DCS216 Operating Systems

Lecture 15
Synchronization (4)

Apr 17th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

Content

- Liveness
- Starvation vs. Deadlock
- Deadlock in Multithreaded Applications
- System Model & Resource-Allocation Graph
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock **Prevention**
    - Invalidate one of the necessary conditions
  - Deadlock **Detection**
    - Deadlock Detection Algorithm
    - Recovery from Deadlock
  - Deadlock **Avoidance**
    - Banker's Algorithm

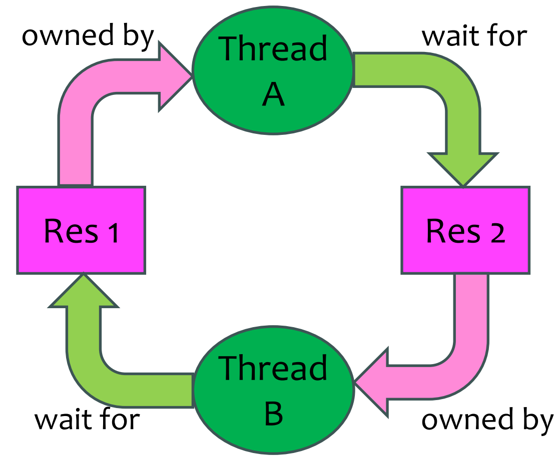
- **Liveness (活性)**
  - Liveness refers to the properties that a system must satisfy to ensure that processes make progress during their execution life cycle.
    - a guarantee that something good **eventually** happens
  - Key aspects of liveness:
    - Progress
    - Freedom from Deadlock
    - Freedom from Starvation
    - Fairness

Liveness (活性)

- Liveness refers to the properties that a system must satisfy to ensure that processes make progress during their execution life cycle.
  - a guarantee that something good **eventually** happens
- Key aspects of liveness:
  - **Progress:** ensures that if a process needs to perform an action, it will eventually be able to do so.
    - A failure in this property might result in deadlock or livelock.
  - **Freedom from Deadlock:** Deadlock occurs when processes are stuck waiting indefinitely for resources that are held by each other.
  - **Freedom from Starvation:** This property ensures that every process gets a chance to proceed. Starvation occurs when a process is perpetually denied necessary resources, usually because of scheduling or resource allocation policies.
  - **Fairness:** This is often considered as part of liveness, ensuring that all processes are treated in a fair manner over time. This means that all processes will eventually be given CPU time and access to resources.

Deadlock: A Deadly type of Starvation

- **Starvation vs Deadlock**
  - Starvation: thread waits indefinitely
    - Example: low-priority thread waiting for resources constantly in use by high-priority threads, also known as **Priority Inversion (优先级反转)**.
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1, and is waiting for Res 2
    - Thread B owns Res 2, and is waiting for Res 1



  - Deadlock $\Rightarrow$ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

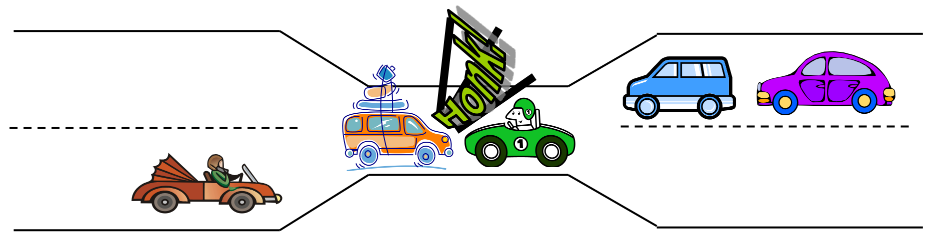
- **Single-Lane Bridge Crossing**

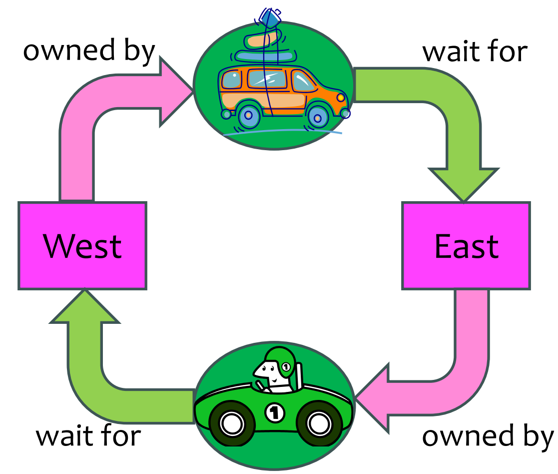
Only one car can cross at a time

Two lanes converged into one

Example: Single-Lane Bridge Crossing

- **Single-Lane Bridge Crossing**
  - Each segment of road can be viewed as a resource
    - Car must own the segment under them
    - Must acquire segment that they are moving into
  - For bridge: must acquire both halves
    - Traffic only in one direction at a time

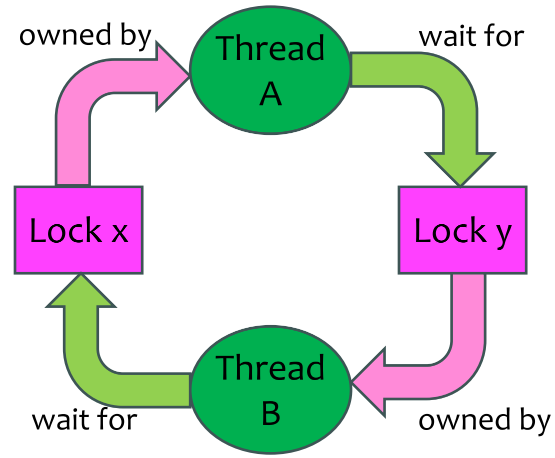




  - Deadlock: Shown above when two cars from opposite directions meet in the middle
    - Each owns one segment and requests to acquire the other
    - Deadlock resolved if one car backs up (preempt resources and rollback)
      - Several cars may have to be backed up
  - Starvation: (not deadlock)
    - East-going traffic really fast ⇒ no one can go to west

Deadlock with Locks

- This lock pattern exhibits non-deterministic deadlock
  - Sometimes it happens, sometimes it doesn't
- Really hard to debug!



Thread A:

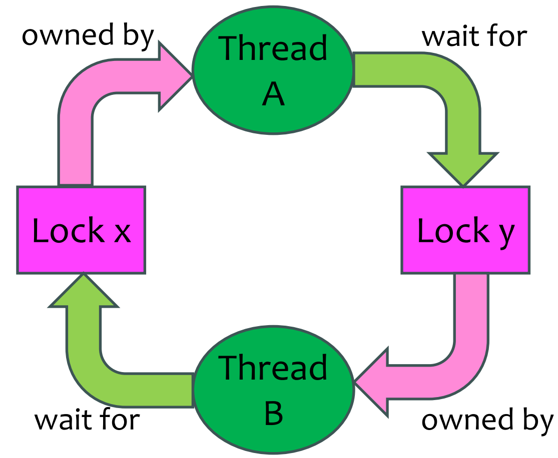
```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

Thread B:

```
acquire(&y);
acquire(&x);
...
release(&x);
release(&y);
```

Deadlock with Locks

- **Deadlock with Locks (Unlucky Case)**
  - This lock pattern exhibits non-deterministic deadlock
    - Sometimes it happens, sometimes it doesn't
  - Really hard to debug!



Thread A:

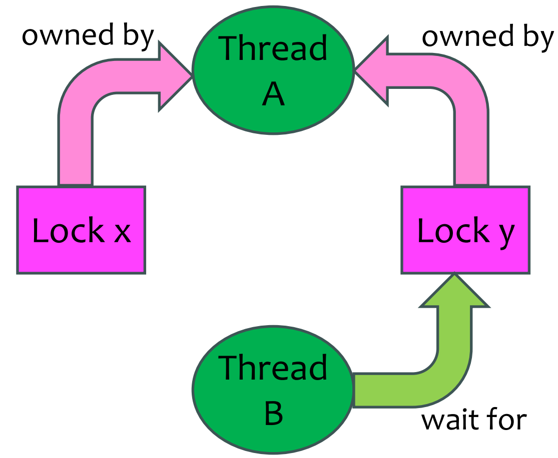
```
acquire(&x);

acquire(&y); <stalled>
<unreachable>
...
release(&y);
release(&x);
```

Thread B:

```

acquire(&y);

acquire(&x); <stalled>
<unreachable>
...
release(&x);
release(&y);
```

Deadlock with Locks (Lucky Case)

- Sometimes, deadlock won't occur with proper scheduling



Thread A:

```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

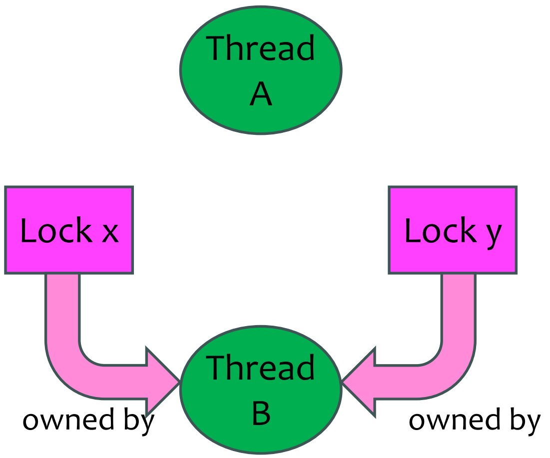
Thread B:

```
acquire(&y);
```

Deadlock with Locks

Deadlock with Locks (Lucky Case)

- Sometimes, deadlock won't occur with proper scheduling



Thread A:

```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

Thread B:

```
acquire(&y);

<y acquired>
acquire(&x);
...
release(&x);
release(&y);
```

Deadlock with Locks

Deadlock with Locks

```c
/* deadlock.c */
pthread_mutex_t lock1, lock2;

void *threadA(void *arg) {
    pthread_mutex_lock(&lock1);
    usleep(100);
    pthread_mutex_lock(&lock2);

    printf("Thread A working...\n");

    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    pthread_exit(NULL);
}
void *threadB(void *arg) {
    pthread_mutex_lock(&lock2);
    usleep(100);
    pthread_mutex_lock(&lock1);

    printf("Thread B working...\n");

    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    pthread_exit(NULL);
}
```

```c
int main() {
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_t p1, p2;
    pthread_create(&p1, NULL, threadA,NULL);
    pthread_create(&p2, NULL, threadB,NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    return 0;
}
```

```
$ ./deadlock
<process stuck...>
^C
```

Livelock

```
$ ./livelock
Thread A acquired Lock 1
Thread B acquired Lock 2
Thread A released Lock 1
Thread A acquired Lock 1
Thread B released Lock 2
Thread B acquired Lock 2
...
<continue forever>
^C
```

