Locks

Jinkyu Jeong ([jinkyu@skku.edu](mailto:jinkyu@skku.edu))
Computer Systems Laboratory
Sungkyunkwan University
[http://csl.skku.edu](http://csl.skku.edu)
The Classic Example (1)

• Withdrawing money from a bank account
  – Suppose you and your girl (or boy) friend share a bank account with a balance of 1,000,000won
  – What happens if both go to separate ATM machines and simultaneously withdraw 100,000won from the account?

```c
int withdraw (account, amount)
{
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    return balance;
}
```
The Classic Example (2)

- The execution of the two threads can be interleaved, assuming preemptive scheduling:

```c
balance = get_balance (account);
balance = balance - amount;

balance = get_balance (account);
balance = balance - amount;
put_balance (account, balance);

put_balance (account, balance);
```
The Real Example

```c
extern int g;
void inc()
{
    g++;
}
```

Thread T1

```
 movl 0x1000, %eax
 addl $1, %eax
 movl %eax, 0x1000
```

Thread T2

```
 movl 0x1000, %eax
 addl $1, %eax
 movl %eax, 0x1000
```

context switch
Sharing Resources

• Local variables are not shared among threads
  – Refer to data on the stack
  – Each thread has its own stack
  – Never pass/share/store a pointer to a local variable on another thread’s stack

• Global variables are shared among threads
  – Stored in static data segment, accessible by any thread

• Dynamic objects are shared among threads
  – Stored in the heap, shared through the pointers

• Also, processes can share memory (shmem)
Synchronization Problem

• Concurrency leads to non-deterministic results
  – Two or more concurrent threads accessing a shared resource create a race condition
  – The output of the program is not deterministic; it varies from run to run even with same inputs, depending on timing
  – Hard to debug

• We need synchronization mechanisms for controlling access to shared resources
  – Synchronization restricts the concurrency
  – Scheduling is not under programmer’s control
Concurrency in the Kernel

User Space

Kernel

Hardware

system call handlers

interrupt handlers

background kernel threads

...
Critical Section

• A **critical section** is a piece of code that accesses a shared resource, usually a variable or data structure.

```
movl 0x1000, %eax
addl $1, %eax
movl %eax, 0x1000
```

• Need **mutual exclusion** for critical sections
  – Execute the critical section atomically (all-or-nothing)
  – Only one thread at a time can execute in the critical section
  – All other threads are forced to wait on entry
  – When a thread leaves a critical section, another can enter
Locks

• A lock is an object (in memory) that provides mutual exclusion with the following two operations:
  – acquire(): wait until lock is free, than grab it
  – release(): unlock and wake up any thread waiting in acquire()

• Using locks
  – Lock is initially free
  – Call acquire() before entering a critical section, and release() after leaving it
  – acquire() does not return until the caller holds the lock
  – On acquire(), a thread can spin (spinlock) or block (mutex)
  – At most one thread can hold a lock at a time
Using Locks

```c
int withdraw (account, amount)
{
    acquire (lock);
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    release (lock);
    return balance;
}
```
Requirements for Locks

• Correctness
  – **Mutual exclusion**: only one thread in critical section at a time
  – **Progress** (deadlock-free): if several threads want to enter the critical section, must allow one to proceed
  – **Bounded waiting** (starvation-free): must eventually allow each waiting thread to enter

• Fairness
  – Each thread gets a fair chance at acquiring the lock

• Performance
  – Time overhead for a lock without and with contentions (possibly on multiple CPUs)?
An Initial Attempt

• An initial implementation of a spinlock

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

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

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

• Does this work?

The caller “busy-waits”, or spins for locks to be released
Implementing Locks

• Software-only algorithms
  – Dekker’s algorithm (1962)
  – Peterson’s algorithm (1981)
  – Lamport’s Bakery algorithm for more than two processes (1974)

• Hardware atomic instructions
  – Test-And-Set
  – Compare-And-Swap
  – Load-Linked (LL) and Store-Conditional (SC)
  – Fetch-And-Add

• Controlling interrupts
Software-only Algorithm

• The second attempt to implement spinlocks
  – Note: each load and store instruction is atomic

```c
int interested[2];

void acquire (int process) {
    int other = 1 - process;
    interested[process] = TRUE;
    while (interested[other]);
}

void release (int process) {
    interested[process] = FALSE;
}
```

