Operating System Concepts

Lecture 21: Deadlock

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Previous class

- discussed variants of the Readers-Writers problem
  - let a writer enter its critical section as soon as possible

The Readers-Writers Problem

```
class ReadWrite {
  public:
  void Read();
  void Write();
  private:
  int readers;      // number of readers (shared between readers)
  Semaphore mutex; // controls access to readers
  Semaphore wrt;   // controls entry to first writer or reader
}

ReadWrite::ReadWrite {
  readers = 0;
  mutex.value = 1;
  wrt.value = 1;
}
```

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

```
void ReadWrite::Write(){
  write_mutex.Wait(); // ensure mutual exclusion
  writers += 1;       // another pending writer
  if(writers == 1)    // first writer blocks readers
    read_block.Wait();
  write_mutex.Signal();

  write_block.Wait(); // ensure mutual exclusion
  <perform write>
  write_block.Signal();

  write_mutex.Wait(); // ensure mutual exclusion
  writers -= 1;       // writer done
  if(writers == 0)    // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

first critical section

second critical section

```
void ReadWrite::Write(){
  write_mutex.Wait(); // ensure mutual exclusion
  writers += 1;       // another pending writer
  if(writers == 1)    // first writer blocks readers
    read_block.Wait();
  write_mutex.Signal();

  write_block.Wait(); // ensure mutual exclusion
  <perform write>
  write_block.Signal();

  write_mutex.Wait(); // ensure mutual exclusion
  writers -= 1;       // writer done
  if(writers == 0)    // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

Favouring writers

to block readers as soon as a writer enters, we need to keep track of the number of writers and use a writer mutex lock to update this number

```
void ReadWrite::Write(){
  write_mutex.Wait(); // ensure mutual exclusion
  writers += 1;       // another pending writer
  if(writers == 1)    // first writer blocks readers
    read_block.Wait();// wait if there is a reader
  write_mutex.Signal();

  write_block.Wait(); // ensure mutual exclusion
  <perform write>
  write_block.Signal();

  write_mutex.Wait(); // ensure mutual exclusion
  writers -= 1;       // writer done
  if(writers == 0)    // enable readers
    read_block.Signal();
  write_mutex.Signal();
}
```

Favouring writers

```
void ReadWrite::Read(){
  write_pending.Wait(); // at most one reader will enter
                        // before a pending write
  read_block.Wait();    // write in progress; wait
  read_mutex.Wait();    // ensure mutual exclusion
  readers += 1;         // another reader
  if(readers == 1)      // synchronize with writers
    write_block.Wait();
  read_mutex.Signal();
  read_block.Signal();
  write_pending.Signal();

  <perform read>

  read_mutex.Wait();    // ensure mutual exclusion
  readers -= 1;         // reader done
  if(readers == 0)      // enable writers
    write_block.Signal();
  read_mutex.Signal();
}
```

Today’s class

- Definition
  - conditions leading to deadlock
- Dealing with deadlock
  - Deadlock prevention
  - Deadlock detection
    - one instance of each resource type
  - Deadlock recovery

Motivating example

- two producers share a buffer but use a different protocol for accessing the buffer

Thread 1

```
Produce() {
  lock1.acquire();
  lock2.acquire();
  ...
}
```

Thread 2

```
Produce2() {
  lock2.acquire();
  lock1.acquire();
  ...
}
```

Motivating example

- two producers share a buffer but use a different protocol for accessing the buffer

Thread 1

context switch

```
Produce() {
  lock1.acquire();
  lock2.acquire();
  ...
}
```

Thread 2

```
Produce2() {
  lock2.acquire();
  lock1.acquire();
  ...
}
```

Motivating example

- two producers share a buffer but use a different protocol for accessing the buffer

