Deadlocks

CSD2180 Operating Systems
BSc in Computer Science (IMGD / RTIS)
Singapore Institute of Technology / DigiPen Institute of Technology
September 2021

Attendance Taking

https://forms.office.com/r/s4HNNrnPLG

- Log in to your SIT account to submit the form
- You can only submit the form once
- The codeword is **codeword**

System Model

System Model

A system consists of finite resources, partitioned into several resource types (or classes)

Each process utilizes a resource as follows

Request

Use

Release

CPU cycles, memory space, I/O devices, etc. are examples of resource types

Let's label resource types as R_1, R_2, R_3, ..., R_m

Each resource type R_i has W_i instances, e.g., 4 instances of CPU, etc.

System Model

Request

The thread requests the resource; if the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource

Use

The thread can operate on the resource (e.g., if the resource is a mutex lock, the thread can access its critical section)

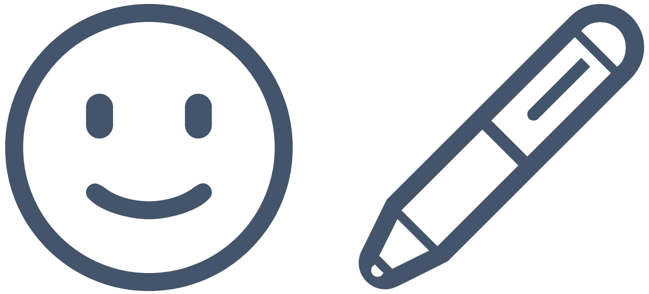
Release

The thread releases the resource

Deadlock

Deadlock

Deadlock



Deadlock

Deadlock with Mutex Locks

```c
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

```c
/* thread one runs in this function */
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /* do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
```

```c
/* thread two runs in this function */
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /* do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Livelock with Mutex Locks

```c
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

```c
/* thread one runs in this function */
void *do_work_one(void *param) {
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex))
        {
            /* do some work */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}
```

```c
/* thread two runs in this function */
void *do_work_two(void *param) {
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex))
        {
            /* do some work */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously

Four Necessary Conditions for Deadlock

Mutual exclusion
Only one process at a time can use a resource

Hold and wait
A process holding at least one resource is waiting to acquire additional resources held by other processes

No preemption
A resource can be released only voluntarily by the process holding it, after that process has completed its task

Circular wait
There exists a set of waiting processes P_0, P_1, ..., P_n, such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

Resource Allocation Graph

A set of vertices V and a set of edges E

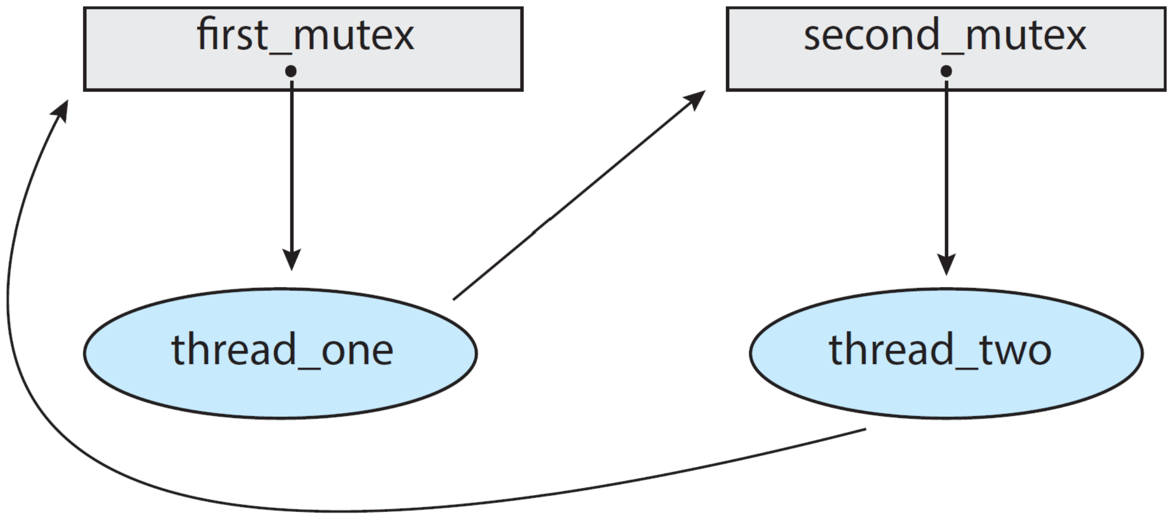
Two types of vertices V

- $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system; sometimes T used to denote threads instead
- $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system

