```
6.S081 2020 Lecture 11: Thread switching

Topic: more "under the hood" with xv6
  Previously: system calls, interrupts, page tables, locks
  Today: process/thread switching

Why support multiple tasks?
  time-sharing: many users and/or many running programs.
  program structure: prime number sieve.
  parallel speedup on multi-core hardware.

Threads are an abstraction to simplify programming when there are many tasks.
  thread = an independent serial execution -- registers, pc, stack
  the threading system interleaves the execution of multiple threads
  two main strategies:
    multiple CPUs, each CPU runs a different thread
    each CPU "switches" between threads, runs one at a time

threads can share memory, or not
  xv6 kernel threads: they share kernel memory (thus locks)
  xv6 user processes: one thread per process, so no sharing
  linux: supports multiple threads sharing a user process's memory

there are other techniques for interleaving multiple tasks
  look up event-driven programming, or state machines
  threads are not the most efficient, but they are usually the most convenient

thread design challenges
  how to interleave many threads on a few CPUs?
    how to make interleaving transparent?
    "scheduling" = the process of choosing which thread to run next
  what to save while a thread isn't running?
  how to cope with compute-bound threads?

how to cope with compute-bound threads?
  each CPU has timer hardware, which interrupts periodically
  kernel uses timer interrupts to grab control from looping threads
  kernel saves thread state, switches, eventually resumes,
    restores that saved state for transparency
  RUNNING vs RUNNABLE
  this is "pre-emptive" scheduling -- a forced yield of unaware code
    as opposed to cooperative scheduling, in which code yields voluntarily

what to do with a thread that isn't running?
  we need to set aside its state: registers, stack, memory
    though no need to worry about memory, it won't go anywhere
  so implementation provides each thread with a stack and register save area
  need to track status of each thread
    RUNNING vs RUNNABLE vs SLEEPING

in xv6:
  [simple diagram: processes, user stack, trapframe, kernel stack]
  each process has two threads, one user, one kernel
    a process is *either* executing its user thread,
    *or* in a system call or interrupt in its kernel thread
  kernel threads share kernel memory / data structures
    thus the kernel is a parallel program
  we'll use "process" and "kernel thread" and "thread" as synonyms

overview of thread switching in xv6
  (the point: switch among threads to interleave many threads on each CPU)
  [diagram: P1, TF1, STACK1, swtch(), CTX1;
            CTXs, swtch(), STACKs, scheduler(), &c]
  TF = trapframe = saved user registers
```

```
  CTX = context = saved RISC-V registers
  getting from one process to another involves multiple transitions:
    user -> kernel; saves user registers in trapframe
    kernel thread -> scheduler thread; saves kernel thread registers in context
    scheduler thread -> kernel thread; restores kernel thread registers from ctx
    kernel -> user; restores user registers from trapframe

"context switch" -- the switch from one thread to another

scheduler threads
  there's one per core; each has a stack and a struct context
  kernel threads (processes) always switch to the current core's scheduler thread
    which switches to another kernel thread, if one is RUNNABLE
    there are never direct kernel thread to kernel thread switches
  the reason: the scheduler's separate stack simplifies
    cases like switching away from an exiting process
  the scheduler thread keeps scanning the process table until
    it finds a RUNNABLE thread (there may not be one!)
  if there is not RUNNABLE thread, the scheduler is "idle"

note:
  each core is either running its scheduler thread, or some other thread
    a given core runs only one thread at a time
  each thread is either running on exactly one core, or its registers
    are saved in its context
  if a thread isn't running, its saved context refers to a call
    to swtch()

struct proc in proc.h
  p->trapframe holds saved user thread's registers
  p->context holds saved kernel thread's registers
  p->kstack points to the thread's kernel stack
  p->state is RUNNING, RUNNABLE, SLEEPING, &c
  p->lock protects p->state (and other things...)

# Code

pre-emptive switch demonstration
  user/spin.c -- two CPU-bound processes
  my qemu has only one CPU
  let's watch xv6 switch between them

make qemu-gdb
gdb
(gdb) c
show user/spin.c
spin

you can see that they alternate, despite running continuously.
xv6 is switching its one CPU between the two processes.
how does the switching work?

I'm going to cause a break-point at the timer interrupt.
(gdb) b trap.c:207
(gdb) c
(gdb) finish
(gdb) where

we're in usertrap(), handling a device interrupt from the timer
(timerinit() in kernel/start.c configures the RISC-V timer hardware).

what was running when the timer interrupt happened?

(gdb) print p->name
(gdb) print p->pid
```

```
(gdb) print/x *(p->trapframe)
(gdb) print/x p->trapframe->epc

let's look for the saved epc in user/spin.asm
timer interrupted user code in the increment loop, no surprise

(gdb) step ... into yield() in proc.c
(gdb) next
(gdb) print p->state

change p->state from RUNNING to RUNNABLE -> give up CPU but want to run again.
note: yield() acquires p->lock
  since modifying p->state
  and to prevent another CPU from running this RUNNABLE thread!

(gdb) next 2
(gdb) step (into sched())

sched() makes some sanity checks, then calls swtch()

(gdb) next 7

this is the context switch from a process's kernel thread to the scheduler thread
  swtch will save the current RISC-V registers in first argument (p->context)
  and restore previously-saved registers from 2nd argument (c->context)
let's see what register values swtch() will restore

(gdb) print/x cpus[0].context

where is cpus[0].context.ra?
  i.e. where will swtch() return to?
  kernel.asm says it's in the scheduler() function in proc.c

(gdb) tbreak swtch
(gdb) c

we're in kernel/swtch.S
a0 is the first argument, p->context
a1 is the second argument, cpus[0].context
swtch() saves current registers in xx(a0) (p->context)
swtch() then restores registers from xx(a1) (cpus[0].context)
then swtch returns

Q: swtch() neither saves nor restores $pc (program counter)!
   so how does it know where to start executing in the target thread?

