CS3210: Processes and Switching

Tim Andersen

Administrivia

- (Oct 24) Team Proposal Day
  - Problem statement
  - Idea
  - Demo plan (aka evaluation)
  - Timeline
- **DUE** submit slides (as a team) by 10 am, Oct 24
- All team members should arrive on time to class.
- Approx. 30 presentations will each be timed to 2 minutes.

General comments on pre-proposals

- Many good ideas. Some were lacking clear steps to achieving the stated goal.
- 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
  - Is sets the standard for proposals.

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

Today's plan

- About process
  - For multiplexing (e.g., more processes than CPUs)
  - In particular, switching and scheduling

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?

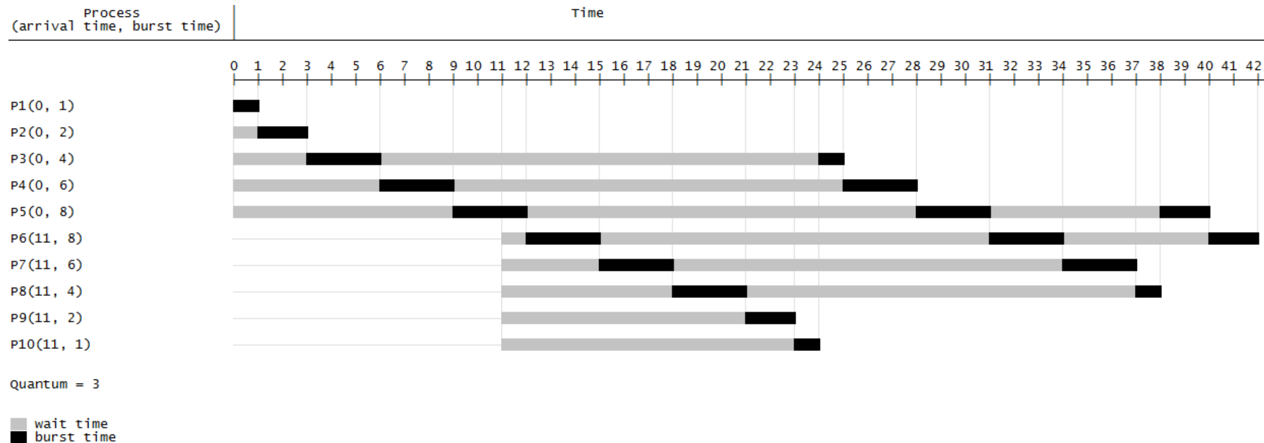
Scheduling: design space

- 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?

```
01      for (j = 1; j <= NENV; j++) {
02        k = (j + i) % NENV;
03        if (envs[k].env_status == ENV_RUNNABLE)
04          env_run(&envs[k]);
05      }
```

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"
...
```

Context Switching

Implementation challenges

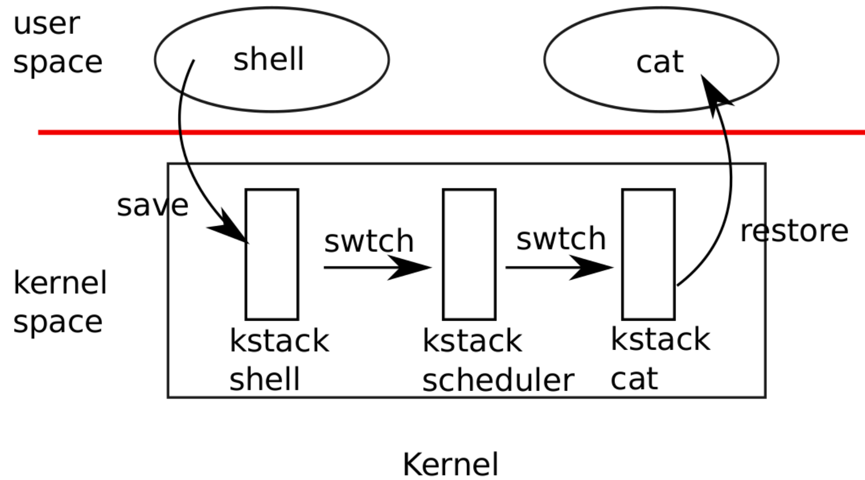
- Q: How to switch from one process to another?
    - A: Context switching
- Q: How to make context switching transparent?
    - A: Timer interrupts
- Q: How to switch among processes running concurrently?
    - A: Locking
- Q: How to coordinate processes?
    - A: Sleep on events (e.g., pipe, child exit)

Two kinds of context switch

1. From a process's kernel thread to CPU scheduler thread
2. From the scheduler thread to a process's kernel thread.

- xv6 never directly switches from user-space to user-space
  - user-kernel transition (system call or interrupt)
  - context switch to scheduler
  - context switch to new process's kernel thread
  - trap return

Big picture: switching



Context switching

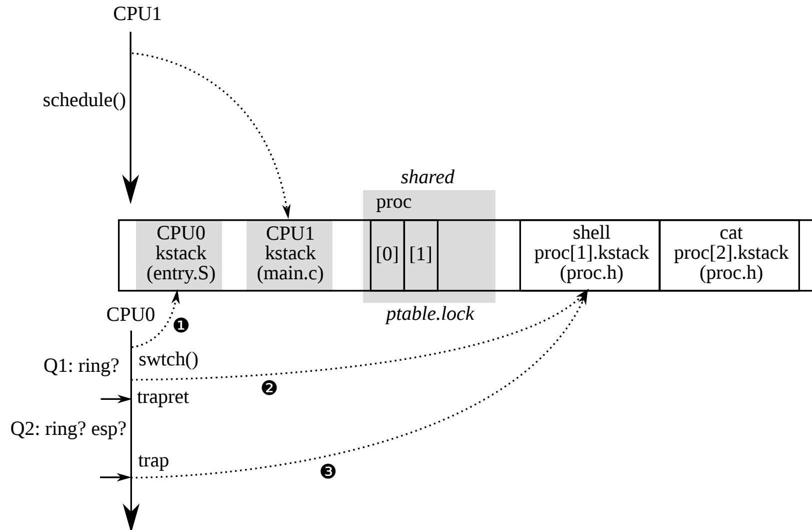
- Every xv6 process has its own kernel stack and register set
- Every CPU has its own scheduler thread
- Switching from one thread to another
    - Save and load CPU registers, including %esp, %eip.

`swtch`

