Implementing Locks

Questions answered in this lecture:
- Why use higher-level synchronization primitives?
- What is a lock?
- How can locks be implemented?
- When to use spin-waiting vs. blocking?

Synchronization Layering

Build higher-level synchronization primitives in OS
- Operations that ensure correct ordering of instructions across threads
  - Motivation: Build them once and get them right
    - Don’t make users write entry and exit code

<table>
<thead>
<tr>
<th>Monitors</th>
<th>Locks</th>
<th>Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Condition Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable Interrupts</td>
</tr>
</tbody>
</table>

Locks

Goal: Provide mutual exclusion (mutex)
Three common operations:

Allocate and Initialize
- Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

Acquire
- Acquire exclusion access to lock; Wait if lock is not available
- Pthread_mutex_lock(&mylock);

Release
- Release exclusive access to lock
- Pthread_mutex_unlock(&mylock);

Lock Example

After lock has been allocated and initialized:

Void deposit(int amount) {
    Pthread_mutex_lock(&mylock);
    balance += amount;
    Pthread_mutex_unlock(&mylock);
}

Allocate one lock for each bank account:

Void deposit(int accountid, int amount) {
    Pthread_mutex_lock(&locks[accountid]);
    balance[accountid] += amount;
    Pthread_mutex_unlock(&locks[accountid]);
}
Implementing Locks: Version #1

Build locks using atomic loads and stores

```c
typedef struct {
    bool lock[2] = {false, false};
    int turn = 0;
} lockT;

void acquire(lockT *l) {
    l->lock[tid] = true;
    l->turn = 1-tid;
    while (l->lock[1-tid] && l->turn==l-tid) /* wait */;
}

void release(lockT *l) {
    l->lock[tid] = false;
}
```

Disadvantages??

Implementing Locks: Version #2

Turn off interrupts for critical sections
- Prevent dispatcher from running another thread
- Code executes atomically

```c
void acquire(lockT *l) {
    disableInterrupts();
}

void release(lockT *l) {
    enableInterrupts();
}
```

Disadvantages??

Implementing Locks: Version #3

Leverage atomic "Test of Lock" and "Set of Lock"

Hardware instruction: TestAndSet addr val (TAS)
- Returns previous value of addr and sets value at addr to val

Example: C=10;
Old = TAS(&C, 15)
Old == ?? C == ??

```c
typedef bool lockT;

void acquire(lockT *l) {
    while (TAS(l, true)) /* wait */;
}

void release(lockT *l) {
    *l = false;
}
```

Disadvantages??

Lock Implementation #4: Block when Waiting

```c
typedef struct {
    bool lock = false;
    bool guard = false;
    queue q = queue_alloc();
} LockT;

void acquire(LockT *l) {
    while (TAS(l->guard, true));
    if (!l->lock) {
        qadd(l->q, tid);
        l->guard = false;
        call dispatcher;
    } else {
        l->lock = true;
        l->guard = false;
    }
}

void release(LockT *l) {
    while (TAS(l->guard, true)) {
    if (qempty(l->q)) l->lock=false;
        else WakeFirstProcess(l->q);
        l->guard = false;
    }
```
How to stop Spin-Waiting?

Option 1: Add sleep(time) to while(TAS(&guard,1));
- Problems?

Option 2: Add yield()
- Problems?

Option 3: Don’t let thread give up CPU to begin with
- How?
- Why is this acceptable here?

Spin-Waiting vs Blocking

Each approach is better under different circumstances

Uniprocessor
- Waiting process is scheduled → Process holding lock isn’t
- Waiting process should relinquish processor
- Associate queue of waiters with each lock

Multiprocessor
- Waiting process is scheduled → Process holding lock might be
- Spin or block depends on how long, t, before lock is released
  - Lock released quickly → Spin-wait
  - Lock released slowly → Block
  - Quick and slow are relative to context-switch cost, C

When to Spin-Wait?
When to Block?

If know how long, t, before lock released, can determine optimal behavior

How much CPU time is wasted when spin-waiting?

How much wasted when block?

What is the best action when t<C?

When t>C?

Problem: Requires knowledge of future

Lock Implementation #5: Final Optimization

```c
void acquire(LockT *l) {
  while (TAS(&l->guard, true)) {
    if (l->lock) {
      qadd(l->q, tid);
      l->guard = false;
      call dispatcher;
    } else {
      l->lock = true;
      l->guard = false;
    }
  }
}

void release(LockT *l) {
  while (TAS(&l->guard, true)) {
    if (qempty(l->q)) l->lock = false;
    else WakeFirstProcess(l->q);
    l->guard = false;
  }
}
```
Two-Phase Waiting

Theory: Bound worst-case performance
When does worst-possible performance occur?

Spin-wait for $C$ then block $\rightarrow$ Factor of 2 of optimal

Two cases:
- $T < C$: optimal spin-waits for $t$; we spin-wait $t$ too
- $T > C$: optimal blocks immediately (cost of $C$); we pay spin $C$ then block (cost of $2C$)

Example of competitive analysis