

## 5 Concurrent Errors

### 5.1 Race Condition

- A **race condition** occurs when there is missing:
  - synchronization
  - mutual exclusion
- Two or more tasks race along assuming synchronization or mutual exclusion has occurred.
- Can be very difficult to locate (thought experiments).

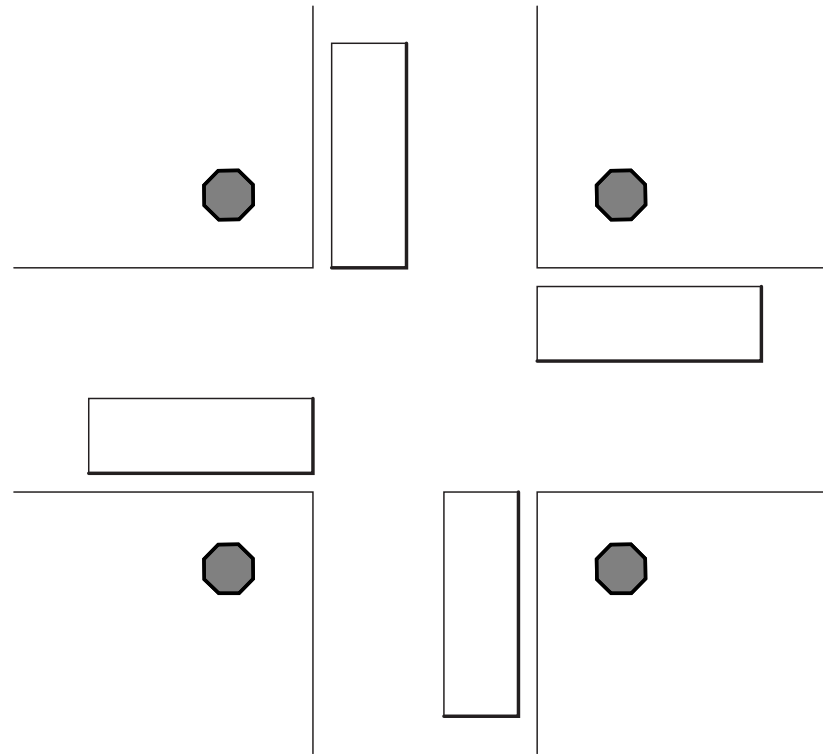
### 5.2 No Progress

#### 5.2.1 Live-lock

- indefinite postponement: the “You go first” problem on simultaneous arrival
- Caused by poor scheduling:

---

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.



- There always exists some scheduling algorithm that can deal effectively with live-lock.

## 5.2.2 Starvation

- A selection algorithm ignores one or more tasks so they are never executed.
- I.e., lack of long-term fairness.

- Long-term (infinite) starvation is extremely rare, but short-term starvation can occur and is a problem.

### 5.2.3 Deadlock

- **Deadlock** is the state when one or more processes are waiting for an event that will not occur.

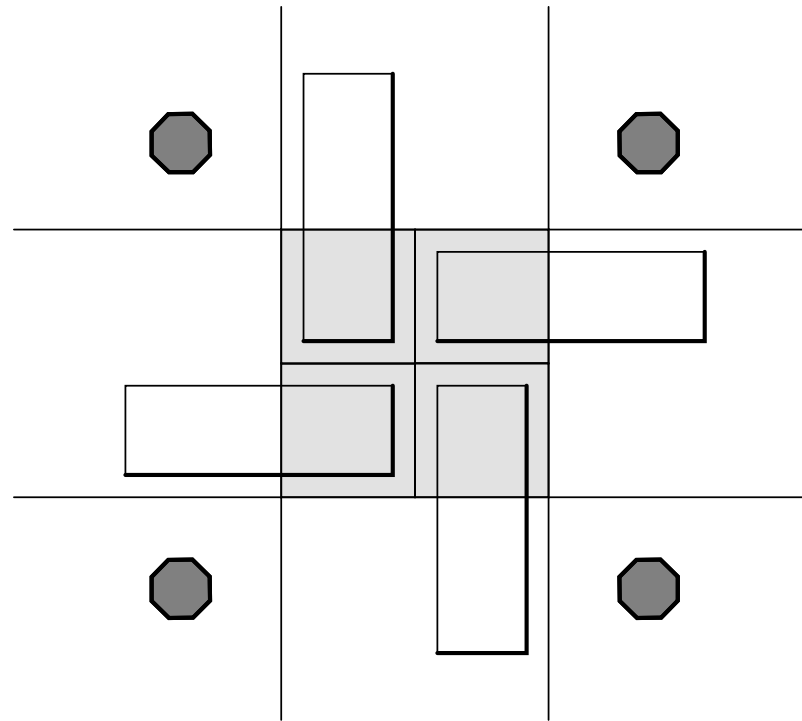
#### 5.2.3.1 Synchronization Deadlock

- Failure in cooperation, so a blocked task is never unblocked (stuck waiting):

```
void uMain::main() {  
    uSemaphore s(0);    // closed  
    s.P();              // wait for lock to open  
}
```

#### 5.2.3.2 Mutual Exclusion Deadlock

- Failure to acquire a resource protected by mutual exclusion.



