Operating System Concepts

Lecture 22: Deadlock — Part 2

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

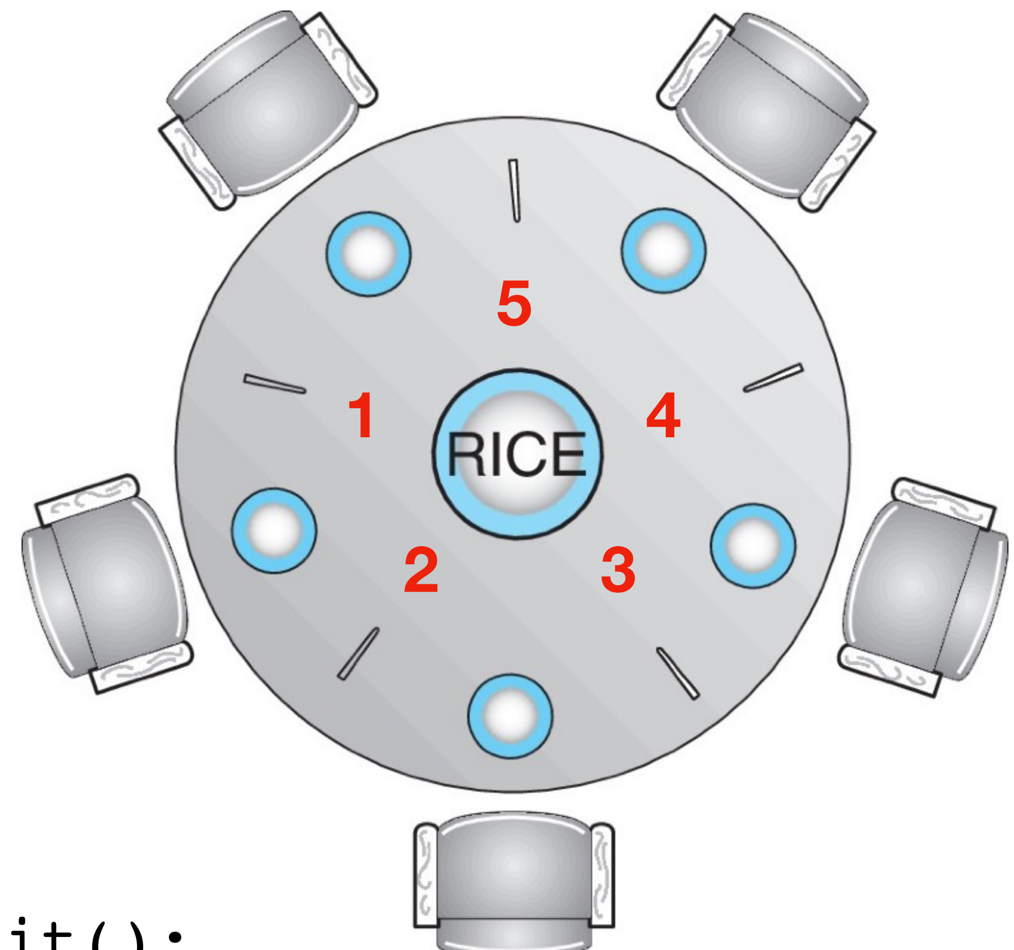
MWF 12:00-12:50 VVC 2 215

Imposing a total ordering of resources

- eliminate circular waiting by ordering all locks (or semaphores, or resources)
  - define a one-to-one function $F : R \rightarrow N$
  - all code grabs locks in a predefined order
  - if several instances of the same resource are needed, a single request is issued
- problems?
  - maintaining global order is difficult, especially in a large project
  - programmers may not follow the ordering…
  - the global order can force the programmer to grab a lock earlier than it would like, tying up a resource for too long

Imposing a total ordering of resources

- in the dinning philosopher problem
  - number chopsticks from 1 to 5
  - pick up the lower number chopstick fist
  - this will break the circular wait condition

