

6.828 2014 : Locking

Alarm homework

recall: call user process "alarm handler" every X timer interrupts
 timer interrupts arrive in trap()

Q: how to transfer execution to handler function?

Q: where is user EIP stored while in trap?

Q: how did user EIP get there?

Q: when will handler start executing?

Q: how to have handler return to interrupted user instructions?

Q: on what stack should handler execute (user or kernel)?

Q: why not call handler directly from trap()? e.g. (*proc->alarmhandler)()
 ("works" but write() sys call arguments pushed on wrong stack)

Q: (challenge) security attack if kernel doesn't check tf->esp?

Q: (challenge) how to save/restore caller-saved registers?

Why talk about locking?

kernels are full of shared data (buffer cache, process list, &c)

how to access shared data from multiple cores?

locks help sort concurrent access

big source of bugs

critical to getting parallel speedup => high performance

Locking homework

recall: ph.c, multi-threaded hash table, put(), get()

Q: why run ph.c on multiple cores?

diagram: CPUs, bus, RAM

assumption: CPU is bottleneck, can divide work between CPUs

Q: how to measure parallel speedup?

Q: where are the missing keys?

Q: specific scenario?

diagram...

table[0] = 10, 25

concurrent put(15), put(20)

both insert()s allocate new entry, point next to 25

both set 10->next to their new entry

last inserter wins, other one is lost!

called a "race" or a "lost update"

Q: where to put the lock/unlock?

Q: one lock covering the whole hash table?

why? why not?

called a "big" or "coarse-grained" lock

Q: one lock per table[] entry?

this lock is "finer grained"

what might be harder with this per-row lock?

Q: one lock per struct entry, protecting the next pointer?

why? why not?

this lock is even more fine-grained

tricky: 20 minutes wasn't enough to get my version of this right

Q: should get() also lock? if so, how?

Q: will per-row locks show parallel speedup w/ 10 cores? NBUCKETS=5...

The lock abstraction:

lock l

acquire(l)

x = x + 1 -- "critical section"

release(l)

a lock is itself an object

only one core can hold a given lock

acquire() forces other cores to "block" waiting for a held lock

a program can have lots of locks

different locks can be held at same time

so cores can operate on different data in parallel

note that lock l is not specifically tied to data x

the program keeps track of the correspondence
this kind of locking is popular in kernels but not the only possibility!

One way to think about locking

locks are part of a method for writing correct parallel code

1. programmer writes "serial" code
code that is correct assuming only one CPU
`int x; void inc() { x = x + 1; }`
but perhaps not correct if executed in parallel
2. programmer add locks to FORCE serial execution
since acquire/release allows execution by only one CPU at a time
the point:
it's easier for programmers to reason about serial code
locks can cause your serial code to be correct despite parallelism

When do you need locks?

any time two cores use a memory location, and at least one is a write
rule: don't touch data unless you hold the right lock!

Could locking be automatic?

perhaps the language could associate a lock with every data object
compiler adds acquire/release around every use
less room for programmer to forget!

that idea is often too rigid:

```
rename("d1/x", "d2/y"):
    lock d1, erase x, unlock d1
    lock d2, add y, unlock d2
```

problem: the file didn't exist for a while!

rename() should be atomic

other system calls should see before, or after, not in between
otherwise too hard to write programs

we need:

```
lock d1 ; lock d2
erase x, add y
unlock d2; unlock d1
```

that is, programmer often needs explicit control over
the region of code during which a lock is held
in order to hide awkward intermediate states

More ways to think about locks

locks help you create atomic multi-step operations
locks help operations maintain invariants on a data structure
assume the invariants are true at start of operation
operation uses locks to hide temporary violation of invariants
operation restores invariants before releasing locks

Problem: deadlock

notice rename() held two locks

what if:

core A	core B
rename(d1/x, d2/y)	rename(d2/a, d1/b)
lock d1	lock d2
lock d2 ...	lock d1 ...

solution:

programmer works out an order for all locks
all code must acquire locks in that order
i.e. predict locks, sort, acquire -- complex!

