No efficient solution** to the deadlock problem in the general case.
- In deadlock
  - a set of **processes are in a wait** state, because each process is waiting for a resource that is held by some other waiting process
- All deadlocks involve conflicting resource needs by two or more processes

Classification of resources—I

- Resources grouped into two classes:
  - **Reusable**: something that can be safely used by one process at a time and is ***not depleted*** by that use. Processes obtain resources that they later release for reuse by others.

    E.g., CPU, memory, specific I/O devices, or files.
  - **Consumable**: these can be ***created and destroyed***. When a resource is acquired by a process, the resource ceases to exist.

    E.g., interrupts, signals, or messages.

Classification of resources—II

- Another classification of resources:
  - **Preemptable**: these can be taken away from the process owning it with no ill effects (needs save/restore).

    E.g., memory or CPU.
  - **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail.

    E.g., printer or floppy disk.
- Deadlocks mostly occur when sharing ***reusable*** and ***non-preemptable*** resources.

Conditions for deadlock

- Four conditions that must hold for a deadlock to be possible:
  - **Mutual exclusion**: processes require exclusive control of its resources (not sharing)
  - **Hold and wait**: process may wait for a resource while holding others
  - **No preemption**: process will not give up a resource until it is finished with it
  - **Circular wait**: each process in the chain holds a resource requested by another

Conditions for deadlock

- ***Also, a process*** *cannot be reset to an earlier state where resources not held*
- Conditions 1—3 are necessary, but not sufficient. All 4 are needed

Solving deadlocks

- If a necessary condition is prevented a deadlock cannot occur. For example:
  - Systems with only shared resources cannot deadlock.
    - – Negates *mutual exclusion*.
  - Systems that abort processes which request a resource that is in use.
    - – Negates *hold and wait*.

Solving deadlocks (contd.)

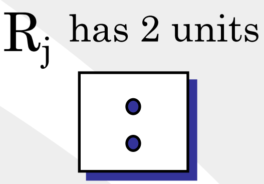
- If a necessary condition is prevented a deadlock cannot occur. For example:
  - Systems that detect or avoid deadlocks.
    - Prevents *cycle*.
  - Transaction processing systems provide checkpoints so that processes may back out of a transaction.
    - Negates *irreversible process*.

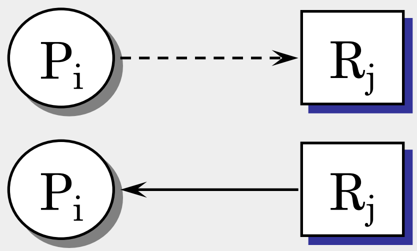
Resource allocation graphs

Set of Processes $P = \{P_1, P_2, \ldots, P_n\}$

Set of Resources $R = \{R_1, R_2, \ldots, R_m\}$

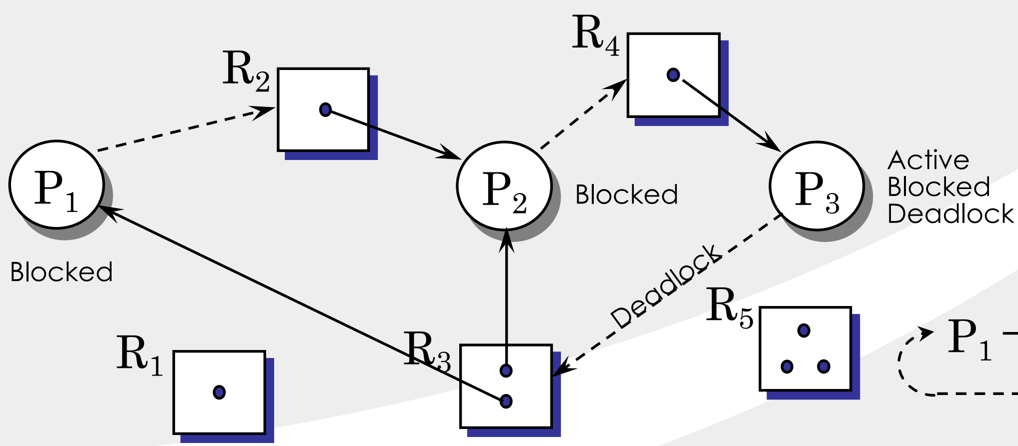
Some resources come in multiple units.





