CS3210: Multiprocessors and Locking

Kyle Harrigan

Administrivia

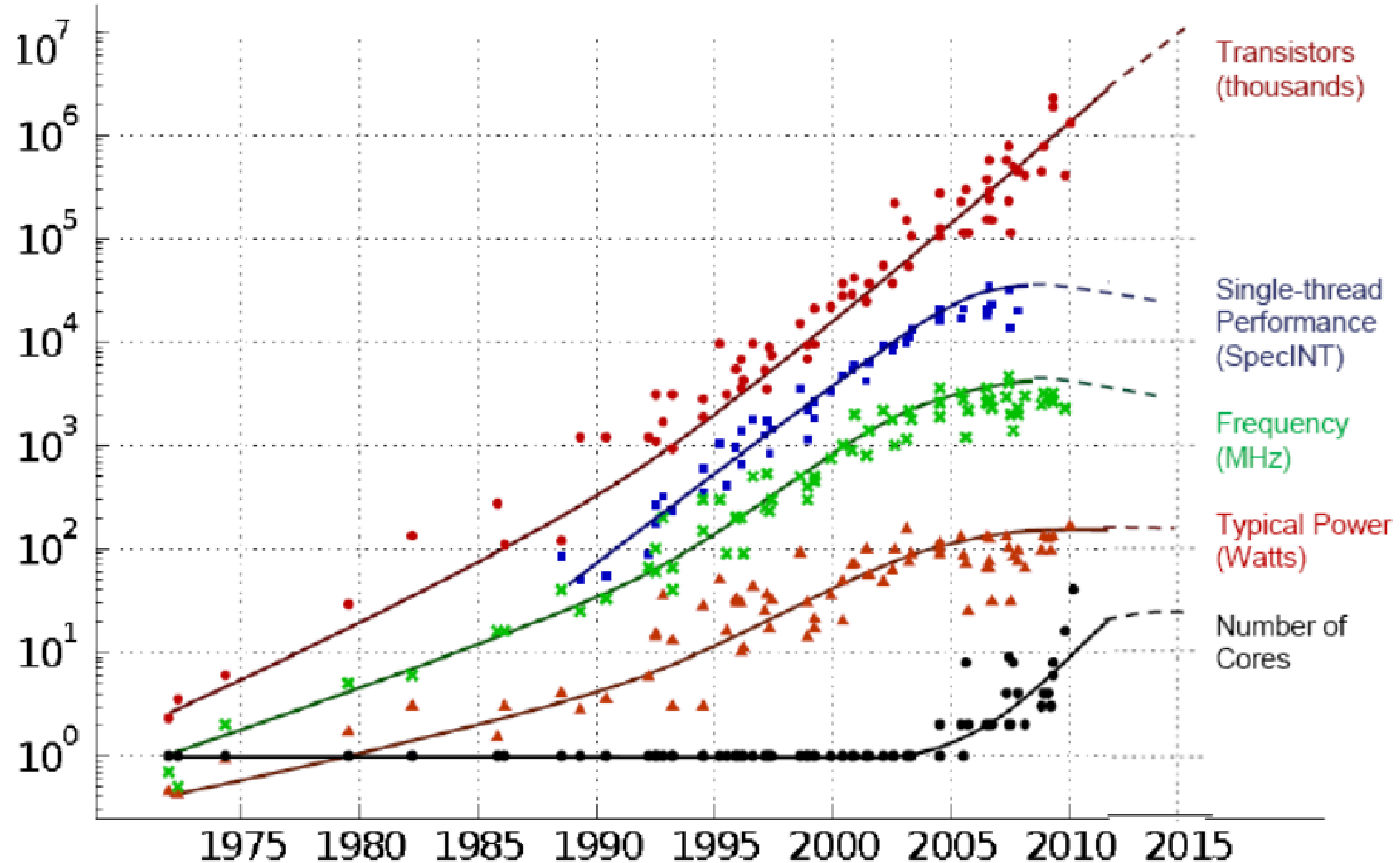
- Quiz errata discussion
- Lab 3 (Part B) due tomorrow
- Drop Date approaching (Oct 28)
- Team Proposal (3-5 min/team) - Oct 24

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
- Lazy allocation

Today: Multiprocessor (and locking?)