Thread 1

```
Produce() {
  lock1.acquire();
  lock2.acquire();
  ...
}
```

Thread 2

```
Produce2() {
  lock2.acquire();
  lock1.acquire();
  ...
}
```

context switch

wait queue of lock2

thread 1

wait queue of lock1

thread 2

First attempt

```
#define N  5              // number of philosophers
#define L  i              // index of i's left neighbour
#define R  (i+1)%N        // index of i's right neighbour

semaphore chopstick[N]; // one semaphore per chopstick

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    chopstick[L].wait();
    chopstick[R].wait();
    /* eat for awhile */
    chopstick[L].signal();
    chopstick[R].signal();
    /* think for awhile */
  }
}
```

First attempt

```
#define N  5              // number of philosophers
#define L  i              // index of i's left neighbour
#define R  (i+1)%N        // index of i's right neighbour

semaphore chopstick[N]; // one semaphore per chopstick

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    chopstick[L].wait();
    chopstick[R].wait();
    /* eat for awhile */
    chopstick[L].signal();
    chopstick[R].signal();
    /* think for awhile */
  }
}
```

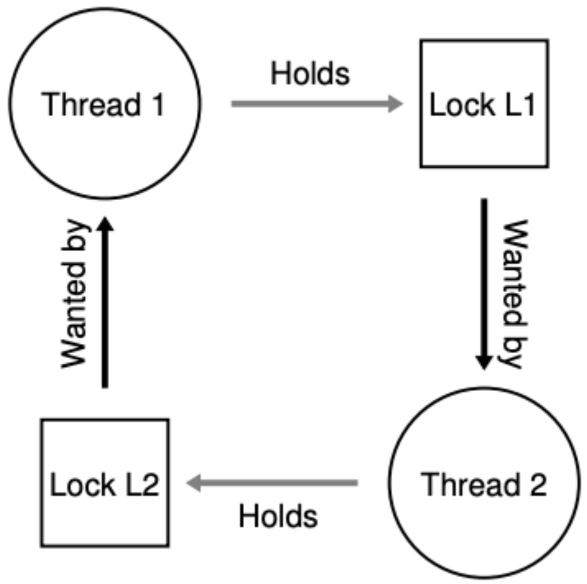
What's the problem with this solution? **could create a deadlock**

First attempt

```
#define N  5             // number of philosophers
#define L  i             // index of i's left neighbour
#define R  (i+1)%N       // index of i's right neighbour

semaphore chopstick[N]; // one semaphore per chopstick

void philosopher(int i) { // i is the index of the philosopher
  while(true) {
    chopstick[L].wait();
    chopstick[R].wait();
    /* eat for awhile */
    chopstick[L].signal();
    chopstick[R].signal();
    /* think for awhile */
  }
}
```



What's the problem with this solution? **could create a deadlock**

System model

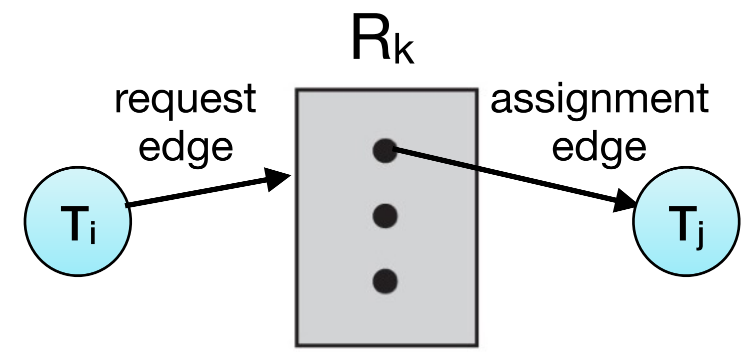
- the system has $\mathtt{m}$ resource types: $\mathtt{R_1, \ldots, R_M}$
  - e.g., CPU cycles, disk space, memory, I/O devices, lock
- there are $\mathtt{W_i}$ instances from resource type $\mathtt{R_i}$
  - assume instances are interchangeable
- to utilize a resource, each process/thread must
  a. request: it is met instantly if the resource is available, otherwise the requestor has to wait
  b. use: can operate on that resource
  c. release: has to release the resource when finished