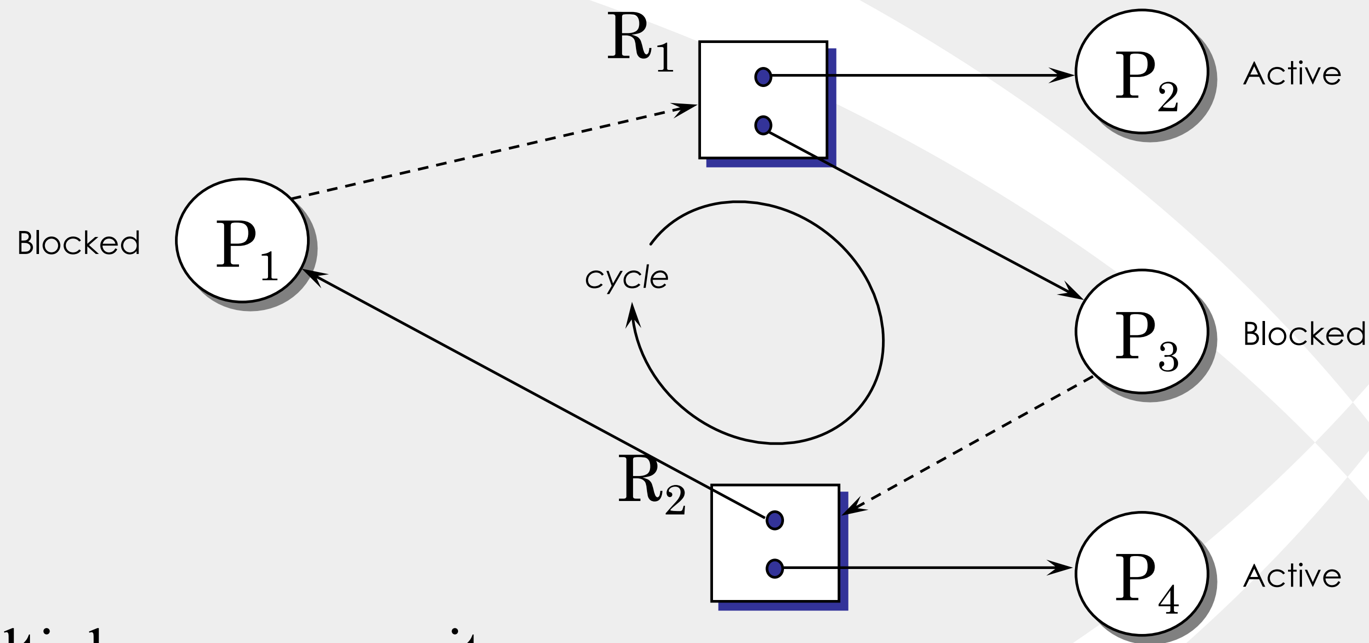
Process P_i waits for (has requested) R_j

Resource R_j has been allocated to P_i



$P_1 \to R_2 \to P_2 \to R_4 \to P_3 \to R_3$

Cycle is necessary, but …



Multiple resource unit case:

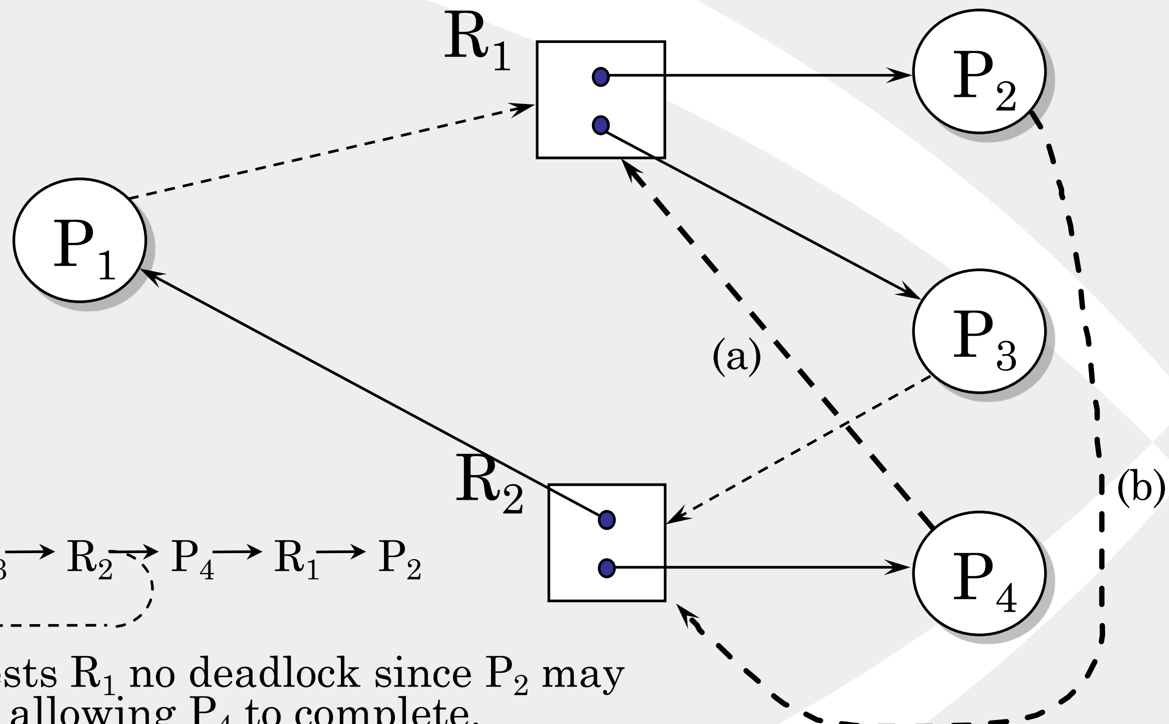
No Deadlock—yet!