```
void swtch(struct context**, struct context*);
```

- Doesn't know about threads just saves and loads sets of registers called contexts
- When time to give up CPU, kernel thread calls `swtch` to save itself and return to scheduler context
- Context is a `struct context*`, stored on the kernel stack
- CPU pushed onto stack and saves stack pointer to `*old`
- Copies `new` to `%esp`, pops previous registers, and returns

Switching overview: CPU perspective



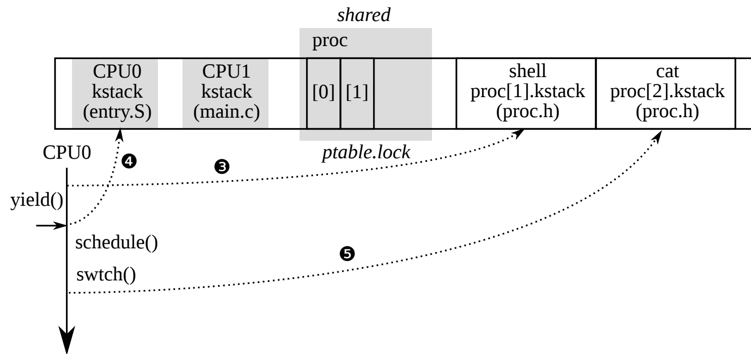
`yield`

- At the end of each interrupt, trap can call `yield`
- `yield` -> `sched` -> `swtch`
- Switches from `proc=>context` to `cpu=>scheduler`

`swtch`: Detailed Look

- Loads its arguments off the stack into `%eax` and `%edx` before it loses its arguments when it changes `%esp`
- Only callee-save registers saved
  - `%ebp`, `%ebx`, `%esi`, `%ebp`, and `%esp`
  - First four pushed, `%esp` saved as `*old`
- `%eip` was saved by the `call` instruction and is just above `%ebp`
- Moves pointer to new context into `%esp`
- Inverts sequence of steps to load context

Switching overview: CPU perspective



Show xv6 Code

- `swtch()`, `scheduler()`, `sched()`
- about `ptable.lock`

Scheduling

- Process giving up the CPU must
  - aquire `ptable.lock`
  - release any other locks
  - update its own state (e.g., RUNNABLE, SLEEPING)
  - call `sched`
- `yield`, `sleep`, and `exit` all do these steps

DEMO: sched

```
br sched
commands
p cpus[cpunum()].proc.pid
c
end
```

`ptable.lock`

- Held across calls to `swtch`
- Caller holding lock passes control to switched to code
- Needed because process state and context must be kept invariant across `swtch`
- Without lock, a different CPU might try to run a process after RUNNABLE but before kernel stack switch.
    - Result is two CPUs with same stack.

`sched` and `scheduler`

- Kernel thread always gives up in `sched` and switches to same location in scheduler
- Almost always switches to a process in `sched`.
- Thread switches follow a simple pattern between sched and scheduler
  - Coroutines
- Exception is `forkret` when process is first scheduled

Scheduler

- Loops over process table looking for RUNNABLE processes
- Finding one, sets current per-CPU process to proc
- Switches page table with `switchuvm`, marks as RUNNING, and calls `swtch`

References

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