Deadlock, unless one of the cars is willing to backup.

- Simple example using semaphores:

```
uSemaphore L1(1), L2(1); // open
```

task <sub>1</sub>	task <sub>2</sub>	
L1.P()	L2.P()	<i>// acquire opposite locks</i>
R1	R2	<i>// access resource</i>
L2.P()	L1.P()	<i>// acquire opposite locks</i>
R1 & R2	R2 & R1	<i>// access resource</i>

- There are 5 conditions that must occur for a set of processes to get into Deadlock.
  1. There exists a shared resource requiring mutual exclusion.
  2. A process holds a resource while waiting for access to a resource held by another process (hold and wait).
  3. Once a process has gained access to a resource, the O/S cannot get it back (no preemption).
  4. There exists a circular wait of processes on resources.
  5. These conditions must occur simultaneously.

## **5.3 Deadlock Prevention**

- Eliminate one or more of the conditions required for a deadlock from an algorithm  $\Rightarrow$  deadlock can never occur.

### **5.3.1 Synchronization Prevention**

- Eliminate all synchronization from a program
- $\Rightarrow$  no communication

- all tasks must be completely independent, generating results through side-effects.

### 5.3.2 Mutual Exclusion Prevention

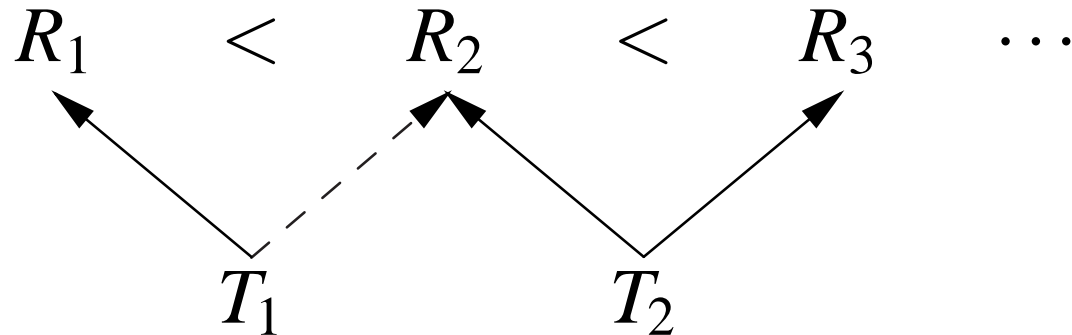
- Deadlock can be prevented by eliminating one of the 5 conditions:
  1. no mutual exclusion: impossible in many cases
  2. no hold & wait: do not give any resource, unless all resources can be given
    - $\Rightarrow$  poor resource utilization
    - possible starvation
  3. allow preemption:
    - Preemption is dynamic  $\Rightarrow$  cannot apply statically.
  4. no circular wait:
    - Control the order of resource allocations to prevent circular wait:

```

uSemaphore L1(1), L2(1); // open
task1                task2
L1.P()                L1.P() // acquire same locks
R1                    R1 // access resource
L2.P()                L2.P() // acquire same locks
R2                    R2 // access resource

```

- Use an **ordered resource** policy:



- divide all resources into classes  $R_1, R_2, R_3$ , etc.
- rule: can only request a resource from class  $R_i$  if holding no resources from any class  $R_j$  for  $j \geq i$
- unless each class contains only one resource, requires requesting several resources simultaneously
- denote the highest class number for which  $T$  holds a resource by  $h(T)$

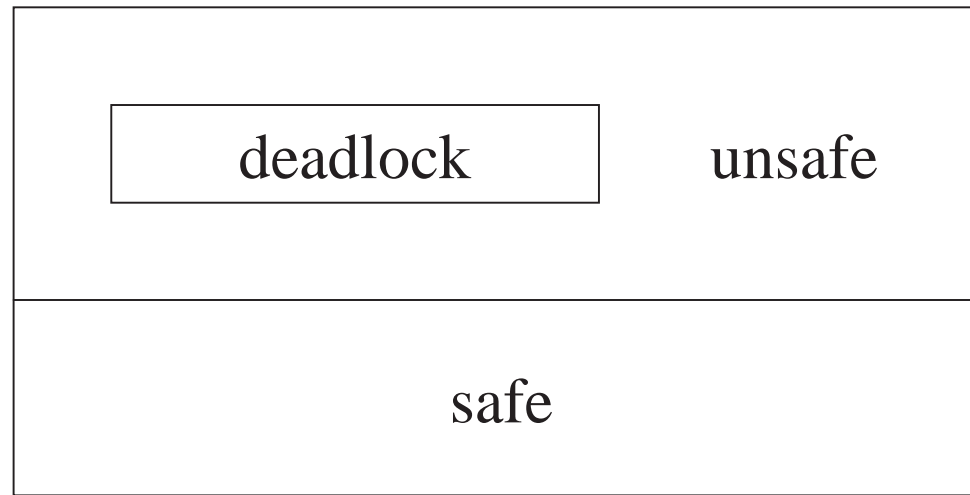
- if process  $T_1$  is requesting a resource of class  $k$  and is blocked because that resource is held by process  $T_2$ , then  $h(T_1) < k \leq h(T_2)$
- as the preceding inequality is strict, a circular wait is impossible
- in some cases there is a natural division of resources into classes that makes this policy work nicely
- in other cases, some processes are forced to acquire resources in an unnatural sequence, complicating their code and producing poor resource utilization

