

CS3210: Multiprocessors and locking

Taesoo Kim edited by Kyle Harrigan

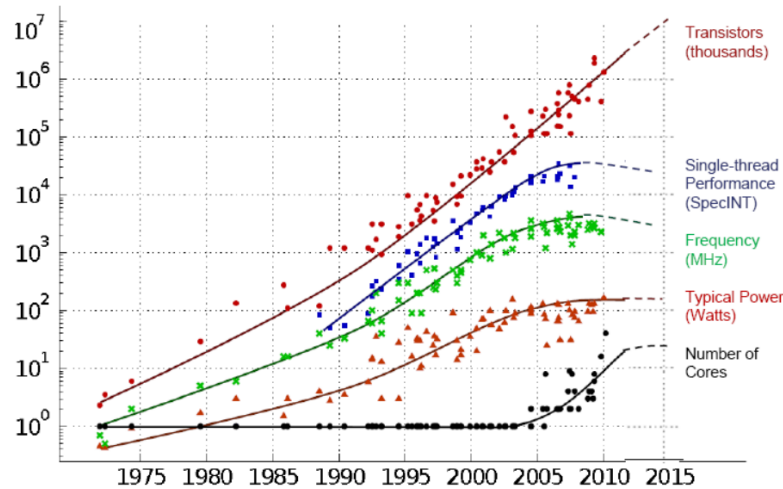
Administrivia

- Quiz graded, will present results
- Pre-proposal rework, due 25 October by ~9pm
- Drop Date approaching
- (Nov 1) Team Proposal (3-5 min/team)
- **DUE** : submit slides (as a team) by 10 pm, Oct 31

Summary of last lectures

- Power-on → BIOS → bootloader → kernel → init (+ user bins)
- OS: abstraction, multiplexing, isolation, sharing
- Design: monolithic (xv6) vs. micro kernels (jos)
- Abstraction: process, system calls
- Isolation mechanisms: CPL, segmentation, paging
- Interrupts, exceptions

Today: Multiprocessor (and locking)



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

- A mechanism to implement abstraction and multiplexing
 - Locking: how to correctly serialize executions?

Example: counting

- DEMO: `count.c`

```
$ ./count 1 10  
cpu = 1, count = 10
```

```
$ ./count 1 100  
cpu = 1, count = 100
```

```
$ ./count 2 10  
cpu = 2, count = 20
```

Example: measuring execution time

- Execution time reduces by half (x2 utilization)
- Q: problem?

```
$ time ./count 1 1000000000
cpu = 1, count = 1000000000
./count 1 1000000000 2.25s user 0.00s system 99% cpu 2.258 total

$ time ./count 2 500000000
cpu = 2, count = 502495507
./count 2 500000000 2.31s user 0.00s system 197% cpu 1.165 total
```

Example: analysis in detail

```
$ perf stat ./count 1 1000000000
cpu = 1, count = 1000000000
```

Performance counter stats for './count 1 1000000000':

2251.705855	task-clock (msec)	#	0.999 CPUs utilized
88	context-switches	#	0.039 K/sec
3	cpu-migrations	#	0.001 K/sec
56	page-faults	#	0.025 K/sec
7,135,385,783	cycles	#	3.169 GHz
<not supported>	stalled-cycles-frontend		
<not supported>	stalled-cycles-backend		
4,005,413,202	instructions	#	0.56 insns per cycle
1,000,979,696	branches	#	444.543 M/sec
17,505	branch-misses	#	0.00% of all branches

2.252871308 seconds time elapsed

Example: analysis in detail

```
$ perf stat ./count 2 500000000
cpu = 2, count = 503059602
```

Performance counter stats for './count 2 500000000':

2349.797354	task-clock (msec)	#	1.992 CPUs utilized
19	context-switches	#	0.008 K/sec
4	cpu-migrations	#	0.002 K/sec
58	page-faults	#	0.025 K/sec
7,274,653,523	cycles	#	3.096 GHz
<not supported>	stalled-cycles-frontend		
<not supported>	stalled-cycles-backend		
4,003,964,870	instructions	#	0.55 insns per cycle
1,000,732,490	branches	#	425.880 M/sec
19,942	branch-misses	#	0.00% of all branches

1.179731295 seconds time elapsed

Q: How to fix this problem?

- Q: plan?
- Two goals:
 - Correctness : no missing counts
 - Performance : execution time

Attempt 1: use only one CPU

- `pin_cpu(0)` : fix its execution to the first CPU (id = 0)

```
01 void pin_cpu(int cpu) {
02     cpu_set_t cpuset;
03     CPU_ZERO(&cpuset);
04     CPU_SET(cpu, &cpuset);
05
06     if (pthread_setaffinity_np(pthread_self(), \
07                               sizeof(cpu_set_t), &cpuset) < 0)
08         err(1, "failed to set affinity");
09 }
```

Result

- Q: correctness? performance?

```
$ time ./count 1 1000000000  
cpu = 1, count = 1000000000  
2.26s user 0.00s system 99% cpu 2.266 total
```

```
$ time ./count 2 500000000  
cpu = 2, count = 1000000000  
2.31s user 0.00s system 99% cpu 2.316 total
```

Attempt 2: use atomic operation

- Add a lock prefix (all memory ops)

```
01  asm volatile("lock incl %0"  
02                : "+m"(count)  
03                : "m"(count)  
04                : "memory");
```

