Operating Systems Engineering

Recitation 4: Processes and switching

Based on MIT 6.828 (2014, lec10)
Plan

- Processes
- Threads
- Scheduling
Process

- Abstract virtual machine
- Motivated by isolation – provides the illusion to application of a dedicated computer, but an abstract one convenient for application developer
- One process cannot affect another accidentally
Process State

- **SLEEPING** - was in kernel and went to sleep
- **EMBRYO** - creation started, not ready to run yet
- **UNUSED** – slot is unused
- **RUNNABLE** – ready to run
- **RUNNING** – already running on CPU
- **ZOMBIE** – after exit(), waiting for parent's wait()
- enum procstat (line 2100)
Process API

- fork (2304)
- exec (5910)
- exit (2354)
- wait (2403)
- kill (2625)
- sbrk (3451)
- getpid (3438)
Challenge

- More processes than processors
- xv6 picture:
  - 1 user thread and 1 kernel thread per process;
  - 1 scheduler thread per processor; n processors
Terms

• **A process**: address space plus one or more threads
• **A thread**: thread of execution
• **Kernel thread**: thread running in kernel mode
• **User thread**: thread running in user mode
Thread

• Thread of execution
  – an abstraction that contains enough state of a running program that it can be stopped and resumed

• Switch between two threads
  – saving the old thread’s CPU registers, and restoring previously-saved registers of the new thread.

• xv6 API
  – yield (2522)
  – sched (2503)
Goals for solution

- Switching transparent to user threads
- User thread cannot hog a processor
  - kernel thread assumed to be correct, so not a goal
Switch between two user threads

- User threads
  - user -> kernel transition
  - kernel -> kernel switch
  - kernel -> user transition

- Guaranteed U->K transitions
  - timing interrupt every 100 ms
  - switches to different kernel thread on yield
  - the different kernel thread returns to a different user thread
Switch between two user threads: an example
Challenges in implementing

- Opaque code
  - "You are not supposed to understand this"
- Concurrency
  - several processors switching between threads
- Terminating a thread, always need a valid stack
xv6 design (1)

- **One scheduler thread per processor**
- **Simplifies implementation**
  - Need a stack to run scheduler on, if there are no other threads anymore
- **Downside**
  - More switches. To switch from one thread to another requires two switches: thread 1 -> scheduler -> thread 2
- **Advantage**
  - Scheduling organized as co-routines
  - Simplifies reasoning about concurrency. e.g., it is clear lock is always passed from current thread to scheduler thread
xv6 design (2)

• xv6 schedules user threads preemptively
  – Every 100ms a timer interrupt
• xv6 schedules kernel threads cooperatively
  – Assume kernel programmer doesn't make mistakes (e.g., no infinite loop)
xv6 code

• proc.c:scheduler() – 2458
• swtch.S:swtch() – 2708
• trap.c:trap() – 3101
• proc.c:yield() – 2522
swtch() and struct context

- swtch.S:swtch() – 2708
- swtch doesn't know about threads
  - only saves and restores registers sets called context
- swtch(old context, new context)
- struct context (2093):
  - %edi, %esi, %ebx, %ebp, %eip
U→K switch example

- Timer interrupt → trap() 3101
  - 3117: ticks++ and 3174: yield()

- yiled() 2522
  - proc->state = RUNNABLE
  - sched()

- sched() 2503
  - 2516: swtch(&proc->context, cpu->scheduler)
  - switch from current process's to scheduler (will run new process)
  - %eip was saved on stack by call instruction
  - when return from scheduler will continue from 2517!
K→U switch example

- When return from scheduler will continue from `yeild()` line 2517!
- Return from `sched()` back to `yiled()`
- `yield()` back to `trap()`
- K→U from the trap
  - `trap()` back to `trapret()` 3027 in `trapasm.S:alltraps()`
  - restore user context (popal + segments)
  - `iret`, user process continue
swtch (2708)

swtch(struct context **old, struct context *new)
2709: movl 4(%esp), %eax ← store old in %eax
2710: movl 8(%esp), %edx ← store new in%edx
2713-2716: pushl %ebp, %edx, %esi, %edi
  - save current context, %eip already saved on stack by call
2719: movl %esp, (%eax) ← save current %esp in old context
2720: movl %edx, %esp ← load %esp from new context
2723-2726: popl %edi, %esi, %edx, %ebp
  - load rest of new context, %eip will be restored with ret in 2727
Scheduling

- `sched()` 2516 and `scheduler()` 2478 are coroutines
  - use `swtch()` to switch between them
- `scheduler()` 2458 runs a simple loop
  - find a RUNNABLE process
  - run it until it stops
  - repeat
  - if no RUNNABLE
    - release lock
    - let other CPUs to update process table
    - acquire lock and start the loop again
Scheduling – locks

• yield() 2522 must acquire ptable.lock
  – same for sleep() 2553 or exit() 2354
  – and call sched() → scheduler() 2458
    • but scheduler() takes same lock again in 2467!
    • how is that?
    • we resume in 2478
  – xv6 transfers locks
    • Remember Mesa/Hoare semantic?
  – forkret() 2533 exists only to release lock
kill

- Killing Is My Business... And Business Is Good!
  - SYS_kill 3206 (sys_kill() 3428) is the only signal in xv6
- kill() 2625
  - must hold ptable.lock
  - find proc and set proc→killed = 1
  - process will be “killed” when it returns to user space
    - trap() kills in 3104, 3108, 3168, 3177 (calls exit())
exit

• exit() 2354
  – sys_exit() 3415
• Doesn't return
• Exited process remains in ZOMBIE state
• Wakes up parent with wakeup1() 2603
  – parent will clean after its child
• Makes init to be a parent of children of the exited process
• Jumps to scheduler and never returns
wait

- wait() 2403
  - sys_wait() 3422
  - cleans up after its children (stack 2419, pg table 2421, proc slot 2422)

- Flow:
  - acquiring ptable.lock
  - scan the process table looking for children
  - if finds one child in ZOMBIE state
    - perform cleaning, release ptable.lock and return child pid
  - if not, go to sleep() 2553 on proc channel (self) with ptable.lock
    - wait for childer to exit
    - special case when sleep() runs without holding ptable.lock
Questions?