COMP35112 Chip Multiprocessors

More about Locks

Pierre Olivier

Dangers with Locks

Dangers with Locks

Dangers with Locks

```c
typedef struct {
    double balance;
    pthread_mutex_t lock;
} account;

void initialise_account(account *a, double balance) {
    a->balance = balance;
    pthread_mutex_init(&a->lock, NULL);  // return value checks omitted for brevity
}

void transfer(account *from, account *to, double amount) {
    if(from == to) return;  // can't take a standard lock twice, avoid account transfer to self

    pthread_mutex_lock(&from->lock);
    pthread_mutex_lock(&to->lock);

    if(from->balance >= amount) {
        from->balance -= amount;
        to->balance += amount;
    }

    pthread_mutex_unlock(&to->lock);
    pthread_mutex_unlock(&from->lock);
}
```

09-more-about-locks/deadlock.c

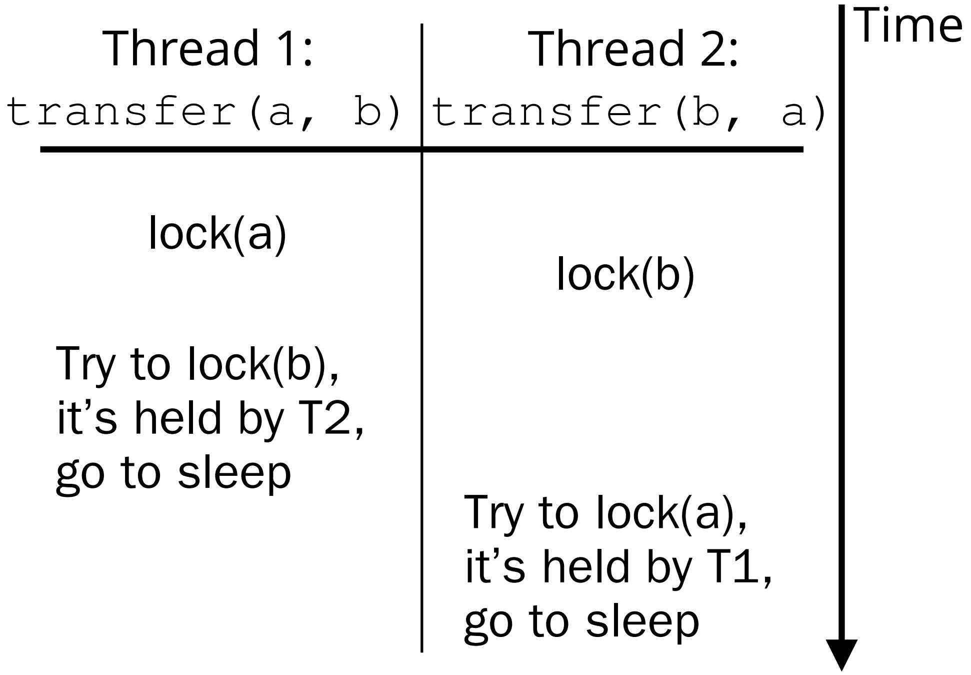
Dangers with Locks

```
void transfer(account *from, account *to,
        double amount) {
    if(from == to) return;

    pthread_mutex_lock(&from->lock);
    pthread_mutex_lock(&to->lock);

    if(from->balance >= amount) {
        from->balance -= amount;
        to->balance += amount;
    }

    pthread_mutex_unlock(&to->lock);
    pthread_mutex_unlock(&from->lock);
}
```



Thread 1:
`transfer(a, b)`

Thread 2:
`transfer(b, a)`

Time

lock(a)

lock(b)

Try to lock(b),
it’s held by T2,
go to sleep

Try to lock(a),
it’s held by T1,
go to sleep

Deadlock! 🙃

Dangers with Locks

```
typedef struct {
    int id;                 // unique integer id, used to sort accounts
    double balance;
    pthread_mutex_t lock;
} account;

void transfer(account *from, account *to, double amount) {
    if(from == to) return;
    pthread_mutex_t *lock1 = &from->lock, *lock2 = &to->lock;

    if(from->id < to->id) {   // always lock the accounts in the same order
        lock1 = &to->lock;
        lock2 = &from->lock;
    }

    pthread_mutex_lock(lock1);
    pthread_mutex_lock(lock2);
    if(from->balance >= amount) {
        from->balance -= amount;
        to->balance += amount;
    }
    pthread_mutex_unlock(lock2);
    pthread_mutex_unlock(lock1);
}
```