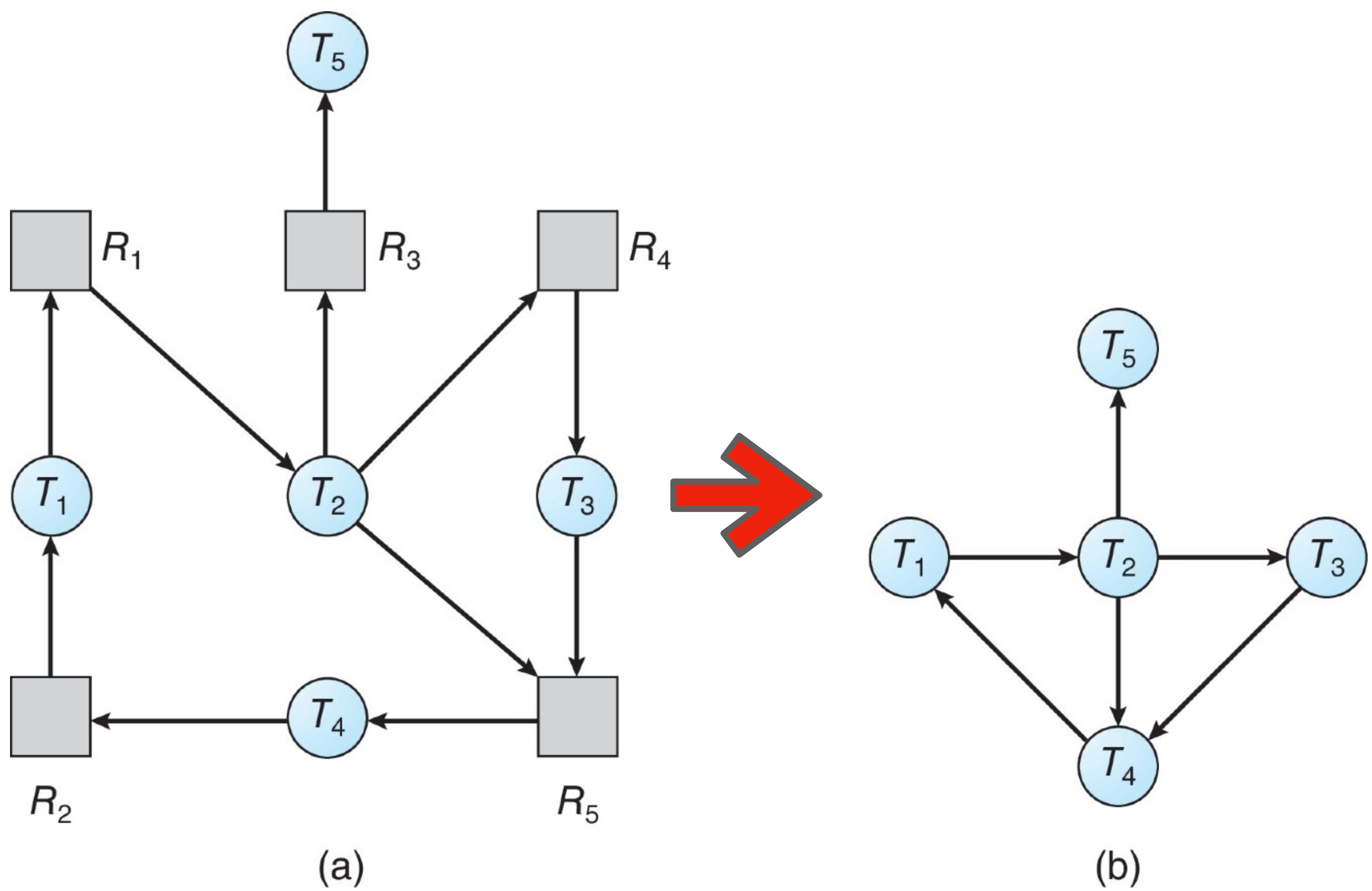



```
void philosopher(int i) {
  while(true) {
    chopstick[LowerNum(i)].wait();
    chopstick[HigherNum(i)].wait();
    ...
  }
}
```

Deadlock detection

if there is a single instance of each resource type

$\mathtt{T_i} \rightarrow \mathtt{R_j}$ and $\mathtt{R_j} \rightarrow \mathtt{T_k}$ yields $\mathtt{T_i} \rightarrow \mathtt{T_K}$



resource allocation graph **wait-for graph**

Deadlock detection (single instance of each resource type)

- scan the resource allocation graph for cycles
  - detecting cycles is $O(|E|+|V|)$
- when should we execute this algorithm?
  - whenever a resource request can't be filled
  - on a regular schedule (hourly or ...)
  - when CPU utilization drops below some threshold
  - just before granting a resource, check if granting it would lead to a cycle (it is deadlock avoidance rather than detection in this case)

Deadlock detection (single instance of each resource type)

