departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Monitoring Concurrency Errors: Detection of Deadlocks, Atomicity Violations, and Data Races (3)

Concurrency and Parallelism — 2017-17

Master in Computer Science

(Mestrado Integrado em Eng. Informática)

Joao Lourenço <joao.lourenco@fct.unl.pt>

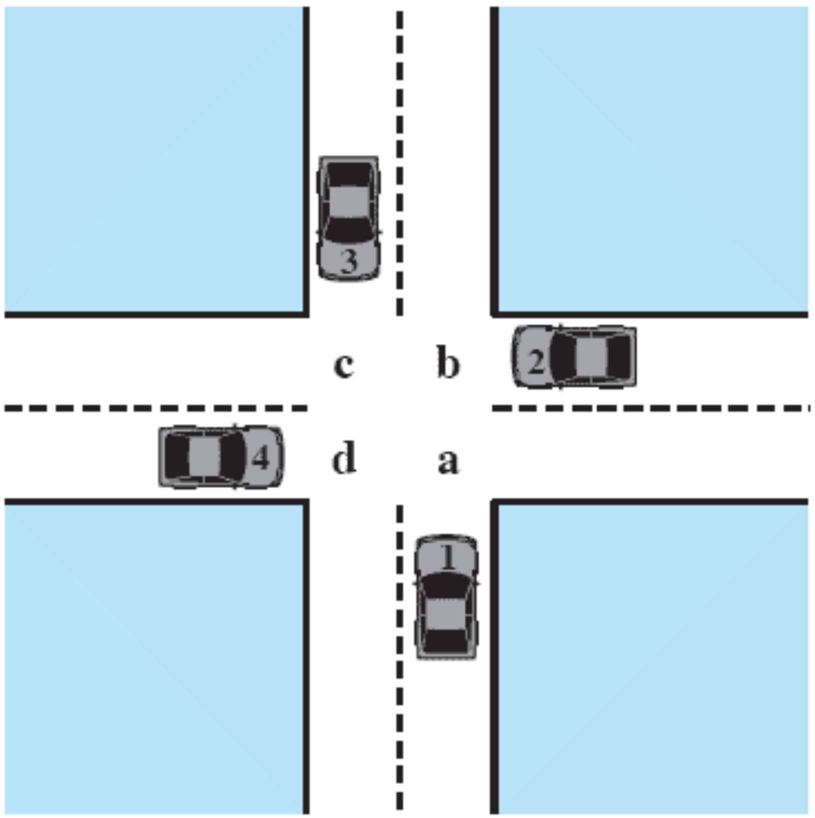
Agenda

- Why are we here?
- Concurrency Anomalies
- Assigning Semantics to Concurrent Programs
- Concurrency Errors
  – Detection of data races
  – Detection of high-level data races and stale value errors
  – Detection of deadlocks