Deadlock

- Definition: a set of processes (or threads) is deadlocked when every member of the set is waiting for an **event** that can only be generated by another member of the set
  - e.g., releasing a system resource
- Example: a computer has 2 tape drives
  - there are two processes, P1 and P2; each process needs both tape drives to do some operation
  - each holds one tape drive and waits for the other process to release the other tape drive!
  - implies starvation because none of these processes can finish execution

image from: https://www.javahelps.com/2015/06/thread-deadlock.html

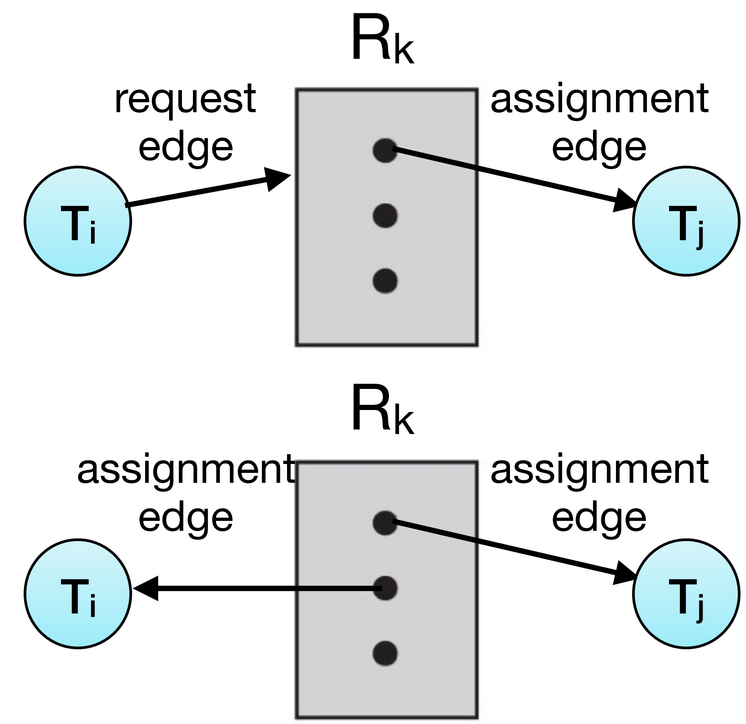
Resource-allocation graph



- the set of all vertices: $\{R_1, \ldots, R_M\} \cup \{T_1, \ldots, T_N\}$
  - the set of resource types: $\{R_1, \ldots, R_M\}$
    - resource instances are represented by dots
  - the set of threads in the system: $\{T_1, \ldots, T_N\}$
- a directed edge from a thread to a resource indicates a request: $T_i \rightarrow R_k$
- a directed edge from a resource instance to a thread indicates an assignment: $R_k \rightarrow T_j$