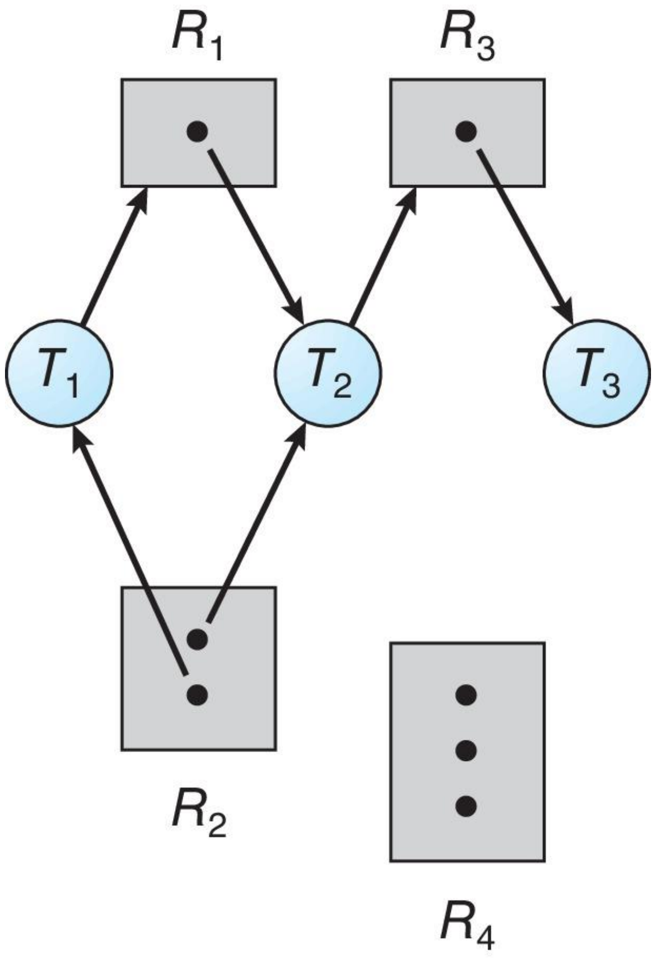
Two types of edges E

- **Request edge:** Directed edge $P_i \rightarrow R_j$
- **Assignment edge:** Directed edge $R_j \rightarrow P_i$

Instances of a resource type represented as dots inside resource type vertices



Resource Allocation Graph



Threads $T = \{T_1, T_2, T_3\}$

Resource types $R = \{R_1, R_2, R_3, R_4\}$

Edges $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$

Resource instances

- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4

Thread states

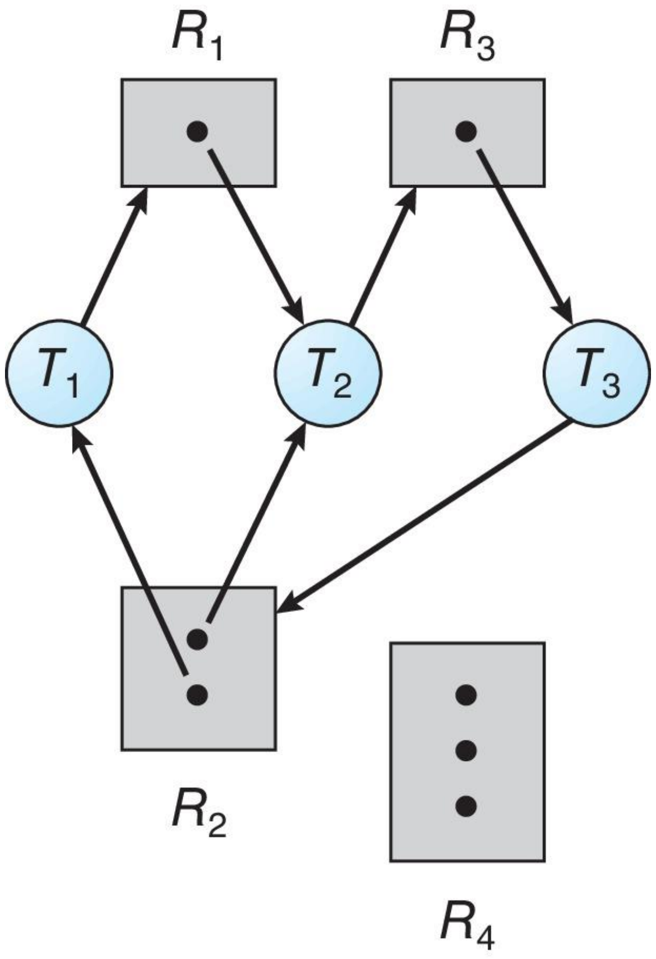
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3

Is there a deadlock?