Q: why does swtch() save only 14 registers (ra, sp, s0..s11)?
   the RISC-V has 32 registers -- what about the other 18?
     zero, gp, tp
     t0-t6, a0-a7
   note we're talking about kernel thread registers
     all 32 user register have already been saved in the trapframe

registers at start of swtch:
(gdb) print $pc  -- swtch
(gdb) print $ra  -- sched
(gdb) print $sp

registers at end of swtch:
(gdb) stepi 28   -- until ret
(gdb) print $pc  -- swtch
(gdb) print $ra  -- scheduler
(gdb) print $sp  -- stack0+??? -- entry.S set this up at boot
(gdb) where
(gdb) stepi
```

```
we're in scheduler() now, in the "scheduler thread",
  on the scheduler's stack

scheduler() just returned from a call to swtch()
  it made that call a while ago, to switch to our process's kernel thread
  that previous call saved scheduler()'s registers
  our processes's call to swtch() restored scheduler()'s saved registers
  p here refers to the interrupted process

(gdb) print p->name
(gdb) print p->pid
(gdb) print p->state

remember yield() acquired the process's lock
  now scheduler releases it
  the scheduler() code *looks* like an ordinary acquire/release pair
    but in fact scheduler acquires, yield releases
    then yield acquires, scheduler releases
  unusual: the lock is released by a different thread than acquired it!

Q: why hold p->lock across swtch()?
   yield() acquires
   scheduler() releases
   could we release p->lock just before calling swtch()?

p->lock protects a few things:
  makes these steps atomic:
    * p->state=RUNNABLE
    * save registers in p->context
    * stop using p's kernel stack
    so other CPU's scheduler won't start running p until all steps complete
  makes these steps atomic and uninterruptable:
    * p->state=RUNNING
    * move registers from context to RISC-V registers
    so an interrupt won't yield() and save not-yet-initialized
      RISC-V registers in context.

scheduler()'s loop looks at all processes, finds one that's RUNNABLE
  keeps looping until it finds something -- may be idle for a while
  in this demo, will find the other spin process
let's fast-forward to when scheduler() finds a RUNNABLE process

(gdb) tbreak proc.c:474
(gdb) c

scheduler() locked the new process, then set state to RUNNING
  now another CPUs' scheduler won't run it

it's the other "spin" process:

(gdb) print p->name
(gdb) print p->pid
(gdb) print p->state

let's see where the new thread will start executing after swtch()
  by looking at $ra (return address) in its context

(gdb) print/x p->context
(gdb) x/4i p->context->ra

new thread will return into sched()

look at kernel/swtch.S (again)
```

```
(gdb) tbreak swtch
(gdb) c
(gdb) stepi 28 -- now just about to execute swtch()'s ret
(gdb) print $ra
(gdb) where

now we're in a timer interrupt in the *other* spin process
  in the past it was interrupted, called yield() / sched() / swtch()
  but now it will resume, and return to user space

note: only swtch() writes contexts (except for initialization)
      only sched() and scheduler() call swtch()
      so for a kernel thread, context.ra always points into sched()
      and for a scheduler thread, context.ra always points into scheduler()

note: sched() calls swtch() -- then swtch() returns to sched()
      but it's typically a *different* thread returning

sched() and scheduler() are "co-routines"
   each knows what it is swtch()ing to
   each knows where swtch() return is coming from
   e.g. yield() and scheduler() cooperate about p->lock and p->state
   different from ordinary thread switching, where neither
     party typically knows which thread comes before/after

Q: what is the "scheduling policy"?
   i.e. how does xv6 decide what to run next if multiple threads are RUNNABLE?
   is it a good policy?

Q: is there pre-emptive scheduling of kernel threads?
   yes -- timer interrupt and yield() can occur while in kernel.
   yield() called by kerneltrap() in kernel/trap.c
   where to save registers of interrupted kernel code?
     not in p->trapframe, since already has user registers.
     not in p->context, since we're about to call yield() and swtch()
     kernelvec.S pushes them on the kernel stack (since already in kernel).
   is pre-emption in the kernel useful?
     not critical in xv6.
     valuable if some system calls have lots of compute.
     or if we need a strict notion of thread priority.

Q: why does scheduler() briefly enable interrupts, with intr_on()?
   There may be no RUNNABLE threads
     They may all be waiting for I/O, e.g. disk or console
   Enable interrupts so device has a chance to signal completion
     and thus wake up a thread
   Otherwise, system will freeze

Q: why does sched() forbid locks from being held when yielding the CPU?
   (other than p->lock)
   i.e. sched() checks that noff == 1
   suppose process P1 holding lock L1, yields CPU
   process P2 runs, tries acquire(L1)
   P2's acquire spins with interrupts turned off
     so timer interrupts won't occur
     so P2 won't yield the CPU
     so P1 can't execute
     so P1 won't release L1, ever

Q: can we get rid of the separate per-cpu scheduler thread?
   could sched() directly swtch() to a new thread?
   so that sched() looks for next process to run?
   that would be faster -- avoids one of the swtch() calls
   yes -- but:
     scheduling loop would run on a thread's kernel stack
```

```
      what if that thread is exiting?
      what if another cpu wants to run the thread?
      what if there are fewer threads than CPUs -- i.e. too few stacks?
      can be dealt with -- give it a try!

 Summary
   xv6 provides a convenient thread model for kernel code
   pre-emptive via timer interrupts
   transparent via switching registers and stack
   multi-core requires careful handling of stacks, locks
   next lecture: mechanisms for threads to wait for each other
```