Because, either P_2 or P_4 could relinquish a resource allowing P_1 or P_3 (which are currently <u>blocked</u>) to continue. P_2 is still <u>executing</u>, even if P_4 requests R_1.

… a knot is required

- Cycle is a *necessary condition* for a deadlock. With multiple unit resources—*not sufficient.*
- A *knot* must exist—a cycle with no non-cycle outgoing path from any involved node.
- At the moment assume that:
  - a process *halts* as soon as it waits for one resource, and
  - processes can wait for only *one* resource at a time

Further requests



$P_1 \rightarrow R_2 \rightarrow P_3 \rightarrow R_2 \rightarrow P_4 \rightarrow R_1 \rightarrow P_2$

(a) If P_4 requests R_1 no deadlock since P_2 may release R_1, allowing P_4 to complete.

$P_1 \rightarrow R_2 \rightarrow P_3 \rightarrow R_2 \rightarrow P_4 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2$

(b) If P_2 requests R_2: Deadlock—*Cycle*—*Knot*.

No active processes to release resources.

Strategies for deadlocks

- In general, four strategies are used for dealing with deadlocks:
  - **Ignorance**: pretend there is no problem at all.
  - **Prevention**: design a system in such a way deadlock is excluded <u>a priori</u>.
  - **Avoidance**: make a decision dynamically checking whether a request will, if granted, potentially lead to a deadlock or not.
  - **Detection**: let the deadlock occur and detect when it happens, and take some action to recover after the fact.

Dealing with Deadlocks

- Different people react to this strategy in different ways:
  - **Mathematicians**: find deadlock totally unacceptable, and say that it must be prevented at all costs.
  - **Engineers**: ask how serious it is, and do not want to pay a penalty in performance and convenience.
- UNIX approach
  - ignore the problem
  - Prevention price is high – inconvenient restrictions

Deadlock prevention

- Deadlock prevention is to design a system to exclude the possibility of a deadlock
  - *indirect methods* -- prevent the occurrence of one of the necessary conditions listed earlier.
  - *direct methods* -- prevent the occurrence of a circular wait condition.
- Deadlock prevention strategies are very ***conservative*** -- solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes

More on deadlock prevention

- **Mutual exclusion**
  - In general, this condition cannot be disallowed.
- **Hold-and-wait**
  - The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously.
- **No preemption**
  - One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource.
- **Circular Wait**
  - The circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.

Deadlock avoidance

- Allows the necessary conditions
- Makes judicious resource allocation choices to ensure a deadlock-free system
  - System evaluates current request and denies it if granting it will lead to potential deadlock
  - Requires knowledge of future requests
- Deadlock avoidance approaches:
  - Resource trajectories
  - Safe/unsafe states
  - Dijkstra’s Banker's algorithm