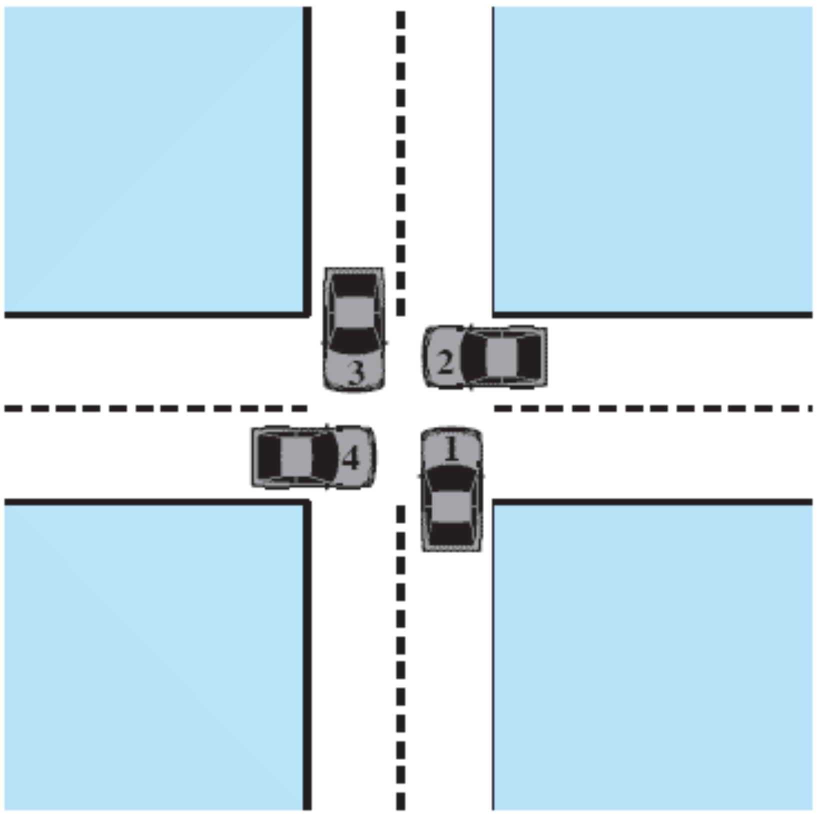
Deadlocks

Deadlock

Permanent blocking of a set of processes that either compete for system resources or communicate with each other.



(a) Deadlock possible



(b) Deadlock

System Model

- Finite number of resources
- Resources are organized into classes
  - Each class only contain identical resource instances
- Processes compete for accessing resources
- If a process request an instance of a resource class, *any* instance of that class must satisfy the process

Protocol to Use a Resource

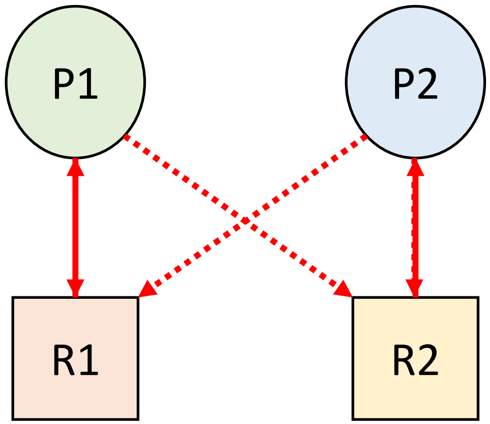
- **Request** — The process either gets an instance of the resource immediately; or waits until one is available (and gets it)
- **Use** — The process can operate on its resource instance
- **Release** — The process releases its resource instance

- Examples: malloc() & free() — open() & close()

Deadlock

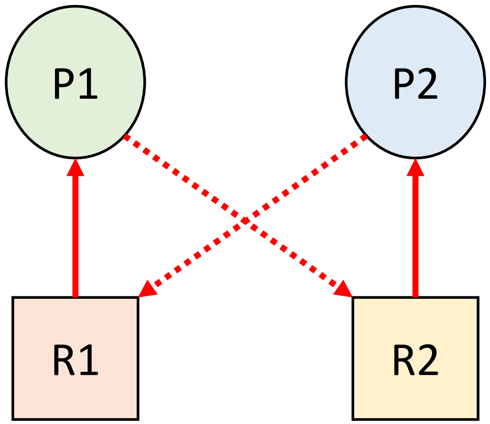
A set of two or more processes are deadlocked if:

1. They are blocked (i.e., in the waiting state)
2. Each is holding a resource
3. Each is waiting to acquire a resource held by another process in the set



Deadlock

- Deadlock depends on the dynamics of the execution
- Is difficult to identify and test for deadlocks, which may occur only under certain circumstances



Conditions Necessary for Deadlock

- **mutual exclusion**: only one process can use a resource at a time
- **hold and wait**: a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes
- **no preemption**: a resource can only be released voluntarily by the process holding it
- **circular wait**: a cycle of process requests exists (i.e., $P_0 \dashrightarrow P_1 \dashrightarrow P_2 \dashrightarrow \ldots \dashrightarrow P_{n-1} \dashrightarrow P_0$).

Example

Thread 1

```
void *do_work_one(void *param) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param) {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

Example

Will deadlock happen?

Thread 1

```
void *do_work_one(void *param) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param) {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

Example

Only if executed in order:
— 1, 3, 2, 4; or
— 1, 3, 4, 2; or
— 3, 1, 2, 4; or
— 3, 1, 4, 2

Thread 1

```
void *do_work_one(void *param) {
1 pthread_mutex_lock(&m1);
2 pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

read 2

```
void *do_work_two(void *param) {
3 pthread_mutex_lock(&m2);
4 pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

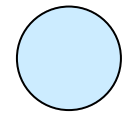
These orderings are ok:
— 1, 2, 3, 4; and
— 3, 4, 1, 2

Resource Allocation Graph

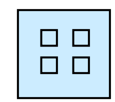
- A set of vertices **V** and a set of edges **E**
- **V** is partitioned into two types:
  - P = {P_1, P_2, …, P_n}, the set of all the processes in the system
  - R = {R_1, R_2, …, R_m}, the set of all resource types in the system
- **E** is partitioned into two types:
  - Request edge – directed edge $P_i \rightarrow R_j$
  - Assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

