Critical Sections and Race Conditions

Administrivia

Install and build Nachos if you haven’t already.
http://www.cs.duke.edu/~anderson/110/building.html

“I need more space to build Nachos.”
The ball is in ACPUB’s court.
Until then, only build in the threads directory (needs 1.2 MB).

Lab #1 is due ...
Follow submission instructions in the project guide.
E-mail your project group info before the deadline!

Reading:
Nutt: chapter 8 (Synchronization)
Birrell, “An Introduction to Programming with Threads”
Warmup

1. What does it mean for threads to execute “concurrently”? How does this happen?
2. Why/when do context switches occur?
3. What happens on a context switch?
4. What causes programs to “crash”?
5. If a multithreaded program “crashes” and I show a stack trace in \texttt{gdb}, what will I see?
6. Can I use \texttt{gdb} to look at all the threads in my program?
7. What happens if I delete a thread while it’s running?
8. What happens if a thread makes recursive calls “forever”?

Race Conditions Defined

1. Every data structure defines \textit{invariant} conditions.
   defines the space of possible \textit{legal} states of the structure
   defines what it means for the structure to be “well-formed”
2. Operations depend on and preserve the invariants.
   The invariant must hold when the operation begins.
   The operation may temporarily violate the invariant.
   The operation restores the invariant before it completes.
3. Arbitrarily interleaved operations violate invariants.
   Rudely interrupted operations leave a mess behind for others.
4. Therefore we must constrain the set of possible schedules.
Resource Trajectory Graphs

Resource trajectory graphs (RTG) depict the thread scheduler's "random walk" through the space of possible system states.

RTG for N threads is N-dimensional.

Thread $i$ advances along axis $I$.

Each point represents one state in the set of all possible system states.

cross-product of the possible states of all threads in the system

(But not all states in the cross-product are legally reachable.)

Avoiding Races #1

1. Identify *critical sections*, code sequences that:
   - rely on an invariant condition being true;
   - temporarily violate the invariant;
   - transform the data structure from one legal state to another;
   - or make a sequence of actions that assume the data structure will not "change underneath them".

2. *Never sleep or yield in a critical section.*
   Voluntarily relinquishing control may allow another thread to run and "trip over your mess" or modify the structure while the operation is in progress.
Critical Sections in the Color Stack

```c
InitColorStack() {  
    push(blue);  
    push(purple);  
}

PushColor() {  
    if (s[top] == purple) {  
        ASSERT(s[top-1] == blue);  
        push(blue);  
    } else {  
        ASSERT(s[top] == blue);  
        ASSERT(s[top-1] == purple);  
        push(purple);  
    }  
}
```

OK, so we never `Sleep` or `Yield` in a critical section.

Is that sufficient to prevent races?
Avoiding Races #2

Is caution with *yield* and *sleep* sufficient to prevent races?
No!

Concurrency races may also result from:
- involuntary context switches (timeslicing) e.g., caused by the Nachos thread scheduler with -rs flag
- interrupts (inside the kernel) or signals (outside the kernel)
- physical concurrency (on a multiprocessor)

How to ensure atomicity of critical sections in these cases?
Synchronization primitives!

Mutual Exclusion and Locks
**Administrivia**

Lab #1 is due at midnight.
- Carefully follow submission instructions in the project guide.
- E-mail your project group info before midnight.
- You may continue to work on projects after the deadline for less-than-full credit (tell us what you did).

A new experiment: Web-based signup for demos.
- Look for it on the course Web page.
- At least one group member must have a password (see Rajiv).

**Reading:**
- Nutt: chapter 8 (Synchronization)
- Birrell, “An Introduction to Programming with Threads”

**Avoiding Races**

1. Don’t *yield* or *sleep* in a critical section.
   - Necessary but not sufficient...

2. Concurrency races may also result from:
   - involuntary context switches (timeslicing)
     - e.g., caused by the Nachos thread scheduler with -rs flag
   - interrupts (inside the kernel) or signals (outside the kernel)
   - physical concurrency (on a multiprocessor)

How to ensure atomicity of critical sections in these cases?
- Synchronization primitives!
Mutual Exclusion

Race conditions can be avoiding by ensuring *mutual exclusion* in critical sections.

- Critical sections are code sequences that contribute to races.
  
  Every race (possible incorrect interleaving) involves two or more threads executing related critical sections concurrently.

- To avoid races, we must *serialize* related critical sections.
  
  Never allow more than one thread in a critical section at a time.

Locks

*Locks* can be used to ensure mutual exclusion in conflicting critical sections.

- A lock is an object, a data item in memory.
  
  Methods: `Lock::Acquire` and `Lock::Release`.

- Threads pair calls to `Acquire` and `Release`.
- `Acquire` before entering a critical section.
- `Release` after leaving a critical section.
- Between `Acquire/Release`, the lock is *held*.
- `Acquire` does not return until any previous holder releases.
- Waiting locks can spin (a *spinlock*) or block (a *mutex*).
/* shared by all threads */
int counters[N];
int total;

/*
* Increment a counter by a specified value, and keep a running sum.
* This is called repeatedly by each of N threads.
* tid is a thread identifier unique across all threads.
* value is just some arbitrary number.
*/
void
TouchCount(int tid, int value)
{
    counters[tid] += value;
    total += value;
}
Using a Lock for the Counter/Sum Example

```c
int counters[N];
int total;
Lock *lock;

/*
 * Increment a counter by a specified value, and keep a running sum.
 */
void
TouchCount(int tid, int value)
{
    lock->Acquire();
    counters[tid] += value; /* critical section code is atomic...*/
    total += value; /* ...as long as the lock is held */
    lock->Release();
}
```

Reading Between the Lines of C

```c
/*
    counters[tid] += value;
    total += value;
*/
load counters, R1 ; load counters base
load 8(SP), R2 ; load tid index
shl R2, #2, R2 ; index = index * sizeof(int)
add R1, R2, R1 ; compute index to array
load 4(SP), R3 ; load value
load (R1), R2 ; load counters[tid]
add R2, R3, R2 ; counters[tid] += value
store R2, (R1) ; store back to counters[tid]
load total, R2 ; load total
add R2, R3, R2 ; total += value
store R2, total ; store total
```
The Need for an Atomic Primitive

To implement safe mutual exclusion, we need support for some sort of “magic toehold” for synchronization.

- The lock and mutex primitives themselves have critical sections to test and/or set the lock flags.
- These primitives must somehow be made *atomic*.
  - uninterruptible
    - a sequence of instructions that execute “all or nothing”