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

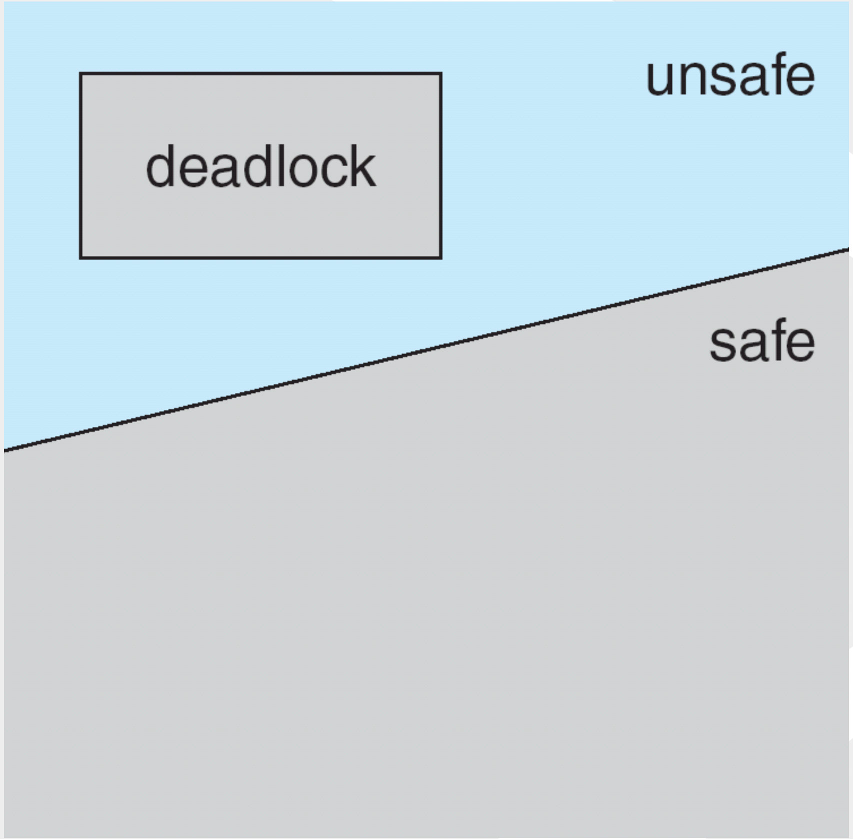
Safe State

- When a process requests an available resource, 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, …, P_n> of ALL the processes in the systems 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
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Basic Facts

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

Safe, Unsafe, Deadlock State



Avoidance Algorithms

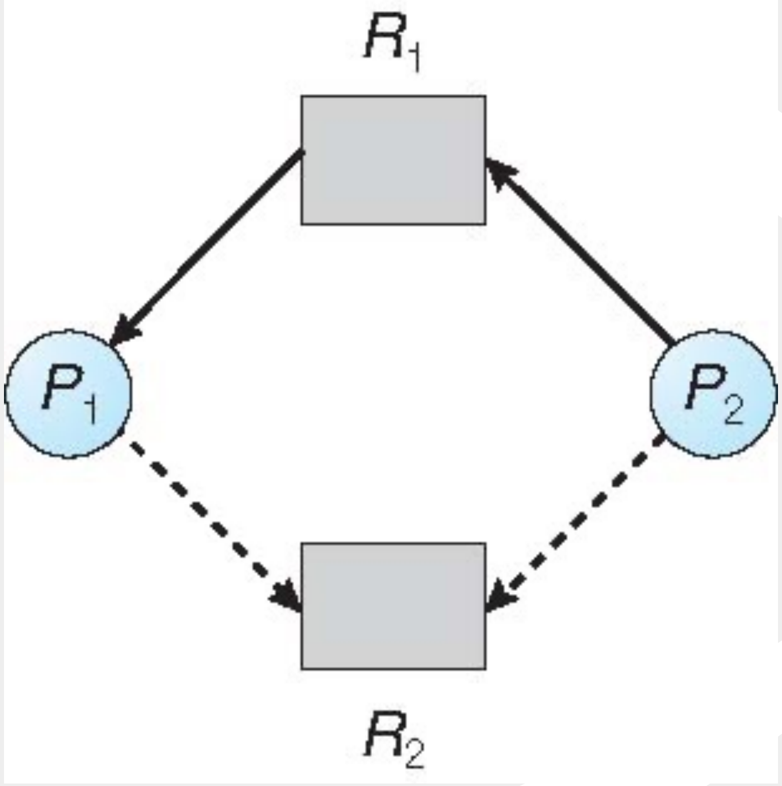
- Single instance of a resource type
  - Use a resource-allocation graph

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

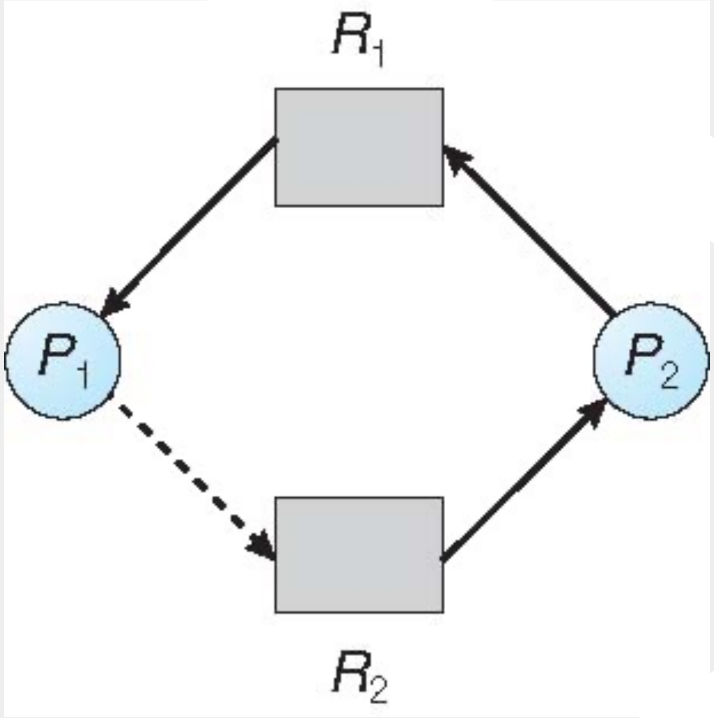
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request/wait resource R_j; represented by a dashed line
- Claim edge converts to request/wait edge when a process requests a resource
- Request edge converted to an assignment/allocation edge when the resource is allocated to the process
- When a resource is released by a process, assignment/allocation edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



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

- Multiple instances
- 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 algorithm—*definitions*

Assume N Processes $\{P_i\}$
M Resources $\{R_j\}$

Availability vector $\mathbf{Avail}_j$, units of each resource (initialized to maximum, changes dynamically).

Let $[\mathbf{Max}_{ij}]$ be an N x M matrix.

$\mathbf{Max}_{ij} = \mathbf{L}$ means Process P_i will request at most $\mathbf{L}$ units of R_j.

$[\mathbf{Hold}_{ij}]$ Units of R_j currently held by P_i

$[\mathbf{Need}_{ij}]$ Remaining need by P_i for units of R_j

$\mathbf{Need}_{ij} = \mathbf{Max}_{ij} - \mathbf{Hold}_{ij}$, for all i & j

Banker' s algorithm—*resource request*

- At any instance, P_i posts its request for resources in vector $\mathbf{REQ}_j$.
- **Step 1:** verify that a process matches its needs.
- ***if*** $\mathbf{REQ}_j > \mathbf{Need}_{ij}$ ***abort***—*error, impossible*
- **Step 2:** check if the requested amount is available.
- ***if*** $\mathbf{REQ}_j > \mathbf{Avail}_j$ ***goto Step 1***—*P_i must wait*
- **Step 3:** provisional allocation.
- $\mathbf{Avail}_j = \mathbf{Avail}_j - \mathbf{REQ}_j$
- $\mathbf{Hold}_{ij} = \mathbf{Hold}_{ij} + \mathbf{REQ}_j$
- $\mathbf{Need}_{ij} = \mathbf{Need}_{ij} - \mathbf{REQ}_j$
- ***if*** **isSafe()** ***then*** grant resources—*system is safe*
- ***else*** cancel allocation; ***goto Step 1***—*P_i must wait*

Banker's algorithm—*isSafe*

Find out whether the system is in a safe state.
`Work` and **`Finish`** are two temporary vectors.

`Step 1:` initialize.
`Work`$_j$ **`= Avail`**$_j$ for all j; **`Finish`**$_i$ **`= false`** for all i.

`Step 2:` find a process P_i such that
`Finish`$_i$ **`= false`** and **`Need`**$_{ij}$ $\leq$ **`Work`**$_j$
if no such process, ***`goto Step 4`***.

`Step 3:` **`Work`**$_j$ **`= Work`**$_j$ **`+ Hold`**$_{ij}$
`Finish`$_i$ **`= true`**
`goto Step 2`.

`Step 4:` ***`if`*** **`Finish`**$_i$ **`= true`** for all i
`then return` **`true`**—*yes, the system is safe*
`else return` **`false`**—*no, the system is NOT safe*

Banker's algorithm—*what <u>is</u> safe?*

- <u>Safe</u> with respect to some resource allocation.
  - very safe

    `NEED`$_i$ `<=` `AVAIL` for <u>all</u> Processes P_i.

    Processes can run to completion in any order.
  - safe (but take care)

    `NEED`$_i$ `>` `AVAIL` for <u>some</u> P_i

    `NEED`$_i$ `<=` `AVAIL` for <u>at least one</u> P_i such that

    There is at least one correct order in which the processes may complete their use of resources.
  - unsafe (deadlock inevitable)

    `NEED`$_i$ `>` `AVAIL` for <u>some</u> P_i

    `NEED`$_i$ `<=` `AVAIL` for <u>at least one</u> P_i

    But some processes <u>cannot</u> complete successfully.
  - deadlock

    `NEED`$_i$ `>` `AVAIL` for <u>all</u> P_i

    Processes are already blocked or will become so as they request a resource.

Example—safe allocation

	Max	Hold	Need	Finish	Avail	Work
P_1	5	~~2~~ 3	~~3~~ 2	F	2	2
P_2	4	1	3	F		
P_3	2	1	1	F		

For simplicity, assume that all the resources are identical.

Assume P_1 acquires one unit. *Very safe?* **No!** $\mathtt{\mathbf{Need_2 > 2}}$

Safe? Let us see with the safe/unsafe algorithm...

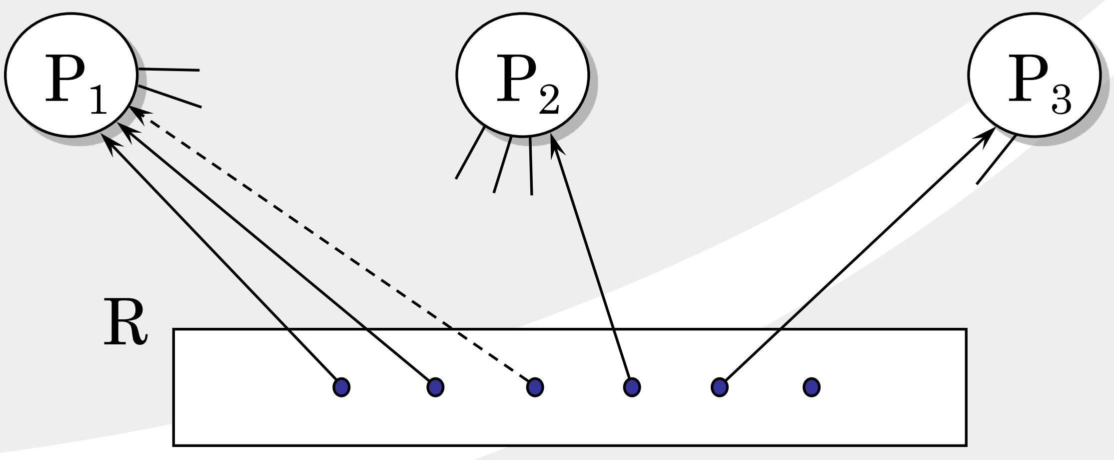
$\mathtt{\mathbf{i = 1}}$; does P_1 agree with $\mathtt{\mathbf{Step\ 2}}$? No.
$\mathtt{\mathbf{i = 2}}$; does P_2 agree with $\mathtt{\mathbf{Step\ 2}}$? No.
$\mathtt{\mathbf{i = 3}}$; does P_3 agree with $\mathtt{\mathbf{Step\ 2}}$? Yes. $\mathtt{\mathbf{Work = Work{+}Hold_3;\ Finish_3 = T}}$
$\mathtt{\mathbf{i = 1}}$; does P_1 agree with $\mathtt{\mathbf{Step\ 2}}$? Yes. $\mathtt{\mathbf{Work = Work{+}Hold_1;\ Finish_1 = T}}$
$\mathtt{\mathbf{i = 2}}$; does P_2 agree with $\mathtt{\mathbf{Step\ 2}}$? Yes. $\mathtt{\mathbf{Work = Work{+}Hold_2;\ Finish_2 = T}}$

No more (unfinished) P_i , therefore *safe*.

Example—safe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	~~2~~ 3	~~3~~ 2	F	2	~~2~~ 1
P_2	4	1	3	F		
P_3	2	1	1	F		

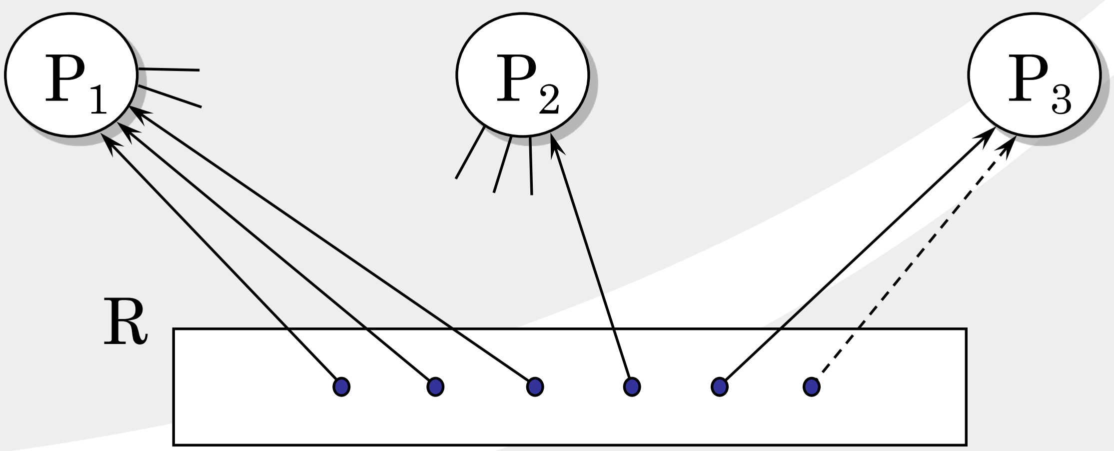
Assume P_1 acquires one unit.



