-*- mode: org -*-
#+STARTUP: indent

how to run connect to qemu from gdb
how to pass a break point multiple times (c n)

* <2011-10-03 Mon>: processes, threads, and scheduling

* Plan:
 process
 threads
 scheduling

* Process:
** 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
** API:
fork
exec
exit
wait
kill
sbrk
getpid

* Problem: 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 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
Scheduling organized as co-routines
Scheduler thread performs cleanup

* Code
** Forced switching:
*** demo of two processes who don't invoke system calls
**** look at process states
*** clock interrupt
lapic.c for SMP
timer.c for uniprocessor
*** walk through what xv6 does to guarantee switching
breakpoint in trap
get hog running (c 100)
look at tf, in particular tf->eip
look at tf->trapno (timer interrupt), gets to yield
get to swtch, look at contexts (p /x *cpus[0]->scheduler)
look at eip before return from swtch (we switched to scheduler thread)
scheduler: switches to selected thread (set b proc.c:278)
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 runnable, p must complete switch before another scheduler choses p
 - hard to reason about; coroutine style helps
 - can two schedulers select the same runnable process?
 - why does scheduler release after loop, and re-acquire it immediately? (run with interrupts!)
** 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

6.828 Operating System Engineering

Fall 2012