```c
/* livelock.c */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


pthread_mutex_t lock1, lock2;

void* threadA(void* arg) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock(&lock1);
        printf("Thread A acquired Lock 1\n");
        // Try to acquire lock2
        if (pthread_mutex_trylock(&lock2) == 0) {
            printf("Thread A acquired Lock 2\n");
            // Simulate work
            pthread_mutex_unlock(&lock2);
            printf("Thread A released Lock 2\n");
            done = 1;
        }
        usleep(100);
        pthread_mutex_unlock(&lock1);
        printf("Thread A released Lock 1\n");
    }
    pthread_exit(0);
}
```

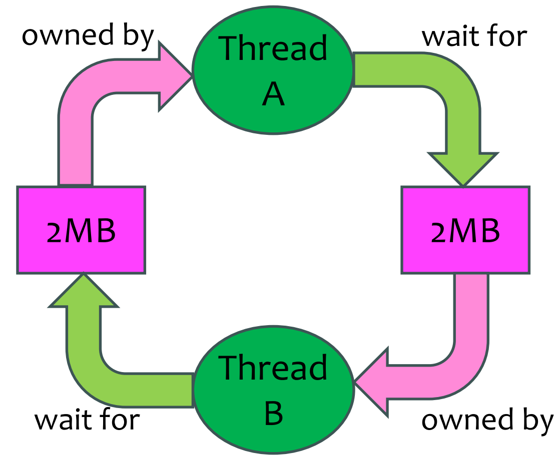
```c
void* threadB(void* arg) {
    int done = 0;
    while (!done) {
        pthread_mutex_lock(&lock2);
        printf("Thread B acquired Lock 2\n");
        // Try to acquire lock1
        if (pthread_mutex_trylock(&lock1) == 0) {
            printf("Thread B acquired Lock 1\n");
            // Simulate work
            pthread_mutex_unlock(&lock1);
            printf("Thread B released Lock 1\n");
            done = 1;
        }
        usleep(100);
        pthread_mutex_unlock(&lock2);
        printf("Thread B released Lock 2\n");
    }
    pthread_exit(0);
}
```

Deadlocks

- **Other Types of Deadlock**
  - Threads often block waiting for resources
    - Mutex Locks
    - Terminals
    - Printers
    - Memory
    - CD Drives
  - Threads often block waiting for other threads
    - Pipes
    - Sockets
  - Deadlocks can occur on any of these!

Deadlock with Memory Space

- If there're only **3MB** of free space, we get the same deadlock situation

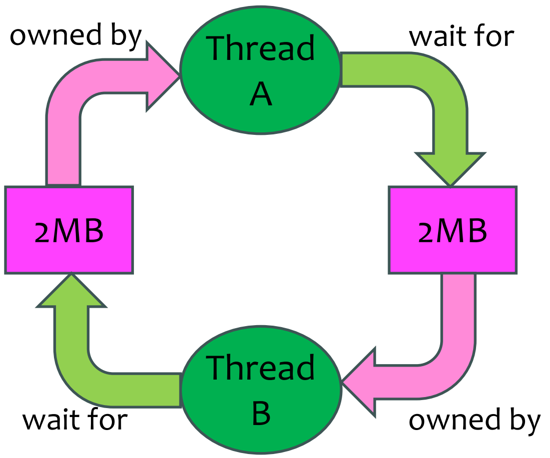


Thread A:
```
malloc_wait(2MB);
malloc_wait(2MB);
...
free(2MB);
free(2MB);
```

Thread B:
```
malloc_wait(2MB);
malloc_wait(2MB);
...
free(2MB);
free(2MB);
```

Deadlock with Memory Space

- If there're only **3MB** of free space, we get the same deadlock situation



Thread A:

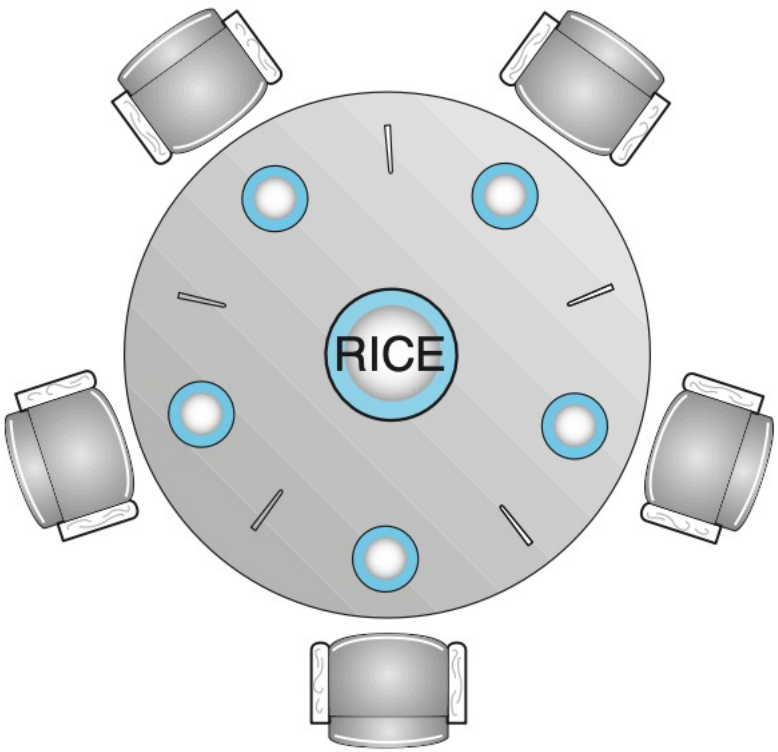
```
malloc_wait(2MB);

malloc_wait(2MB);
...
free(2MB);
free(2MB);
```

Thread B:

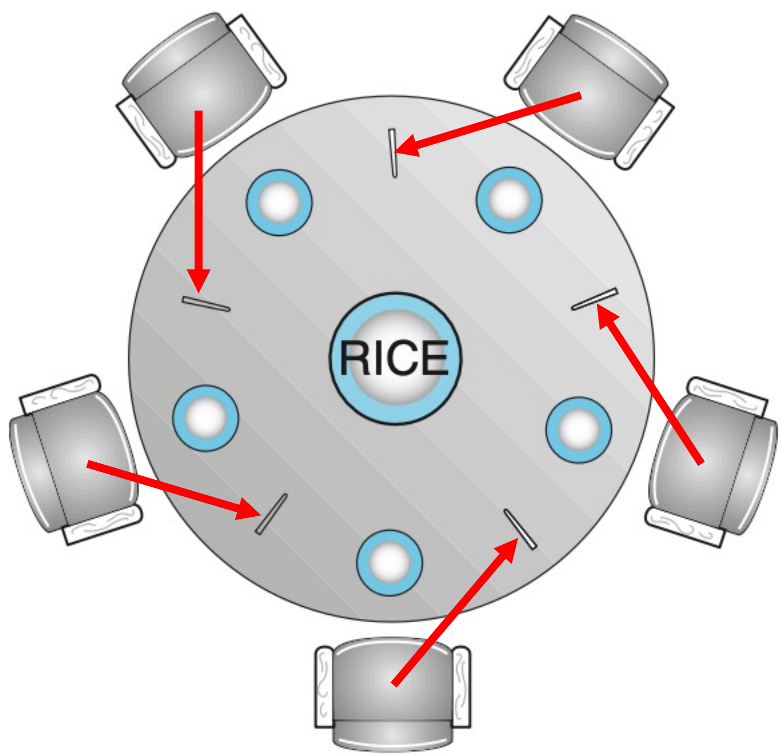
```
malloc_wait(2MB);