5. prevent simultaneous occurrence:

- Show previous 4 rules cannot occur simultaneously.

## 5.4 Deadlock Avoidance

- Monitor all lock blocking and resource allocation to detect any potential formation of deadlock.



- Achieve better resource utilization, but additional overhead to avoid deadlock.

### 5.4.1 Banker's Algorithm

- Demonstrate a safe sequence of resource allocations to processes that  $\Rightarrow$  no deadlock.
- However, to do this requires that a process state its maximum resource needs.

	R1	R2	R3	R4	
T1	4	10	1	1	maximum needed
T2	2	4	1	2	for execution
T3	5	9	0	1	(M)
T1	2	5	1	0	currently
T2	1	2	1	0	allocated
T3	1	2	0	0	(C)

resource request (T1, R1)  $2 \rightarrow 3$

T1	1	5	0	1	needed to
T2	1	2	0	2	execute
T3	4	7	0	1	( $N = M - C$ )

- Is there a safe order of execution that avoids deadlock should each process require its maximum resource allocation?

total available resources				
R1	R2	R3	R4	
6	12	4	2	(TR)

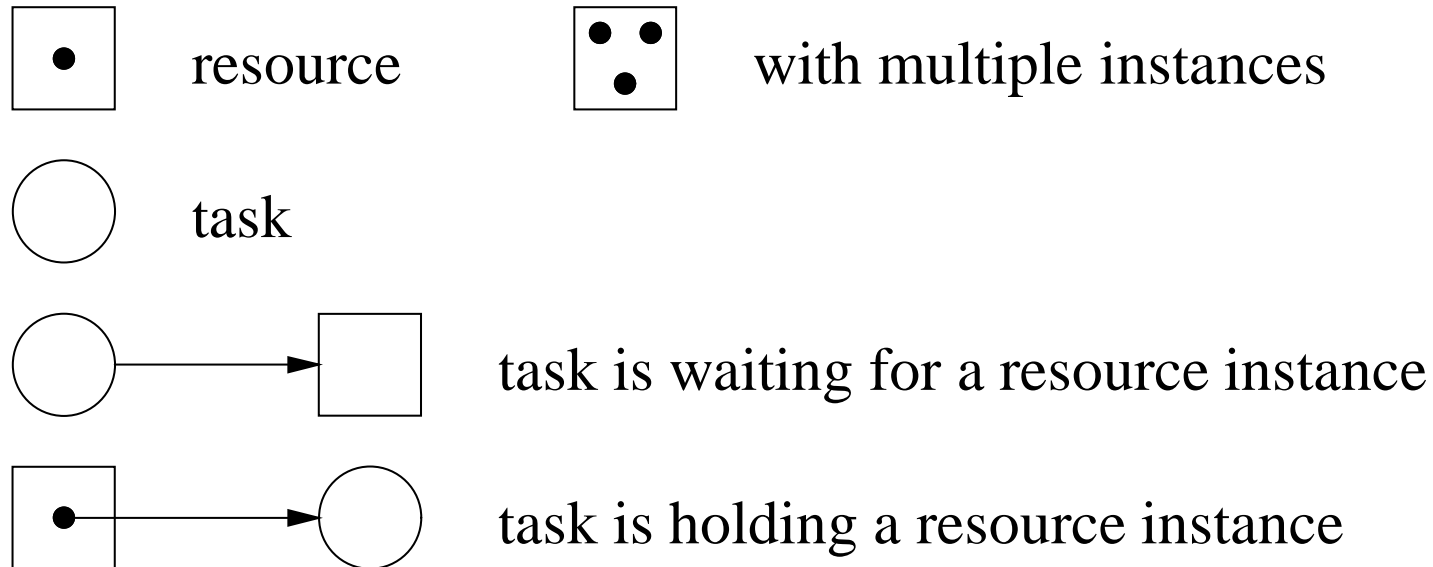
current available resources					
	1	3	2	2	$(CR = TR - \sum C_{cols})$
T2	0	1	2	0	$(CR = CR - N_{T2})$
	2	5	3	2	$(CR = CR + M_{T2})$
T1	1	0	3	1	$(CR = CR - N_{T1})$
	5	10	4	2	$(CR = CR + M_{T1})$
T3	1	3	4	1	$(CR = CR - N_{T3})$
	6	12	4	2	$(CR = CR + M_{T3})$

