香港城市大學
City University of Hong Kong

Tutorial 5: Deadlocks

CS3103

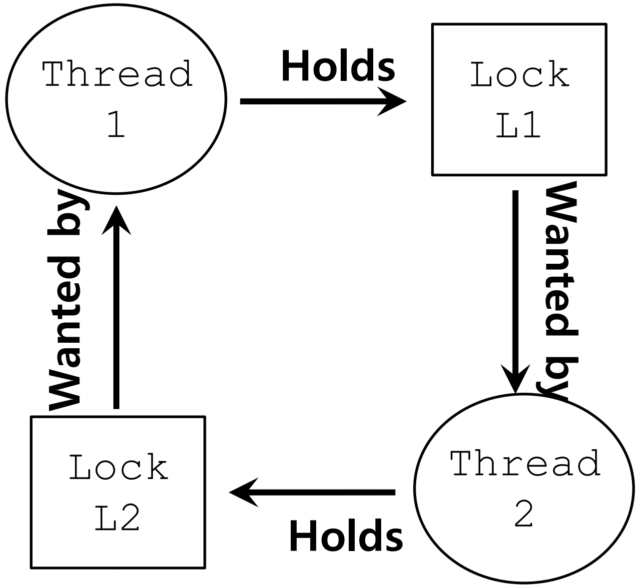
Operating Systems

專業 創新 胸懷全球
Professional · Creative
For The World

Disclaimer: This slide was initially developed for Operating System course in Department of Computer Science at Hanyang University.
This slide is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

Deadlock

```
Thread 1:      Thread 2:
lock(L1);      lock(L2);
lock(L2);      lock(L1);
```



- The presence of a cycle
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`
  - `Thread2` that holds lock `L2` is waiting for `L1` to be release

```c
void vector_add(vector_t *v_dst, vector_t *v_src) {
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
    v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

vector-deadlock.c

Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, complex dependencies arise between components
- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way
    - Such modularity *does not mesh* well with <u>locking</u>

Why Do Deadlocks Occur?

- **Example**: Java Vector class and the method `AddAll()`

```
1   Vector v1,v2;
2   v1.AddAll(v2);
```

  - **Locks** for both the vector being added to (`v1`) and the parameter (`v2`) *need to be acquired*.
    - The routine acquires said locks in some arbitrary order (v1 then v2).
    - If some other thread `calls v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock.

CityU

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur.

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

  - If any of these four conditions are not met, **deadlock cannot occur**.

Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies.
- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks at once, atomically.

```
void vector_add(vector_t *v_dst, vector_t *v_src) {
    // put GLOBAL lock around all lock acquisition...
    Pthread_mutex_lock(&global);
    Pthread_mutex_lock(&v_dst->lock);
    Pthread_mutex_lock(&v_src->lock);
    Pthread_mutex_unlock(&global);
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
    v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

vector-avoid-hold-and-wait.c

  – This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.
  – **Problem**:
    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
    - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.
- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```c
void vector_add(vector_t *v_dst, vector_t *v_src) {
  top:
    if (pthread_mutex_trylock(&v_dst->lock) != 0) {
    goto top;
    }
    if (pthread_mutex_trylock(&v_src->lock) != 0) {
    retry++;
    Pthread_mutex_unlock(&v_dst->lock);
    goto top;
    }
    int i;
    for (i = 0; i < VECTOR_SIZE; i++) {
    v_dst->values[i] = v_dst->values[i] + v_src->values[i];
    }
    Pthread_mutex_unlock(&v_dst->lock);
    Pthread_mutex_unlock(&v_src->lock);
}
```

vector-try-wait.c

Prevention – No Preemption (Cont.)

- livelock
  - Both systems are running through the code sequence *over and over again*.
  - Progress is not being made.
  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

```
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
            *address = new;
            return 1; // success
    }
    return 0;
}
```

Prevention – Mutual Exclusion (Cont.)

- We now wanted to atomically increment a value by a certain amount:

```
void AtomicIncrement(int *value, int amount){
    do{
            int old = *value;
    }while( CompareAndSwap(value, old, old+amount)==0);
}
```

  - Repeatedly tries to update the value to *the new amount* and uses the `compare-and-swap` to do so.

  - **No lock** is acquired
  - **No deadlock** can arise
  - **livelock** is still a possibility.

Prevention – Mutual Exclusion (Cont.)

- **Solution**:
  - Surrounding this code with a lock acquire and release.

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    lock(listlock);       // begin critical section
    n->next     = head;
    head   = n;
    unlock(listlock) ;   //end critical section
}
```

  - wait-free manner using the `compare-and-swap` instruction

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
          n->next = head;
    } while (CompareAndSwap(&head, n->next, n)== 0);
}
```

Deadlock Avoidance via Scheduling

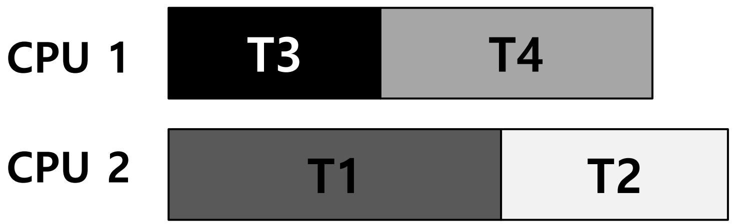
- In some scenarios deadlock avoidance is preferable.
  - **Global knowledge** is required:
    - Which locks various threads might grab during their execution.
    - Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
  - Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

  - A smart scheduler could compute that as long as <u>T1 and T2 are not run at the same time</u>, no deadlock could ever arise.

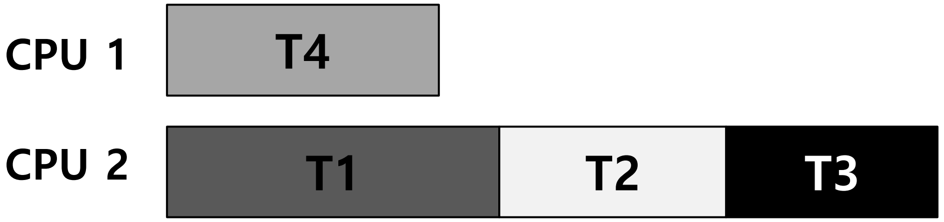


Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

  – A possible schedule that guarantees that *no deadlock* could ever occur.



    • The total time to complete the jobs is lengthened considerably.

Detect and Recover

- Allow deadlock to occasionally occur and then *take some action*.
  - **Example**: if an OS froze, you would reboot it.

- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for cycles.
  - In deadlock, the system **need to be restarted**.