Example—safe allocation
cont.

	Max	Hold	Need	Finish	Avail	Work
P_1	5	3	2	F	1	$\cancel{1}$ 0
P_2	4	1	3	F		
P_3	2	$\cancel{1}$ 2	$\cancel{1}$ 0	F		

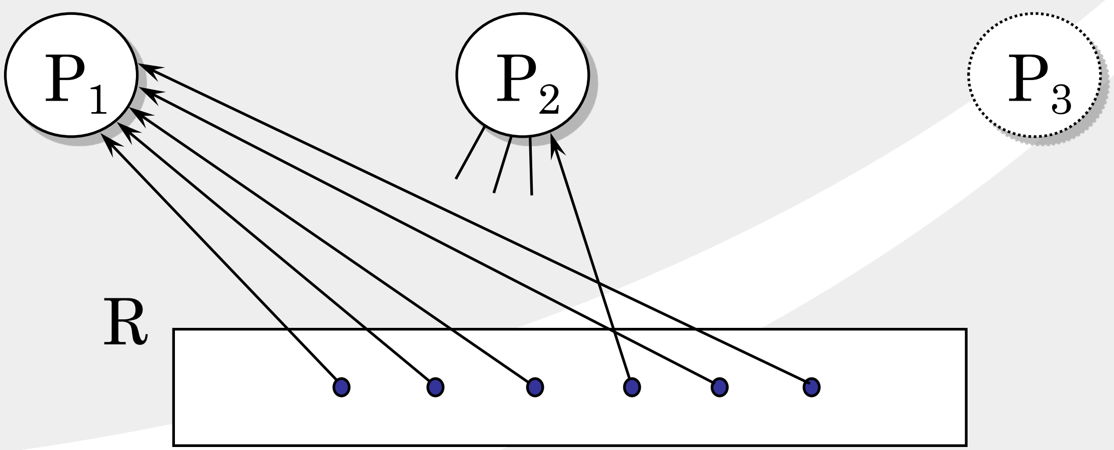
P_3 can acquire the last unit and finish.



Example—safe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	5	0	F	2	~~2~~ 0
P_2	4	1	3	F		
P_3	2	0	0	T		

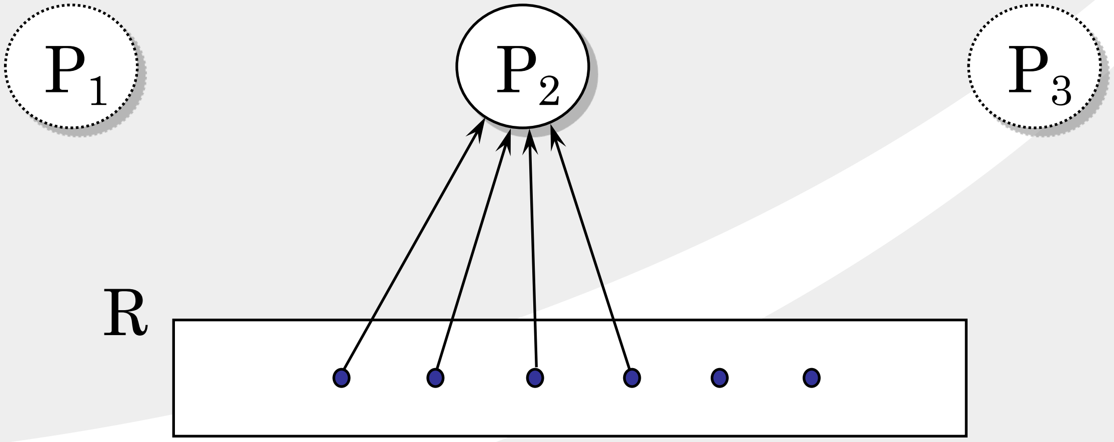
Then, P_1 can acquire two more units and finish.



Example—safe allocation
cont.

	Max	Hold	Need	Finish	Avail	Work
P_1	5	0	0	T	5	$\cancel{5}$ 2
P_2	4	$\cancel{1}$ 4	$\cancel{3}$ 0	F		
P_3	2	0	0	T		

Finally, P_2 can acquire three more units and finish.