- If there is a choice of processes to choose for execution, it does not matter which path is taken.
- Example: If T1 or T3 could go to their maximum with the current resources, then choose either. A safe order starting with T1 exists if and only if a safe order starting with T3 exists.
- So a safe order exists (the left column in the table above) and hence the Banker's Algorithm allows the resource request.

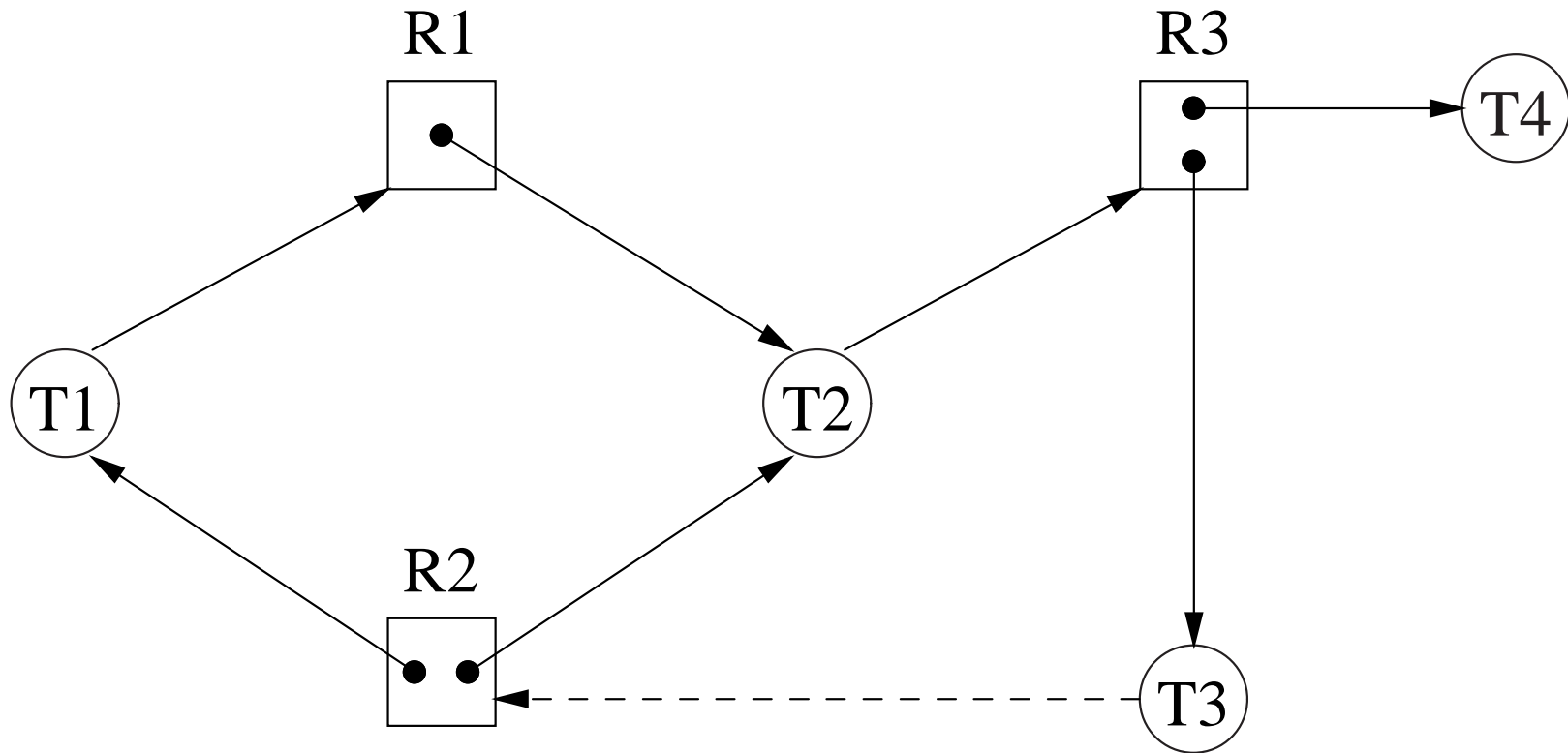
- The check for a safe order is performed for every allocation of resource to a process and then process scheduling is adjusted appropriately.

## 5.4.2 Allocation Graphs

- One method to check for potential deadlock is to graph processes and resource usage at each moment a resource is allocated.