Result

- Q: correctness? performance?

```
$ time ./count 1 1000000000  
cpu = 1, count = 1000000000  
6.64s user 0.00s system 99% cpu 6.644 total
```

```
$ time ./count 2 500000000  
cpu = 2, count = 1000000000  
49.76s user 0.00s system 199% cpu 24.893 total
```

Analysis (see stall cycles)

```
$ perf stat ./count 2 500000000
cpu = 2, count = 1000000000
```

Performance counter stats for './count 2 500000000':

62475.069100	task-clock (msec)	#	1.988 CPUs utilized	
5,228	context-switches	#	0.084 K/sec	
3	cpu-migrations	#	0.000 K/sec	
80	page-faults	#	0.001 K/sec	
134,913,649,220	cycles	#	2.159 GHz	[83.34%]
133,127,752,850	stalled-cycles-frontend	#	98.68% frontend cycles idle	[83.34%]
78,451,841,095	stalled-cycles-backend	#	58.15% backend cycles idle	[66.67%]
4,103,848,320	instructions	#	0.03 insns per cycle	
		#	32.44 stalled cycles per insn	[83.34%]
1,018,681,684	branches	#	16.305 M/sec	[83.34%]
474,657	branch-misses	#	0.05% of all branches	[83.32%]

31.427313911 seconds time elapsed

Attempt 3: compute locally (per CPU)

- Q: correctness? performance?
- Q: how to improve perf even further?
- Q: how to trigger a race?

```
01  int local = 0;
02  for (register int i = 0; i < cnt; i++)
03      local ++;
04
05  count += local;
```

Attempt 4: using locks

```
01  int local = 0;
02  for (register int i = 0; i < cnt; i++)
03      local ++;
04
05  acquire(&lock);
06  count += local;
07  release(&lock)
```


Locks

- **Mutual exclusion** : only one core can hold a given lock
 - concurrent access to the same memory location, at least one write
 - example: `acquire(l); x = x + 1; release(l);`
- **Serialize** critical section: hide intermediate state
 - another example: transfer money from account A to B
 - `put(a + 100)` and `put(b - 100)` must be both effective, or neither

Strawman: locking

```
01  struct lock { int locked; };
02
03  void acquire(struct lock *l) {
04      for (;;) {
05          if (l->locked == 0) { // A: test
06              l->locked = 1;    // B: set
07              return;
08          }
09      }
10  }
11
12  void release(struct lock *l) {
13      l->locked = 0;
14  }
```

Relying on an atomic operation

- Q: correctness? performance?

```
01  struct lock { int locked; };
02
03  void acquire(struct lock *l) {
04      for (;;) {
05          if (xchg(&l->locked, 1) == 0)
06              return;
07      }
08  }
09
10  void release(struct lock *l) {
11      xchg(&l->locked, 0);
12  }
```

Using xchg: an atomic operation (primitive)

- `x86.h` in `xv6`

```
01  int xchg(volatile int *addr, int newval) {
02      int result;
03      // The + in "+m" denotes a read-modify-write operand.
04      asm volatile("lock; xchgl %0, %1" :
05                  "+m" (*addr), "=a" (result) :
06                  "1" (newval) :
07                  "cc");
08      return result;
09  }
```

Result

```
$ time ./count 1 1000000000  
cpu = 1, count = 1000000000  
0.35s user 0.00s system 99% cpu 0.349 total  
  
$ time ./count 2 500000000  
cpu = 2, count = 1000000000  
0.35s user 0.00s system 186% cpu 0.188 total
```

Different aspects of performance?

- Q: what about its performance with the increasing number of cpus?
- DEMO: in 80-core machine (160!)

```
; in optimus:  
$ cd lock  
$ ./count-mp.py 160 1000000  
; in host  
$ scp optimus:lock/plot.dat .  
$ plot.py
```

Attempt 5: lockless? add (atomically)

- Q: correctness? performance?

```
asm volatile("mov %1, %%eax\n\t"  
             "lock add %%eax, %0\n\t"  
             : "+m"(count)  
             : "m"(local), "m"(count)  
             : "memory");
```

```
__atomic_add_fetch(&count, local, __ATOMIC_RELEASE);
```


acquire() in xv6

```
01 void acquire(struct spinlock *lk) {
02     pushcli(); // disable interrupts to avoid deadlock.
03     if (holding(lk))
04         panic("acquire");
05
06     // The xchg is atomic.
07     // It also serializes, so that reads after acquire are not
08     // reordered before it.
09     while (xchg(&lk->locked, 1) != 0)
10         ;
11
12     // Record info about lock acquisition for debugging.
13     lk->cpu = cpu;
14     getcallerpcs(&lk, lk->pcs);
15 }
```

release() in xv6

```
01 void release(struct spinlock *lk) {
02     if (!holding(lk))
03         panic("release");
04
05     lk->pcs[0] = 0;
06     lk->cpu = 0;
07
08     // Q?
09     xchg(&lk->locked, 0);
10
11     popcli();
12 }
```

Quiz results

Quiz results, +18.25 correction

Quiz policy

- We have provided an 18.25 point adjustment (basically toss one question)
- We will allow you to choose the best of Quiz 1 and Quiz 2 grades, and use this for your quiz grade

References

- [Intel Manual](#)
- [UW CSE 451](#)
- [OSPP](#)
- [MIT 6.828](#)
- Wikipedia
- The Internet