CS3210: Processes and Switching

Kyle Harrigan

Administrivia

- (Mar 7) Team Proposal Day (just slides, target 3-5 min/team)
  - Problem statement
  - Idea
  - Demo plan (aka evaluation)
  - Timeline
- **DUE** submit slides (as a team) by 12 pm, Mar 7

General comments on pre-proposals

- Many too short, some were very good
- What (can) drive(s) a project?
  - Identified need / shortfall (not solutions looking for problems)
  - New approach (optimize, refine)
  - Curiosity / proof of concept (can we do it?)
- A sensible project flow:
  - Identify problem, propose potential solutions
  - Refine and choose approch
  - Implement, then document

Heilmeier's Catechism

- What are you trying to do? Articulate your objectives using absolutely no jargon.
- How is it done today, and what are the limits of current practice?
- What's new in your approach and why do you think it will be successful?
- Who cares? If you're successful, what difference will it make? What are the risks and the payoffs?
- How many people do you need? How long will it take? What are themilestones to check for success?

Heilmeier Credentials

- Why listen to Heilmeier?
  - Pioneering contributor to liquid crystal display (LCD)
  - RCA labs, DARPA (director), TI, Bellcore/Telcordia, others
- Notable awards (too many to list on slide)
  - IEEE Founders Medal
  - National Medal of Science
  - IRI Medal
  - IEEE Medal of Honor
  - John Fritz Medal
  - Kyoto Price

Other thoughts

- Plan your presentation in advance
  - Who will say what
  - Smooth transitions
- Don't read the slides to us
- Pictures and diagrams are good
- Enthusiasm can go a long way

Summary of last lectures

- Power-on -> BIOS -> bootloader -> kernel -> **user programs**
- OS: abstraction, **multiplexing**, isolation, sharing
- Design: monolithic (xv6) vs. micro kernels (jos)
- Abstraction: **process**, system calls
- Isolation mechanisms: CPL, segmentation, paging
- Multiprocessors and Locking, Threads and Spinlocks

Today's plan

- A few more notes on locking in xv6
- About process
    - For multiplexing (e.g., more processes than CPUs)
    - In particular, switching and scheduling

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);`

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);`

Example: why do we need a lock?

```
struct file* filealloc(void) {
  struct file *f;

  acquire(&ftable.lock);
  for(f = ftable.file; f < ftable.file + NFILE; f++){
    if(f=>ref == 0){
      f=>ref = 1;
      release(&ftable.lock);
      return f;
    }
  }
  release(&ftable.lock);
  return 0;
}
```

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);`
- **Atomic execution**: 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

A different way to think about locks

```
"In computer science, an invariant is a condition that can be relied upon
to be true during execution of a program, or during some portion of it.
It is a logical assertion that is held to always be true during a
certain phase of execution. For example, a loop invariant is a condition
that is true at the beginning and end of every execution of a loop"
```

source: wikipedia

- 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
- Q: `put(a + 100)` and `put(b - 100)`?

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;
}
```

Problem: concurrent executions on line 05

```
// process A                    // process B
if (l=>locked == 0)             if (l=>locked == 0)
  l=>locked = 1;                  l=>locked = 1;
```

- Recall:

```
$ while true; do ./count 2 10 | grep 10 ; done
cpu = 2, count = 10
...
```

Relying on an atomic operation

```
struct lock { int locked; };

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

void release(struct lock *l) {
  // Q?
  xchg(&l=>locked, 0);
}
```

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) {
  // Q1?
  pushcli();
  // Q2?
  if (holding(lk))
    panic("acquire");

  while (xchg(&lk=>locked, 1) != 0)
    ;

  lk=>cpu = cpu;
  getcallerpcs(&lk, lk=>pcs);
}
```

`release()` in xv6

```
void release(struct spinlock *lk) {
  // Q1?
  if (!holding(lk))
    panic("release");

  // Q2?
  lk->pcs[0] = 0;
  lk->cpu = 0;

  xchg(&lk->locked, 0);

  // Q3?
  popcli();
}
```

Why spinlocks?

- 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?

Spinlock guidelines

- hold for very short times
- don't yield CPU while holding lock
- (un)fairness issues: FIFO ordering?
- **NOTE** "blocking" locks for longer critical sections
  - waiting threads yield the CPU
  - but overheads are typically higher (later)

Problem 1: deadlock (e.g., double acquire)

- Q: what happens in xv6?

```
struct spinlock lk;
initlock(&lk, "test lock");
acquire(&lk);
acquire(&lk);
```

Problem 2: interrupt (preemption)

- Race in `iderw()` (`ide.c`)
  - `sti()` after `acquire()`
  - `cli()` before `release()`

iderw()

- What goes wrong with adding sti/cli in iderw?
- What ensures atomicity between processors
- What ensures atomicity within a single processor?

What about racing in file.c

- Race in `filealloc()` (`file.c`)
- Q: `ftable.lock`?
    - `sti()` after `acquire()`
    - `cli()` before `release()`

filealloc()

- Could the disk interrupt handler run while interrupts are enabled?
- Does any any interrupt handler grab the `ftable.lock`?
- What interrupt could cause trouble?

Scheduling - Motivation

Why are we here?

- OS typically has more processes than processors
- This implies a time-sharing mechanism
- You will implement basic scheduler (round-robin) in Lab 4
    - Cooperative -> Preemptive

Scheduling

- Which process to run?
  - Pick one from a set of RUNNABLE processes (or env in jos)
  - What have you seen from lab?
- (next lecture) Switching/scheduling in detail

Scheduling: design space

- Preemptive vs. cooperative?
- Global queue vs. per-CPU queue?