Example—unsafe allocation

	Max	Hold	Need	Finish	Avail	Work
P_1	5	2	3	F	2	~~2~~ 1
P_2	5	~~1~~ 2	~~4~~ 3	F		
P_3	2	1	1	F		

Assume P_2 acquires one unit.
As before, P_3 can finish and release its resources.

BUT…

`i = 1`; does P_1 agree with **`Step 2`**? No.
`i = 2`; does P_2 agree with **`Step 2`**? No.
`i = 3`; does P_3 agree with **`Step 2`**? Yes. **`Work = Work+Hold`**$_2$**`;`** **`Finish`**$_2$ **`= T`**

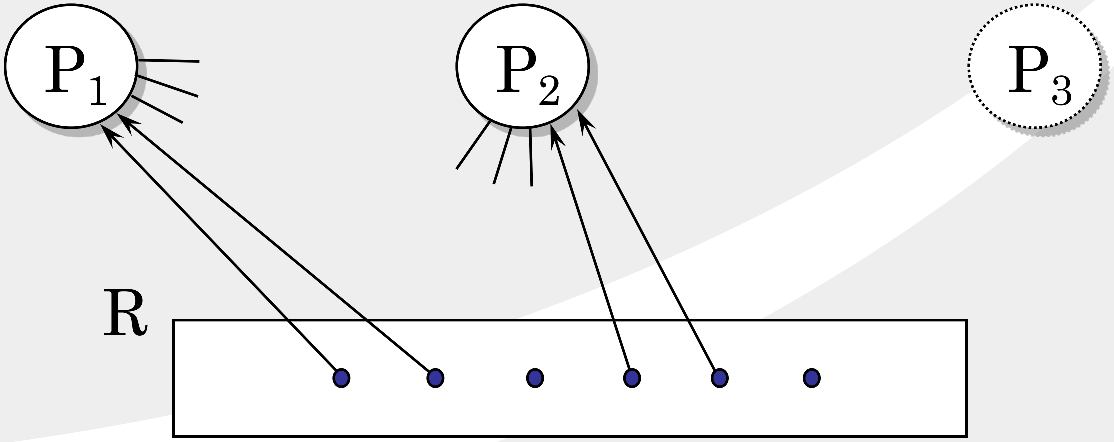
Any more unfinished P_i? Yes.

P_1 and P_2 cannot finish. Therefore *unsafe*.

Example—unsafe allocation *cont.*

	Max	Hold	Need	Finish	Avail	Work
P_1	5	2	3	F	2	2
P_2	5	2	3	F		
P_3	2	0	0	T		

NOW…



Deadlock detection

- This technique
  - does not attempt to prevent deadlocks;
  - instead, it lets them occur.
- The system
  - detects when this happens, by periodically running an algorithm to detect a **circular wait** condition,
  - then takes some action to recover after the fact.
- With deadlock detection, requested resources are granted to processes whenever possible.

Deadlock detection
cont.

- A check for deadlock can be made
  - as frequently as for each resource request, or
  - less frequently, depending on how likely it is for a deadlock to occur.
- Checking at each resource request has two advantages:
  - it leads to early detection, and
  - the algorithm is relatively simple because it is based on incremental changes to the state of the system.
- On the other hand,
  - such frequent checks consume considerable processor time.

Recovering from deadlocks

- Once the deadlock algorithm has successfully detected a deadlock, some strategy is needed for recovery. There are various ways:
  - Recovery through *Preemption*

    In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another.
  - Recovery through *Rollback*

    If it is known that deadlocks are likely, one can arrange to have processes *checkpointed* periodically. For example, can undo transactions, thus free locks on database records.
  - Recovery through *Termination*

    The most trivial way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. Warning! *Irrecoverable losses may occur, even if this is the least advanced process.*

Summary of strategies

Principle	*Resource Allocation Strategy*	*Different Schemes*	*Major Advantages*	*Major Disadvantages*
DETECTION	• Very liberal; grant resources as requested.	• Invoke periodically to test for deadlock.	• Never delays process initiation. • Facilitates on-line handling.	• Inherent preemption losses.
PREVENTION	• Conservative; under-commits resources.	• Requesting all resources at once.	• Works well for processes with single burst of activity. • No preemption is needed.	• Inefficient. • Delays process initiation.
		• Preemption	• Convenient when applied to resources whose state can be saved and restored easily.	• Preempts more often then necessary. • Subject to cyclic restart.
		• Resource ordering	• Feasible to enforce via compile-time checks. • Needs no run-time computation.	• Preempts without immediate use. • Disallows incremental resource requests.
AVOIDANCE	• Selects midway between that of detection and prevention.	• Manipulate to find at least one safe path.	• No preemption necessary.	• Future resource requirements must be known. • Processes can be blocked for long periods.