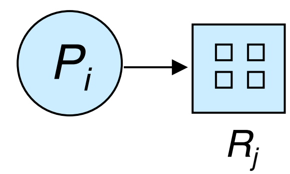
- Process



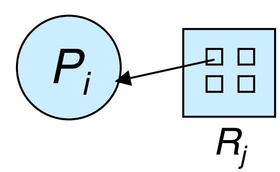
- Resource Type with 4 instances



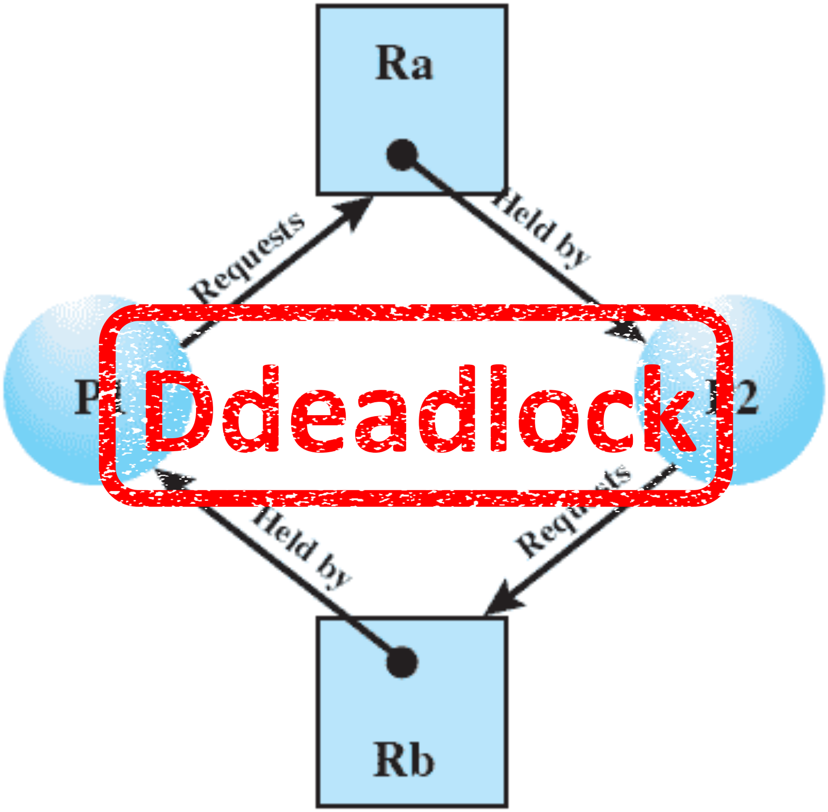
- P_i requests instance of R_j



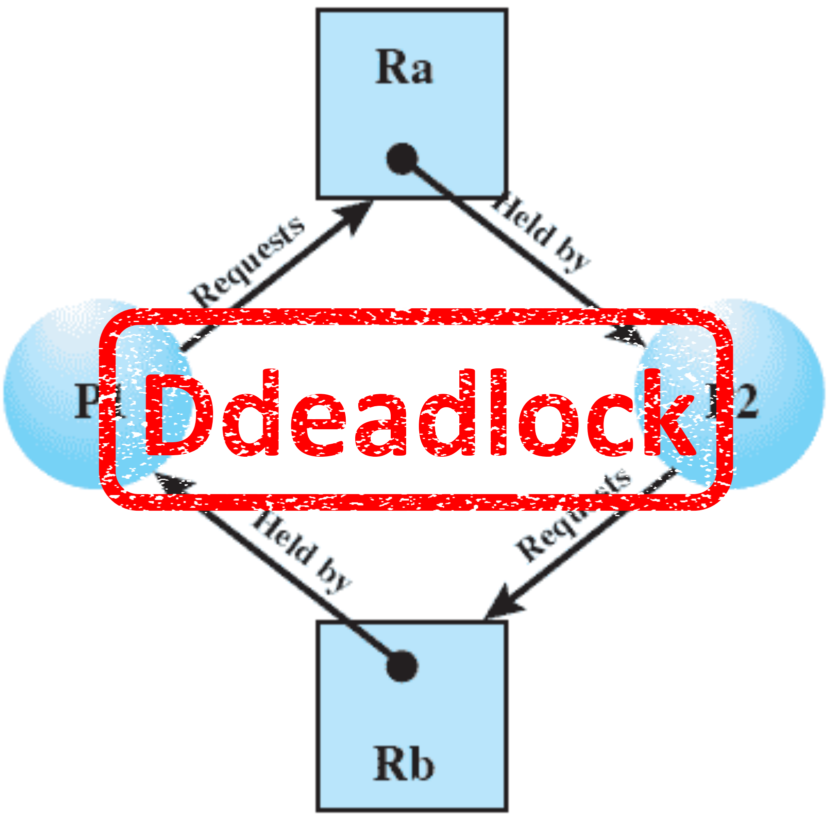
- P_i is holding an instance of R_j

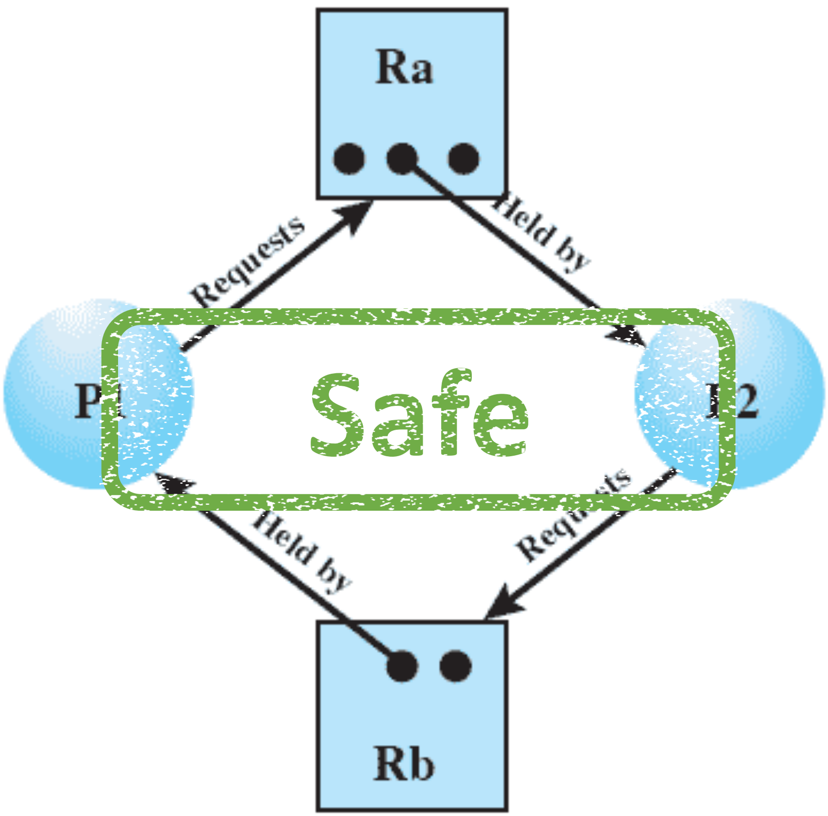


Example of Resource Allocation Graphs

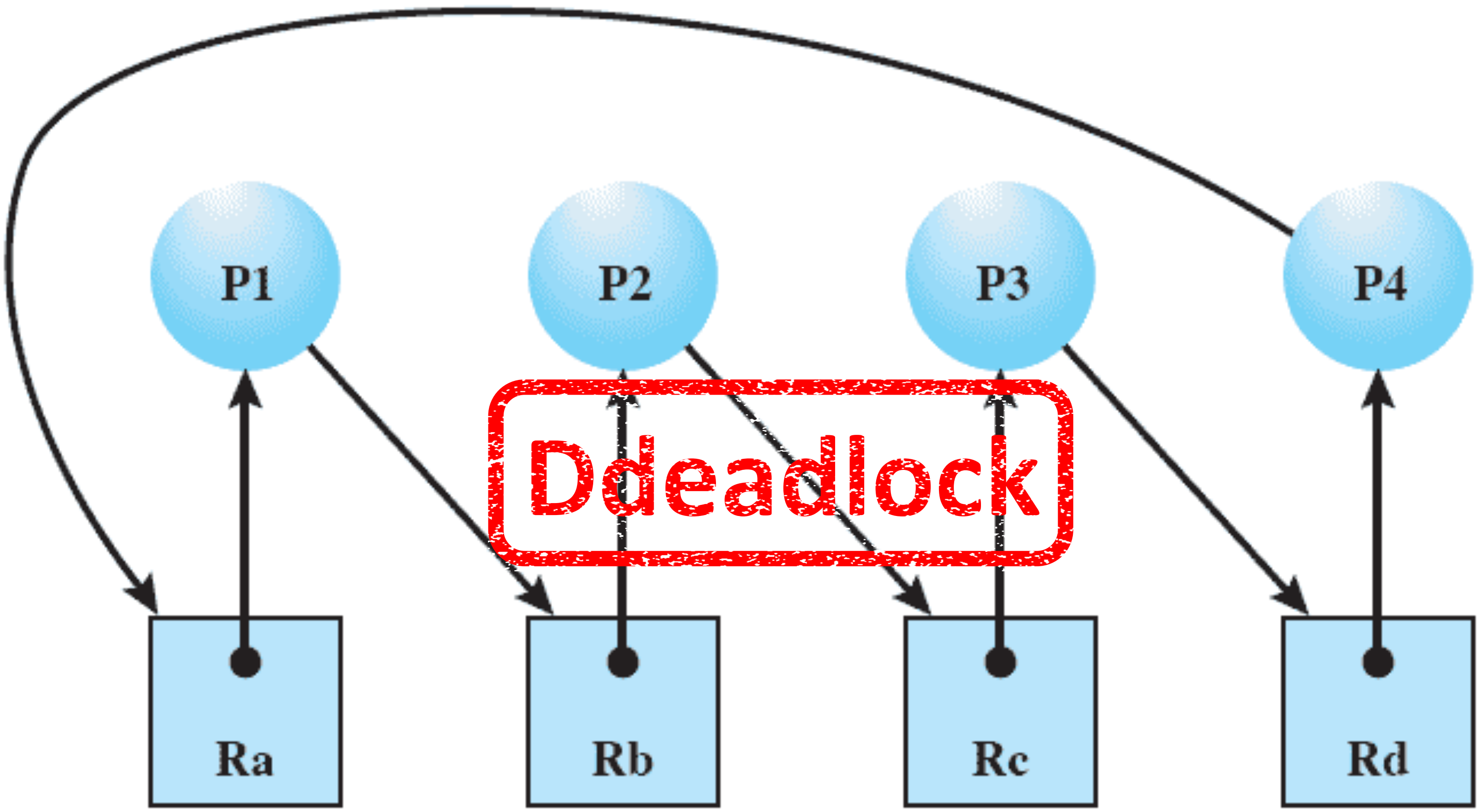


Example of Resource Allocation Graphs

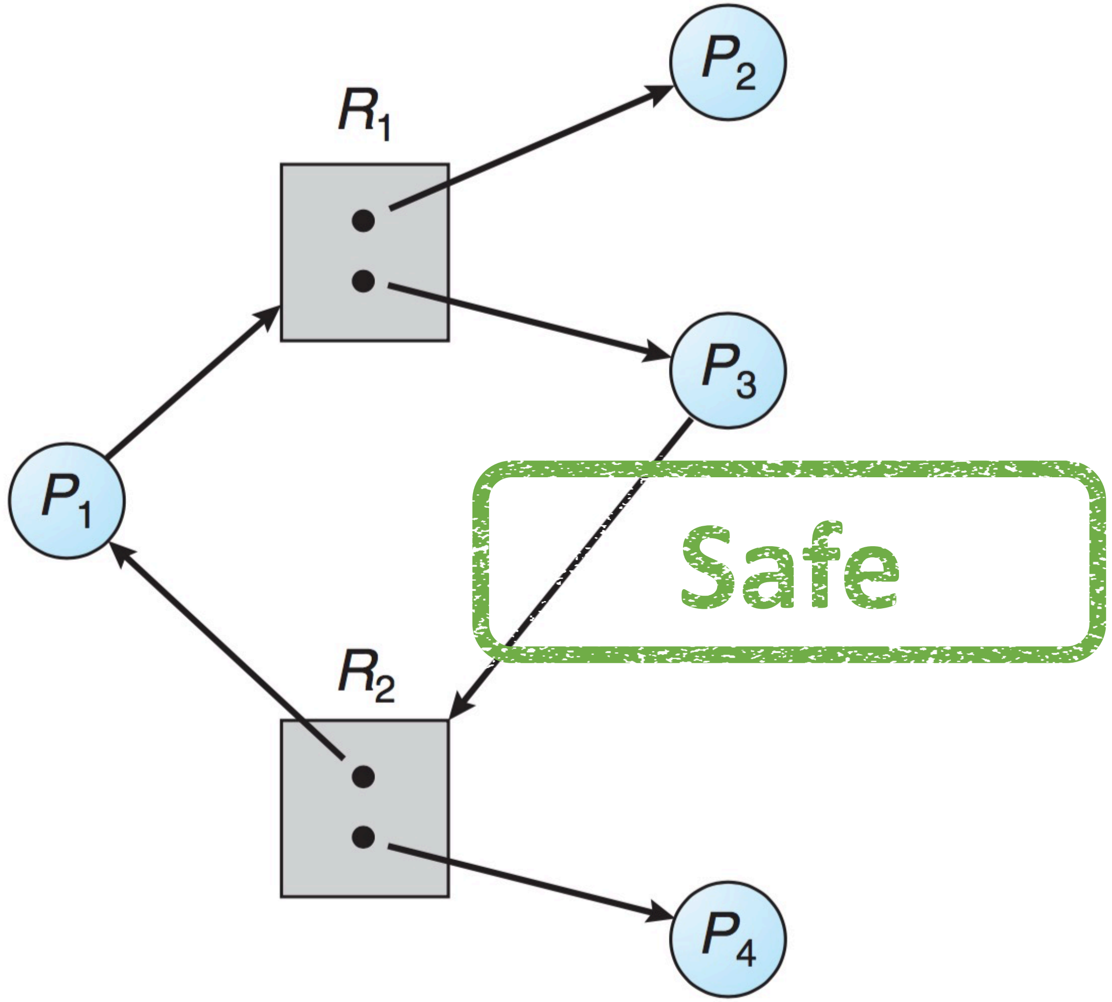




Example of Resource Allocation Graphs



Example of Resource Allocation Graphs



