Synchronization

CSCI 444/544 Operating Systems
Fall 2008

Agenda

• Why is synchronization necessary?

• What are race conditions, critical sections, and atomic operations?

• How to protect critical sections with atomic loads and stores?

• Synchronization primitives
  – locks

Synchronization

Threads cooperate in multithreaded programs
• to share resources, access shared data structures
  – e.g., threads accessing a memory cache in a web server
  – data consistency must be maintained
• to coordinate their execution
  – e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer
  – ensure the orderly execution of cooperation processes

Cooperation requires Synchronization

Example:
Two threads share account balance in memory
Each runs common code, deposit()
void deposit (int amount) {
  balance = balance + amount;
}
Compile to sequence of assembly instructions
load R1, balance
add R1, amount
store R1, balance

Which variables are shared? Which private?
**Concurrent Execution**

What happens if 2 threads deposit concurrently?
- Assume any interleaving of instructions is possible
- Make no assumptions about scheduler

Initial balance: $100
Thread 1: deposit(10)  
Thread 2: deposit(-5)

<table>
<thead>
<tr>
<th>Load R1, balance</th>
<th>Load R1, balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add R1, amount</td>
<td>Add R1, amount</td>
</tr>
<tr>
<td>Store R1, balance</td>
<td>Store R1, balance</td>
</tr>
</tbody>
</table>

What is the final balance?

---

**The Crux of the Matter**

The problem is that two concurrent threads (or processes) access a shared resource (account) without any synchronization
- creates a race condition
  - output is non-deterministic, depends on timing

We need mechanisms for controlling access to shared resources in the face of concurrency
- so we can reason about the operation of programs
  - essentially, re-introducing determinism

Synchronization is necessary for any shared data structure
- buffers, queues, lists, hash tables, ...

---

**Definitions**

| Critical section: a section of code which reads or writes shared data |
| Race condition: result depends upon ordering of execution |
| • Potential for interleaved execution of a critical section by multiple threads |
| • Non-deterministic bug, very difficult to find |
| Mutual exclusion: synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections |
| Deadlock: permanent blocking of threads |
| Livelock: execution but no progress |

---

**Critical Section Requirements**

Critical sections have the following requirements
- mutual exclusion
  - at most one thread is in the critical section
- progress (deadlock-free and livelock-free)
  - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
- bounded waiting (no starvation)
  - if thread T is waiting on the critical section, then T will eventually enter the critical section
  - assumes threads eventually leave critical sections
  - vs. fairness?
- performance
  - the overhead of entering and exiting the critical section is small with respect to the work being done within it
Mutual Exclusion

- Only one thread at a time can execute in the critical section
- Execution time is finite
- All other threads are forced to wait on entry
- When a thread leaves a critical section, another can enter
- A process/thread not in the critical section cannot prevent others to enter
- Waiting time is limited: no deadlock or starvation

Implementing Critical Sections

To implement, need atomic operations

**Atomic operation:** No other instructions can be interleaved

Examples of atomic operations

- Loads and stores of words
  - Load r1, B
  - Store r1, A
- Special hw instructions
  - Test&Set
  - Compare&Swap

Critical Section: Attempt #1

Code uses a single shared lock variable

```plaintext
Boolean lock = false; // shared variable
Void deposit(int amount) {
    while (lock) /* wait */ ;
    lock = true;
    balance += amount; // critical section
    lock = false;
}
```

Why doesn’t this work? Which principle is violated?

Attempt #2

Each thread has its own lock; lock indexed by tid (0, 1)

```plaintext
Boolean lock[2] = {false, false}; // shared
Void deposit(int amount) {
    lock[tid] = true;
    while (lock[1-tid]) /* wait */ ;
    balance += amount; // critical section
    lock[tid] = false;
}
```

Why doesn’t this work? Which principle is violated?
Attempt #3

Turn variable determines which thread can enter
Int turn = 0; // shared
Void deposit(int amount) {
    while (turn == 1-tid) /* wait */ ;
    balance += amount; // critical section
    turn = 1-tid;
}
Why doesn’t this work? Which principle is violated?

Peterson's Algorithm: Solution for Two Threads

Combine approaches 2 and 3: Separate locks and turn variable
Int turn = 0; // shared
Boolean lock[2] = {false, false};
Void deposit(int amount) {
    lock[tid] = true;
    turn = 1-tid;
    while (lock[1-tid] && turn == 1-tid) /* wait */ ;
    balance += amount; // critical section
    lock[tid] = false;
}

Peterson's Algorithm: Intuition

Mutual exclusion: Enter critical section if and only if
• Other thread does not want to enter
• Other thread wants to enter, but your turn
Progress: Both threads cannot wait forever at while() loop
• Completes if other process does not want to enter
• Other process (matching turn) will eventually finish
Bounded waiting
• Each process waits at most one critical section

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

Monitors Semaphores
Locks
Loads Stores Test&Set
Disable Interrupts
Mechanisms for building critical sections

Locks
- very primitive, minimal semantics; used to build others

Semaphores
- basic, easy to get the hang of, hard to program with

Monitors
- high level, requires language support, implicit operations
- easy to program with; Java "synchronized()" as an example

Locks

A lock is an object (in memory) that provides the following two operations:
- acquire(): a thread calls this before entering a critical section
- release(): a thread calls this after leaving a critical section

Threads pair up calls to acquire() and release()
- between acquire() and release(), the thread holds the lock
- acquire() does not return until the caller holds the lock
- at most one thread can hold a lock at a time
- so: what can happen if the calls aren’t paired?

Two basic flavors of locks
- spinlock
- blocking (a.k.a. "mutex")

Spinlocks

How do we implement locks? Here’s one attempt:

```c
struct lock {
    int held = 0;
}

void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}

void release(lock) {
    lock->held = 0;
}
```

Why is the problem?
- where is the race condition?

Implementing locks

Problem is that implementation of locks has critical sections, too!
- the acquire/release must be atomic
  - atomic == executes as though it could not be interrupted
  - code that executes "all or nothing"

Need help from the hardware
- atomic instructions
  - test-and-set, compare-and-swap, ...
- disable/reenable interrupts
  - to prevent context switches
Spinlocks redux: Test-and-Set

Now spinlocks

```c
struct lock {
    int held = 0;
}
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```

Problem with Spinlocks

Spinlocks work, but are wasting CPU times
- if a thread is spinning on a lock, the thread holding the lock cannot make progress (does not make much sense in uniprocessor scenario)
- On multiprocessor, completely waste the time of the requesting CPU

How does a thread blocked on an "acquire" (that is, stuck in a test-and-set loop) yield the CPU?
- calls yield() (spin-then-block)
- Add sleep(time) to while(test_and_set(&lock->held));
- there’s an involuntary context switch

Mutex (blocking)

Disabling interrupts

```c
struct lock {
    }
void acquire(lock) {
    cli(); // disable interrupts
}
void release(lock) {
    sti(); // reenable interrupts
}
```

• On uniprocessor, operation is atomic as long as context switch doesn’t occur in the middle of the operation

Problems with disabling interrupts

Only available to the kernel
- Can’t allow user-level to disable interrupts!

Insufficient on a multiprocessor
- Each processor has its own interrupt mechanism

“Long” periods with interrupts disabled can wreak havoc with devices
Spin-waiting vs Blocking

Uniprocessor: normally use Blocking

Multiprocessor: depend on
- how long the lock will be released (T)
- the overhead of context-switch (O)
  - Lock released quickly -> spin-waiting
  - Lock released slowly -> block
  - quick and slow (T) are relative to O