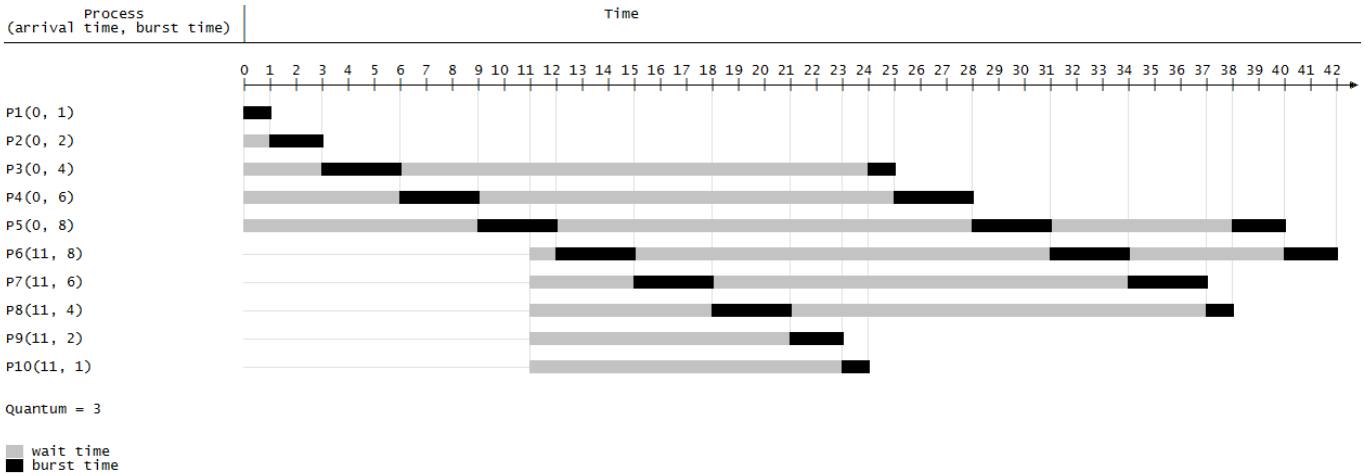
Scheduling: design space

- Scalability: w/ many runnable processes?
- Granularity (timeslice, quantum): 10ms vs 100ms? (dynamic? tickless?)
- Fairness: time quota, epoch (inversion? group?)
- QoS: priority? (e.g., nice)
- Constraints: realtime, deadlines (e.g., airplane)
- etc: resource starvation, performance consolidation (e.g., cloud)

Scheduling: difficult in practice

- No perfect/universal solution/policy
- Contradicting goals:
  - maximizing throughput vs. minimizing latency
  - minimizing response time vs. maximizing scalability
  - maximizing fairness vs. maximizing scalability

Example: round-robin scheduling



- Simple: assign fixed time unit per process
- Starvation-free (no priority)

Complexity in real scheduling algorithms

- Linux?

Complexity in real scheduling algorithms

- Linux
  - `kernel/sched/*.c`: 17k LoC with 7k lines of comments
  - vs. your RR in jos? 10 LoC?

```
    for (j = 1; j <= NENV; j++) {
      k = (j + i) % NENV;
      if (envs[k].env_status == ENV_RUNNABLE)
        env_run(&envs[k]);
    }
```

Summary (Wikipedia)

Operating System	*Preemption*	*Algorithm*
Amiga OS	Yes	Prioritized round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux kernel before 2.6.0	Yes	Multilevel feedback queue
Linux kernel 2.6.0–2.6.23	Yes	O(1) scheduler
Linux kernel after 2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative scheduler
Mac OS 9	Some	Preemptive scheduler for MP tasks, and cooperative for processes and threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative scheduler
Windows 95, 98, Me	Half	Preemptive scheduler for 32-bit processes, and cooperative for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

Example: available options in Linux

```
sysctl -A | grep "sched" | grep -v "domain"
kernel.sched_cfs_bandwidth_slice_us = 5000
kernel.sched_child_runs_first = 0
kernel.sched_compat_yield = 0
kernel.sched_latency_ns = 6000000
kernel.sched_migration_cost_ns = 500000
kernel.sched_min_granularity_ns = 2000000
kernel.sched_nr_migrate = 32
kernel.sched_rr_timeslice_ms = 25
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
kernel.sched_shares_window_ns = 10000000
kernel.sched_time_avg_ms = 1000
kernel.sched_tunable_scaling = 1
kernel.sched_wakeup_granularity_ns = 2500000
...

$ less /proc/sched_debug
$ less /proc/[pid]/sched
```

To tinker:

```
$ sysctl variable=value
```

Characterizing processes

- CPU-bound vs IO-bound
- Interactive processes (e.g., vim, emacs)
- Batch processes (e.g., cronjob)
- Real-time processes (e.g., audio/video players)

Scheduling policies in Linux

- **SCHED_FIFO**: first in, first out, real time processes
- **SCHED_RR**: round robin real time processes
- **SCHED_OTHER**: normal time/schedule sharing (default)
- **SCHED_BATCH**: CPU intensive processes
- **SCHED_IDLE**: Very low prioritized processes

Example

- Q: `count.c`?

```
$ sudo ./count 3 1000000000
8522: runs
8524: runs
8523: runs
8523: 2.05 sec
8522: 2.34 sec
8524: 2.49 sec
```

Example: available policies

```
$ chrt -m
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
```

Example: FIFO (real time scheduling)

```
$ sudo ./count 10 1000000000 "chrt -f -p 99"
...
```

References

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