- scan the resource allocation graph for cycles
  - detecting cycles is O(|E|+|V|)
- when should we execute this algorithm?
  - whenever a resource request can't be filled
  - on a regular schedule (hourly or ...)
  - when CPU utilization drops below some threshold
  - just before granting a resource, check if granting it would lead to a cycle (it is deadlock avoidance rather than detection in this case)
- what do Linux and Windows do?
  - ignore the problem altogether; leave it to the programmer/application
  - why? **because it's cheaper,** especially if deadlock occurs infrequently

Deadlock recovery

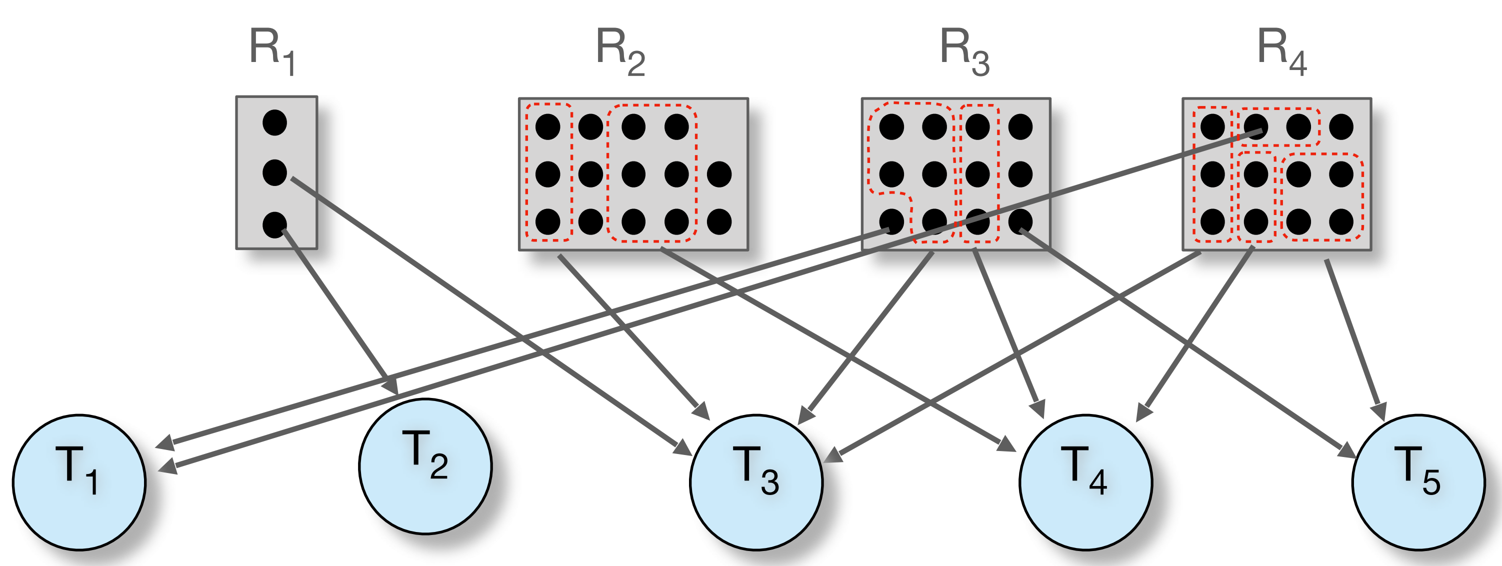
- inform the operator (manual recovery)

Deadlock recovery

- inform the operator (manual recovery)
- abort all deadlocked threads and reclaim their resources

Deadlock recovery

- inform the operator (manual recovery)
- abort all deadlocked threads and reclaim their resources
- abort one thread at a time and reclaim its resources; continue until all cycles in the resource allocation graph are eliminated
  - where to start? minimum cost is not well-defined? (a) low priority thread, (b) thread with most allocation of resources, (c) thread running for the longest time



Deadlock recovery

- inform the operator (manual recovery)
- abort all deadlocked threads and reclaim their resources
- abort one thread at a time and reclaim its resources; continue until all cycles in the resource allocation graph are eliminated
  - where to start? minimum cost is not well-defined? (a) low priority thread, (b) thread with most allocation of resources, (c) thread running for the longest time
- preempt some resources from one of multiple deadlocked threads?
  - which resources?