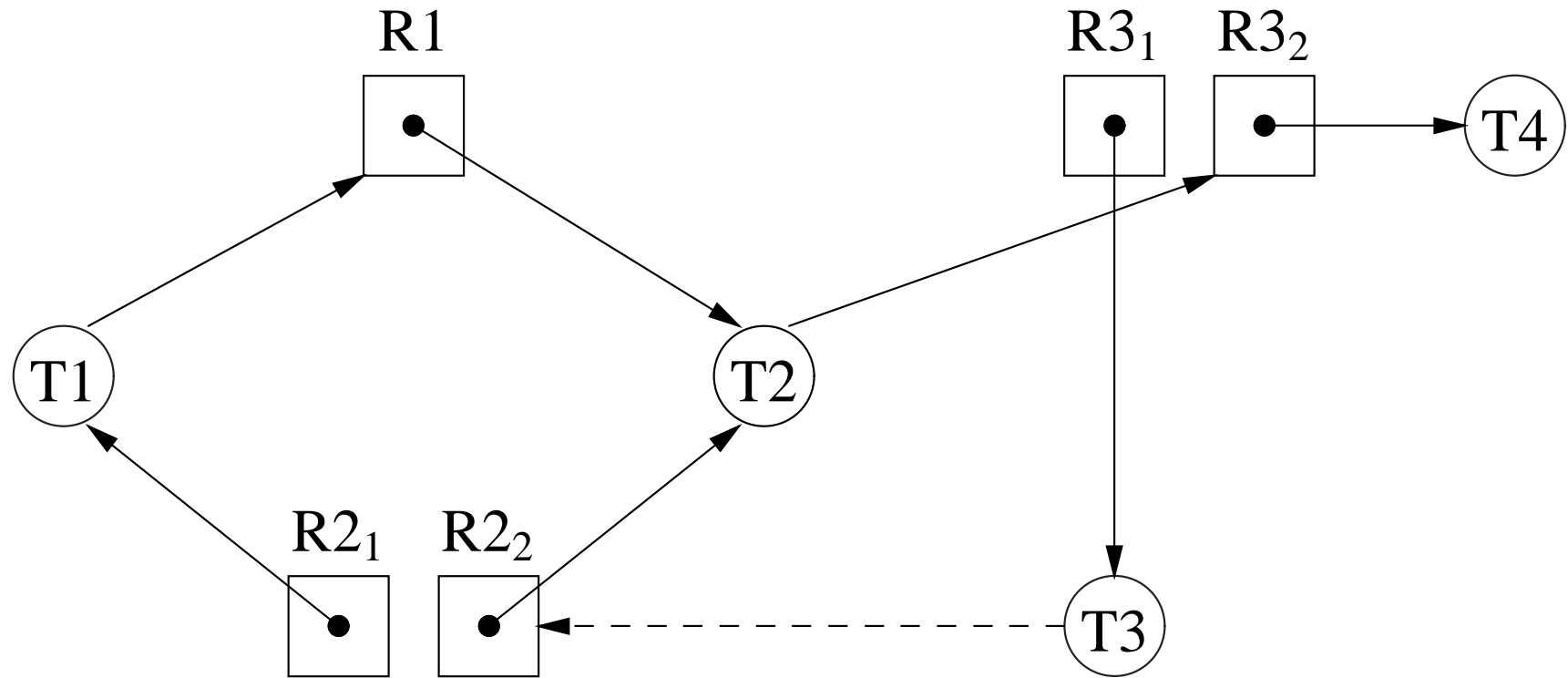


- Multiple instances are put into a resource so that a specific resource does not have to be requested. Instead, a generic request is made.