If a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state

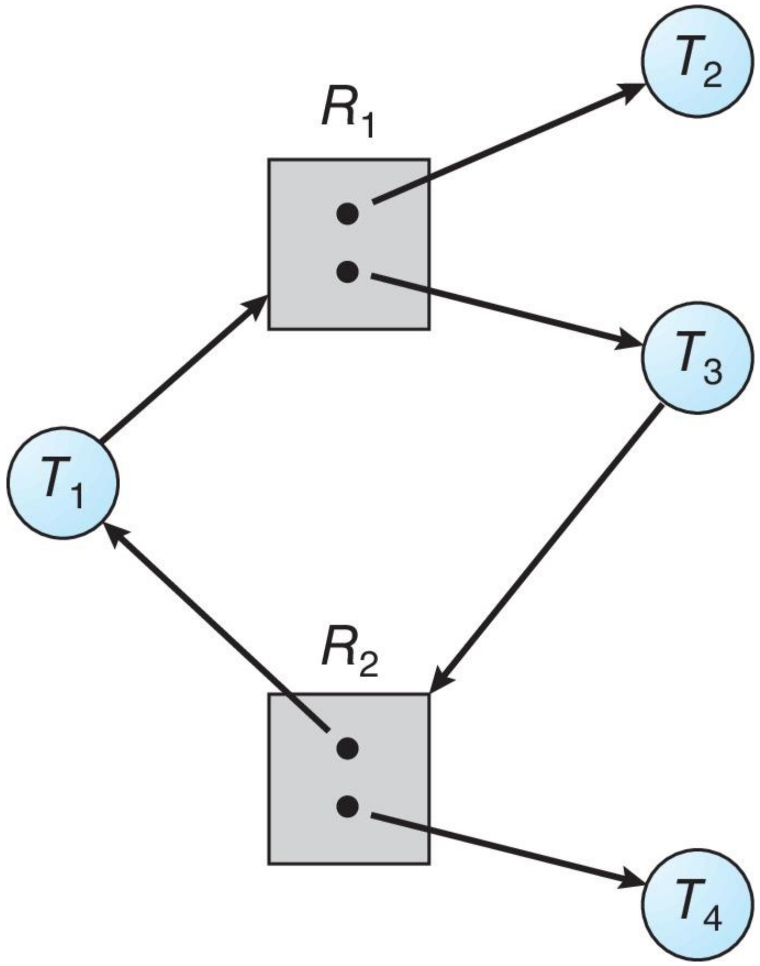
If the graph does contain a cycle, then the system *may* or *may not* be in a deadlocked state

Resource Allocation Graphs and Deadlocks



Is there a deadlock?

Resource Allocation Graphs and Deadlocks



What about this one? Is there a deadlock?

Basic Facts About Resource-Allocation Graph

If graph contains no cycles, then there is no deadlock

If graph contains a cycle, and

- There is only one instance per resource type, then there is deadlock
- There is at least one resource type with several instances, then there is *possibility* of deadlock

Methods for Handling Deadlocks

Methods for Handling Deadlocks

Ignore the problem altogether and pretend that deadlocks never occur in the system ☺

Use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state

Allow the system to enter a deadlocked state, detect it, and recover

The first solution is used by most operating systems, including Linux and Windows ⌛

Left to kernel and application developers to write programs that handle deadlocks ⏳

So how? ⌛

Deadlock Prevention

Deadlock Prevention

Recall four necessary conditions for deadlock, i.e., mutual exclusion, hold and wait, no preemption, and circular wait

Goal of deadlock prevention is to invalidate one of these conditions for deadlock

Mutual exclusion: Not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

Hold and wait: Must guarantee that whenever a process requests a resource, it does not hold any other resources

- Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it
- Results in low resource utilization since resources will be allocated but may not be used immediately
- May also lead to starvation if a process cannot obtain all required resources

Deadlock Prevention

Recall four necessary conditions for deadlock, i.e., mutual exclusion, hold and wait, no preemption, and circular wait

Goal of deadlock prevention is to invalidate one of these conditions for deadlock

No preemption: If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Circular wait: Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Invalidating Circular Wait

Resources are assigned a "sequence number", and must be acquired in order

```
/* valid: thread acquire first_mutex before second_mutex */
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /* do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}
```

Consider `first_mutex` has a sequence number of 1, and `second_mutex` has a sequence number of 2