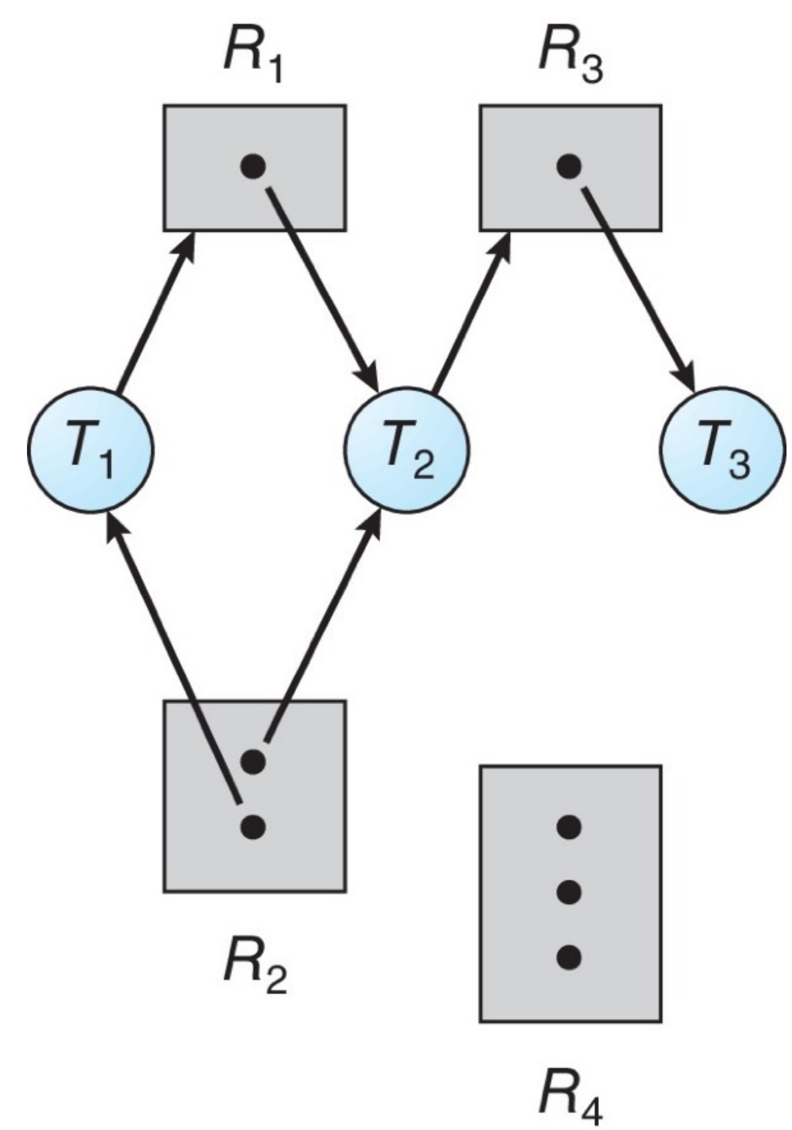
Resource-allocation graph

- the set of all vertices: $\{R_1, \ldots, R_M\} \cup \{T_1, \ldots, T_N\}$
  - the set of resource types: $\{R_1, \ldots, R_M\}$
    - resource instances are represented by dots
  - the set of threads in the system: $\{T_1, \ldots, T_N\}$
- a directed edge from a thread to a resource indicates a request: $T_i \rightarrow R_k$
- a directed edge from a resource instance to a thread indicates an assignment: $R_k \rightarrow T_j$



