IE411: Operating Systems

Deadlock bugs

Deadlocks

- Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
- Could arise when we need to coordinate access to more than one shared resource
  - e.g., acquiring multiple locks can result in deadlock

Example

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

- If this code runs, deadlock does not necessarily occur
- It may occur, if, for example
  - Thread 1 grabs lock L1 and then a context switch occurs to Thread 2
  - At that point, Thread 2 grabs L2, and tries to acquire L1

Conditions for deadlock

- Four conditions need to hold for a deadlock to occur
    1. Mutual exclusion: threads claim exclusive control of resources that they require
    2. Hold-and-wait: threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks)

Conditions for deadlock

- Four conditions need to hold for a deadlock to occur
  1. Mutual exclusion: threads claim exclusive control of resources that they require
  2. Hold-and-wait: threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks)
  3. No preemption: resources (e.g., locks) cannot be forcibly removed from threads that are holding them

Conditions for deadlock

- Four conditions need to hold for a deadlock to occur
  1. Mutual exclusion: threads claim exclusive control of resources that they require
  2. Hold-and-wait: threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks)
  3. No preemption: resources (e.g., locks) cannot be forcibly removed from threads that are holding them
  4. Circular wait: there exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain

Deadlock prevention

- (mutual exclusion) and (hold-and-wait) and (no-preemption) and (circular wait)
- If any of these four conditions are not met deadlock cannot occur

Deadlock prevention

- (mutual exclusion) and (hold-and-wait) and (no-preemption) and (circular wait)
- If any of these four conditions are not met deadlock cannot occur
- Deadlock prevention: prevent one of the above conditions from arising
    - $\sim$ (mutual exclusion) or $\sim$ (hold-and-wait) or $\sim$ (no preemption) or $\sim$ (circular wait) $\rightarrow$ no deadlock

Preventing circular wait

- Avoid circular wait by acquiring locks in the same order
  - called a locking hierarchy
- Example
  - Assume two threads need locks L1 and L2
  - Use a locking hierarchy, e.g., L1 $\rightarrow$ L2

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

Preventing circular wait

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

- Only one thread will be able to acquire L1, the other thread has to block
- No deadlock possible, because the blocking thread did not acquire L2 before if it obeys the locking hierarchy
- The blocking thread can continue only after the locks have been released

Enforce lock ordering

- Example: `set_t *intersect(set_t *s1, set_t *s2)`
- Locks `s1->lock` and `s2->lock` for the sets `s1` and `s2` need to be acquired
- Assume `intersect()` acquires the said locks in some fixed order
  - `s1->lock` before `s2->lock` (or `s2->lock` before `s1->lock`)
- Deadlock possible?

Enforce lock ordering

- Example: `set_t *intersect(set_t *s1, set_t *s2)`
- Locks `s1->lock` and `s2->lock` for the sets `s1` and `s2` need to be acquired
- Assume `intersect()` acquires the said locks in some fixed order
  - `s1->lock` before `s2->lock` (or `s2->lock` before `s1->lock`)
- Deadlock possible?
  - one thread calls `intersect(s1, s2)`
  - another thread calls `intersect(s2, s1)`

Enforce lock ordering

- One possible solution is to acquire the locks in the order of their virtual addresses

```
if (&s1->lock > &s2->lock) {
    // grab locks in high-to-low address order
    lock(s1->lock);
    lock(s2->lock);
} else {
    lock(s2->lock);
    lock(s1->lock);
}
```

Preventing hold-and-wait

- Avoid hold-and-wait by acquiring all locks at once atomically, say by acquiring a master lock first

```
lock(prevention);
lock(L1);
lock(L2);
...
unlock(prevention);
```

Preventing hold-and-wait

- Avoid hold-and-wait by acquiring all locks at once atomically, say by acquiring a master lock first

```
lock(prevention);
lock(L1);
lock(L2);
...
unlock(prevention);
```

- Guarantees that no untimely thread switch can occur in the midst of lock acquisition
- But this method method may reduce concurrent execution and performance gains

Preventing no preemption

- Strategy: If a thread can't get what it wants, release what is holds
- trylock()
  - grabs the lock if it is available, otherwise returns -1

```
top:
        lock(L1);
        if (trylock(L2) == -1) {
            unlock(L1);
            goto top;
        }
```

Preventing no preemption

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
     unlock(L1);
     goto top;
  }
```

```
top:
  lock(L2);
  if (trylock(L1) == -1) {
     unlock(L2);
     goto top;
  }
```

- Another thread could follow the same protocol but grab locks in the other order (L2 then L1)
  - still be deadlock free

New issue: Livelock

- It can happen that both threads repeatedly attempt this sequence and repeatedly fail to acquire both locks
- Livelock: the state of the threads is constantly changing (so not a deadlock), but there is no progress

New issue: Livelock

- It can happen that both threads repeatedly attempt this sequence and repeatedly fail to acquire both locks
- Livelock: the state of the threads is constantly changing (so not a deadlock), but there is no progress
- solution: add a random delay looping back and trying the entire thing over again

Preventing mutual exclusion

- Avoid the need for mutual exclusion at all
- But how, the code we wish to run does have critical sections
- Use lock-free approaches to code data structures without explicit locking

Compare and swap

Yet another atomic instruction (pseudo C code):

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

Atomic increase

- Goal: 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);
}
```

  - no lock is required and no deadlock can arise (livelock is possible)

Other solutions to deadlock

- Deadlock avoidance
  - requires global knowledge of which locks various threads might grab during their execution, and subsequently schedule threads in a way as to guarantee no deadlock can occur
  - Banker's algorithm is a well-known deadlock avoidance algorithm
- Detect and recover (reboot or kill deadlock threads)

Next time

- Lock-free linked lists!

Exercie: Spinlock using compare-and-swap

```
typedef struct {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (CAS(?, ?, ?) == ?)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```