Last lecture on concurrency

→ Please read and follow Mike Dahlin’s coding standard

→ Today

→ Finishing up

→ Chance to ask question about concurrency.

So far

- Critical sections
- Mutexes
- Condition Variables
- Semaphores
- Building mutexes
  → Spinlock (Compare-and-swap)
  → Wait queue

Safety concerns
- Unsafe concurrent access

get(x)

\[ \text{section } x \rightarrow \]

inc = \[ x ( \cdot ) \]

\[ a = q(m) \]

\[ x = 2 + x' \]

\[ x += 1 \]

\[ \text{rel}(n) \]

- Deadlocks

\( t_0 \leftrightarrow t_1 \)

\( \text{wait on} \)

Performance Concerns

- What is desirable

- Correctness

- Maximize concurrency

- When possible allow as many threads to run as possible

- Bounded waiting

  \( \Rightarrow \) No one should wait too long

- Fairness (?)

  \( \Rightarrow \) All threads have equal chance to execute critical section

- Minimize overheads when using mutexes, cond vars, etc.

- Concerns

  For more details on coarse-grained locking
Fine-grained vs coarse-grained

head → [ ] → [ ] → [ ] → ....

insert
find
delete

head → [ ] → [ ] → [ ] → [ ] → ....

S
find

S
insert

S
find(2)
S
insert(6)

Σ
1

head

head

Σ
1

1, 2, 3
- Priority Inversion
- How to handle
- Starvation
Locks are Expensive

Programmability Issues
- Loss of modularity

void f(...) { 
    ... 
}

Cond-wait

void g(...) { 
    ... 
    x = 22; 
}
The coding guidelines are meant to avoid problems due to lack of modularity.

Getting Started

1. Identify shared data
   - Decide how to split it (granularity) into objects that a single thread accesses at a time.

2. Associate a monitor (1 mutex + 1 or more C.V.) with each object.

3. Acquire the mutex when entering a function that accesses the object
   - Release it when returning

4. Use C.V. when waiting for some condition to hold
\( \textbf{L} \to \text{use while (condition) \& cond-wait} \)

\( \Downarrow \)

\( \text{⑤ DO NOT USE SLEEP!} \)

ASSUMPTIONS ABOUT EXECUTION ORDER

- We assumed Seq Cst but...

- Compiler reordering

- Reality
1. Simple deadlock example

T1:
acquire(mutexA);
acquire(mutexB);
// do some stuff
release(mutexB);
release(mutexA);

T2:
acquire(mutexB);
acquire(mutexA);
// do some stuff
release(mutexA);
release(mutexB);

2. More subtle deadlock example

Let M be a monitor (shared object with methods protected by mutex)
Let N be another monitor

class M {
    private:
        Mutex mutex_m;
        // instance of monitor N
        N another_monitor;
        // Assumption: no other objects in the system hold a pointer
        // to our "another_monitor"
    public:
        M();
        ~M();
        void methodA();
        void methodB();
    }

class N {
    private:
        Mutex mutex_n;
        Cond cond_n;
        int navailable;
    public:
        N();
        ~N();
        void* alloc(int nwanted);
        void free(void* returning_mem);
    }

int N::alloc(int nwanted) {
    acquire(&mutex_n);
    while (navailable < nwanted) {
        wait(&cond_n, &mutex_n);
    }
    // peel off the memory
    navailable -= nwanted;
    release(&mutex_n);
}

void N::free(void* returning_mem) {
    acquire(&mutex_n);
    // put the memory back
    navailable += returning_mem;
    broadcast(&cond_n, &mutex_n);
    release(&mutex_n);
}
void M::methodA() {
    acquire(&mutex_m);
    void* new_mem = another_monitor.alloc(int nbytes);
    // do a bunch of stuff using this nice
    // chunk of memory n allocated for us
    release(&mutex_m);
}

void M::methodB() {
    acquire(&mutex_m);
    // do a bunch of stuff
    another_monitor.free(some_pointer);
    release(&mutex_m);
}

QUESTION: What’s the problem?
The point is: fine-grained locking leads to complexity.

...
Finally, here are some references on this topic:

  explores issues with this pattern in C++

  The "Double-Checked Locking is Broken" Declaration:

  C++11 provides a way to implement the pattern correctly and portably (again, using memory barriers):