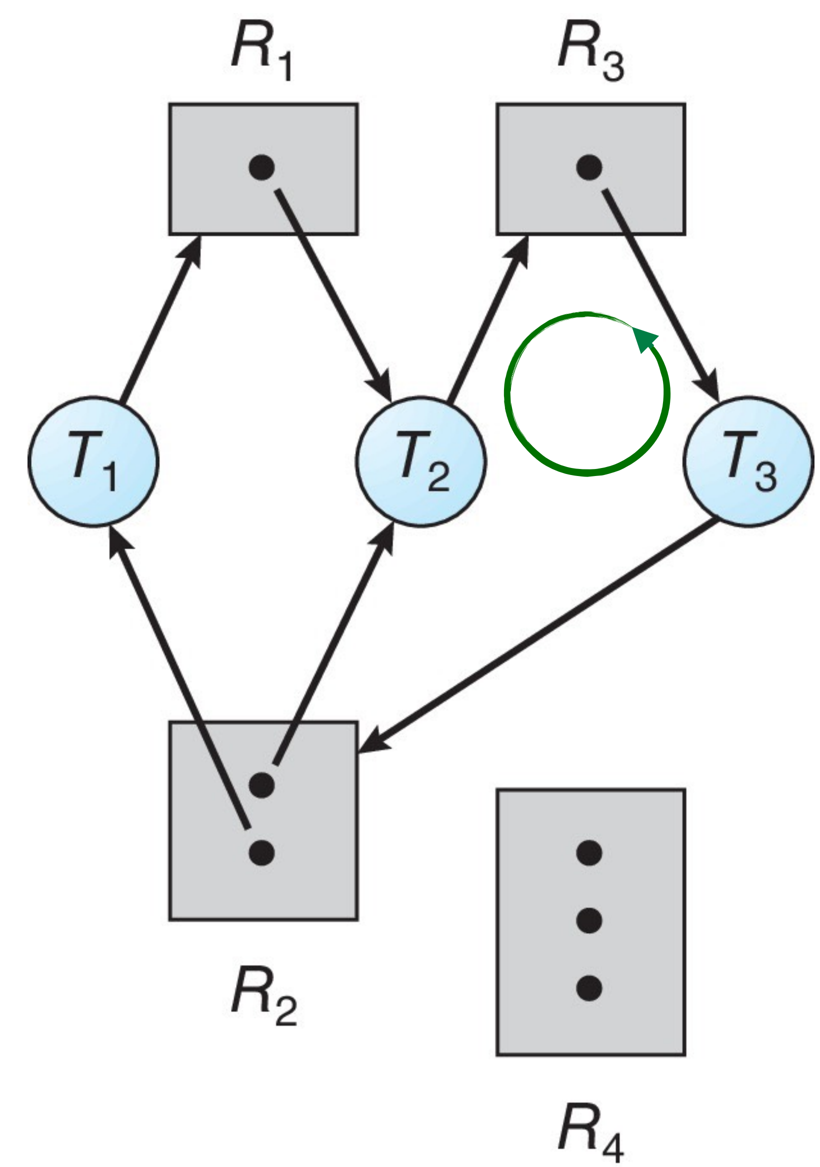
Resource-allocation graph

- the set of all vertices: $\{R_1, \ldots, R_M\} \cup \{T_1, \ldots, T_N\}$
  - the set of resource types: $\{R_1, \ldots, R_M\}$
    - resource instances are represented by dots
  - the set of threads in the system: $\{T_1, \ldots, T_N\}$
- a directed edge from a thread to a resource indicates a request: $T_i \rightarrow R_k$
- a directed edge from a resource instance to a thread indicates an assignment: $R_k \rightarrow T_j$
- example
  - there are two instances of R_2
  - T_3 holds one instance of R_3
  - T_2 requests one instance of R_3 (not available at the moment)



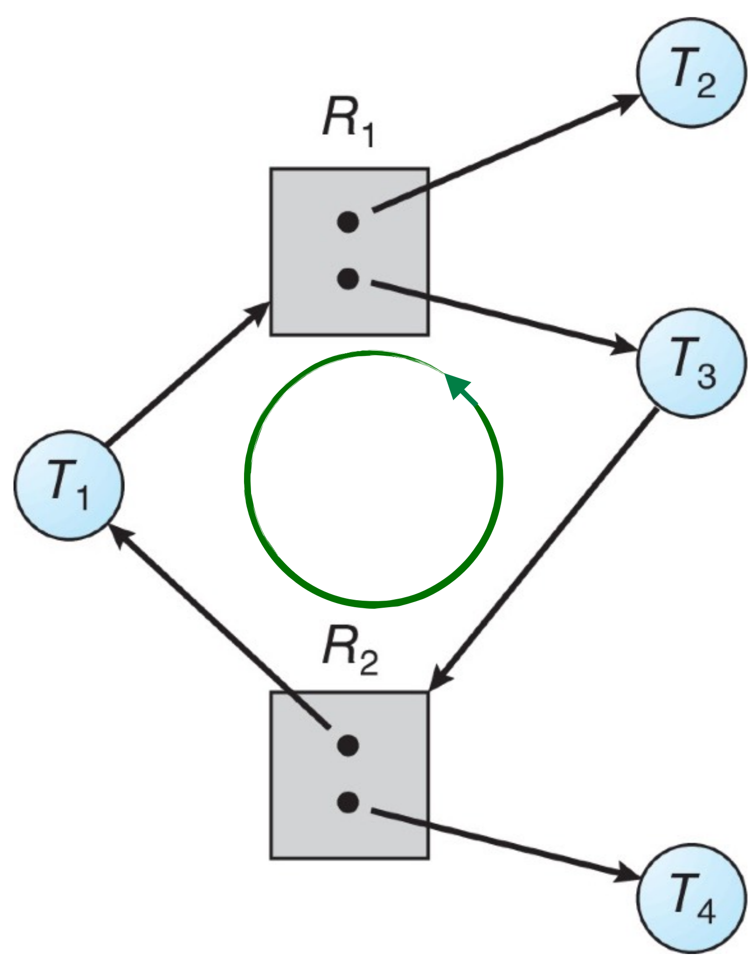
Example of resource-allocation graph

- we enter a deadlock when T_3 requests an instance of R_2
  - this graph has two cycles



Example of resource-allocation graph

- we enter a deadlock when T_3 requests an instance of R_2
  - this graph has two cycles
- but not all graphs with cycle show a deadlock



Resource-allocation graph of the example

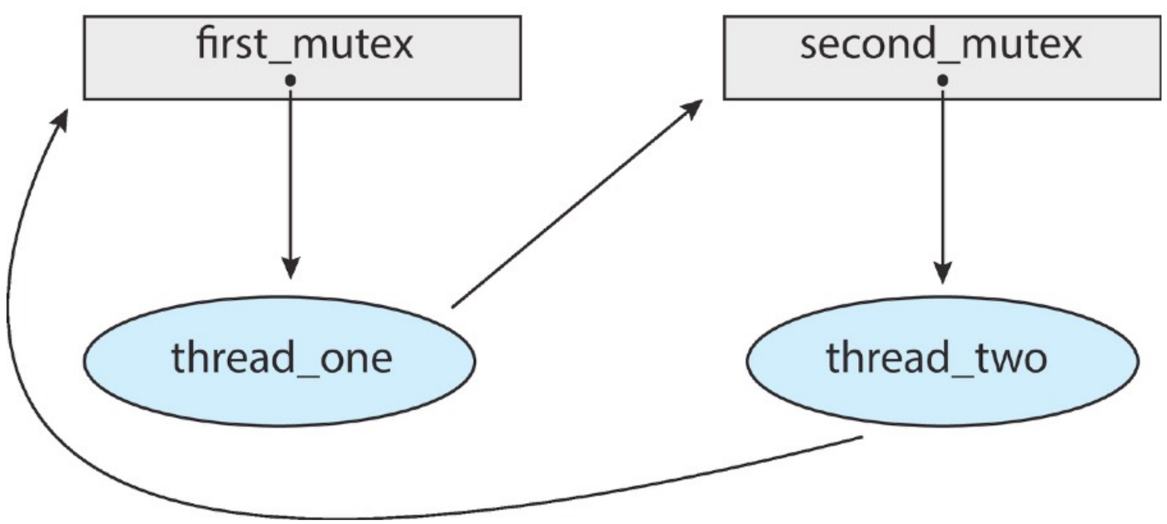
- two producers share a buffer but use a different protocol for accessing the buffer

Thread 1

```
Produce() {
   lock1.acquire();
   lock2.acquire();
   ...
}
```

Thread 2

```
Produce2() {
   lock2.acquire();
   lock1.acquire();
   ...
}
```



Takeaways

- if graph has no cycles
  - no deadlock; hence a cycle in the resource allocation graph is a necessary condition for deadlock (**is it a necessary condition?**)
- if graph has a cycle
  - if there is only one instance per resource type, then deadlock
  - if there are several instances per resource type, there is a possibility of deadlock

Necessary conditions for deadlock

- Deadlock **can** arise if the four conditions below hold simultaneously. Note that these conditions imply each other
  - Mutual exclusion — at least one resource is held in a non-sharable mode and the system doesn't have more resources of that type available
  - Hold and wait — a thread holds at least one resource and waits to acquire additional resources currently being held by other threads
  - No preemption — resources can be released only voluntarily by a thread that holds it (they cannot be forcibly taken from a thread)
  - Circular wait — there exists a set of waiting threads, each waiting for a resource held by the next one (**this is usually the weak link**)
    - the last thread waits for a resource held by the first thread