Deadlock recovery

- inform the operator (manual recovery)
- abort all deadlocked threads and reclaim their resources
- abort one thread at a time and reclaim its resources; continue until all cycles in the resource allocation graph are eliminated
  - where to start? minimum cost is not well-defined? (a) low priority thread, (b) thread with most allocation of resources, (c) thread running for the longest time
- preempt some resources from one of multiple deadlocked threads?
  - which resources?
- caveat: make sure that the system is in consistent state

Today's class

- Dealing with deadlock
  - Deadlock avoidance
  - Deadlock detection with several instances of a resource type

Reserve resources instead of obtaining them

- obtaining all resources at startup reduces resource utilization

Reserve resources instead of obtaining them

- obtaining all resources at startup reduces resource utilization
- a better approach is to state maximum resources that will be needed at startup and allocate resources dynamically when needed
  - assumptions: (a) the maximum resource need is known a priori; (b) when a thread gets all its resources it must return them in a finite amount of time

Reserve resources instead of obtaining them

- obtaining all resources at startup reduces resource utilization
- a better approach is to state maximum resources that will be needed at startup and allocate resources dynamically when needed
  - assumptions: (a) the maximum resource need is known a priori; (b) when a thread gets all its resources it must return them in a finite amount of time
  - when is a request granted?
    - the sum of the stated maximum resources exceeds the total available resources? **No, threads do not request resources at once**

Reserve resources instead of obtaining them

- obtaining all resources at startup reduces resource utilization
- a better approach is to state maximum resources that will be needed at startup and allocate resources dynamically when needed
  - <u>assumptions:</u> (a) the maximum resource need is known a priori; (b) when a thread gets all its resources it must return them in a finite amount of time
  - when is a request granted?
    - the sum of the stated maximum resources exceeds the total available resources? **No, threads do not request resources at once**
    - some sequential ordering of threads is deadlock free? **Yes!**

Reserve resources instead of obtaining them

- obtaining all resources at startup reduces resource utilization
- a better approach is to state maximum resources that will be needed at startup and allocate resources dynamically when needed
  - <u>assumptions:</u> (a) the maximum resource need is known a priori; (b) when a thread gets all its resources it must return them in a finite amount of time
  - when is a request granted?
    - the sum of the stated maximum resources exceeds the total available resources? **No, threads do not request resources at once**
    - some sequential ordering of threads is deadlock free? **Yes!**
  - there just needs to be some way for all threads to finish

Deadlock avoidance with resource reservation

- threads provide advance information about the maximum resources they may need during execution

Deadlock avoidance with resource reservation

- threads provide advance information about the maximum resources they may need during execution
- define a sequence of threads $\{t_1, ..., t_n\}$ as **safe** if for each t_i, the resources that t_i will eventually request can be satisfied by the currently available resources plus the resources held by all t_j where $j < i$
  - hence a safe state is a state in which there is at least one safe sequence for the threads

Deadlock avoidance with resource reservation

- threads provide advance information about the maximum resources they may need during execution
- define a sequence of threads $\{t_1, ..., t_n\}$ as **safe** if for each t_i, the resources that t_i will eventually request can be satisfied by the currently available resources plus the resources held by all t_j where $j < i$
  - hence a safe state is a state in which there is at least one safe sequence for the threads
- an unsafe state is not equivalent to deadlock, it **may** lead to deadlock
  - since some threads might not actually use the maximum resources they have declared

Deadlock avoidance with resource reservation

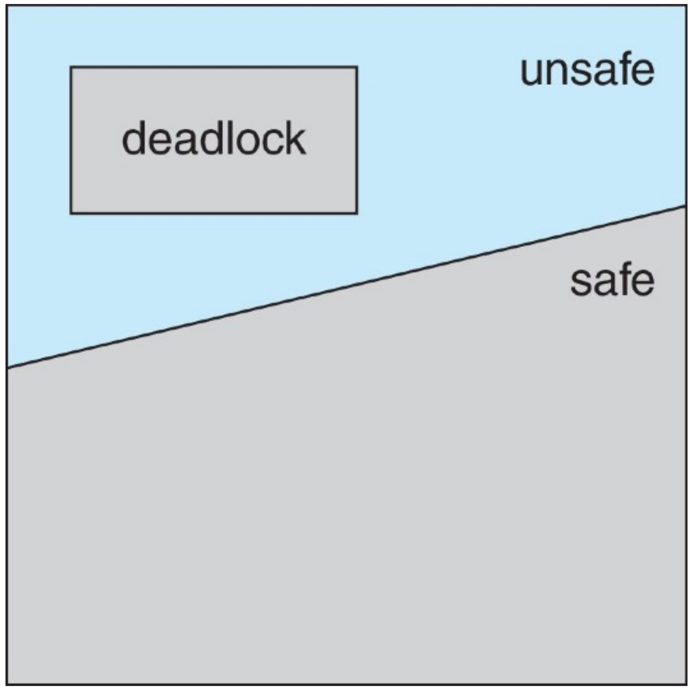
- threads provide advance information about the maximum resources they may need during execution
- define a sequence of threads $\{t_1, \ldots, t_n\}$ as **safe** if for each t_i, the resources that t_i will eventually request can be satisfied by the currently available resources plus the resources held by all t_j where $j < i$
  - hence a safe state is a state in which there is at least one safe sequence for the threads
- an unsafe state is not equivalent to deadlock, it **may** lead to deadlock
  - since some threads might not actually use the maximum resources they have declared
- to summarize
  - a resource is given to a thread if the new state is safe
  - otherwise, the thread must wait **even if the resource is currently available**

