Soongsil Univ.
http://design.ssu.ac.kr

Digital System Design Lab.

Ch. 08 Deadlocks

Chanho Lee
Soongsil University

1

Objectives

- Deadlock with mutex locks.
- Four necessary conditions for deadlock.
  - ► Deadlock situation in a resource allocation graph.
- Solution to deadlock
  - ► preventing deadlocks.
  - ► deadlock avoidance.
  - ► deadlock detection.
  - ► recovering from deadlock.

8.1 System Model

- Resources
  - ►CPU cycles, files, and I/O devices (ethernet, Wifi, SSD, …)
  - ►Multiple instances for each resource type: allocation to any instance should satisfy the request
  - ►Mutex locks and semaphores are also system resources
    - the most common sources of deadlock
- utilization sequence of a thread for a resource
  1. Request. May wait until available
  2. Use.
  3. Release
  - ►Request and release may be system calls
    - request() and release() of a device,
    - open() and close() of a file,
    - allocate() and free() memory
    - wait() and signal() operations on semaphores
    - acquire() and release() of a mutex lock
  - ►Resource table: resource state: free or allocated – tid, waiting queue

8.2 Deadlock in Multithreaded Applications

- Deadlock:
  - ►Every thread in a set is blocked waiting for an event caused only by another thread in the set
- Illustration of deadlock in a multithreaded Pthread program
  - ►deadlocks may occur only under certain scheduling circumstances
    - Simultaneous thread_one and thread_two
    - Sequential thread_one and thread_two

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

```
/* 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);
}
```

```
/* 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);
}
```

8.2 Deadlock in Multithreaded Applications

- Livelock
  - ►a thread continuously attempts an action that fails
  - ►not blocked, but no progress
  - ►Less common than deadlock

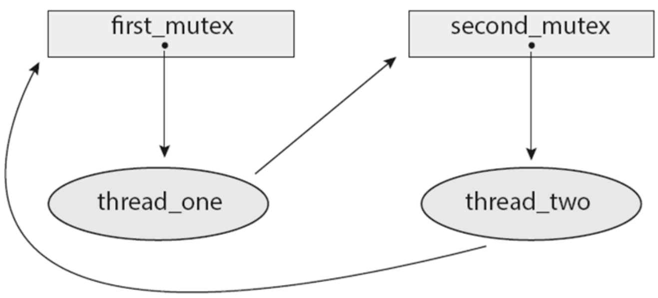
```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
  int done = 0;
  while (!done) {
    pthread_mutex_lock(&first_mutex);
    if (pthread_mutex_trylock(&second_mutex)) {
      /** Do some work */
      pthread_mutex_unlock(&second_mutex);
      pthread_mutex_unlock(&first_mutex);
      done = 1;
    }
    else
      pthread_mutex_unlock(&first_mutex);
  }
  pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_one(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);
}
```

Solution: retry the failing operation at random times (ex. Ethernet)

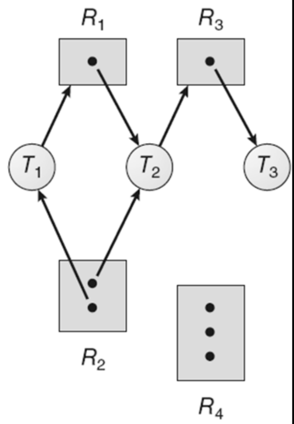
8.3 Deadlock Characterization

- Necessary Conditions: holding simultaneously
  - ►Mutual exclusion: resource in a nonsharable mode
  - ►Hold and wait: A thread holding a resource and waiting additional resources held by other
  - ►No preemption: a resource can be released only voluntarily
  - ►Circular wait. Implies hold-and-wait: not completely independent
- Resource-Allocation Graph