Terminology

- Detection
  - allow the system to enter a deadlocked state, periodically check for it (if threads are making progress), and recover
  - undetected deadlock will cause the system's performance to deteriorate; eventually the system will stop functioning

Terminology

- Detection
  - allow the system to enter a deadlocked state, periodically check for it (if threads are making progress), and recover
  - undetected deadlock will cause the system's performance to deteriorate; eventually the system will stop functioning
- Recovery
  - simplest approach: if OS froze once a year, you would just reboot it
  - this breaks the no-preemption condition as we will see

Terminology

- Detection
  - allow the system to enter a deadlocked state, periodically check for it (if threads are making progress), and recover
  - undetected deadlock will cause the system’s performance to deteriorate; eventually the system will stop functioning
- Recovery
  - simplest approach: if OS froze once a year, you would just reboot it
  - this breaks the no-preemption condition as we will see
- Prevention
  - goal is to guarantee that deadlock will never occur
  - ensure that at least one of the necessary conditions cannot hold

Terminology

- Detection
  - allow the system to enter a deadlocked state, periodically check for it (if threads are making progress), and recover
  - undetected deadlock will cause the system's performance to deteriorate; eventually the system will stop functioning
- Recovery
  - simplest approach: if OS froze once a year, you would just reboot it
  - this breaks the no-preemption condition as we will see
- Prevention
  - goal is to guarantee that deadlock will never occur
  - ensure that at least one of the necessary conditions cannot hold
- Avoidance
  - use an algorithm to check resource requests and availability to ensure that deadlock will never occur
  - additional information is needed: what resources a thread will request during its lifetime

Terminology

- Detection
  - allow the system to enter a deadlocked state, periodically check for it (if threads are making progress), and recover
  - undetected deadlock will cause the system's performance to deteriorate; eventually the system will stop functioning
- Recovery
  - simplest approach: if OS froze once a year, you would just reboot it
  - this breaks the no-preemption condition as we will see
- Prevention
  - goal is to guarantee that deadlock will never occur
  - ensure that at least one of the necessary conditions cannot hold
- Avoidance
  - use an algorithm to check resource requests and availability to ensure that deadlock will never occur
  - additional information is needed: what resources a thread will request during its lifetime
- Starvation: threads wait indefinitely (e.g., because some other thread is using a resource)
  - deadlock leads to starvation
  - but starvation is not caused necessarily by a deadlock situation

Deadlock prevention

- mutual exclusion:
  - open files in read-only model
  - in general we cannot guarantee that it doesn't hold
- hold-and-wait
  - protocol 1: request all resources at once→not practical due to the dynamic nature of requesting resources; even if it works, reduces resource utilization
  - protocol 2: request resources when it has none→may lead to starvation when one of these resources is popular

Deadlock prevention

- no preemption
  - protocol 1: preempt all resources when waiting
    - when a thread waits for a resource, all resources that it is currently holding are preempted (they are also added to the list of resources for which the thread is waiting)
  - protocol 2: preempt part of resources when waiting and some other thread requests those resources
    - when a thread requests a resource that is not currently available, check whether it is allocated to some other thread and if that thread is waiting for a resource, then preempt the desired resource from the waiting thread
  - both protocols work only for resources whose state can be saved, e.g. CPU registers
- circular wait
  - impose a total ordering of all resource types, i.e., threads request resources in an increasing order of enumeration

