Operating Systems Engineering

Locking & Synchronization
[chapter #4]

Dan Tsafrir, 2017-03-29
Granularity

◆ The manner by which we synchronize
  ❖ Depends on the granularity of the computation
  ❖ “Granularity is the extent to which a material or system is composed of distinguishable pieces or grains. It can either refer to the extent to which a larger entity is subdivided, or the extent to which groups of smaller indistinguishable entities have joined together to become larger distinguishable entities. For example, a kilometer broken into centimeters has finer granularity than a kilometer broken into meters.”
◆ Fine
  ❖ Short (i) critical sections and/or (ii) wait times to gain access to those sections
◆ Coarse
  ❖ Opposite of fine
When a critical section is short

FINE-GRAINED SYNCHRONIZATION
Agenda: mutex locks

❖ Locks
  ❖ Why do we need them?
  ❖ How do we implement them? (In xv6 in particular)
  ❖ How do we use them?
Why locks?

◆ **Multicores**
  ❖ Goal – improve performance
  ❖ Means – concurrency

◆ **Problem**
  ❖ Simultaneous use of the same data structures might cause incorrect results

◆ **Solution**
  ❖ For short critical sections
    ▪ Sync to serialize access with *spinlocks*
Example

```
struct List {
    int data;
    struct List *next;
};

List *list = 0;

insert(int data) {
    List *l = new List;
    l->data = data;
    l->next = list;
    list = l;
}
```

- Works correctly when serial
- Breaks when two cores doing it simultaneously
Race condition

Core #1:

```c
insert(int data) { // data=3
    List *l = new List;
    l->data = data;
    l->next = list; // = null
    list = l; // list = [3 -> null]
}
```

Core #2:

```c
insert(int data) { // data=2
    List *l = new List;
    l->data = data;
    l->next = list; // = null
    list = l; // list = [2 -> null]
}
```
Race condition

◆ Why “race”?
  ❖ Nondeterministic nature
  ❖ “Heisenbug” – SW bug that seems to disappear or alter its behavior when one attempts to study it
  ❖ E.g., debug print might eliminate race or introduce a new one
  ❖ Nondeterminism makes it very hard to debug

◆ How to avoid races?
  ❖ Introduce a lock to protect a collection of invariants
    ▪ Data structure properties that are maintained across ops
    ▪ Invariant of linked lists:
      – ‘list’ points to head + each ‘next’ points to next element
  ❖ Op might temporarily violate invariants
    ▪ Which is why we need it to be (logically) atomic
Serialize with lock abstraction

```c
insert(int data) {
    acquire( &list_lock );
    List *l = new List;
    l->data = data;
    l->next = list;
    list = l;
    release( &list_lock )
}
```

- Associate a lock object with the list
- 2 ops + 2 matching states:
  - acquire => “locked”
  - release => “unlocked”
- Semantics:
  - Only one thread can ‘acquire’ lock
  - Other threads must wait until it’s ‘release’-d
Serialize with lock abstraction

```c
insert(int data) {
    List *l = new List;
    l->data = data;
    acquire( &list_lock );
    l->next = list;
    list = l;
    release( &list_lock )
}
```

- Why would we want to do this?
- Would it do the job?
LET’S TRY TO IMPLEMENT THE ABSTRACTION
Acquire & Release (broken)

```c
struct Lock {
    int locked; // 0=free; 1=taken
};

void broken_acquire(Lock *lock) {
    while(1) {
        if(lock->locked == 0) {
            lock->locked = 1;
            break;
        }
    }
}

void release (Lock *lock) {
    lock->locked = 0;
}
```
Acquire & Release (broken)

```c
struct Lock {
    int locked; // 0=free; 1=taken
};

void broken_acquire(Lock *lock) {
    while(1) {
        if(lock->locked == 0) {
            lock->locked = 1;
            break;
        }
    }
}

void release (Lock *lock) {
    lock->locked = 0;
}

Can be acquired simultaneously, a time-of-check-to-time-of-use ("TOCTTOU") race.
We want these 2 lines to be atomic.
```
Atomic instructions

◆ Nowadays, CPUs overwhelmingly provide atomic operations
  ❖ Sometimes doing exactly what we needed in the previous slide

◆ In x86
  ❖ xchg %eax, addr
  ❖ Is equivalent to
  1. freeze any memory activity for address addr
  2. temp := *addr
  3. *addr := %eax
  4. %eax = temp
  5. un-freeze memory activity

\[\text{swap}(\%eax,*\text{addr})\]
Acquire & Release (now works)

```c
int xchg(addr, value) {
    %eax = value
    xchg %eax, (addr)
    return %eax
}

void acquire(Lock *lock) {
    while(1)
        if( xchg(&lock->locked,1) == 0 )
            break;
}

void release (Lock *lock) {
    xchg(&lock->locked,0);
    // why not do:
    // lock->locked = 0 ?
    // discussed in following
    // slides
}
now its atomic
```
Reminder: memory barriers

Thread X:

a = 0
// ...
a = 1
if (b == 0)
   // alone => fast path

Thread Y:

b = 0
// ...
b = 1
if (a == 0)
   // alone => fast path

Can both X and Y enter the fast path?
Reminder: memory barriers

Thread X:

\[
a = 0 \\
// \ldots \\
a = 1 \\
if( b == 0 ) \\
    // alone => fast path
\]