malloc_wait(2MB);
...
free(2MB);
free(2MB);
```

Dining-Philosopher Problem

- Five chopsticks for five philosophers
  - Philosopher will grab any one they can.
  - One chopstick at a time.
  - Need two chopsticks to eat

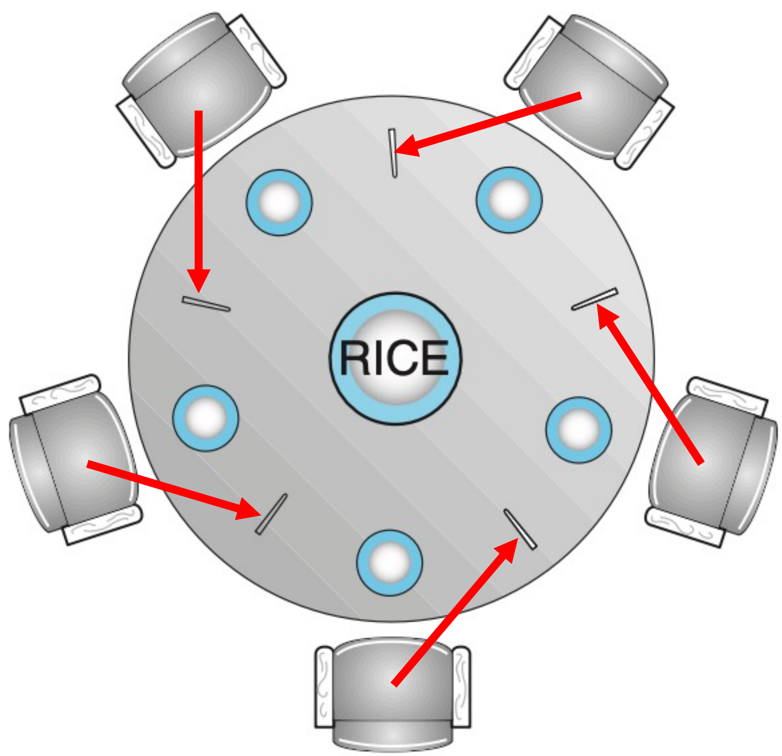


- **Dining-Philosopher Problem**
  - Five chopsticks for five philosophers
    - What if they all grab the one chopstick on their right at the same time?
      - Deadlock!



Dining-Philosopher Problem

- Deadlock
  - How to fix deadlock?
  - How to prevent deadlock?



Deadlock Characterization

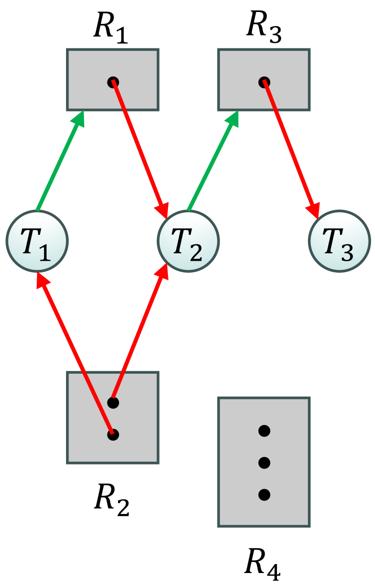
- **Four (necessary) requirements for Deadlock**
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

Deadlock Characterization

- **Four (necessary) requirements for Deadlock**
  - **Mutual Exclusion**
    - Only one thread at a time can use a resource
  - **Hold and Wait**
    - Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - **No Preemption**
    - Resources are released only voluntarily by the thread holding that resources, after thread is finished with it
  - **Circular Wait**
    - There exists a set $\{T_1, T_2, \ldots, T_n\}$ of waiting threads
      - T_1 is waiting for a resource held by T_2
      - T_2 is waiting for a resource held by T_3
      - …
      - T_n is waiting for a resource held by T_1

System Model

- A set of n Threads $\{T_1, T_2, \ldots, T_n\}$
- Resource types $\{R_1, R_2, \ldots, R_m\}$
  - CPU cycles, Memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - `Request() / Use() / Release()`

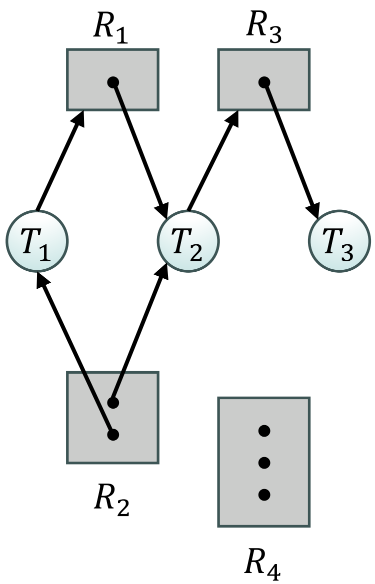


Resource-Allocation Graph: $(\boldsymbol{V}, \boldsymbol{E})$

- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$ are the set of **threads** in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$ are the set of **resource types** in the system.
- E can be categorized into two types:
  - **Request** Edge – directed edge $T_i \rightarrow R_j$
  - **Assignment** Edge – directed edge $R_j \rightarrow T_i$

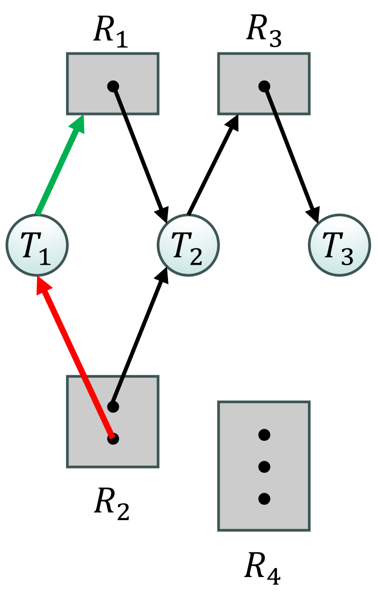
Resource-Allocation Graph Example

- The sets T, R, and E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Resource instances:
  - 1 instance of resource type R_1
  - 2 instances of resource type R_2
  - 1 instance of resource type R_3
  - 3 instances of resource type R_4
- Thread states:
  - T_1 is holding an instance of R_2 and is waiting for an instance of R_1
  - T_2 is holding an instance of R_1 and an instance of R_2, and is waiting for an instance of R_3
  - T_3 is holding an instance of R_3



Resource-Allocation Graph Example

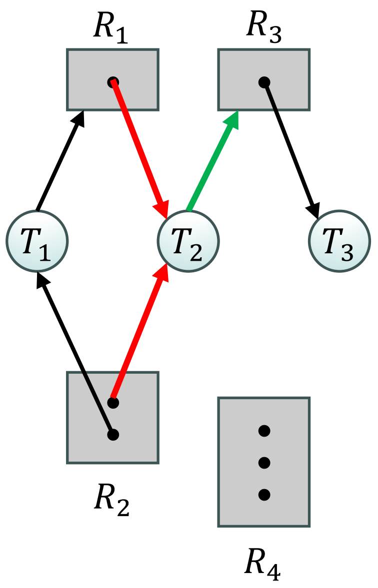
- The sets T, R, and E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3,$
    $R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$



- Resource instances:
  - 1 instance of resource type R_1
  - 2 instances of resource type R_2
  - 1 instance of resource type R_3
  - 3 instances of resource type R_4
- Thread states:
  - T_1 is holding an instance of R_2 and is waiting for an instance of R_1
  - T_2 is holding an instance of R_1 and an instance of R_2, and is waiting for an instance of R_3
  - T_3 is holding an instance of R_3

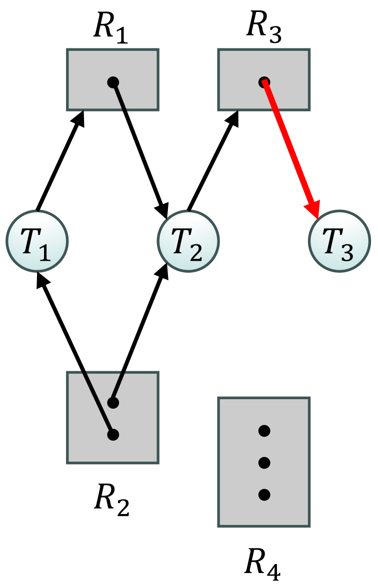
Resource-Allocation Graph Example

- The sets T, R, and E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3,$
    $R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$



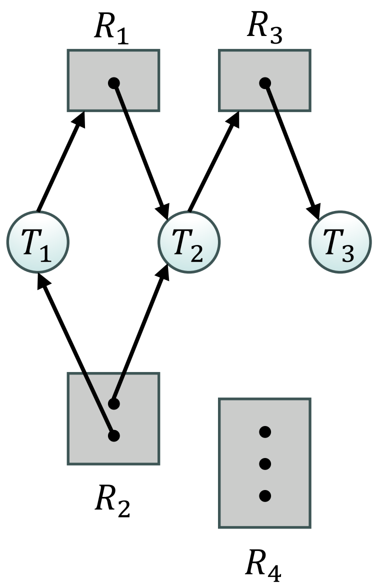
- Resource instances:
  - 1 instance of resource type R_1
  - 2 instances of resource type R_2
  - 1 instance of resource type R_3
  - 3 instances of resource type R_4
- Thread states:
  - T_1 is holding an instance of R_2 and is waiting for an instance of R_1
  - T_2 is holding an instance of R_1 and an instance of R_2, and is waiting for an instance of R_3
  - T_3 is holding an instance of R_3

Resource-Allocation Graph Example

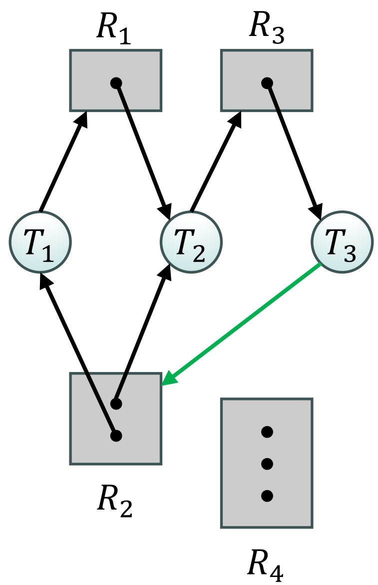


- The sets T, R, and E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{T_1 \to R_1, T_2 \to R_3, R_1 \to T_2, R_2 \to T_2, R_2 \to T_1, R_3 \to T_3\}$
- Resource instances:
  - 1 instance of resource type R_1
  - 2 instances of resource type R_2
  - 1 instance of resource type R_3
  - 3 instances of resource type R_4
- Thread states:
  - T_1 is holding an instance of R_2 and is waiting for an instance of R_1
