Using Mutex Locks
Overview

- Which type of mutex lock to use and when?
- How to use mutex locks correctly?
Locking Solutions

- Notice how locking solutions depend on lower-level locking.

<table>
<thead>
<tr>
<th>Uniprocessor</th>
<th>Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocking lock</td>
<td>blocking lock</td>
</tr>
<tr>
<td>interrupt disabling lock</td>
<td>spin lock yielding lock</td>
</tr>
<tr>
<td>interrupt disabling</td>
<td>atomic instruction</td>
</tr>
</tbody>
</table>

- Lower-level locks are more efficient.
  - So why use higher-level locks?
### Which Lock to Use?

<table>
<thead>
<tr>
<th>Uniprocessor</th>
<th>Multiprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocking lock</td>
<td>blocking lock</td>
</tr>
<tr>
<td>interrupt disabling lock (only in kernel)</td>
<td>spin lock</td>
</tr>
<tr>
<td>interrupt disabling</td>
<td>atomic instruction</td>
</tr>
</tbody>
</table>

**When critical sections are long, may block**

**When critical sections are short, will not block**

**When critical sections perform atomic operation**

**When critical sections are long, may block**
Overview

- Which type of mutex lock to use?
- How to use mutex locks correctly?
Using a Mutex Lock

```
// counter is located in shared address space
int counter;
// mutex is located in shared address space
struct lock mutex; // same mutex used by both threads

while() {
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}

while() {
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}
```

Thread 1

Thread 2
Will This Code Work?

// counter is located in shared address space
int counter;
// mutex1 and mutex2 are located in shared address space
struct lock mutex1, mutex2;

while() {
    lock(&mutex1);
    // critical section;
    counter++;
    unlock(&mutex1);
    // remainder section;
}

while() {
    lock(&mutex2);
    // critical section;
    counter++;
    unlock(&mutex2);
    // remainder section;
}

Thread 1

Thread 2
Will This Code Work?

```c
// counter is located in shared address space
int counter;
// mutex1 and mutex2 are located in shared address space
struct lock mutex1, mutex2;

while() {
    lock(&mutex1);
    // critical section;
    counter++;
    unlock(&mutex1);
    // remainder section;
}

while() {
    lock(&mutex2);
    // critical section;
    counter--;
    unlock(&mutex2);
    // remainder section;
}
```

Thread 1

Thread 2
Will This Code Work?

```
// counter is located in shared address space
int counter;
// struct lock mutex; // commented out

while() {
    struct lock mutex;
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}
```

Thread 1

```
while() {
    struct lock mutex;
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
    // remainder section;
}
```

Thread 2
Will This Code Work?

```c
int counter;

update_counter() {
    struct lock mutex;
    lock(&mutex);
    // critical section;
    counter++;
    unlock(&mutex);
}

while() {
    update_counter();
    // remainder section
}
```

Thread 1

```c
while() {
    update_counter();
    // remainder section
}
```

Thread 2
Will This Code Work?

// counter is located in shared address space
int counter;
// mutex is located in shared address space
struct lock mutex; // same mutex used by both threads

while() {
  lock(&mutex);
  // critical section;
  counter++;
  unlock(&mutex);
  // remainder section;
}

while() {
  lock(&mutex);
  // critical section;
  if (counter <= 0)
    continue;
  counter--;
  unlock(&mutex);
}

Thread 1

Thread 2
Using Mutex Locks With Data Structures

- Say, 3 threads access a linked list
  - Thread 1 adds elements to the list
  - Thread 2 deletes elements from the list
  - Thread 3 reads an element from the list

- Questions:
  - Should Threads 1 and 2 use the same lock or different locks?
  - Should Thread 3 use a lock at all?
  - Should we create one lock for the entire list, or one lock per list node?
Summary

- Mutual exclusion is implemented using locks

- How to choose locks:
  - Use interrupt disabling and spin locks for short critical sections
  - Use blocking locks for long critical sections

- How to use locks correctly:
  - Use shared variable for locks
  - Use the same lock for all code that accesses a data structure
  - Use different locks for different data structures for more concurrency
    - Be careful: fine-grained locks can lead to more bugs
    - Discussed in next lecture
Think Time: To Block or Not

- When and why is a blocking lock better than interrupt disabling or using spin locks?
- Is the blocking lock always better?
- Why would a thread block within a critical section? Shouldn’t a critical section run fast, and so threads shouldn’t block within a critical section?
- Would it make sense to use a spin lock, rather than a blocking lock on a single CPU?
Think Time Answers: To Block or Not

- When and why is a blocking lock better than interrupt disabling or using spin locks?
  - Interrupt disabling is performed for the entire critical section. Similarly, spin locks are held for the entire critical section. With blocking locks, interrupts are disabled or spin locks are held only in the implementation of the lock and unlock operations, not for the entire critical section. This is especially beneficial when critical sections are long, e.g., if critical section is accessing the disk, then it would be best to put the thread to sleep and let other threads run on the same processor, which is accomplished with a blocking lock. Interrupt disabling would not allow other threads to run. Similarly, with spin locks, threads running on other CPUs, that try to enter the critical section will spin in the tset loop and keep using the CPU for a “very” long time, since disks are slow devices.
Is the blocking lock always better?

- Blocking locks have some overhead because they call the thread scheduler to perform a thread switch. When the critical section is short (say a few instructions), it is better to use interrupt disabling (on single CPU) or spin locks (on multi-processor) because the blocking lock has high overhead for short critical sections.

Why would a thread block within a critical section? Shouldn’t a critical section run fast, and so threads shouldn’t block within a critical section?

- We will see later that threads may block in a critical section because they need to synchronize with other threads.
Would it make sense to use a spin lock, rather than a blocking lock on a single CPU?

No. With a spin lock, if a lock is not available, the thread will keep spinning. During this time, this thread will make no progress, but it will also not allow the other thread that holds the lock to make any progress (this is different from multi-processors, where the thread holding the lock can make progress on another CPU). Eventually, the timer interrupt will fire, pre-empt the thread spinning on the held lock, and run the thread holding the lock, and then progress will occur. With blocking locks, the thread that tries to acquire a held lock with sleep right away, allowing the thread that holds the lock to run. So blocking locks are always better to use on a single CPU.
Think Time: Using Locks

- Why should local variables in a function not be used for locks?
- Can you use unlock before lock?
- Say you have two queues, A and B, that are accessed by concurrent threads. Should we use one lock for both queues, or a separate lock for each queue?
Think Time Answers: Using Locks

- Why should local variables in a function not be used for locks?
  - They are not shared between threads, and locks need to be shared.

- Can you use unlock before lock?
  - Lock need to be issued before unlock for mutual exclusion.

- Say you have two queues, A and B, that are accessed by concurrent threads. Should we use one lock for both queues, or a separate lock for each queue?
  - Two locks can be used, if there are no operations that need to operate on both the queues at the same time. Otherwise, a single lock needs to be used for both queues, e.g., if an element needs to be moved from one queue to another queue in a critical section.