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

How to Deal with Deadlocks?

- Deadlock prevention
- Deadlock avoidance

The system never enters a deadlock state

How to Deal with Deadlocks?

- Deadlock prevention
- Deadlock avoidance

The system never enters a deadlock state

- Deadlock detection and recovery
- Ignore the issue! ;)

The system may enter a deadlock state

Deadlocks

Deadlock prevention

Deadlock Prevention

- Provides a set of methods to ensure that at least one of the necessary conditions cannot hold
- These methods prevent deadlocks by constraining how requests for resources can be made

Conditions Necessary for Deadlock

- **mutual exclusion**: only one process can use a resource at a time
- **hold and wait**: a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes
- **no preemption**: a resource can only be released voluntarily by the process holding it
- **circular wait**: a cycle of process requests exists (i.e., $P_0 \dashrightarrow P_1 \dashrightarrow P_2 \dashrightarrow \ldots \dashrightarrow P_{n-1} \dashrightarrow P_0$).

Deadlock Prevention

- Restrict the way requests can be made…
- **Mutual Exclusion**
  - not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait**
  - must guarantee that whenever a process requests a resource, it does not hold any other resources
  - require process to request and allocate all its resources before it begins execution
  - low resource utilization; starvation possible

Deadlock Prevention

- Restrict the way requests can be made…
- **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
- **Circular Wait**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Example of Deadlock 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);
}
```

Thread 1