  - T_2 is holding an instance of R_1 and an instance of R_2, and is waiting for an instance of R_3
  - T_3 is holding an instance of R_3

Resource-Allocation Graph Example

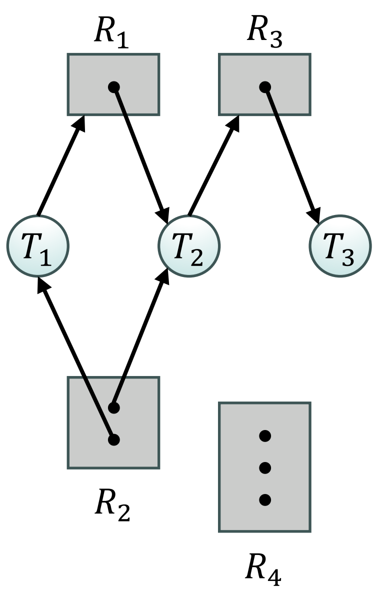


Simple Resource
Allocation Graph

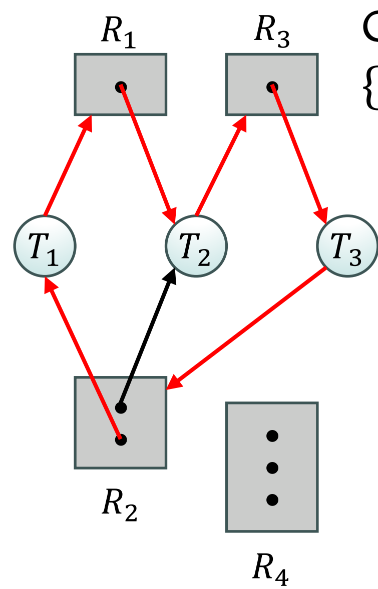


Allocation Graph
with **Deadlock**

Resource-Allocation Graph Example



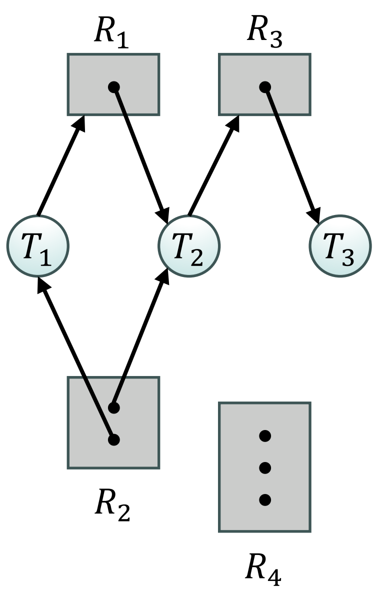
Simple Resource Allocation Graph



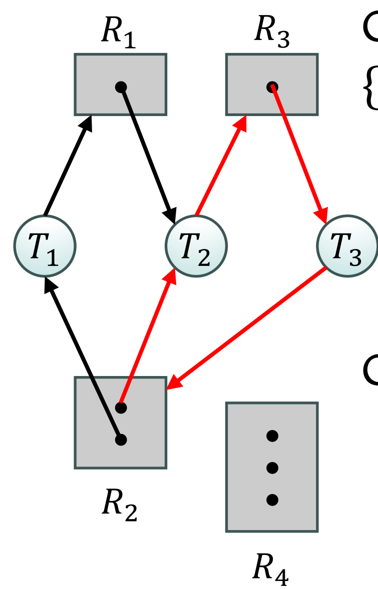
Circle 1:
$\{T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1\}$

Allocation Graph with **Deadlock**

Resource-Allocation Graph Example



Simple Resource Allocation Graph



Allocation Graph with **Deadlock**

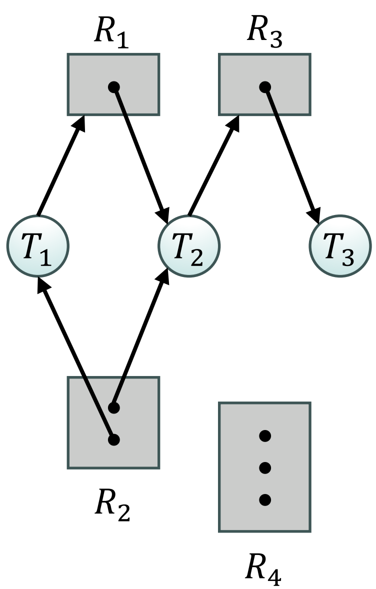
Circle 1:

$\{T_1 \to R_1 \to T_2 \to R_3 \to T_3 \to R_2 \to T_1\}$

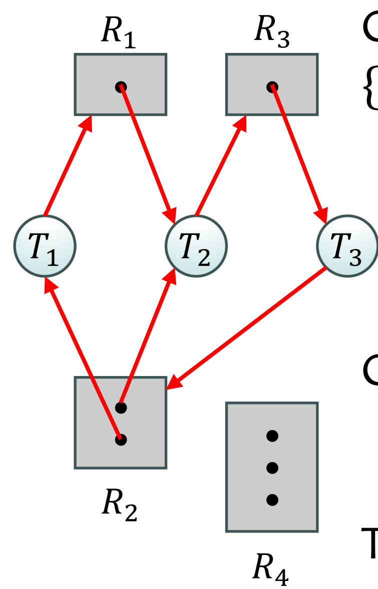
Circle 2:

$\{T_2 \to R_3 \to T_3 \to R_2 \to T_2\}$

Resource-Allocation Graph Example



Simple Resource
Allocation Graph



Allocation Graph
with **Deadlock**

Circle 1:

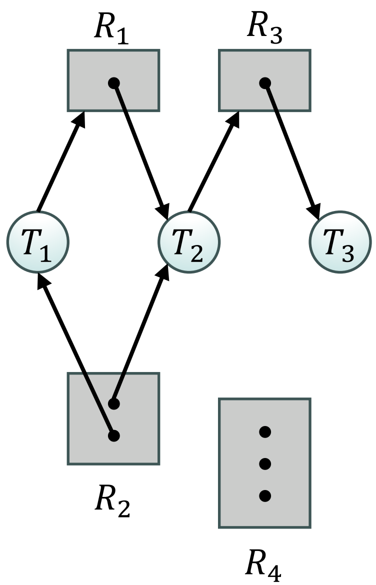
$\{T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1\}$

Circle 2:

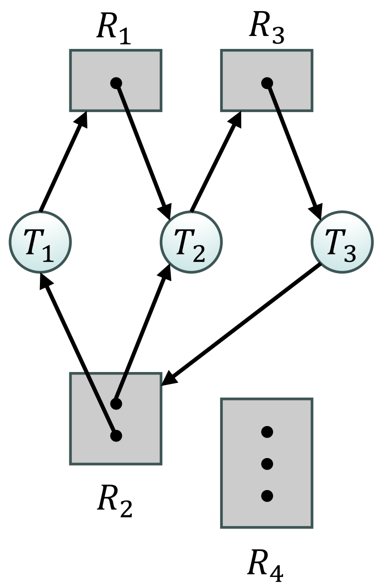
$\{T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2\}$

Threads T_1, T_2 and T_3 are deadlocked.

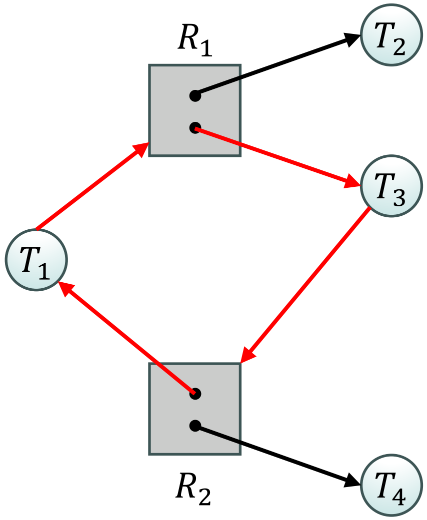
Resource-Allocation Graph Example



Simple Resource Allocation Graph

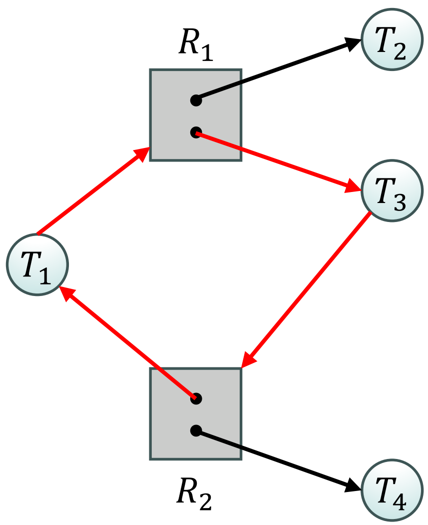


Allocation Graph with **Deadlock**



Allocation Graph with **Cycle**, but No Deadlock

Resource-Allocation Graph Example

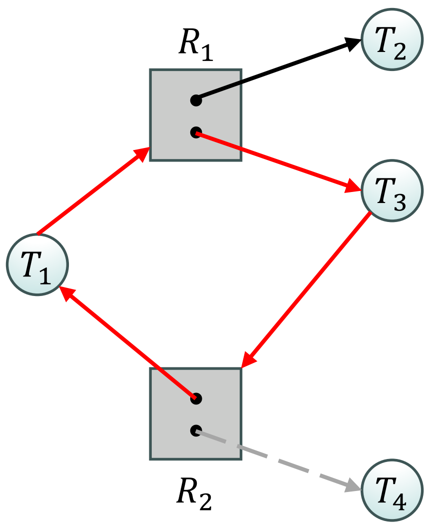


Allocation Graph with
Cycle, but No Deadlock

Circle:
$\{T_1 \to R_1 \to T_3 \to R_2 \to T_1\}$

...but no deadlock...

Resource-Allocation Graph Example



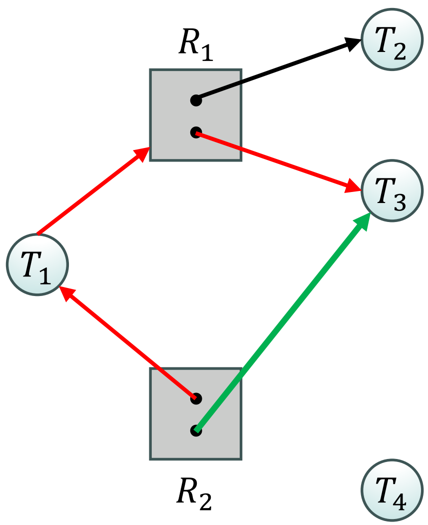
Allocation Graph with
Cycle, but No Deadlock

Circle:
$\{T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1\}$

...but no deadlock...

Because thread T_4 may **release** its instance of resource type R_2...

Resource-Allocation Graph Example



Allocation Graph with
Cycle, but No Deadlock

Circle:
$\{T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1\}$

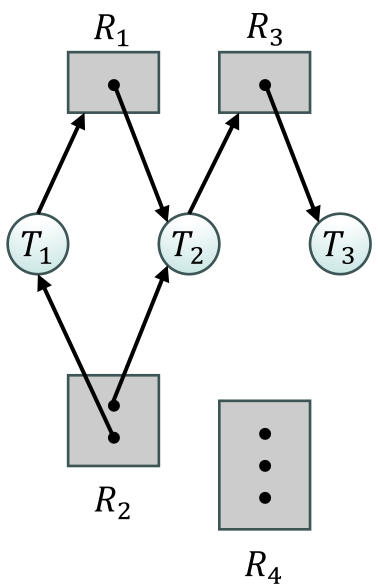
...but no deadlock...

Because thread T_4 may **release** its instance of resource type R_2...

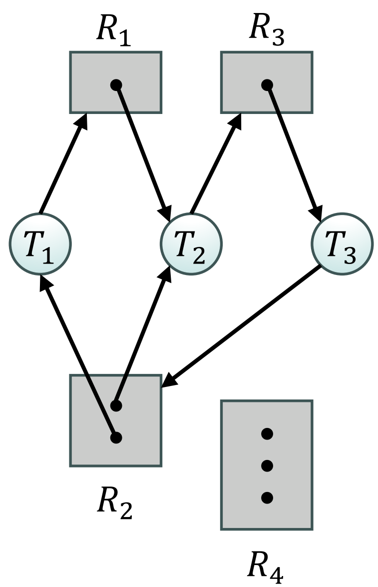
That instance of resource type R_2 can then be allocated to T_3, breaking the cycle.

Resource-Allocation Graph

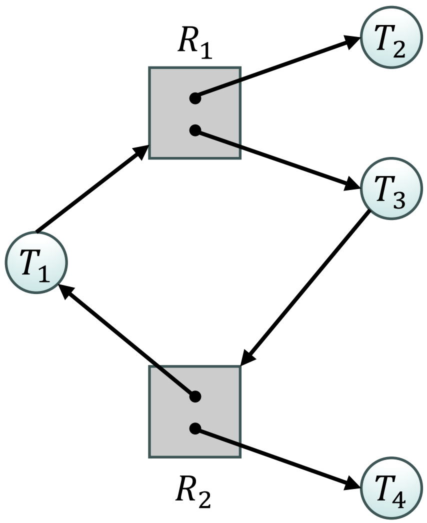
- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if **only one** instance per resource type, then deadlock.
  - if **serveral** instances per resource type, possibility of deadlock.



Simple Resource Allocation Graph



Allocation Graph with **Deadlock**

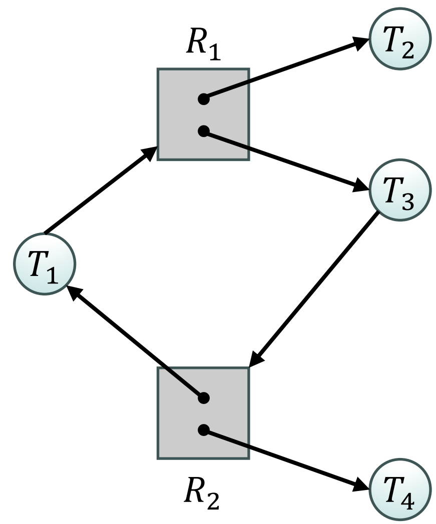


Allocation Graph with **Cycle**, but No Deadlock

Deadlock Detection Algorithm

- Data structures: *([x] represents row vector of size m)*
  - `[Avail]`: Number of available resources of each type
  - `[Alloc`$_i$`]`: Current resources held by thread i ($i \in [1, n]$)
  - `[Request`$_i$`]`: Current requests from thread i ($i \in [1, n]$)
- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



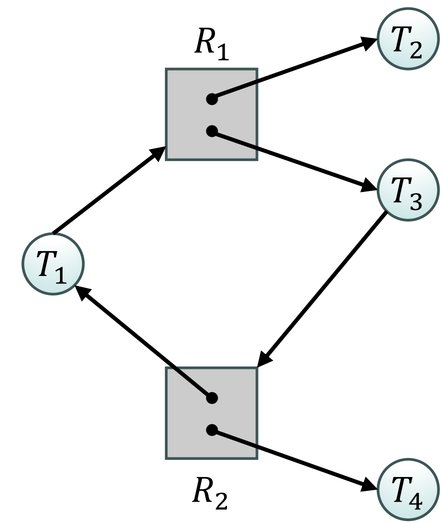
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	0	0	
T1	0	1	1	0			
T2	1	0	0	0			
T3	1	0	0	1			
T4	0	1	0	0			

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



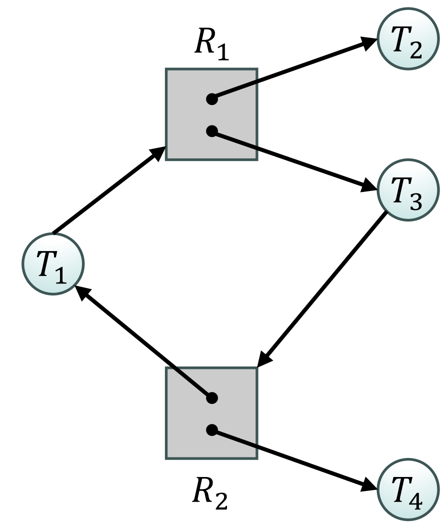
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	0	0	
T1	0	1	1	0			T1
T2	1	0	0	0			T2
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



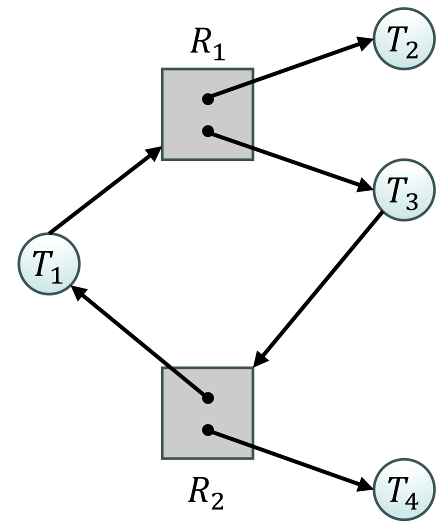
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	R2	R1	R2	0	0	
T1	0	1	1	0			T1
T2	1	0	0	0			T2
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



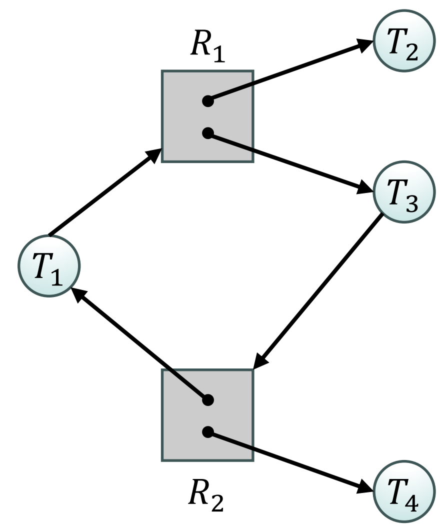
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	0	0	
T1	0	1	1	0			T1
T2	1	0	0	0			T2
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED ⇒ **deadlocked**

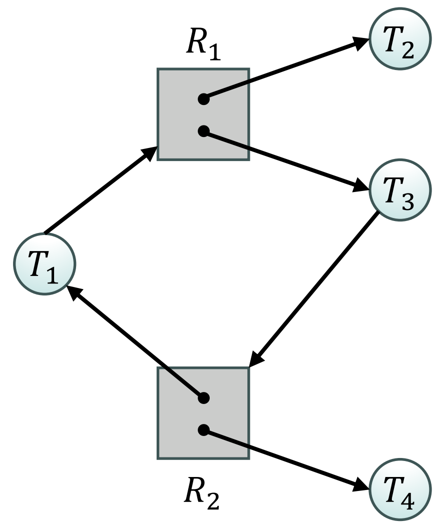
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	0	0	
T1	0	1	1	0			T1
T2	1	0	0	0			T2
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

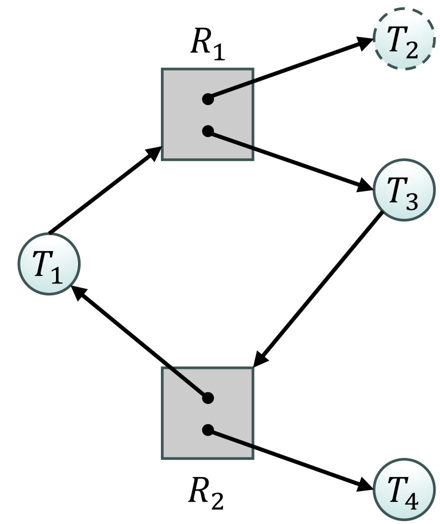


- Nodes left in UNFINISHED $\Rightarrow$ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	0	0	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own