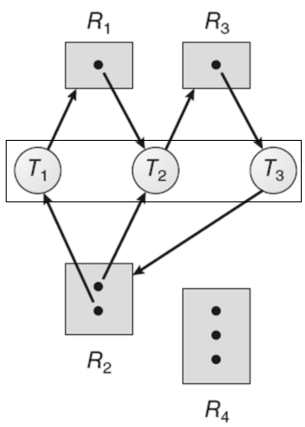
8.3 Deadlock Characterization

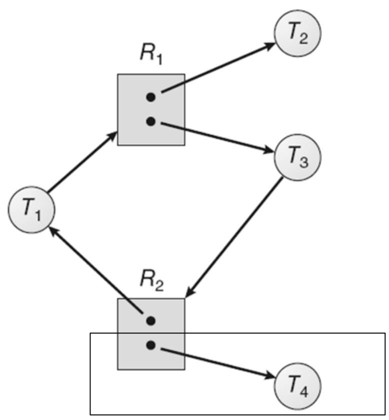
- Resource-Allocation Graph
  - ►system resource-allocation graph
    - a set of vertices V and a set of edges E
    - V : T = {T1, T2, …, Tn}, active threads, and R = {R1, R2, …, Rm}, resource types
    - Request edge, Ti → Rj; Ti has requested a Rj instance and is waiting
    - Assignment edge, Rj → Ti; Rj instance has been allocated to Ti
  - ►Ex. The sets T, R, and E:
    - T = {T1, T2, T3}
    - R = {R1, R2, R3, R4}
    - E = {T1 → R1, T2 → R3, R1 → T2, R2 → T2, R2 → T1, R3 → T3}
    - Resource instances
    - Thread states
  - ►no cycles → no deadlock



8.3 Deadlock Characterization

- ►a cycle → a deadlock may exist
  - one instance for a resource: a cycle implies a deadlock (necessary and sufficient) √
  - several instances for a resource: a cycle does not necessarily imply a deadlock (necessary but not sufficient) √





8.4 Methods for Handling Deadlocks

- Ignore the problem
  - ►Common. Linux and Windows.
  - ►Up to kernel and application developers using the 2nd solution
  - ►Cheapest
- To prevent or avoid deadlocks → never enter a deadlocked state.
  - ►Prevention: at least one of the necessary conditions cannot hold. Constraining requests
  - ►Avoidance: additional information is given to OS concerning request and use of resources → decide allocation : most expensive
- Allow deadlocked state → detect and recover.
  - ►Ex. Database
  - ►Manual recovery: simple
- Performance and expense

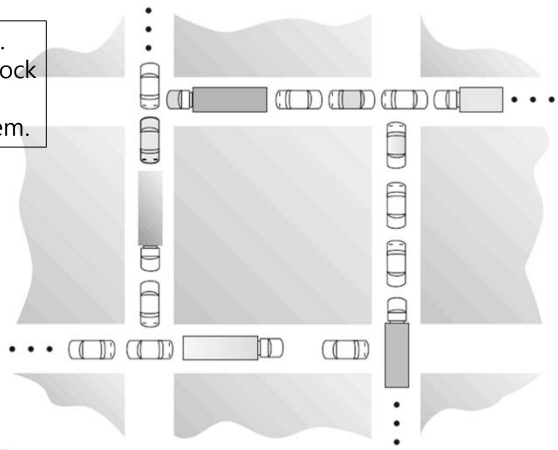
Summary

- Deadlock
  - ►Resources and multithreading
- Livelock
- 4 necessary conditions
- Resource-Allocation Graph
- Handling deadlocks
  - ►Ignoring, preventing or avoiding, recovery

Exercises, problems and projects

- Exercises
  - ►8.1, 8.7, 8.11
- Problems
  - ►8.12, 8.18

8.12 Consider the traffic deadlock depicted in the Figure.
a. Show that the four necessary conditions for deadlock hold in this example.
b. State a simple rule for avoiding deadlocks in this system.



Exercises, problems and projects

8.18 Which of the six resource-allocation graphs shown in the Figures illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution