CS5460: Operating Systems

Lecture 8: Critical Sections

(Chapter 6)

Synchronization

- When two (or more) processes (or threads) may have conflicting accesses to …
  - A memory location
  - A hardware device
  - A file or other OS-level resource
- … synchronization is required
- What makes two accesses conflict?
- Critical section is the most basic solution
  - What are some others?

Critical Section Review

- **Critical section: a section of code that only a single process / thread may be executing at a time**
- **Requirements:**
  - **Cannot allow multiple processes in critical section at the same time (mutual exclusion)**
  - **Ensure progress (lack of deadlock)**
  - **Ensure fairness (lack of livelock)**

Entry code (preamble)

Critical Section code; } **Need to ensure only one thread is ever in this region of code at a time**

Exit code (postscript)

Implementing Critical Sections

- **Critical sections build on lower-level atomic operations**
  - **Loads**
  - **Stores**
  - **Disabling interrupts**
  - **Etc.**

- **Critical section implementation is a bit tricky on modern processors**

Lock Example

- Spinlock from multicore Linux 3.2 for ARM
  - Spinlocks busy wait instead of blocking
  - This is the most basic sync primitive used inside an OS
- Remember: Out-of-order memory models break algorithms like Peterson
  - We're going to need help from the hardware
- ARM offers a powerful synchronization primitive

1. Issue a "load exclusive" to read a value from RAM and put the location into exclusive mode
2. Issue a matching "store exclusive"
   - » Successful if no other processor updated the location between the read and the write
   - » Fails if someone updated the location, and the store does not happen

- Often called "load-link / store-conditional"

- **Spinlock data structure:**

```
typedef struct {
    volatile unsigned int lock;
} arch_spinlock_t;
```

- **lock == 1 if the lock is held**
- **lock == 0 if the lock is free**

- **Why use a whole integer to store 1 bit?**

```
void arch_spin_lock (arch_spinlock_t *lock)
{
        unsigned long tmp;
        __asm__ __volatile__(
"1:     ldrex   %0, [%1]\n"
"       teq     %0, #0\n"
"       wfene\n"
"       strexeq %0, %2, [%1]\n"
"       teqeq   %0, #0\n"
"       bne     1b"
        : "=&r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc");

        smp_mb();
}
```

tmp = *lock

flag = (tmp == 0)

If (!flag) sleep

If (flag) strex (lock, 1)

If (flag), did strex return 0?

If strex failed, start over

```
void arch_spin_unlock
   (arch_spinlock_t *lock)
{
        smp_mb();
        __asm__ __volatile__(
"       str     %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");
}
```

- These functions are in: linux/arch/arm/include/asm/spinlock.h

How to Prove Correctness?

1. **Mutual exclusion**
   a. One process in critical section, another process tries to enter → Show that second process will block in entry code
   b. Two (or more) processes are in the entry code → Show that at most one will enter critical section
2. **Progress (== absence of deadlock)**
   a. No process in critical section, P1 arrives → P1 enters
   b. Two (or more) processes are in the entry code → Show that at least one will enter critical section
3. **Bounded waiting (== fairness)**
   a. One process in critical section, another process is waiting to enter → show that if first process exits the critical section and attempts to re-enter, show that waiting process will be get in

Proving Mutual Exclusion

- One in CS and another wants in → show second will wait
  - Assume P0 in CS, P1 tries to enter
  - What happens?
- Multiple in entry → at most one reaches critical section
  - Assume P0 and P1 try to enter
  - Can both succeed?

```
flag[2] ← {0,0};
turn ← 0;
```

What happens here?

Entry code:

```
flag[me] ← 1;
turn ← them;
while (flag[them] &&
       turn != me);
```

Critical section:

```
Milk ← Milk + 1;
```

P0 here

Exit code:

```
flag[me] ← 0;
```

Proving Mutual Exclusion

- One in CS and another wants in → show second will block
  - Assume P0 in CS, P1 tries to enter
  - What happens?

- **Multiple in entry → <u>at most</u> one reaches critical section**
  - **Assume P0 and P1 try to enter**
  - **Can both succeed?**

```
flag[2] ← {0,0};
turn ← 0;
```

<u>Entry code</u>: **P0 and P1 here…**

```
flag[me] ← 1;
turn ← them;
while (flag[them] &&
       turn != me);
```

<u>Critical section</u>:

```
Milk ← Milk + 1;
```

Can both get here?

<u>Exit code</u>:

```
flag[me] ← 0;
```

Proving Progress

- **No process in critical section → show arriver will always enter**
  - **Nobody in CS and P0 arrives**
  - **What happens?**
- **Multiple in entry → at least one will enter**
  - **Assume P0 and P1 try to enter**
  - **Will at least one succeed?**

```
flag[2] ← {0,0};
turn ← 0;
```

Entry code: **P0 tries to enter…**

```
flag[me] ← 1;
turn ← them;
while (flag[them] &&
       turn != me);
```

Critical section:

```
Milk ← Milk + 1;
```

Will it?

Exit code:

```
flag[me] ← 0;
```

Proving Progress

- No process in critical section → show arriver will always enter
  - Nobody in CS and P0 arrives
  - What happens?

- **Multiple in entry → at least one will enter**
  - **Assume P0 and P1 try to enter**
  - **Will at least one succeed?**

```
flag[2] ← {0,0};
turn ← 0;
```

Entry code: **P0 and P1 here…**

```
flag[me] ← 1;
turn ← them;
while (flag[them] &&
       turn != me);
```

Critical section:

```
Milk ← Milk + 1;
```

Will one get here?

Exit code:

```
flag[me] ← 0;
```

Proving Bounded Wait

- One in CS, another waiting → if 1st exits and tries to re-enter, 2nd will succeed first
  - Assume P0 in CS and P1 waiting
  - P0 exits CS and tries to re-enter
  - Does P1 get in before P2?

```
flag[2] ← {0,0};
turn ← 0;
```

Entry code:

P1 here…

```
flag[me] ← 1;
turn ← them;
while (flag[them] &&
       turn != me);
```

Critical section:

```
Milk ← Milk + 1;
```

P0 starts here…

Exit code:

```
flag[me] ← 0;
```

Proving Correctness

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

Entry code:

```
while (Note) {};
Note ← 1;
```

Critical section:

```
BuyMilk();
```

Exit code:

```
Note ← 0;
```

Which of these requirements does this solution fail?

Milk V.1

Implementing Critical Sections

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
flag[2] = {0,0};
```

Entry code:

```
while (flag[them]) {};
flag[me] ← 1;
```

Critical section:

```
BuyMilk();
```

Exit code:

```
flag[me] ← 0;
```

Which of these requirements does this solution fail?

Milk V.2

Implementing Critical Sections

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
flag[2] = {0,0};
```

Entry code:

```
flag[me] ← 1;
while (flag[them]) {};
```

Critical section:

```
BuyMilk();
```

Exit code:

```
flag[me] ← 0;
```

Which of these requirements does this solution fail?

Milk V.3

Implementing Critical Sections

- **Mutual exclusion**
  - One in CS → another that tries to enter fails
  - Two or more try to enter → at most one can succeed
- **Progress**
  - Nobody waiting and one arrives → it will succeed
  - Two or more try to enter → at least one will succeed
- **Bounded wait**
  - One inside and one waiting, 1st exits → 2nd will enter before 1st

```
turn ← 0;
```

Entry code:

```
while (turn != me) {};
```

Critical section:

```
BuyMilk();
```

Exit code:

```
turn ← them;
```

Which of these requirements does this solution fail?

Milk V.4

Bakery Algorithm (N-thread CS)

```
int choosing[N]; // choosing[i] iff Pi in entry protocol
int number[N];   // value of "ticket" that Pi chooses
```

Entry code:

```
choosing[i] = 1;
number[i] = max(number[0], …, number[N-1]) + 1;
choosing[i] = 0;
for (j = 0; j < N; j++) {
    while (choosing[j]);
    while ((number[j] != 0) && ((number[j],j) < (number[i],i)));
}
```

(a,b) < (c,d) →
(a<c) || (a==c && b<d)

Critical Section!

Exit code:

```
number[i] = 0;
```

Key problem:

Cannot guarantee that no duplicates are selected →
proof of correctness must account for this case

Proof of Correctness (Sketch)

- Need to "prove" each of the five subcases
- Key part of proof:
  - If Pi is in the CS → for all Pj (j != i) that have already chosen a number:

    (number[i], i) < (number[j], j)

  - Use this to show mutual exclusion
- Remainder of proof:
  - Progress: Show that it is FCFS
  - Bounded wait: Show that it is FCFS
- You might get to do the full proof in a homework…

- **Why are we looking at stuff like Bakery if it fails on modern multicores?**
  1. **The concepts are important**
  2. **Can usually fix an existing algorithm using memory barriers and other help from the hardware**
  3. **The old algorithms still apply on less aggressive processors**

```
void arch_spin_lock (arch_spinlock_t *lock)
{
        unsigned long tmp;
        __asm__ __volatile__(
"1:     ldrex   %0, [%1]\n"
"       teq     %0, #0\n"
"       wfene\n"
"       strexeq %0, %2, [%1]\n"
"       teqeq   %0, #0\n"
"       bne     1b"
        : "=&r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc");

        smp_mb();
}
```

tmp = *lock

flag = (tmp == 0)

If (!flag) sleep

If (flag) strex (lock, 1)

If (flag), did strex return 0?

If strex failed, start over

Important Concepts

- Mutual exclusion
- Race conditions
- Meaning of Lock(L) and Unlock(L)
- Requirements for correct synchronization:
  - Mutual exclusion
  - Progress
  - Bounded wait (fairness)
- Proof of correctness for critical sections
- Basic solutions to the critical section problem:
  - 2-process solution: Peterson
  - N-process solution: Bakery

Questions?