```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

- Nodes left in UNFINISHED ⇒ **deadlocked**

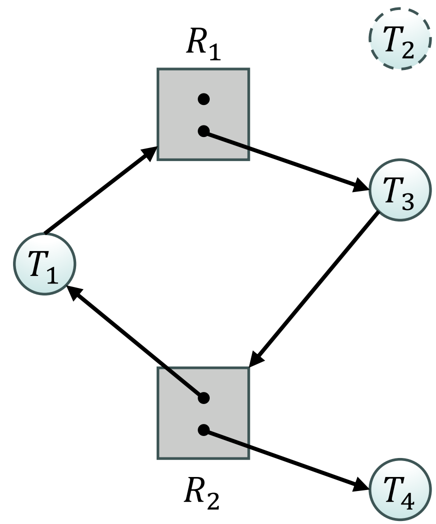
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	0	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



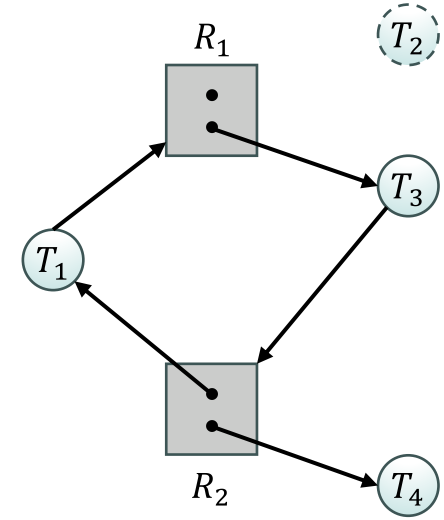
- Nodes left in UNFINISHED $\Rightarrow$ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	0	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED $\Rightarrow$ **deadlocked**

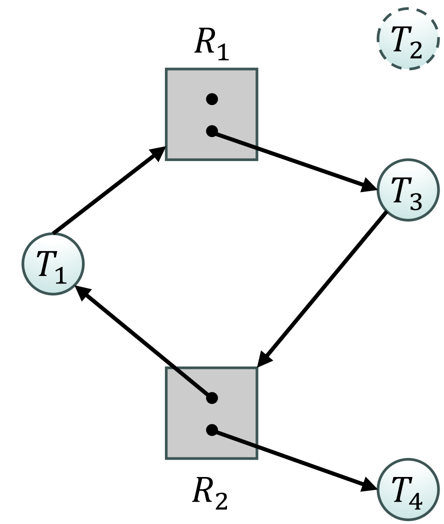
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	0	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
T4	0	1	0	0			T4

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



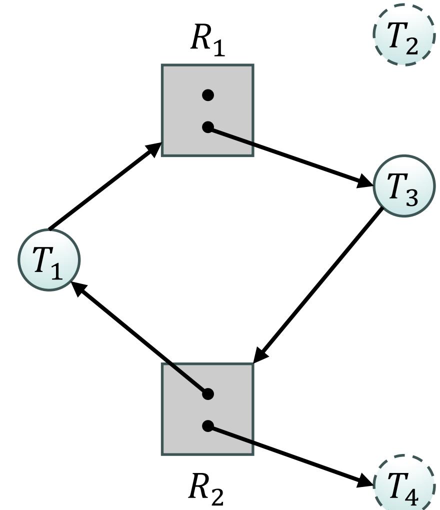
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	R2	R1	R2	1	0	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



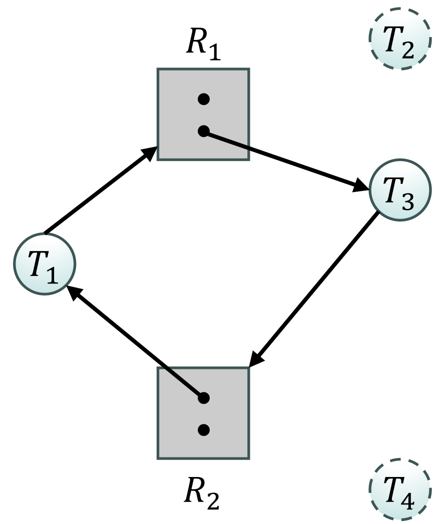
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	1	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



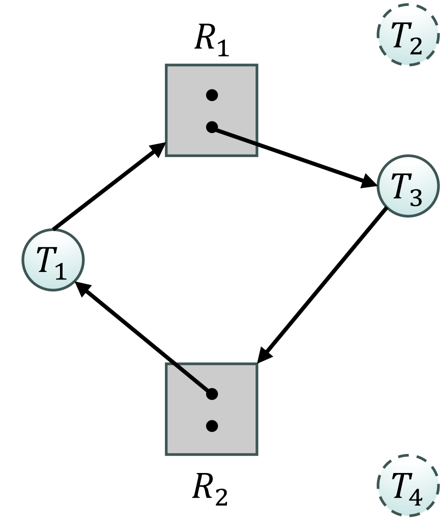
- Nodes left in UNFINISHED $\Rightarrow$ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	1	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