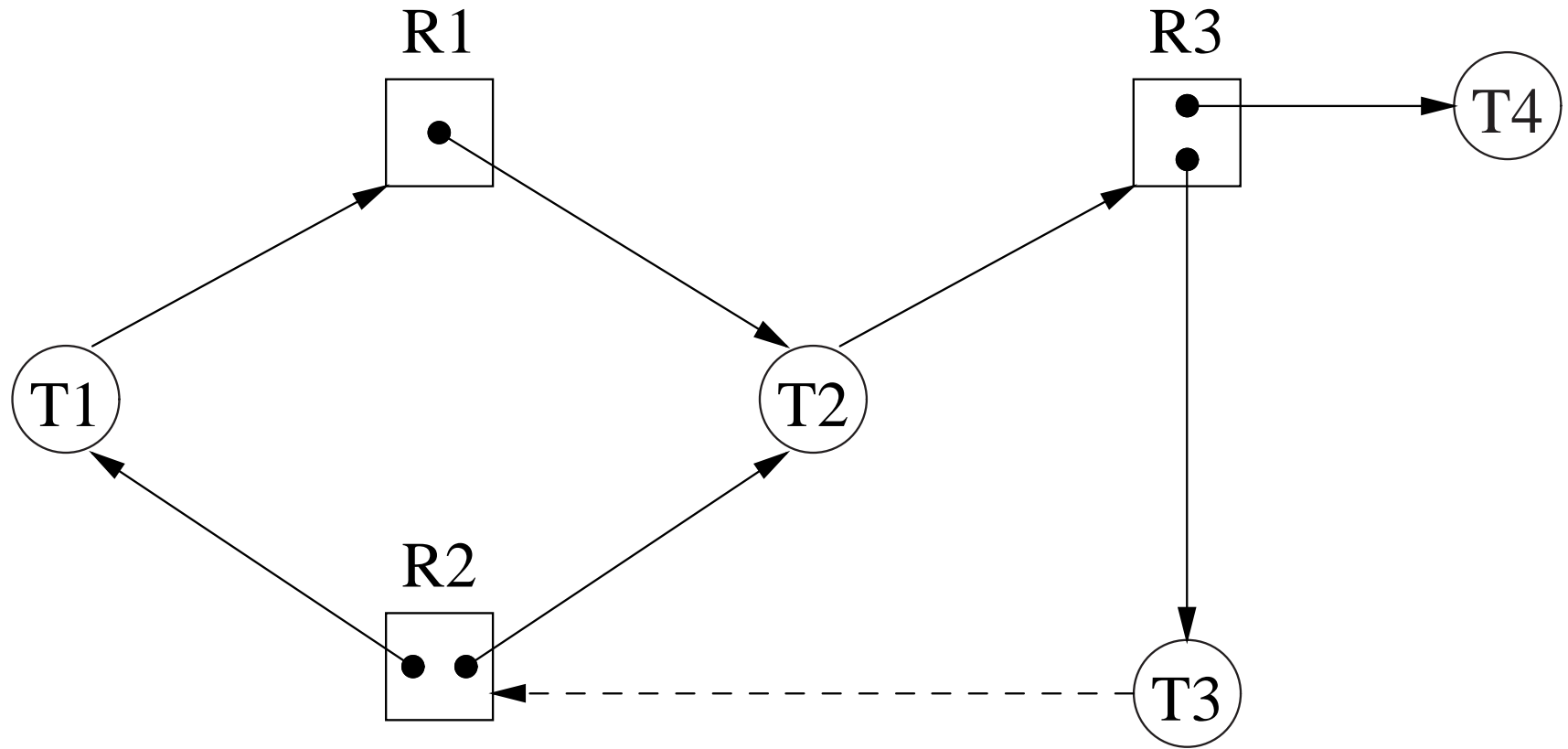


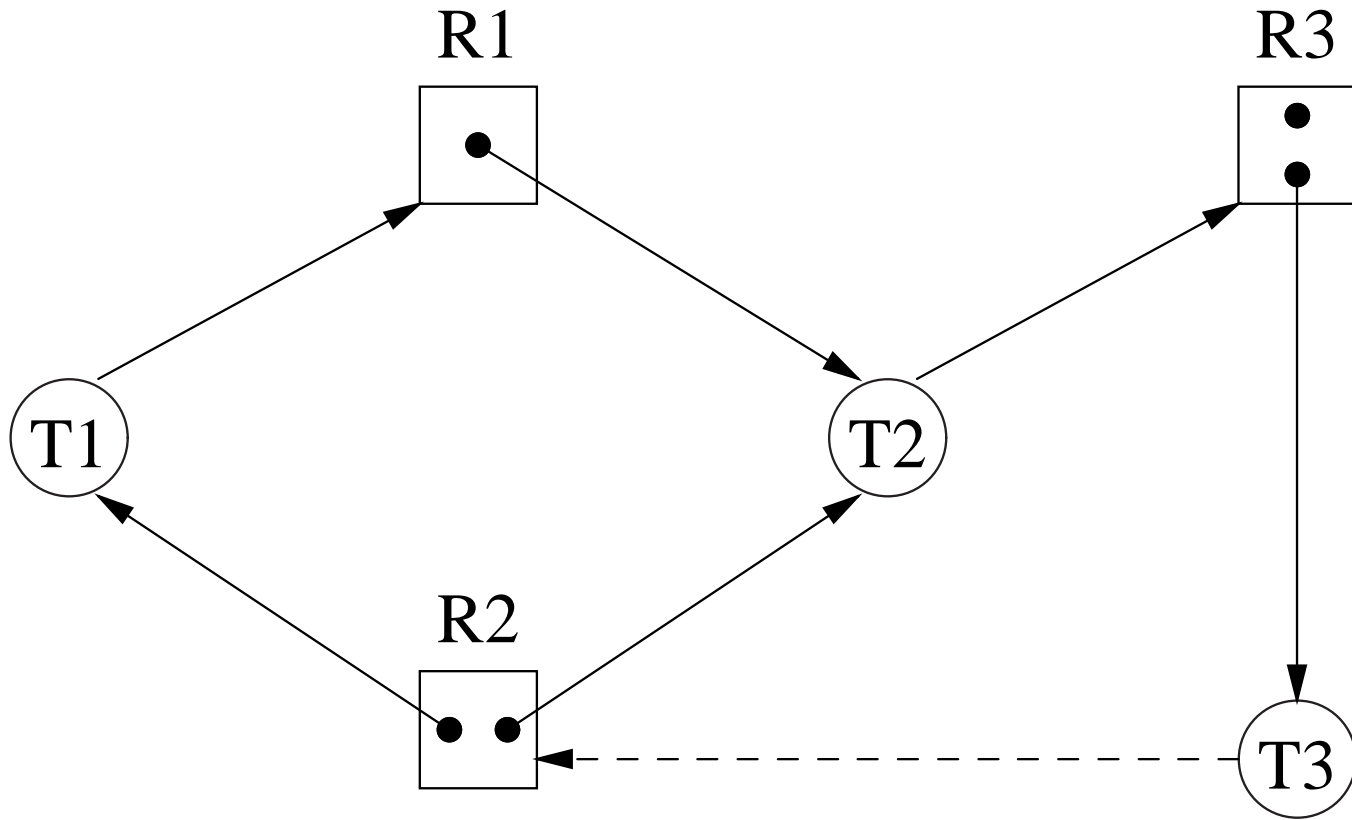
- If a graph contains no cycles, no process in the system is deadlocked.
- If each resource has one instance, a cycle  $\Rightarrow$  deadlock.
  - T1  $\rightarrow$  R1  $\rightarrow$  T2  $\rightarrow$  R3  $\rightarrow$  T3  $\rightarrow$  R2  $\rightarrow$  T1 (deadlock)
  - T2  $\rightarrow$  R3  $\rightarrow$  T3  $\rightarrow$  R2  $\rightarrow$  T2 (deadlock)
- If any resource has several instances, a cycle  $\nRightarrow$  deadlock.
  - If T4 releases its resource, the cycle is broken.

