CISC 3320
Deadlock Prevention

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

- Deadlock Prevention
  - Invalidating necessary conditions for deadlocks
- Deadlock Avoidance
- Deadlock Detection and Recovery

Deadlock Prevention

- By invalidating one of the 4 necessary conditions
  - Mutual Exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Let’s examine each of these 4 prevention strategies

Invalidating Mutual Exclusion?

- Consider two types of resources
  - Sharable resources
    - Example
      - Read-only files
  - Non-sharable resources
    - Example
      - Printers
  - "Sharable" means access simultaneously.
- Mutual exclusion not required for sharable resources
- Mutual exclusion must hold for non-sharable resources
  - Cannot deny the mutual-exclusion condition, thus, cannot prevent deadlocks

Invalidating Hold-and-Wait?

- To do it, we must guarantee that whenever a process requests a resource, it does not hold any other resources
  1. Require process to request and be allocated all its resources before it begins execution
  2. Or allow process to request resources only when the process has none allocated to it (e.g., by releasing it)
- Problem with these two approaches to invalidate Hold-and-Wait
  - Low resource utilization; starvation possible; impractical

Invalidating No-Preemption?

- Implies that we should allow preemption for resource allocation. But how?
  1. 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 (i.e., call it "yielding"?)
  2. We check whether requested resources are allocated to some other thread that is waiting for additional resources. If so, we preempt the desired resources from the waiting thread and allocate them to the requesting thread. (i.e., shall we call it "robbing"?)
     - 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

Resource Preemption

- Invalidating No-Preemption by resource "preemption"
  - Suitable for resources whose state can be easily saved and restored later
    - such as CPU registers and database transactions
  - It cannot generally be applied to such resources as mutex locks and semaphores
    - Precisely the type of resources where deadlock occurs most commonly.

Invalidating Circular Wait?

- Generally impractical in most situations for deadlock prevention by invalidating
  - Mutual exclusion, hold-and-wait, and non-preemption
- Is there any means to invalidate Circular Wait?

Approaches for Invalidating Circular Wait

- Resource ordering
  - Impose a *total ordering* of all resource types by simply assigning each resource (i.e., mutex locks) a unique number
  - Resources must be acquired in order based on the numbers

Resource Ordering: Formulation

- Let R = {R1, R2, …, Rm} be the set of resource types.
- Define a one-to-one function F: R → N to each resource type a unique integer number.
- A thread initially requests an instance of a resource, Ri, can request an instance of resource Rj if and only if F(Rj) > F(Ri).
- Alternatively, a thread requesting an instance of resource Rj must have released any resources Ri such that F(Ri) ≥ F(Rj).
- Note also that if several instances of the same resource type are needed, a single request for all of them must be issued.

Proof by Contradiction

1. Assume that a circular wait exists, i.e., let the set of threads involved in the circular wait be {T0, T1, …, Tn}, where Ti is waiting for a resource Ri, which is held by thread Ti+1.
   - Modulo arithmetic is used on the indexes, so that Tn is waiting for a resource Rn held by T0.
2. Then, since thread Ti+1 is holding resource Ri while requesting resource Ri+1, we must have F(Ri) < F(Ri+1) for all i. But this condition means that F(R0) < F(R1) < … < F(Rn) < F(R0). By transitivity, F(R0) < F(R0), which is impossible. Therefore, there can be no circular wait.

Remarks: Using Resource Ordering

- Resource ordering does not in itself prevent deadlock.
- Application developers must write programs that follow the ordering.

Resource Ordering: Example

- Two resources (i.e., two mutexes) , and their ordering
  - **`Order of first_mutex: 1`**
  - **`Order of second_mutex: 5`**
- Which means first_mutex must be acquired first, and second_mutex second (because 1 < 5)

- Code for **`thread_two`** should ***NOT*** be written as illustrated

```
/* 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);
}
```

Remarks: Using Resource Ordering: Challenges

- However, establishing a lock ordering can be difficult
  - e.g., considering on a system with hundreds or even thousands of locks …
  - To address this challenge, many Java developers have adopted the strategy of using the method System.identityHashCode as the function for ordering lock acquisition.

Remarks: Using Resource Ordering: Dynamic Acquiring

- Imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically
- Example: assume we have a function that transfers funds between two bank accounts.
  - To prevent a race condition, each account has an associated mutex lock that is obtained from a get_lock() function
  - Deadlock is possible if two threads simultaneously invoke the transaction() function, transposing different accounts.
    - Thread 1: transaction(checking_account, savings_account, 25.0)
    - Thread 2: transaction(savings_account, checking_account, 50.0)

The transaction Function

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
        acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

Questions?

- Deadlock prevention
- Invalidating any one of the 4 necessary conditions
  - Mutual exclusion
  - Hold and wait
  - Non-preemption
  - Circular wait
- Approaches and limitations?