Lock granularity

you often have many choices about lock granularity
whole `ph.c` table; each `table[]` row; each entry
whole FS; directory/file; disk block
whole kernel; each subsystem; each object
big locks get you simplicity (= fewer bugs)
less deadlock since less opportunity to hold two locks

less reasoning about invariants/atomicity required
 big locks are sometimes enough
 if chance of parallelism in that code was low
 e.g. Big Kernel Lock OK if most CPU time spent in user code
 fine-grained locks can increase parallel performance
 e.g. processes can write different files in parallel on different cores

Modularity

locks make it hard to hide details inside modules
 to avoid deadlock, I need to know locks acquired by functions I call
 for fine-grained locks, I may need to acquire locks before calling functions
 finer grain -> more ugly

Advice:

start with big locks
 instrument your code -- which locks are preventing parallelism?
 use fine-grained locks only as needed for parallel performance

Let's switch to locking in xv6.

A use of locks: ide.c / iderw()

typical of many O/S's device driver arrangements
 diagram:

```

  user processes, kernel, FS, iderw, append to disk queue
  IDE disk hardware
  ideintr

```

sources of concurrency: processes, interrupt

only one lock in ide.c: idelock

what does idelock protect?

1. no races in idequeue operations
2. IDE h/w always executing head of idequeue
3. IDE h/w only executing one operation
4. only one process touching IDE registers
5. a process is waiting for each buffer in queue

now ideintr(), interrupt handler

acquires lock -- might have to wait at interrupt level!

uses idequeue (1)

touches IDE h/w registers (3, 4)

wakes waiting process (5)

hands next queued request to IDE h/w (2, 4)

How to implement locks?

what is wrong with this:

```

struct lock { int locked; }
acquire(l) {
  while(1){
    if(l->locked == 0){ // A
      l->locked = 1;    // B
      return;
    }
  }
}

```

oops: race between lines A and B

how can we do A and B atomically?

Atomic exchange instruction:

```
mov $1, %eax
```

```
xchg %eax, addr
```

does this in hardware:

lock addr globally (other cores cannot use it)

```
temp = *addr
```

```
*addr = %eax
```

```
%eax = temp
```

```
unlock addr
```

x86 h/w provides a notion of locking a memory location

diagram: cores, bus, RAM, lock thing
 so we are really pushing the problem down to the hardware
 h/w implements at granularity of cache-line or entire bus
 memory lock forces concurrent xchg's to run one at a time, not interleaved

Now:

```
acquire(l){
  while(1){
    if(xchg(&l->locked, 1) == 0){
      break
    }
  }
}
```

if l->locked was already 1, xchg sets to 1 (again), returns 1,
 and the loop continues to spin
 if l->locked was 0, at most one xchg will see the 0; it will set
 it to 1 and return 0; other xchgs will return 1
 this is a "spin lock", since waiting cores "spin" in acquire loop

Look at xv6 spinlock implementation

spinlock.h -- you can see "locked" member of struct lock
 spinlock.c / acquire():
 see while-loop and xchg() call
 what is the pushcli() about?
 why disable interrupts?
 release():
 uses xchg() to set lk->locked = 0
 re-enables interrupts

Detail: memory ordering

suppose two cores use a lock to guard a counter, x

Core A:	Core B:
locked = 1	
x = x + 1	while(locked == 1)
locked = 0	...
	locked = 1
	x = x + 1
	locked = 0

the compiler AND the CPU re-order memory accesses

e.g. the compiler might generate this code for core A:

```
locked = 1
locked = 0
x = x + 1
```

i.e. move the increment outside the critical section!

call to xchg() tells compiler and x86 not to re-order:

intel promises not to re-order past xchg instruction

some junk in x86.h xchg() tells C compiler not to delete or re-order

(volatile asm says don't delete, "m" says no re-order)

thus the xchg() in release()

learn the memory model rules if you like pushing the envelope

to write tricky and efficient parallel code

but the rules are too complex for me to remember

Why spin locks?

don't they waste CPU while waiting?

why not give up the CPU and switch to another process, let it run?

what if holding thread needs to run; shouldn't you yield CPU?

spin lock guidelines:

hold spin locks for very short times

don't yield CPU while holding a spin lock

systems often provide "blocking" locks for longer critical sections

waiting threads yield the CPU

but overheads are typically higher

you'll see some xv6 blocking schemes later

Remember:

don't share if you don't have to
use locks when multiple cores might r/w same data at same time
choice:
 coarse-grained, simple, slow
 fine-grained, complex, fast