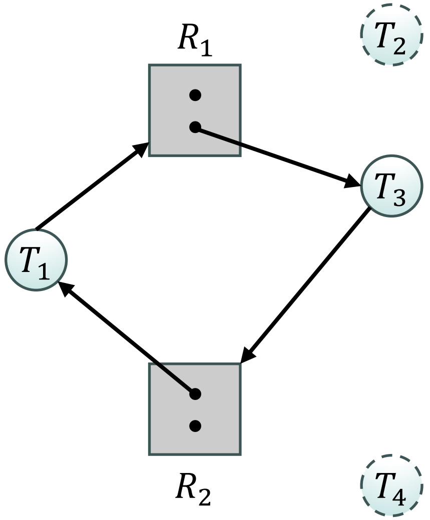
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	1	
T1	0	1	1	0			T1
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



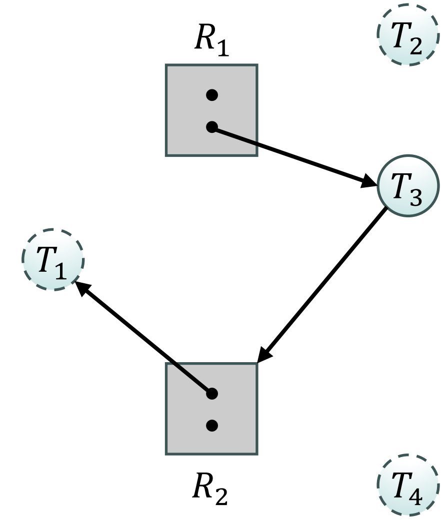
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	1	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



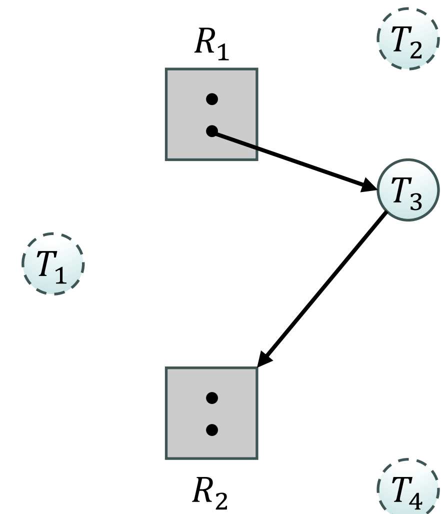
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



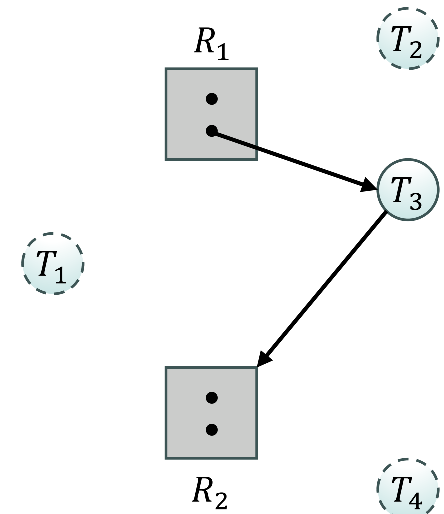
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
T3	1	0	0	1			T3
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



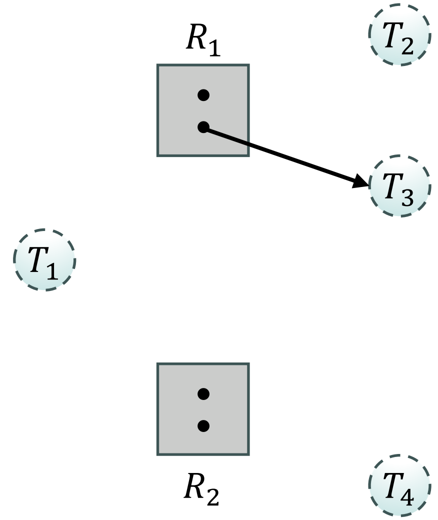
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	1	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
~~T3~~	~~1~~	~~0~~	~~0~~	~~1~~			~~T3~~
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED ⇒ **deadlocked**

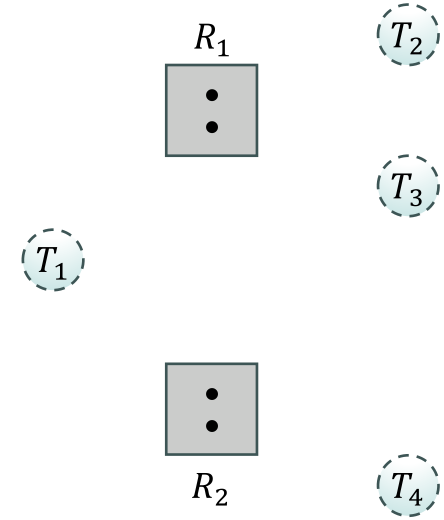
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	2	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
~~T3~~	~~1~~	~~0~~	~~0~~	~~1~~			~~T3~~
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



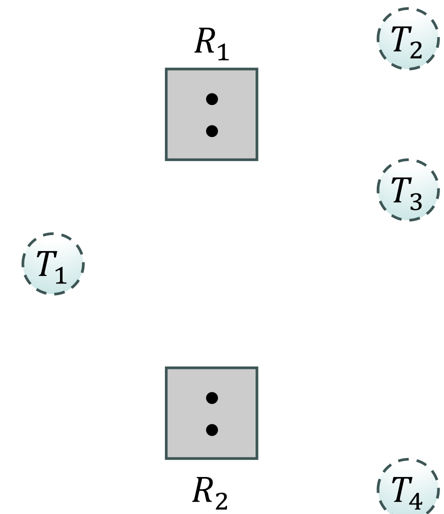
- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	2	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
~~T3~~	~~1~~	~~0~~	~~0~~	~~1~~			~~T3~~
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED ⇒ **deadlocked**

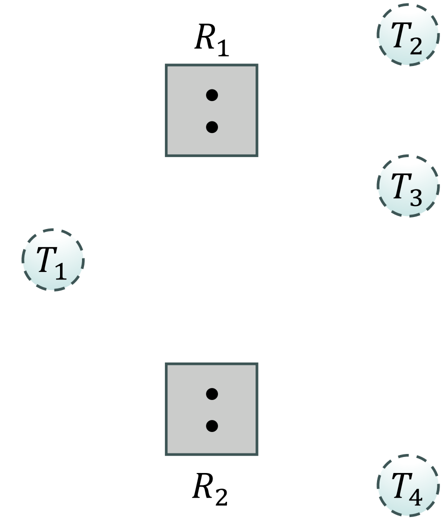
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	2	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
~~T3~~	~~1~~	~~0~~	~~0~~	~~1~~			~~T3~~
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED ⇒ **deadlocked**

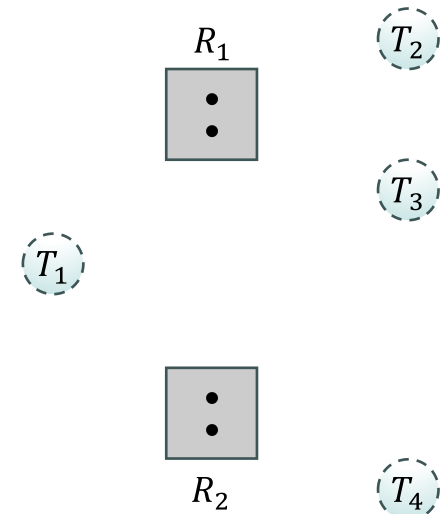
Deadlock Detection

Deadlock Detection Algorithm

	Alloc		Request		Avail		UNFINISHED
	R1	*R2*	*R1*	*R2*	2	2	
~~T1~~	~~0~~	~~1~~	~~1~~	~~0~~			~~T1~~
~~T2~~	~~1~~	~~0~~	~~0~~	~~0~~			~~T2~~
~~T3~~	~~1~~	~~0~~	~~0~~	~~1~~			~~T3~~
~~T4~~	~~0~~	~~1~~	~~0~~	~~0~~			~~T4~~

- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

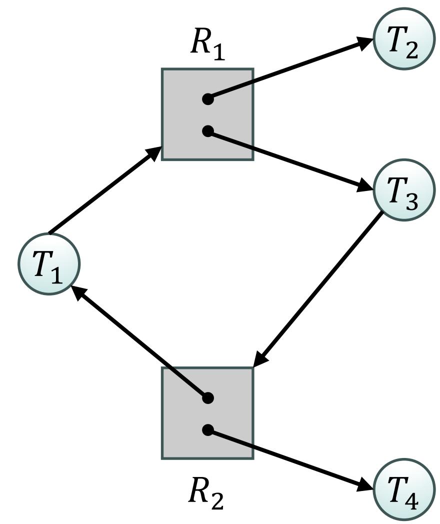


- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Detection Algorithm

- Data structures: *([x] represents row vector of size m)*
  - [Avail]: Number of available resources of each type
  - [Alloc$_i$]: Current resources held by thread i ($i \in [1, n]$)
  - [Request$_i$]: Current requests from thread i ($i \in [1, n]$)
- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



Time Complexity: $O(m \times n^2)$

- Nodes left in UNFINISHED $\Rightarrow$ **deadlocked**

- **Deadlock Detection Algorithm Usage**
  - When, and how often, to invoke the detection algorithm?
  - It depends on:
    - How **often** is a deadlock likely to occur?
    - How **many** threads will be affected by deadlock when it happens?
  - 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 threads "**caused**" the deadlock.

Deadlock Recovery

- **Recovery from Deadlock**
  - **Process and Thread Termination**
    - **Method #1:** Abort all deadlocked processes
    - **Method #2:** Abort one process at a time until the deadlock cycle is eliminated
    - In which order should we choose to abort? **Factors:**
      - Priority of the process
      - How long the process has computed and how much longer?
      - What resources has the process used?
      - How many more resources does the process need?
      - How many processes will need to be terminated?
  - **Resource Preemption**
    - **Selecting a victim:** minimize cost
    - **Rollback:** return to some safe state, restart the process for that state
    - **Starvation:**
      - same process always be picked as victim
      - include number of rollback in cost factor

Methods for Handling Deadlocks

- **Ignore** the problem of deadlock altogether and pretend that deadlocks never occur in the system.
  - **Ostrich** Algorithm (鸵鸟算法)
  - Used by most OSes (e.g., Linux and Windows)
    - Up to the developers to handle deadlocks.

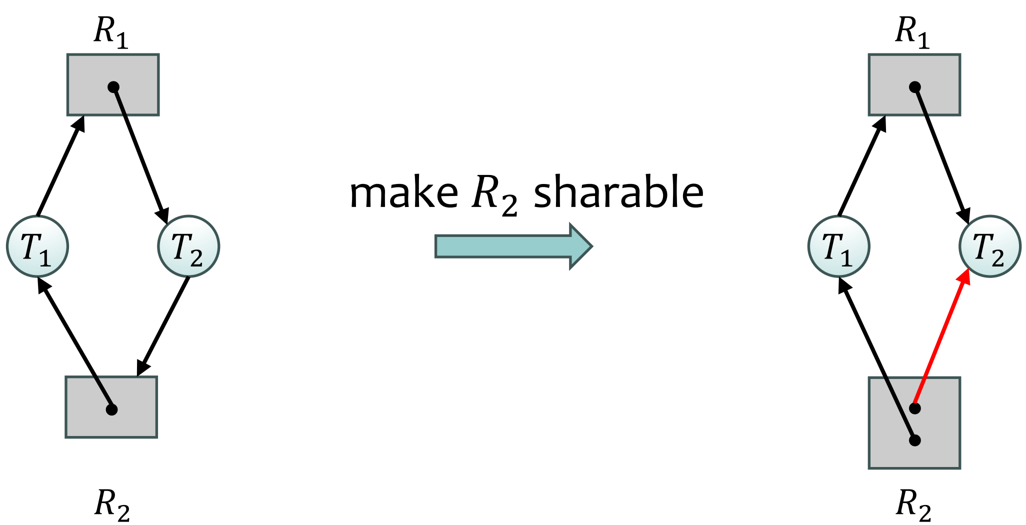
- **Ensure** that the system will never enter a deadlock state:
  - Deadlock **Prevention**
    - by constraining how requests for resources can be made.
    - simple and direct by structurally eliminating deadlocks
  - Deadlock **Avoidance**
    - requires that the OS be given additional info in advance concerning which resources a thread will request and use during its lifetime.
    - dynamic, sophisticated tracking and management of resources
- **Allow** the system to enter a deadlock state, **detect** it, and then **recover**.

Deadlock Prevention

- Invalidate one of the four **necessary** conditions for deadlock:
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait

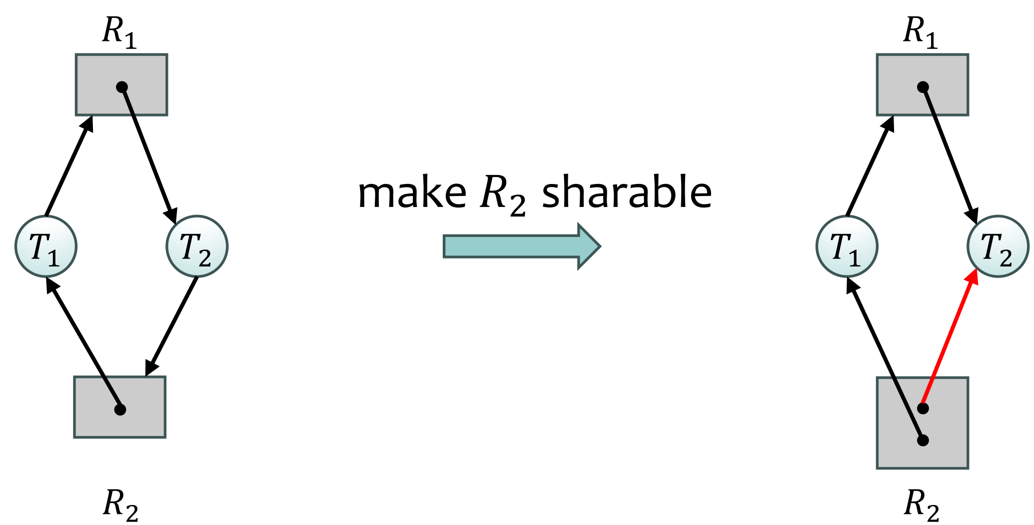
Deadlock Prevention

- Invalidate one of the four **necessary** conditions for deadlock:
- **Mutual Exclusion**
  - not required for sharable resources (e.g., read-only files)



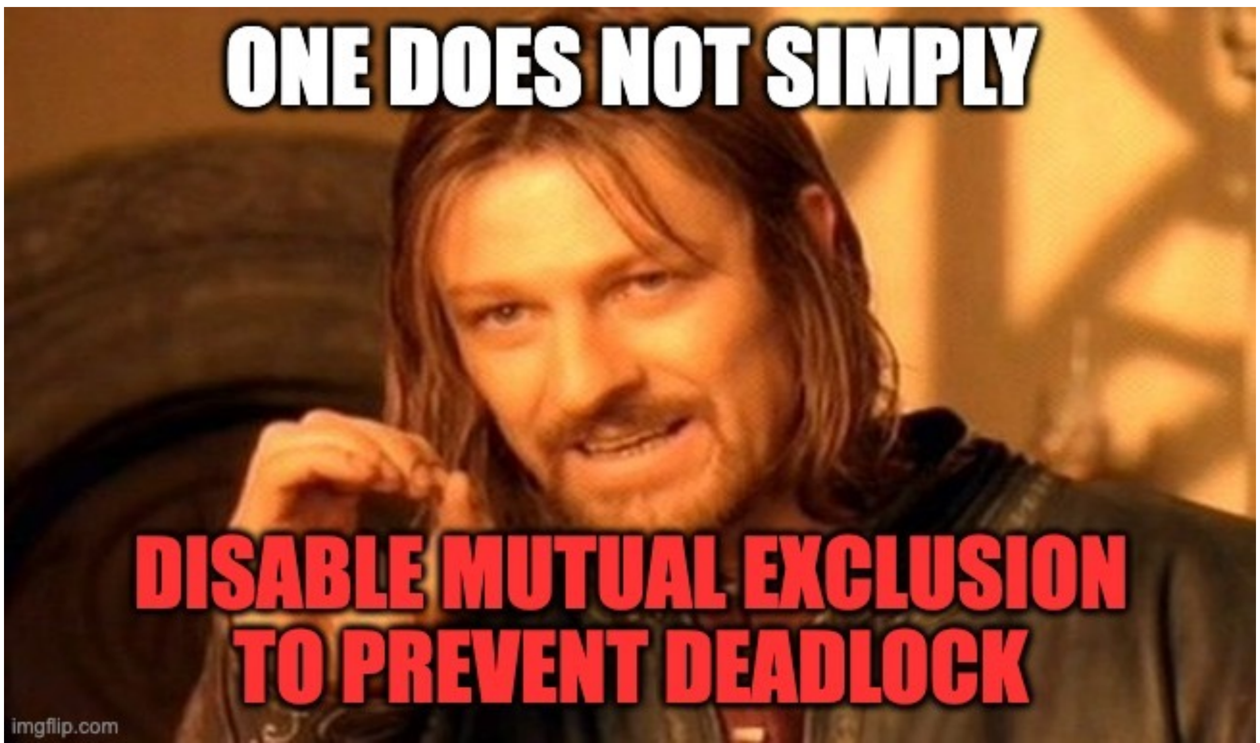
Deadlock Prevention

- Invalidate one of the four **necessary** conditions for deadlock:
- **Mutual Exclusion**
  - not required for sharable resources (e.g., read-only files)
  - In general, we cannot prevent deadlocks simply by denying mutual exclusion, because some resources (e.g., mutex locks) are **intrinsically** nonsharable.



Deadlock Prevention

- Invalidate one of the four **necessary** conditions for deadlock:
- **Mutual Exclusion**
  - not required for sharable resources (e.g., read-only files)
  - In general, we cannot prevent deadlocks simply by denying mutual exclusion, because some resources (e.g., mutex locks) are **intrinsically** nonsharable.



- **Deadlock Prevention**
  - Invalidate one of the four **necessary** conditions for deadlock:
  - **Hodl and Wait**
    - A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
  - ~~**Hodl and Wait**~~
    - must guarantee that whenever a thread requests a resource, it does not hold any other resources
    - **Method #1:** require threads to request and be allocated all its resources **before** it begins execution
    - **Method #2:** allow a thread to request resources **only** when it has none allocated to it.
    - **Disadvantages:**
      - Low resource utilization
      - Starvation is possible

- **Deadlock Prevention**
  - Invalidate one of the four **necessary** conditions for deadlock:
  - **No Preemption**
    - Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
  - ~~**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 thread is waiting
    - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Deadlock Prevention

- Invalidate one of the four **necessary** conditions for deadlock:
- **Circular Wait**
  - There exists a set $\{T_1, T_2, \dots, T_n\}$ of waiting threads such that

$$T_1 \rightarrow R_? \rightarrow T_2 \rightarrow R_? \rightarrow \cdots \rightarrow T_n \rightarrow R_? \rightarrow T_1$$

- ~~**Circular Wait**~~ **(most common)**
  - Impose a total **ordering** of all resource types, and require that each thread requests resources in **an order** of enumeration.
  - Simply assign each resource (i.e., mutex locks) a unique number.
  - Resources must be acquired in ascending order.
  - For example, if F(`mutex1`) = **1**, F(`mutex2`) = **5**, then:

Thread A:

```
acquire(&mutex1);
acquire(&mutex2);
...
release(&mutex2);
release(&mutex1);
```

Thread B:

```
acquire(&mutex2);
acquire(&mutex1);
...
release(&mutex1);
release(&mutex2);
```

Deadlock Avoidance

- **Idea:** When a thread requests a resource, the OS checks if it would result in deadlock
  - If not, it grants the resource right away.
  - If so, it waits for other threads to release resources.
- Does it work?
  - No!