Deadlock avoidance with resource reservation

- threads provide advance information about the maximum resources they may need during execution
- define a sequence of threads $\{t_1, \ldots, t_n\}$ as **safe** if for each t_i, the resources that t_i will eventually request can be satisfied by the currently available resources plus the resources held by all t_j where $j < i$
  - hence a safe state is a state in which there is <u>at least one</u> safe sequence for the threads
- an unsafe state is not equivalent to deadlock, it **may** lead to deadlock
  - since some threads might not actually use the maximum resources they have declared
- to summarize
  - a resource is given to a thread if the new state is safe
  - otherwise, the thread must wait **even if the resource is currently available**
- this algorithm ensures no circular-wait condition exists

Definitions

- safe state: for any possible sequence of future resource requests, it is possible to eventually grant all requests
  - if a system is in safe state, then no deadlock
- unsafe state: some sequence of resource requests can result in deadlock
  - if a system is in unsafe state, possibility of deadlock
- doomed state: all possible computations lead to deadlock
- deadlock avoidance: ensure that the system never enters an unsafe state



Example

- threads t_1, t_2, and t_3 are competing for 12 tape drives; 11 drives are currently allocated to the threads, only 1 tap drive is available at this point
- the current state is **safe**
  - there exists a safe sequence, $\{t_1, t_2, t_3\}$, where all threads may obtain their maximum number of resources

Example

- threads t_1, t_2, and t_3 are competing for 12 tape drives; 11 drives are currently allocated to the threads, only 1 tap drive is available at this point
- the current state is **safe**
  - there exists a safe sequence, $\{t_1, t_2, t_3\}$, where all threads may obtain their maximum number of resources

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	4	8

Example

- if t_3 requests one more drive, then it must wait since allocating the drive would lead to an unsafe state
- granting this request will result in having no resource available, hence none of the threads can finish their tasks

Example

- if t_3 requests one more drive, then it must wait since allocating the drive would lead to an unsafe state
- granting this request will result in having no resource available, hence none of the threads can finish their tasks

	max need	in use	could want
t_1	4	3	1
t_2	8	4	4
t_3	12	5	7

Banker’s algorithm

- each thread must a priori claim maximum use
  - grant a request if and only if it results in a safe state
  - otherwise the thread has to wait

Data structures for the Banker's algorithm

Let `n` be the number of threads and `m` be the number of resource types

- `available` is a vector of length `m`
  - if `available[j]=k` there are `k` instances of resource type R_j available
- `max` is an `n` by `m` matrix
  - if `max[i][j]=k` then thread T_i may request at most `k` instances of resource type R_j
- `allocation` is an `n` by `m` matrix
  - if `allocation[i][j]=k` then thread T_i is currently allocated `k` instances of R_j
- `need` is an `n` by `m` matrix
  - if `need[i][j]=k` then T_i may need `k` more instances of R_j to complete its task
  - thus `need[i][j] = max[i][j] – allocation[i][j]`

these data structures vary over time in size and value

Resource-request algorithm

let `request[i]` be the request vector of thread T_i
if `request[i][j]=k` then T_i wants `k` instances of resource type R_j

1. if `request[i]≤need[i]` go to step 2, otherwise, raise error condition since T_i has exceeded its maximum claim
2. if `request[i]≤available`, go to step 3, otherwise T_i must wait since resources are not available
3. pretend to allocate requested resources to T_i by modifying the state as follows:

```
available-=request[i]
allocation[i]+=request[i]
need[i]-=request[i]
```

run the **safety check algorithm**. If safe, allocate resources to T_i
otherwise, T_i must wait and the old resource-allocation state is restored

Safety check algorithm

Let `work` and `finish` be vectors of length `m` and `n`, respectively.