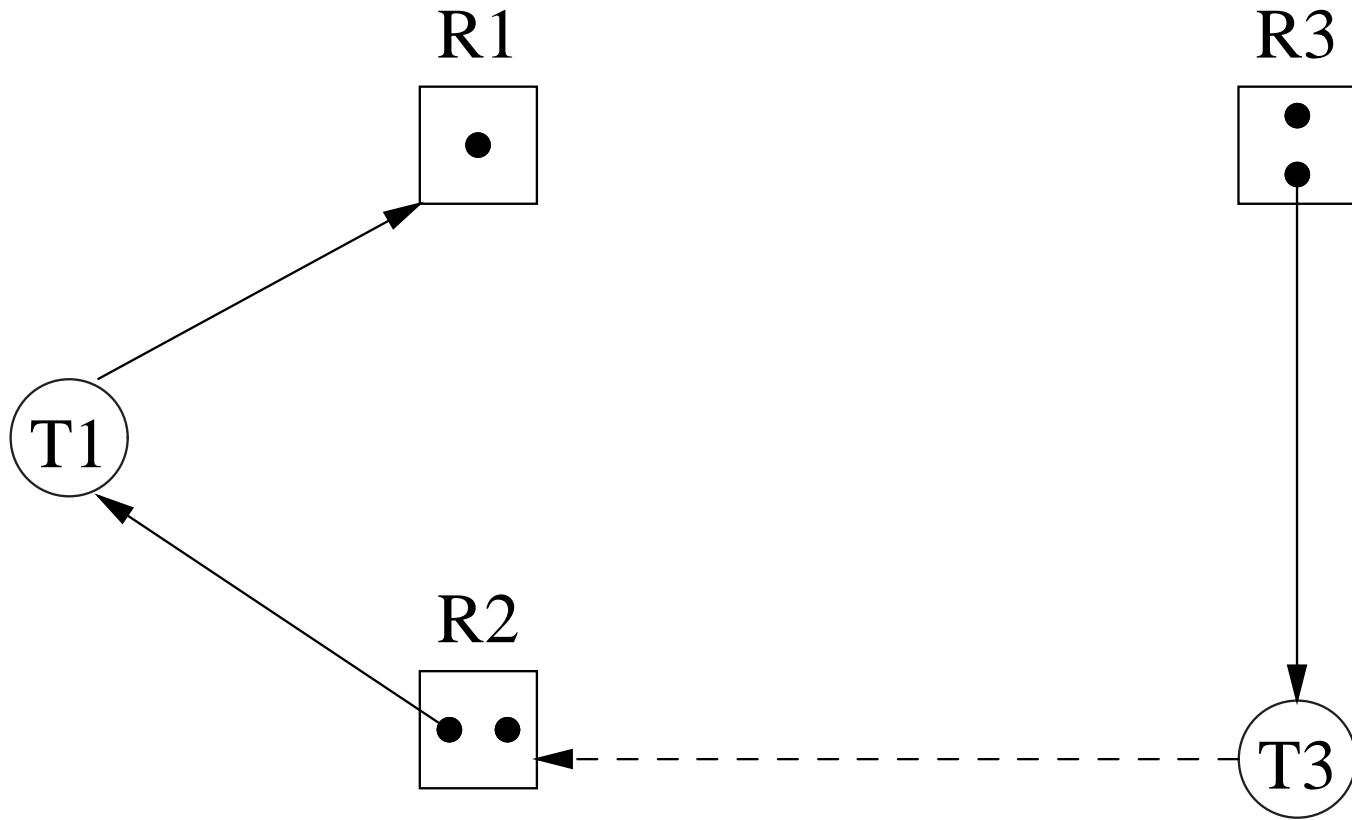
- Create isomorphic graph without multiple instances (expensive and difficult):

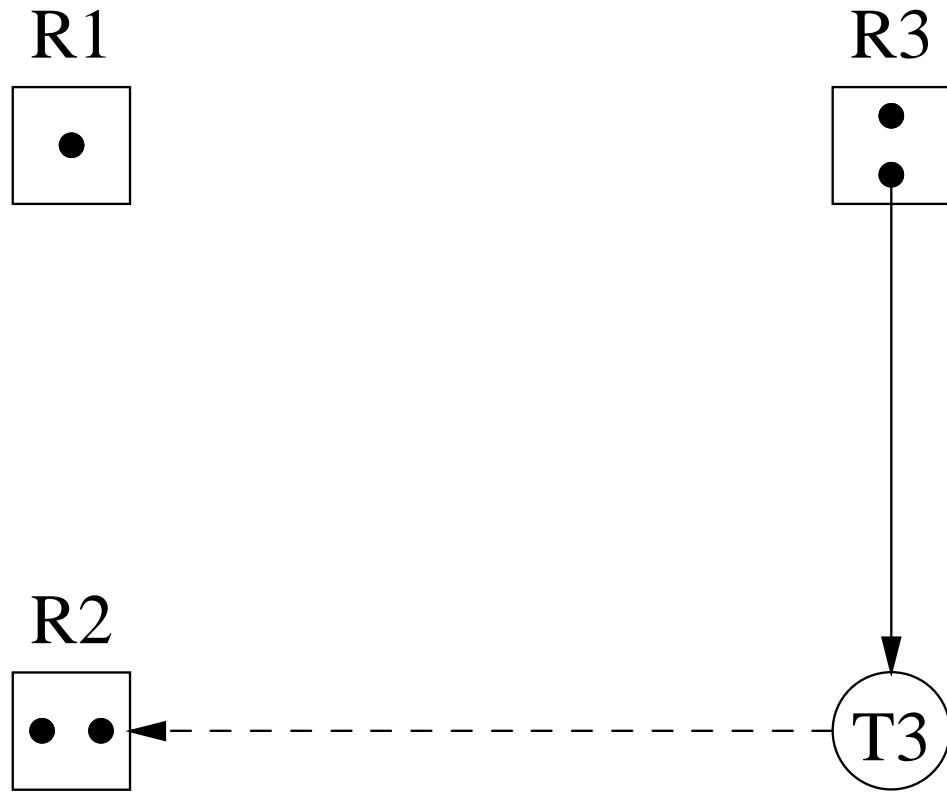


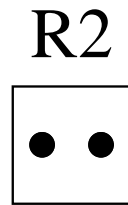
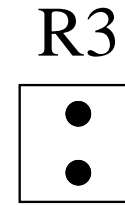
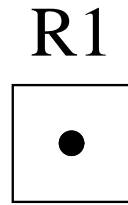
- Use graph reduction to locate deadlocks:











- Problems:

- for large numbers of processes and resources, detecting cycles is expensive.
- there may be large number of graphs that must be examined, one for each particular allocation state of the system.

## 5.5 Detection and Recovery

- Instead of avoiding deadlock let it happen and recover.
  - $\Rightarrow$  ability to discover deadlock
  - $\Rightarrow$  preemption
- Discovering deadlock is not easy, e.g., build and check for cycles in allocation graph.
  - not on each resource allocation, but every  $T$  seconds or every time a resource cannot be immediately allocated
- Recovery involves preemption of one or more processes in a cycle.
  - decision is not easy and must prevent starvation
  - The preemption victim must be restarted, from beginning or some previous checkpoint state, if you cannot guarantee all resources have not changed.
  - even that is not enough as the victim may have made changes before the preemption.

## 5.6 Which Method To Chose?

- Maybe “none of the above”: just ignore the problem

- if some process is blocked for rather a long time, assume it is deadlocked and abort it
- do this automatically in transaction-processing systems, manually elsewhere
- Of the techniques studied, only the ordered-resource policy turns out to have much practical value.