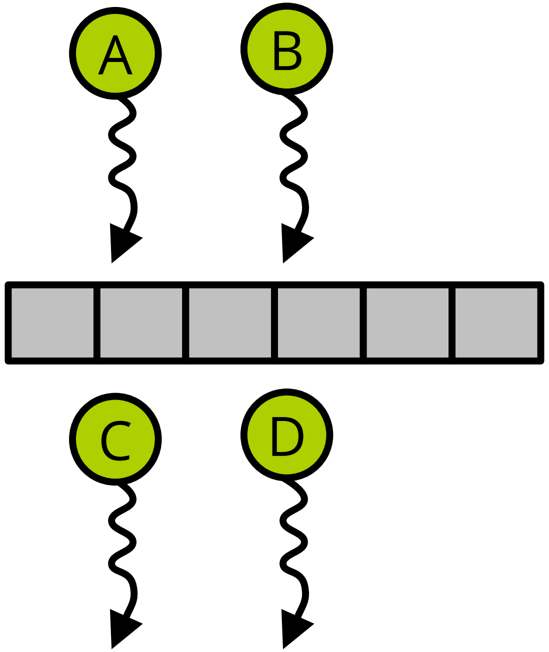
09-more-about-locks/deadlock-fixed.c

Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture

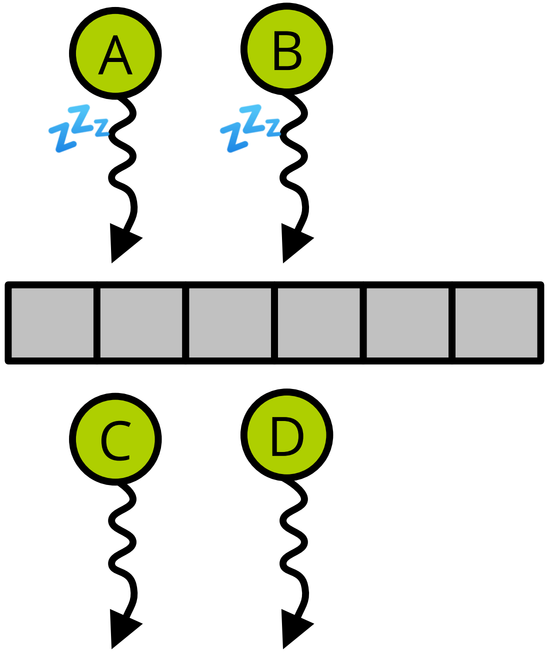
Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture



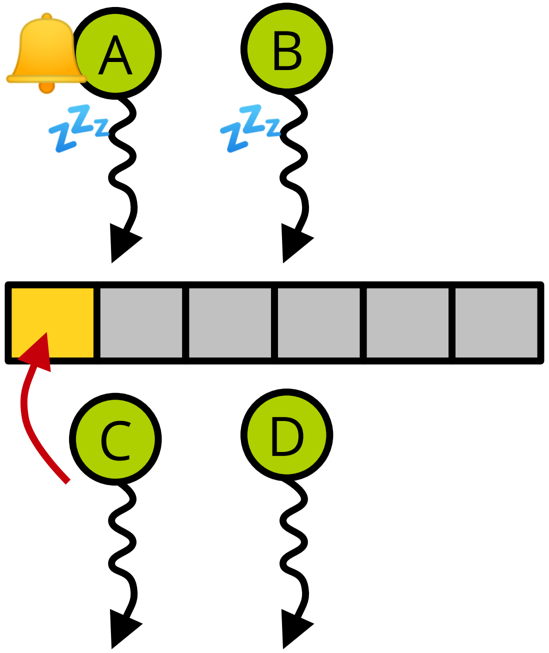
Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture



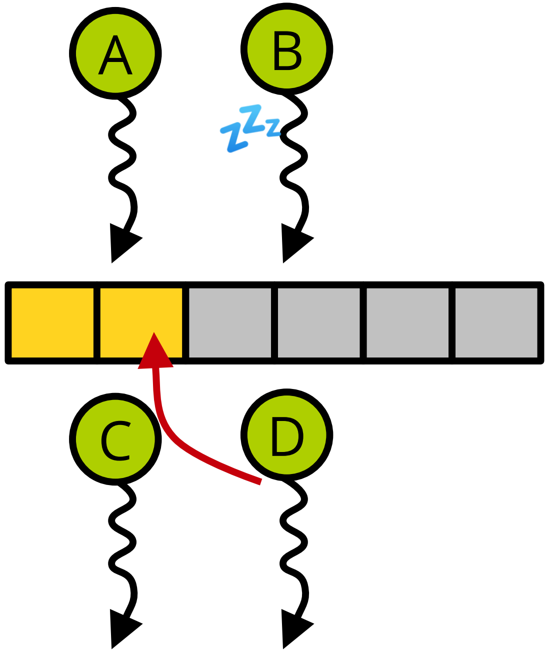
Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture



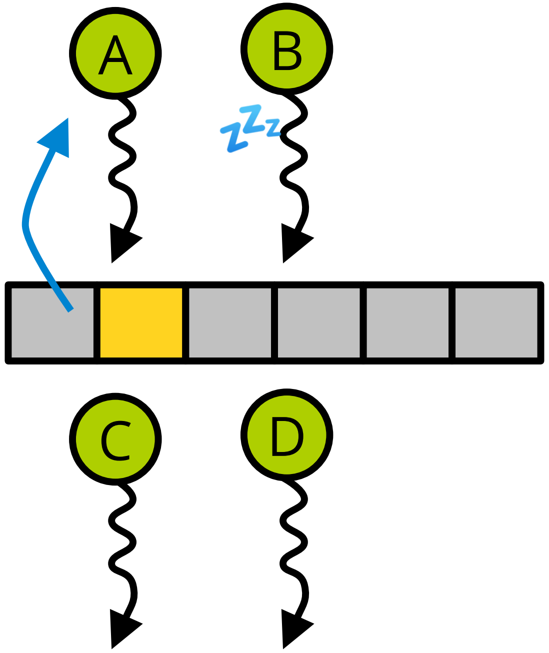
Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture



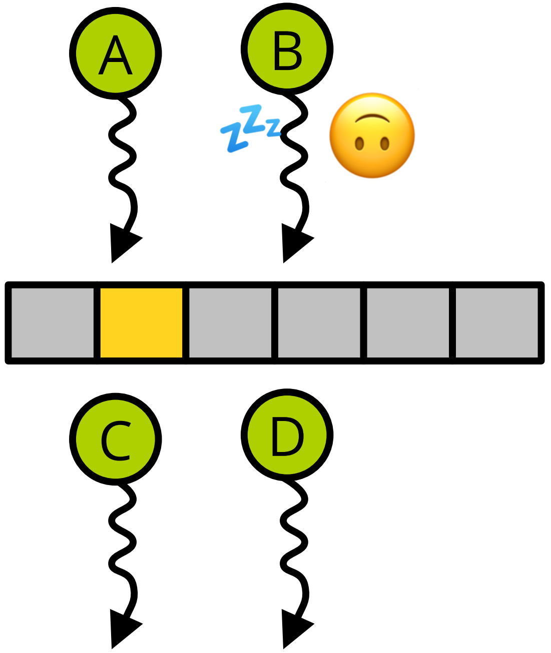
Dangers with Locks

- **Lost wakeup** issue
    - Example with `bounded_buffer` code from last lecture



Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture



Dangers with Locks

- **Lost wakeup** issue
  - Example with `bounded_buffer` code from last lecture

- Fix (here) would be to use `pthread_cond_broadcast()` instead of `pthread_cond_signal()`
  - Wake up **all** threads (vs. a single thread) waiting on a condition variable

Misc. Information about Locks

Granularity

- How big a chunk of code which depends on obtaining a lock should you write?
    - **Coarse-** vs. **fine-** grained

```
/* Coarse-grained locking: */

lock();


/* access a mix of shared and unshared data */


unlock();
```

```
/* Fine-grained locking: */

lock();
/* access shared data */
unlock();
/* access non-shared data */
lock();
/* access shared data */
unlock();
```

Reentrant Lock

- By default, a thread locking a lock it already holds results in **undefined behaviour**

```c
void transfer(account *from, account *to,
    double amount) {
  /* no check if from == to */

  // BUGGY when from == to if lock is
  // not reentrant
  pthread_mutex_lock(from->lock);
  pthread_mutex_lock(to->lock);

  if(from->balance >= amount) {
    from->balance -= amount;
    to->balance += amount;
  }
  /* ... */
}
```

```c
int main(int argc, char **argv) {
  account account1;
  pthread_t t1;

  initialize_account(&account1, 1, INIT_MONEY);

  /* transfer from account1 to account1 */
  worker w1 = {&account1, &account1,
    ITERATIONS};

  pthread_create(&t1, NULL, thread_fn,
    (void *)&w1);
  pthread_join(t1, NULL);

  return 0;
}
```

09-more-about-locks/non-reentrant.c

Reentrant Lock

- A **reentrant** lock can be taken by a thread that already holds it
    - Avoid a thread deadlocking with itself

```c
// Version of the bank account program that allows self transfers
// (return value checks omitted for brevity)

void initialize_account(account *a, int id, double balance) {
    a->id = id;
    a->balance = balance;

    /* Declare the lock as reentrant */
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&a->lock, &attr);
}
```

09-more-about-locks/reentrant.c

]

Other Lock Types

- **Semaphores**
  - Mutexes that can be hold by multiple threads
  - Useful to coordinate access to a fixed number of resources
- **Spinlocks**
  - Threads attempting to hold an unavailable lock will **busy-wait**
    - As opposed to going to sleep for mutexes
    - Monopolises CPU, lower wakeup latency
- **Read-write locks**
  - Allows concurrent reads and exclusive writes

For more information see the multithreaded programming guide:
https://bit.ly/3FGt3k2

Summary

- Locks come with their own issues
    - **Concurrency issues are hard to debug, it's important to get your synchronisation strategy right from the beginning**
- Lock granularity and reentrancy
- Other lock types: semaphores, spinlocks, read-write locks

Next lecture: hardware support for synchronisation