```
transaction (A, B, 25);
```

Thread 2

```
transaction (B, A, 50);
```

Deadlocks

Deadlock avoidance

Deadlock Avoidance

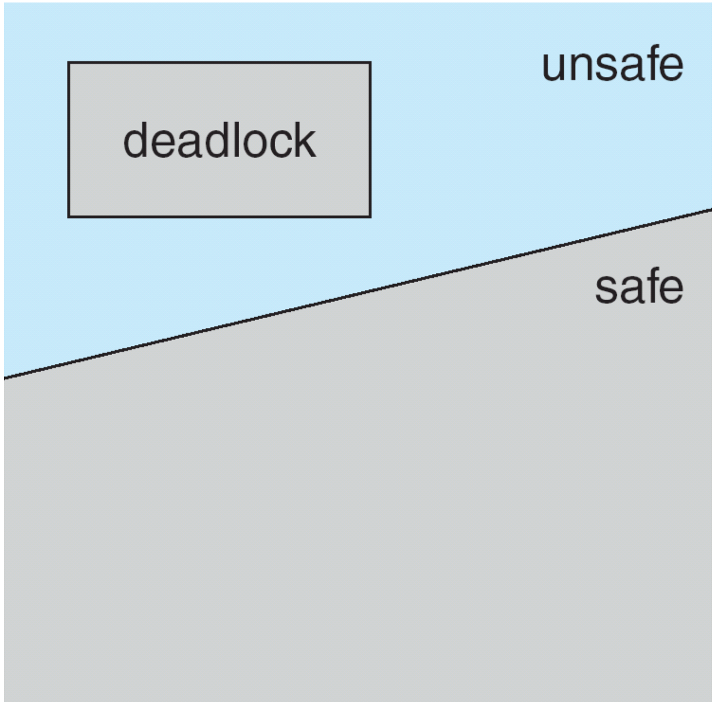
- Requires that the system has some additional *a priori* information available
  - Requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be 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 $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the system such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i 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

Deadlock Avoidance

- 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



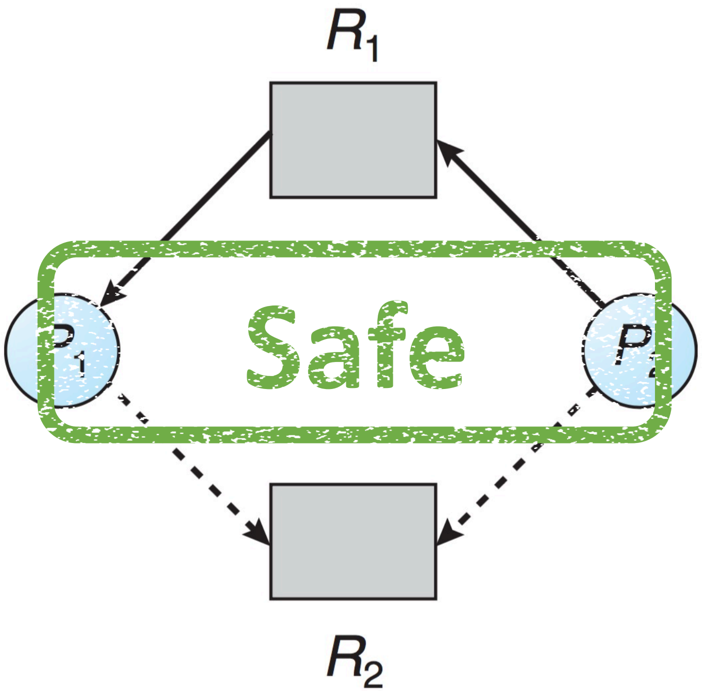
Avoidance Algorithms

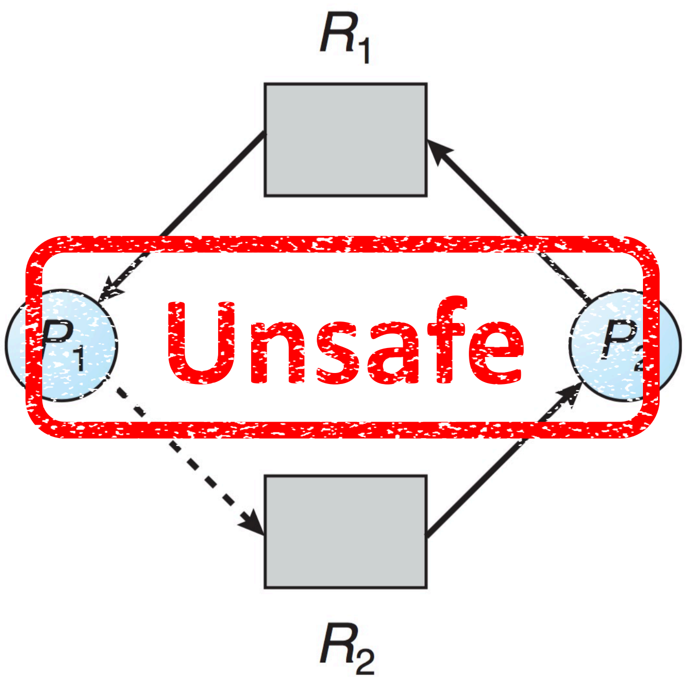
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Scheme

- Claim edge $P_i \dashrightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- 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
- Resources must be claimed a priori in the system

Resource-Allocation Graph





Banker's Algorithm

- Resources may have 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

https://www.youtube.com/watch?v=w0LwGqffUkg

Initial

A	B	C
10	5	7

Available

A	B	C
3	3	2

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
	7	**2**	**5**

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Need (= max – alloc)

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

	Available		
P_0 — finish[0] = F			
P_1 — finish[1] = T	**5**	**3**	**2**
P_2 — finish[2] = F			
P_3 — finish[3] = T	**7**	**4**	**3**
P_4 — finish[4] = T	**7**	**4**	**5**

	Available		
P_0 — finish[0] = T	**7**	**5**	**5**