Thread A:

```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

Thread B:

```
acquire(&y);
acquire(&x);
...
release(&x);
release(&y);
```

Deadlock Avoidance

- **Deadlock Avoidance**
  - **Idea:** When a thread requests a resource, the OS checks if it would result in deadlock
    - If not, it grants the resource right away.
    - If so, it waits for other threads to release resources.
  - Does it work?
    - No!

Thread A:

```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

Blocks…

Thread B:

```
acquire(&y);
acquire(&x);
...
release(&x);
release(&y);
```

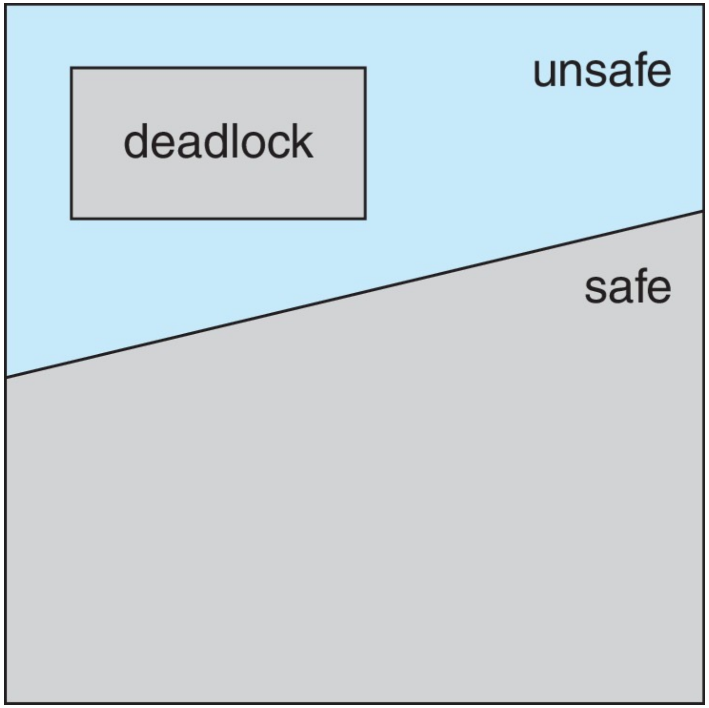
Wait?

But it's already too late…

Deadlock Avoidance: Three States

- **Safe** State
  - System can delay resource acquisition to prevent deadlock
- **Unsafe** State
  - No deadlock yet…
  - But threads can request resources in a pattern that ***unavoidably*** leads to deadlock
- **Deadlocked** State
  - There exists a deadlock in the system.
  - Also considered "unsafe"

Deadlock avoidance: prevent system from reaching an ***unsafe*** state



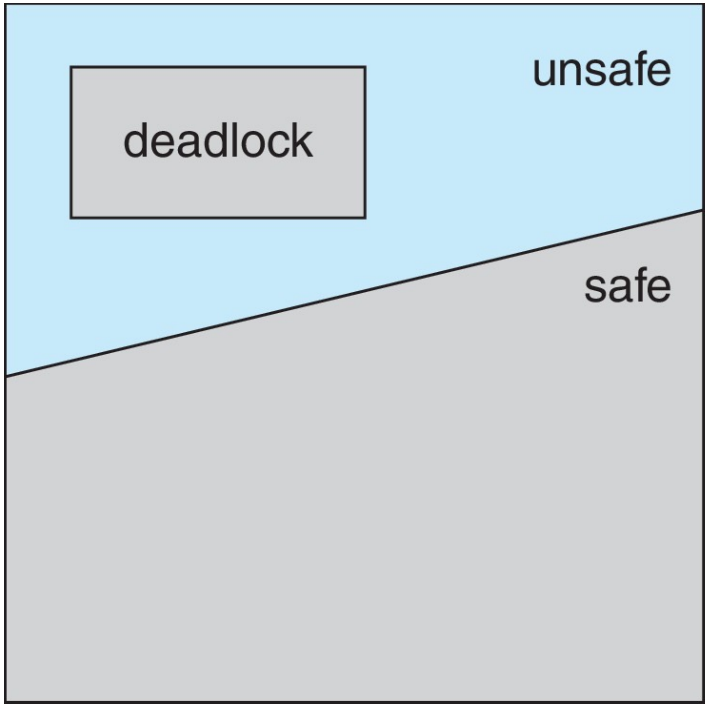
Deadlock Avoidance

Safe State

- When a thread 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 $< T_1, T_2, \dots, T_n >$ of **ALL** threads such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources + resources held by all the T_j, with $j < i$.
- In other words:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on...

Deadlock Avoidance

- **Basic Facts**
  - If a system is in **safe** state $\Rightarrow$ no deadlocks
  - If a system is in **unsafe** state $\Rightarrow$ possibility of deadlocks
  - **Deadlock Avoidance** $\Rightarrow$ ensure that a system will never enter an **unsafe** state



Deadlock Avoidance

- **Idea:** When a thread requests a resource, the OS checks if it would result in ~~deadlock~~ an unsafe state
  - If not, it grants the resource right away.
  - If so, it waits for other threads to release resources.

Thread A:

```
acquire(&x);
acquire(&y);
...
release(&y);
release(&x);
```

Thread B:

```
acquire(&y);
acquire(&x);
...
release(&x);
release(&y);
```

Wait until Thread A releases mutex x

Deadlock Avoidance

- Requires that the system has some additional presumed information available
  - Simpliest and most useful model requires that each thread 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

Banker's Algorithm for Avoiding Deadlocks

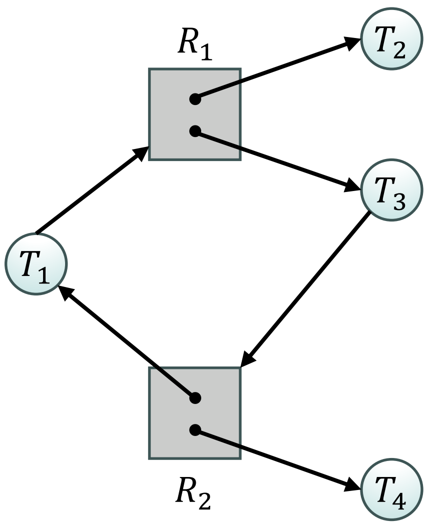
- The idea:
  - Declare maximum resource needs in advance
  - Allow particular threads to proceed if
    - `(available resources - #requested) >= max` remaining that might be needed by any thread

- **Banker's algorithm:**
  - Allocate resources dynamically
  - Evaluate each thread, and grant access if some ordring of threads is still deadlock-free afterwards
  - **Technique:** pretend each request is granted, then run **deadlock detection algorithm** by substituting:
    $([\mathtt{Request}_i] \mathtt{<=} [\mathtt{Avail}]) \Rightarrow ([\mathtt{Max}_i - \mathtt{Alloc}_i] \mathtt{<=} [\mathtt{Avail}])$
  - **Grant request** if the resulting state is deadlock-free (conservative)

Deadlock Detection Algorithm (Revisit)

- Data structures: *([x] represents row vector of size m)*
  - [Avail]: Number of available resources of each type
  - [Alloc$_i$]: Current resources held by thread i ($i \in [1, n]$)
  - [Request$_i$]: Current requests from thread i ($i \in [1, n]$)
- See if tasks can eventually terminate on their own

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```



- Nodes left in UNFINISHED ⇒ **deadlocked**

Deadlock Avoidance

Banker's Algorithm for Avoiding Deadlocks

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

e nee… proc… ces… be ne…

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Max_i - Alloc_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

- **Banker's algorithm:**
  - Allocate resources dynamically
  - Evaluate each thread, and grant access if some ordring of threads is still deadlock-free afterwards
  - **Technique:** pretend each request is granted, then run **deadlock detection algorithm** by substituting:

    $([Request_i] <= [Avail]) \Rightarrow ([Max_i - Alloc_i] <= [Avail])$
  - **Grant request** if the resulting state is deadlock-free (conservative)

Deadlock Avoidance

Banker's Algorithm for Avoiding Deadlocks

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Max_i - Alloc_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

- **Banker's algorithm:**
  - Allocate resources dynamically
  - Evaluate each thread, and grant access if some ordring of threads is still deadlock-free afterwards
  - **Technique:** pretend each request is granted, then run **deadlock detection algorithm** by substituting:

    `([Request_i] <= [Avail])` $\Rightarrow$ `([Max_i - Alloc_i] <= [Avail])`
  - **Grant request** if the resulting state is deadlock-free (conservative)
  - Keep system in a "**SAFE**" state: there exists a sequence $\{T_1, T_2, \ldots, T_n\}$ with T_1 **requesting all remaining resources**, then T_2, then T_3, etc.

Deadlock Avoidance

Banker's Algorithm for Avoiding Deadlocks

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Request_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

```
add all T[i] to UNFINISHED
do {
    done = true
    for each T[i] in UNFINISHED {
        if ([Max_i - Alloc_i] <= [Avail]) {
            remove T[i] from UNFINISHED
            [Avail] = [Avail] + [Alloc_i]
            done = false
        }
    }
} while (done == false)
```

- **Banker's algorithm:**
  - Allocate resources dynamically
  - Evaluate each thread, and grant access if some ordring of threads is still deadlock-free afterwards
  - **Technique:** pretend each request is granted, then run **deadlock detection algorithm** by substituting:
    `([Request_i] <= [Avail])` $\Rightarrow$ `([Need_i] <= [Avail])`
  - **Grant request** if the resulting state is deadlock-free (conservative)
  - Keep system in a "**SAFE**" state: there exists a sequence $\{T_1, T_2, \ldots, T_n\}$ with T_1 **requesting all remaining resources,** then T_2, then T_3, etc.

Banker's Algorithm Example

- 5 threads: $\{T_0, T_1, T_2, T_3, T_4\}$
- 3 resource types:
  - A (10 instances), B (5 instances) and C (7 instances)
- Snapshot at current state of the system:

	Alloc			Max			Avail		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T0	0	1	0	7	5	3	3	3	2
T1	2	0	0	3	2	2			
T2	3	0	2	9	0	2			
T3	2	1	1	2	2	2			
T4	0	0	2	4	3	3			

- Is the system in a **SAFE** state?

Banker's Algorithm Example

- 5 threads: $\{T_0, T_1, T_2, T_3, T_4\}$
- 3 resource types:
  - A (10 instances), B (5 instances) and C (7 instances)
- Snapshot at current state of the system:

	Alloc			Max			Avail		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T0	0	1	0	7	5	3	3	3	2
T1	2	0	0	3	2	2			
T2	3	0	2	9	0	2			
T3	2	1	1	2	2	2			
T4	0	0	2	4	3	3			

	Need		
	A	*B*	*C*
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

$Need_i = [Max_i - Alloc_i]$

- Is the system in a **SAFE** state?
  - Yes. Because there exists a sequence $< T_1, T_3, T_4, T_2, T_0 >$ that satisfies the safety requirement.

Summary

- Four (necessary) conditions for deadlocks
  - Mutual Exclusion
  - Hold and Wait
  - No Preemption
  - Circular Wait
- Techniques for addressing deadlocks
  - Deadlock **Prevention**:
    - write your code in a way that isn't prone to deadlock.
  - Deadlock **Detection** and **Recovery**:
    - Let it happen, and then figure out how to recover once detected.
  - Deadlock **Avoidance**:
    - Dynamically delay resource requests so deadlock doesn't happen
    - Banker's Algorithm provides an algorithmic way to do this
  - Deadlock **Denial**:
    - Ignore the possibility of deadlock (used by most OSes, e.g., Linux)
      - Ostrich Algorithm (鸵鸟算法)

Thank you!