• Does this work?
Peterson’s Algorithm

- Solves the critical section problem for two processes

```c
int turn;
int interested[2];

void acquire (int process) {
    int other = 1 - process;
    interested[process] = TRUE;
    turn = other;
    while (interested[other] && turn == other);
}

void release (int process) {
    interested[process] = FALSE;
}
```
Test-And-Set

• Atomic instructions
  – read-modify-write operations guaranteed to be executed “atomically”

• Test-And-Set instruction
  – Returns the old value of a memory location while simultaneously updating it to the new value
  – e.g. xchg in x86: exchange register/memory with register

```c
int TestAndSet (int *v, int new) {
    int old = *v;
    *v = new;
    return old;
}
```
Using Test-And-Set

• A simple spinlock using Test-And-Set instruction
  – Refer to spinlock.h and spinlock.c in xv6

```c
struct lock { int held = 0; }
void acquire (struct lock *l)
{
    while (l->held);
    l->held = 1;
}
void release (struct lock *l)
{
    l->held = 0;
}
```
```c
struct lock { int held = 0; }
void acquire (struct lock *l)
{
    while (TestAndSet(&l->held, 1));
}
void release (struct lock *l)
{
    l->held = 0;
}
```
Compare-And-Swap

• Supported in x86, Sparc, etc.
  – Update the memory location with the new value only when its old value equals to the “expected” value
  – e.g. cmpxchg in x86: compare and exchange

```c
int CompareAndSwap (int *v, int expected, int new) {
    int old = *v;
    if (old == expected)
        *v = new;
    return old;
}

void acquire (struct lock *l) {
    while (CompareAndSwap(&l->held, 0, 1));
}
```
LL & SC

- Supported in MIPS, Alpha, PowerPC, ARM, etc.
- Load-Locked (LL) fetches a value from memory
- Store-Conditional (SC) succeeds with returning 1 if no intervening store to the address has taken place
- Otherwise, SC returns 0 without updating the memory

```c
void acquire (struct lock *l) {
  while (1) {
    while (LL(&l->held));
    if (SC(&l->held), 1)) return;
  }
}

void release (struct lock *l) {
  l->held = 0;
}
```
Fetch-And-Add

• Supported in x86, etc.
  – Atomically increments a value while returning the old value
  – e.g. xadd in x86: exchange and add

int FetchAndAdd (int *v, int a) {
    int old = *v;
    *v = old + a;
    return old;
}
Ticket Locks Using Fetch-And-Add

• First get a ticket and wait until its turn
• Provides bounded waiting

```c
struct lock {
    int ticket = 0;
    int turn = 0;
};

void acquire (struct lock *l) {
    int myturn = FetchAndAdd(&l->ticket, 1);
    while (l->turn != myturn);
}

void release (struct lock *l) {
    l->turn = l->turn + 1;
}
```
Controlling Interrupts (1)

• Disable interrupts for critical sections

- Disabling interrupts blocks external events that could trigger a context switch (e.g. timer)
- The code inside the critical section will not be interrupted
- There is no state associated with the lock
- Can two threads disable interrupts simultaneously?

```c
void acquire (struct lock *l) {
    cli();  // disable interrupts;
}
void release (struct lock *l) {
    sti();  // enable interrupts;
}
```
Controlling Interrupts (2)

• Pros
  – Simple
  – Useful for a single-processor system

• Cons
  – Only available to kernel
    • Why not provide them as system calls?
  – Insufficient on multi-processor systems
    • Back to atomic instructions
  – When the critical section is long, important interrupts can be delayed or lost (e.g. timer, disks, etc.)
  – Slower than executing atomic instructions on modern CPUs
Summary

• Spinlocks are horribly wasteful
  – If a thread is spinning on a lock, the thread holding the lock cannot make progress
  – The longer the critical section, the longer the spin
  – CPU cycle is wasted
  – Greater the chances for lock holder to be interrupted through involuntary context switch

• Spinlocks (and disabling interrupts on a single CPU) are primitive synchronization mechanisms
  – They are used to build higher-level synchronization constructs
Priority Inversion Problem

- A situation where a higher-priority job is unable to run because a lower-priority job is holding a resource it needs, such as a lock.
- What really happened on Mars?
Solutions to Priority Inversion

• Priority inheritance protocol (PIP)
  – The higher-priority job can donate its priority to the lower-priority job holding the resource it requires.

• Priority ceiling protocol (PCP)
  – The priority of the low-priority thread is raised immediately when it gets the resource.
  – The priority ceiling value must be predetermined.