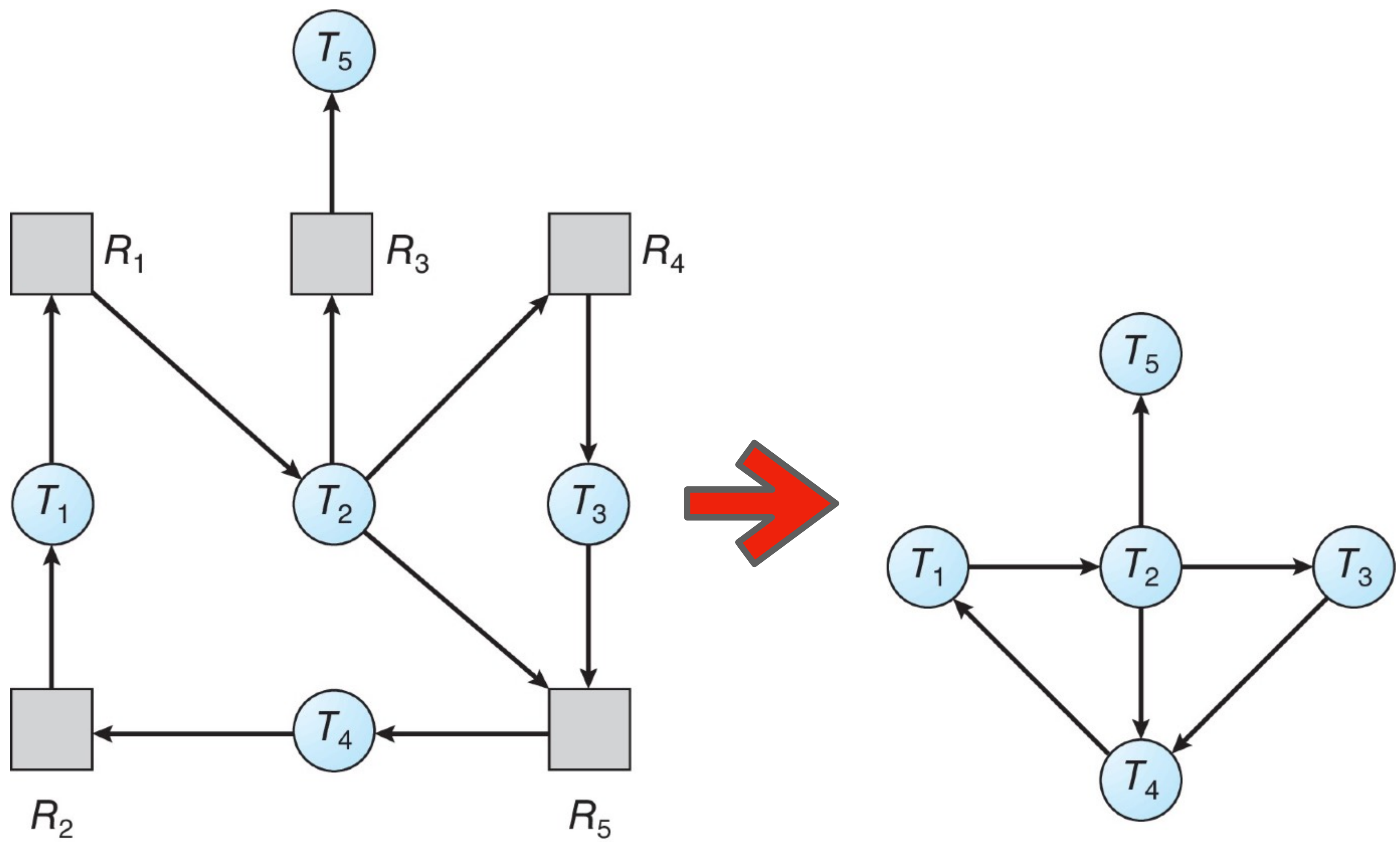
Imposing a total ordering of resources

- 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

Deadlock detection

if there is a single instance of each resource type

$\mathtt{T}_\mathtt{i} \rightarrow \mathtt{R}_\mathtt{j}$ and $\mathtt{R}_\mathtt{j} \rightarrow \mathtt{T}_\mathtt{k}$ yields $\mathtt{T}_\mathtt{i} \rightarrow \mathtt{T}_\mathtt{K}$



(a) **resource allocation graph**

(b) **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

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
- 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