Thread Y:

\[
b = 0 \\
// \ldots \\
b = 1 \\
if( a == 0 ) \\
    // alone => fast path
\]

Yes, e.g., because of performance optimizations:
1) “Store-buffer” of core, and
2) Can reorder load/store ops if no dependencies between them.
Reminder: memory barriers

Thread X:

a = 0
// ...
a = 1
if( b == 0 )
  // alone => fast path

Thread Y:

b = 0
// ...
b = 1
if( a == 0 )
  // alone => fast path

So need an explicit memory “fence” or “barrier” after assignment into ‘a’ and ‘b’; xchg serves as such a “full” memory fence:
- Full fence: all reads/writes prior to fence occur before any reads/writes after fence
- Partial: all writes before fence occur before any reads after

(Also, our C xchg() is “asm volatile”, so no compiler reordering.)
Reminder: memory barriers

insert(int data) {

    acquire( &list_lock );
    // 0. load+store
    //  (lock->locked)

    List *l = new List;
l->data = data;
l->next = list;  // 1. load (list)
list = l;  // 2. store (list)

    release( &list_lock )
    // 3. store (lock->locked)
}

◆ ‘list_lock’ & ‘list’ are independent memory locations
◆ The 1996 PentiumPro manual (Volume 3, 7.2) says reads can be carried out speculatively and in any order, which implies we need to serialize the ‘release’, otherwise might do 3 before 1 or 0 after 2
◆ But the 2007 Intel 64 architecture Memory Ordering White Paper says that Intel 64 and IA−32 will preserve store order and never move a load after a store (“TSO”). So lock−>locked = 0 would work
Spin or block?

- Constructs like `while(1) { xchg }` are called
  - “Busy waiting” or “spinning”
  - Indeed, the lock we’ve used is called a “spinlock”
- **Not advisable when the critical section is long**
  - (critical section = everything between the acquire-release)
  - Because other threads can utilize the core instead
- **The alternative: blocking**
  - In xv6: `sleep()` & `wakeup()`
  - Not advisable when wait is short (otherwise induces overhead)
- **There’s also two-phase waiting**
  - Spin for a while, then block (related: the “ski problem”)
- **Note that kernel & processes are different in this context**
  - Kernels can hold lock with interrupts disabled
  - Sleeping might not make sense for kernel in some cases
Locking granularity

◆ How much code should we lock?
◆ Once, Linux had a “big lock” (for the entire kernel!)
  ❖ (Completely removed only in 2011…)
◆ What’s the appropriate locking granularity for file-system:
  ❖ All of it with a single lock? Every directory? File? Block?
◆ Coarse granularity:
  ☺ Pros: simple, easier, tends to be more correct… (e.g., seL4)
  ☹ Cons: performance degradation, negates concurrency
◆ Fine granularity:
  ☺ If done right, allows for concurrency most of the time
  ☹ Code is harder to write, read, and debug
◆ Xv6
  ❖ To simplify, tends to favor coarse (but no “kernel big lock”)
  ❖ E.g., locking entire process table instead of individual entries
Lock ordering

◆ When utilizing more than one lock
  ❖ All code paths must acquire locks in same order
  ❖ Otherwise => deadlock!

◆ Xv6
  ❖ Simple / coarse => not a lot of locks held together
  ❖ Longest chain is 2, e.g.,
    ▪ When removing a file: lock dir => lock file (in that order)
    ▪ ideinter() / ide lock => wakeup / ptable lock
    ▪ ...

Synchronizing with interrupts

- A core might use locks to synchronize with other cores
  - (interrupt + non-interrupt context)

- A core might also synchronize with itself
  - Namely, its non-interrupt + interrupt code
  - Both are essentially different threads of execution that might manipulate the same data
  - Example: both of the following use tickslock (3063)
    - tick handler (timer interrupt; 3116), and
    - sys_sleep (3473)

- xv6 solution (again, favors simplicity over performance)
  - Disable interrupts while holding a lock
  - Thus, interrupts will never be processed with a lock
  - Which ensures no deadlocks due to interrupts
xv6 locking code

◆ Pages 14 – 15
Locks mess up modularity

◆ In principle…
  ❖ When calling function F, best if we don’t know if it uses a lock
  ❖ Supposedly should be “internal details”

◆ Alas, in the kernel, it oftentimes doesn’t work that way
  ❖ Performance is paramount
    ▪ Typically not trading off elegance for performance
  ❖ If I hold lock L and invoke F, it must not attempt to acquire L
  ❖ If F acquires L, I must not attempt to call it while holding L

❖ => Locks break modularity and contaminate the OS code
❖ => Consider locks as part of the function specification
  ▪ “call this function while holding L1…”
  ▪ “if that function returns -2 release L2…”
  ▪ Ordering of locks needs to be know too
Real world

- **Locking is hard**
  - Despite years of experience, it’s still very challenging!
  - Deadlocks are common when developing even for experts

- **Our spinlock, with xchg, keeps memory busy => suboptimal**
  - We can sometimes do better (see later lecture)
  - RCU (read-copy-update) optimizes for mostly reading shared state; heavily used in Linux

- **Synchronization is an active field of research**
  - Transactional memory (via software, hardware, or both)
  - Lock-free data structures
  - A newly proposed research OS (2010) that is always deterministic (see later lecture)
  - Many papers about automatically identifying races