Motivation:



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

Further motivation: Lab 4

- Multiple CPUs running kernel code can cause race conditions
- We will approach this problem by implementing (utilizing) locks in the proper locations
- Let us further understand the implementation challenges and tradeoffs with locks (very much always an open research area and performance concern)

An Issue

- Multiple CPUs operating on same data opens the possibility of simultaneous reads / writes -> yields incorrect data
    - Any statement in C may be several CPU instructions
- Can also happen in uniprocessor… example?

An Issue

- Multiple CPUs operating on same data opens the possibility of simultaneous reads / writes -> yields incorrect data
    - Any statement in C may be several CPU instructions
- Can also happen in uniprocessor… example?
    - Interrupts

An Issue

- Multiple CPUs operating on same data opens the possibility of simultaneous reads / writes -> yields incorrect data
  - Any statement in C may be several CPU instructions
- Can also happen in uniprocessor… example?
  - Interrupts
- There are many approaches. xv6 approach is on locking / mutual exclusion.

Race conditions

- Ex. File system disk requests
- Use a critical section to protect
  - Locking primitive
  - Acquire and Release
- Invariants
  - Some data is supposed to remain constant
  - Example: linked list assumptions… 1) List should point to first node 2) Next points to next node
  - Some invariants are temporarily violated (ex. during list insertion)
- Race conditions are often hard to reproduce and troubleshoot

Deadlock

- Ex. Dining Philosophers
- May need to hold multiple locks to execute a task
- xv6 uses a max of two locks. Ex:
    - ideintr holds ide locks, but also calls wakeup which acquires the ptable lock
    - More examples in file system (often must lock directory along with file)

Interrupt handlers

- Multiple CPUs and timer ticks (sys_sleep)
- Interrupts on a single processor
  - iderw holds lock, then interrupted to handle ideintr
- Mitigate in xv6 by never holding locks with interrupts enabled

Instruction and Memory Reordering

- Modern compilers and processors support out of order execution
  - Concurrency may expose a hazard due to reordering
  - Solution: Tell compiler not to reorder (`__sync_synchronize()` in `acquire` and `release`)

In acquire():

```
// Tell the C compiler and the processor to not move loads or stores
// past this point, to ensure that the critical section's memory
// references happen after the lock is acquired.
__sync_synchronize();
```

In release():

```
// Tell the C compiler and the processor to not move loads or stores
// past this point, to ensure that all the stores in the critical
// section are visible to other cores before the lock is released.
// Both the C compiler and the hardware may re-order loads and
// stores; __sync_synchronize() tells them both to not re-order.
__sync_synchronize();
```

Spinlock shortfalls

- Complex groups of functions that may call each other (allocproc, fork, userinit, ptable.lock)
- If everyone acquires lock, we have deadlock
- Solution: Force called to hold lock before function all
    - Kernel programmer must have awareness of what locks should be held
    - Another solution: recursive locks (complex)
    - Other examples include pipe read/write complexity (who holds the lock?)

Real world

- Concurrency and parallel programming are active areas of research (grad students rejoice)
- Better to use primitive locks to form higher level constructs
  - Abstract away locking
  - xv6 does not do this
  - libraries like pthreads provide higher level locking capability
- Can implement atomic locks without hardware support, but expensive and complex
- Another option: lock free data structures and algorithms
- Many issues w/ performance related to cache lines, multiple processors, etc.

Extended assembly

How to interpret?

```
asm volatile("incl %0"
              : "+m"(count)
              : "m"(count)
              : "memory");
```

Extended assembly, see https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])
```

Extended assembly

How to interpret?

```
asm volatile("incl %0"
              : "+m"(count)
              : "m"(count)
              : "memory");
```

Extended assembly, see https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

```
asm [volatile] ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])
```

```
asm volatile("incl %0"
              : "+m"(count) // Output operand (reading and writing)
              : "m"(count) //Input operand
              : "memory")  //"tells the compiler that the assembly code
//performs memory reads or writes to items other than those listed in the
//input and output operands" (for example, accessing the memory pointed to
//by one of the input parameters).
```

Preparation Question

```
int count = 0;

void* run(void *arg)
{
  register int cnt = *(int *)arg;
  for (register int i = 0; i < cnt; i++) {
    asm volatile("incl %0"
                 : "+m"(count)
                 : "m"(count)
                 : "memory");
  }
  return NULL;
}
```

Preparation Question

```
int main(int argc, char *argv[])
{
  int ncpu = atoi(argv[1]);
  int upto = atoi(argv[2]);

  pthread_t *tids = malloc(ncpu * sizeof(pthread_t));

  for (int i = 0; i < ncpu ; i ++) {
    if (pthread_create(&tids[i], NULL, run, &upto))
      err(1, "failed to creat a thread");
  }
  for (int i = 0; i < ncpu ; i ++)
    pthread_join(tids[i], NULL);

  printf("cpu = %d, count = %d\n", ncpu, count);

  return 0;
}
```

Example: counting

- DEMO: `count.c`

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

$ ./count 2 5
cpu = 2, count = 10

$ ./count 1 10000
cpu = 1, count = 10000

$ ./count 2 5000
cpu = 2, count = 10000

$ ./count 1 100000
cpu = 1, count = 100000

$ ./count 2 50000
cpu = 2, count = 53494
```

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?

- Two (competing?) 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)

```
void pin_cpu(int cpu) {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(cpu, &cpuset);

  if (pthread_setaffinity_np(pthread_self(), \
                             sizeof(cpu_set_t), &cpuset) < 0)
    err(1, "failed to set affinity");
}
```

Result (attempt 1)

- 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)

```
asm volatile("lock incl %0"
             : "+m"(count)
             : "m"(count)
             : "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?

```
int local = 0;
for (register int i = 0; i < cnt; i++)
  local ++;

count += local;
```

Attempt 3: compute locally (per CPU)

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

```
int local = 0;
for (register int i = 0; i < cnt; i++)
  local ++;

count += local;
```

```
$ time ./count_local 1 1000000000
cpu = 1, count = 1000000000
real    0m1.847s
user    0m1.832s
sys     0m0.012s

$ time ./count_local 2 500000000
cpu = 2, count = 1000000000

real    0m0.896s
user    0m1.780s
sys     0m0.004s
```

Attempt 4: using locks

```
int local = 0;
for (register int i = 0; i < cnt; i++)
  local ++;

acquire(&lock);
count += local;
release(&lock)
```

Attempt 4: using locks

```
int local = 0;
for (register int i = 0; i < cnt; i++)
  local ++;

acquire(&lock);
count += local;
release(&lock)
```

- Perhaps a reasonable solution
    - Lock is localized to where we need it (contention low)
    - Performance is good
    - Correctness is good

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

```
struct lock { int locked; };

void acquire(struct lock *l) {
  for (;;) {
    if (l->locked == 0) { // A: test
      l->locked = 1;      // B: set
      return;
    }
  }
}

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

Strawman: locking

```
struct lock { int locked; };

void acquire(struct lock *l) {
  for (;;) {
    if (l->locked == 0) { // A: test
      l->locked = 1;      // B: set
      return;
    }
  }
}

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

- No, this doesn't work
- Non-atomic test and set has a race condition

Relying on an atomic operation

- Q: correctness? performance?

```
struct lock { int locked; };

void acquire(struct lock *l) {
  for (;;) {
    if (xchg(&l->locked, 1) == 0)
      return;
  }
}

void release(struct lock *l) {
  xchg(&l->locked, 0);
}
```

Using xchg: an atomic operation (primitive)

- `x86.h` in xv6

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

Result

```
$ time ./count_xchg 1 1000000000
cpu = 1, count = 1000000000

real    0m1.876s
user    0m1.872s
sys     0m0.012s

$ time ./count_xchg 2 500000000
cpu = 2, count = 1000000000

real    0m0.925s
user    0m1.832s
sys     0m0.008s
```

Spinlock in xv6

- Pretty much same, but provide debugging info

```
struct spinlock {
  uint locked;       // Is the lock held?

  // Q?
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

`acquire()` in xv6

```
void acquire(struct spinlock *lk) {
  pushcli(); // disable interrupts to avoid deadlock.
  if (holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while (xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

`release()` in xv6

```c
void release(struct spinlock *lk) {
  if (!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Q?
  xchg(&lk->locked, 0);

  popcli();
}
```

References

- Intel Manual
- UW CSE 451
- OSPP
- MIT 6.828
- Wikipedia
- The Internet