```
6.828 2014 Lecture 10: Processes, threads, and scheduling
=

Plan:
  homework
  process
  threads
  scheduling

# Homework

iderw():
  what goes wrong with adding sti/cli in iderw?
  what ensures atomicity between processors
  what ensures atomicity within a single processor?

filealloc():
  Q: could the disk interrupt handler run while interrupts are enabled?
  Q: does any any interrupt handler grab the ftable.lock?
  Q: what interrupt could cause trouble?
  Q: timer interrupt?
    maybe we will find out in this lecture

## Process

Process
 motivated by isolation

 idea: an abstract virtual machine provides the illusion to application of a
 dedicated computer but an abstract one convenient for application developer one
 process cannot effect another accidentally



Process API:
  fork
  exec
  exit
  wait
  kill
  sbrk
  getpid

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
  xv6 API: yield, swtch

Goals for solution:
  Switching transparent to user threads
```

```
    User thread cannot hog a processor  (kernel thread assumed to be correct, so not a goal)

  Overview of 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

  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

  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 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)

  # Code

  Forced switching:
    demo of two processes who don't invoke system calls
      look at process states
    proc.h
    clock interrupt
      lapic.c for SMP
      timer.c for uniprocessor
    walk through what xv6 does to guarantee switching
      breakpoint in trap at yield() (b: trap.c:136)
      get hog running (c 100)
      look at tf, in particular tf->eip
            interrupt arrived in user space, while running hog
      look at tf->trapno (timer interrupt), gets to yield
      get to swtch, look at contexts (p /x *cpus[0]->scheduler)
        Q: why should ncli be 1?
      look at eip before return from swtch  (we switched to scheduler thread)
      scheduler: switches to selected thread (set b proc.c:288)
      will return user space

  What is the scheduling policy?
    will the thread that called yield run immediately again?

  Concurrency
    plock held across swtch; why?
     yield(): p is set to runnable, p must complete switch before another scheduler choses p
    hard to reason about; coroutine style helps
     always enter and leave a thread at the same instruction
    can two schedulers select the same runnable process?
```

```
    why does scheduler release after loop, and re-acquire it immediately?
     To give other processors a chance to look at the proc table
    why does the scheduler enable interrupts?
     The scheduler maybe the only runnable thread
       All other threads maybe sleeping (e.g., waiting on I/O input)
     To briefly run the processor with interrupts enabled
       An interrupt may wakeup a sleeping thread


  Thread clean up
    let's look at kill()
      can we clean up killed process?  (no: it might be running, holding locks, etc.)
          before returning to user space: process kills itself by calling exit
    let's look at exit()
      can thread delete its stack?
        no: it has to switch off it!
      wait() does the cleanup
```