1. Initialize:

```
work = available
finish[i] = false  for  i={0, 1, …, n-1}
```

2. Find an index `i` such that both:
   (a) `finish[i] = false`
   (b) `need[i] ≤ work`
   if no such `i` exists, go to step 4

3. Set

```
work = work + allocation[i]
finish[i] = true
```

and go to step 2

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

Example

- 5 threads (T_0,T_1,T_2,T_3,T_4), 3 resource types (R_0, R_1, R_2), W_0=10, W_1=5, W_2=7

available: (3,3,2)

	allocation (R_0, R_1, R_2)	max (R_0, R_1, R_2)	need (R_0, R_1, R_2)
t_0	(0,1,0)	(7,5,3)	(7,4,3)
t_1	(2,0,0)	(3,2,2)	(1,2,2)
t_2	(3,0,2)	(9,0,2)	(6,0,0)
t_3	(2,1,1)	(2,2,2)	(0,1,1)
t_4	(0,0,2)	(4,3,3)	(4,3,1)

- the system is in a safe state since the sequence {T_1,T_3,T_4,T_2,T_0} is a safe sequence

Example: can request for (3,3,0) by T_4 be granted?

- Pretend that the request is granted

available: (0,0,2)

	allocation (R_0, R_1, R_2)	max (R_0, R_1, R_2)	need (R_0, R_1, R_2)
t_0	(0,1,0)	(7,5,3)	(7,4,3)
t_1	(2,0,0)	(3,2,2)	(1,2,2)
t_2	(3,0,2)	(9,0,2)	(6,0,0)
t_3	(2,1,1)	(2,2,2)	(0,1,1)
t_4	(3,3,2)	(4,3,3)	(1,0,1)

Example: can request for (3,3,0) by T_4 be granted?

- Pretend that the request is granted

available: (0,0,2)

	allocation (R_0, R_1, R_2)	max (R_0, R_1, R_2)	need (R_0, R_1, R_2)
t_0	(0,1,0)	(7,5,3)	(7,4,3)
t_1	(2,0,0)	(3,2,2)	(1,2,2)
t_2	(3,0,2)	(9,0,2)	(6,0,0)
t_3	(2,1,1)	(2,2,2)	(0,1,1)
t_4	(3,3,2)	(4,3,3)	(1,0,1)

- the system is still in an unsafe state
- hence T_4 must wait and the saved state is restored

Example: can request for (1,0,2) by T_1 be granted?

- Pretend that the request is granted

available: (2,3,0)

	allocation (R_0, R_1, R_2)	max (R_0, R_1, R_2)	need (R_0, R_1, R_2)
t_0	(0,1,0)	(7,5,3)	(7,4,3)
t_1	(3,0,2)	(3,2,2)	(0,2,0)
t_2	(3,0,2)	(9,0,2)	(6,0,0)
t_3	(2,1,1)	(2,2,2)	(0,1,1)
t_4	(0,0,2)	(4,3,3)	(4,3,1)

Example: can request for (1,0,2) by T_1 be granted?

- Pretend that the request is granted

available: (2,3,0)

	allocation (R_0, R_1, R_2)	max (R_0, R_1, R_2)	need (R_0, R_1, R_2)
t_0	(0,1,0)	(7,5,3)	(7,4,3)
t_1	(3,0,2)	(3,2,2)	(0,2,0)
t_2	(3,0,2)	(9,0,2)	(6,0,0)
t_3	(2,1,1)	(2,2,2)	(0,1,1)
t_4	(0,0,2)	(4,3,3)	(4,3,1)

- the system is still in a safe state since the sequence $\{T_1,T_3,T_4,T_2,T_0\}$ is a safe sequence
- so the request will be granted

Problem with the banker algorithm

- the banker's algorithm is slow
  - it's too slow to run on every allocation

Problem with the banker algorithm

- the banker's algorithm is slow
  - it's too slow to run on every allocation
- what else can we do?
  - combine the three basic approaches:
    - prevention: ensure that the system will never enter a deadlock state
    - avoidance: ensure that the system will never enter a deadlock state
    - detection: let the system enter a deadlock state and then recover
  - partition resources into hierarchically ordered classes
    - use the most appropriate technique for handling deadlocks within each class

Homework

modify the safety check algorithm such that it can be used for detecting deadlock when there are <u>multiple instances of each resource</u>