P_2 — finish[2] = T	**10**	**5**	**7**

$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

Deadlocks

Deadlock detection

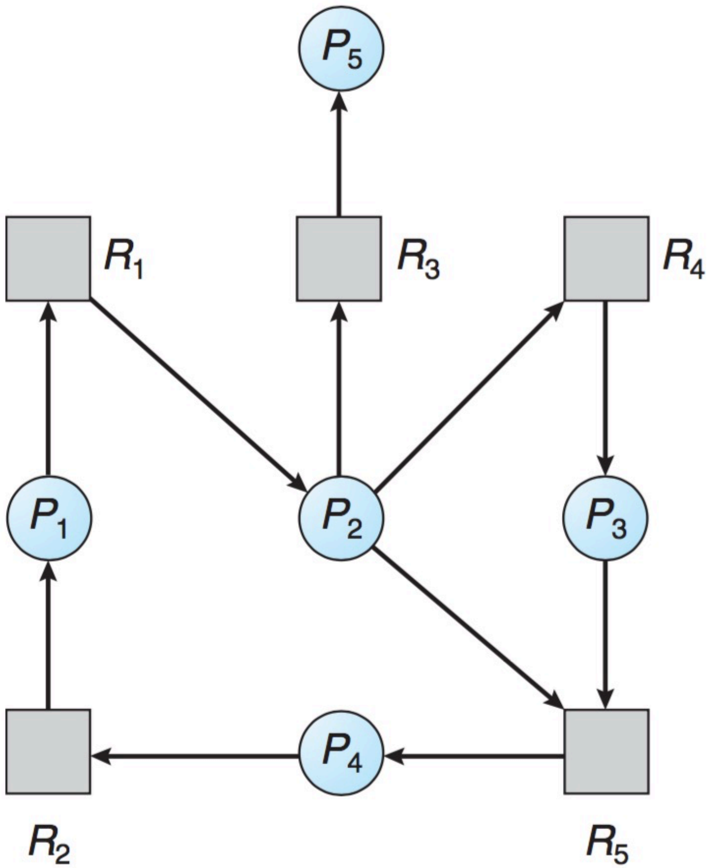
Deadlock Detection

- If neither avoidance nor prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - **Detection**: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - **Recovery**: resolve the deadlock

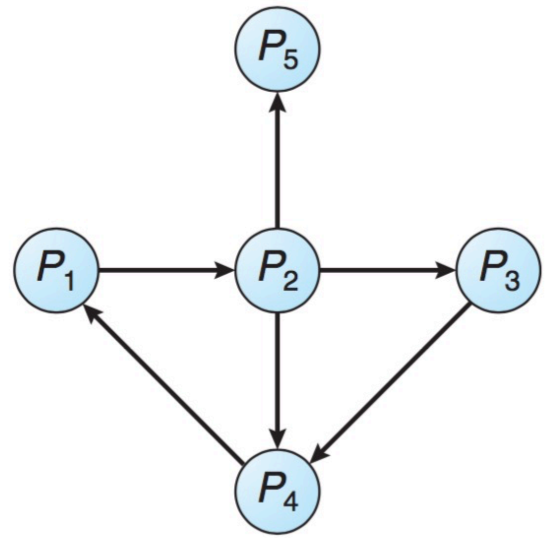
Single Instance of Each Resource Type

- Maintain a *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for a resource held by P_j
- Periodically invoke an algorithm that searches for a cycle in the 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 Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- Yes! It is possible!
- Algorithm inspired in the Banker's algorithm

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Resource preemption
- Roll back each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur

Recovery from Deadlock: Process Termination

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

Recovery from Deadlock: Resource Preemption

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

Roll Back

- Roll back all the processes
  - Possibly to a situation where no locks are being held
- Pray for the deadlock to not happen again

Acknowledgments

- Some parts of this presentation was based in publicly available slides and PDFs
  - www.cs.cornell.edu/courses/cs4410/2011su/slides/lecture10.pdf
  - www.microsoft.com/en-us/research/people/madanm/
  - williamstallings.com/OperatingSystems/
  - codex.cs.yale.edu/avi/os-book/OS9/slide-dir/

The END