A process or thread must hence acquire `first_mutex` before acquiring `second_mutex`, in that order

```
/* invalid: thread acquire second_mutex before first_mutex */
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /* do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Deadlock Avoidance

Deadlock Avoidance

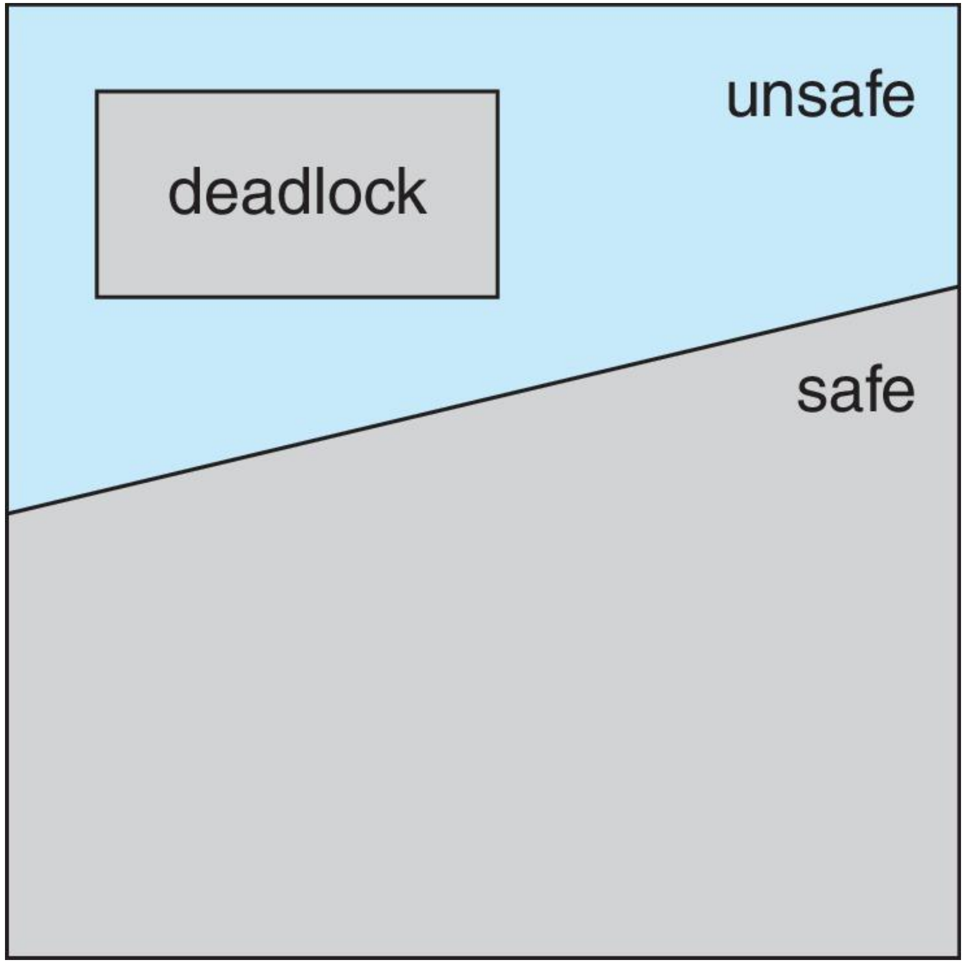
Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*
- System is in safe state if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of all processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all P_j, where $j < i$
- That is
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State and Deadlocks



If a system is in safe state, it has no deadlocks

If a system is in unsafe state, there is possibility of deadlock

Deadlock avoidance algorithms ensure that a system will never enter an unsafe state

Safe State and Deadlocks

Consider a system with twelve units of resources and three threads

Is this system in a safe state?

Thread	Maximum Resource Needs	Current Resource Needs
T_A	10	5
T_B	4	2
T_C	9	2

Safe State and Deadlocks

Consider a system with twelve units of resources and three threads

Is this system in a safe state?

Thread	Maximum Resource Needs	Current Resource Needs
T_A	10	5
T_B	4	2
T_C	9	2

Situation analysis

- Nine units of resources currently used; 3 units free
- T_B can be allocated all its remaining resources (2 units) and then returns them all when done
  - 5 units will be available when T_B is done
- Then, T_A can be allocated all its remaining resources (5 units) and return them all when done
  - 10 units will be available when T_A is done
- Finally, T_C can be allocated all its remaining resources (7 units) and return them all when done
  - 12 units will be available when done

The sequence $\langle \boldsymbol{T_B}, \boldsymbol{T_A}, \boldsymbol{T_C} \rangle$ satisfies the safety condition

Safe State and Deadlocks

Consider another system with twelve units of resources and three threads

Is this system in a safe state?

Thread	**Maximum Resource Needs**	**Current Resource Needs**
T_A	10	5
T_B	4	2
T_C	9	3

Safe State and Deadlocks

Consider another system with twelve units of resources and three threads

Is this system in a safe state?

Thread	Maximum Resource Needs	Current Resource Needs
T_A	10	5
T_B	4	2
T_C	9	3

Situation analysis

- Ten units of resources currently used; 2 units free
- T_B can be allocated all its remaining resources (2 units) and then returns them all when done
  - 4 units will be available when T_B is done
- No other threads can obtain all its resources
  - T_A requires 5 units more, while T_C requires 6 units more

The system is not in a safe state!

Questions?

Thank You!

Weihan.Goh {at} Singaporetech.edu.sg

https://www.singaporetech.edu.sg/directory/faculty/weihan-goh

https://sg.linkedin.com/in/weihan-goh