Assignment 5

CS 6960, Fall 2017

Due: September 21, 2017

Alan Humphrey

Construct a proof – a sound and detailed argument about the xv6 implementation – that in all circumstances it eventually time-slices away from a CPU-bound process. That is, no matter what happens, this process will eventually be descheduled. Write this as a text file or a PDF and submit it to github in an "assignment5" subdirectory.

Do not make generic arguments; instead, refer to specific pieces of code.

Explicitly state any assumptions that you make, such as "the lapic eventually delivers a timer interrupt to each core."

Your proof should run 0.5 to 1 pages of text, but could run longer if you include code snippets.

1 Argument

- Using Assumption 1: Once the bootstrap processor starts running C code within the entry point, `main()` in `main.c`, `lapicinit()` and `ioapicinit()` are both called to setup interrupt controllers that periodically issue interrupts.

```
// The timer repeatedly counts down at bus frequency from lapic[TICR] and then
    issues an interrupt.
// If xv6 cared more about precise timekeeping, TICR would be calibrated using
    an external time source.
lapicw(TDCR, X1);
lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
lapicw(TICR, 10000000);
```

- `startothers()` is also called from `main() in main.c`, which starts all non-boot (AP) processors. At this point we know we will have more than one process in play.
- `mpmain()` is finally called from within `main` in `main.c`, which then calls `scheduler() in proc.c`, which starts running processes.
- Also from within `main.c, mpenter()` is called which makes other CPUs jump here from `entryother.S`.

```
static void
mpenter(void)
{
  switchkvm();
  seginit();
  lapicinit();
  mpmain();
}
```

- At this point, we can claim that "... *the lapic eventually delivers a timer interrupt to each core*".
- Using Assumption 2: We need to now show that a timer interrupt will cause a trapframe to be built.
- As shown in `trap.c` (code below), the case `T_IRQ0 + IRQ_TIMER`, which was set in `lapicinit()`, etc and then calls `lapiceoi()`, acknowledging the interrupt.
- The call to `wakeup()` in turn calls `wakeup1()`, queueing the process (in the run queue we implemented) and setting its state to `RUNNABLE`.

```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
...
// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
```

```
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
...
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      enqueue_proc(&ptable.ready_queue, p);
}
```

- Because of the round-robin scheduling, every process is then guaranteed to be run eventually via this mechanism.
- Using Assumption 3: We can now assert that "in all circumstances xv6 eventually time-slices away from a CPU-bound process". There is clearly some handwaving with this assumption and I need to further strengthen this portion of the argument. However, should this portion be airtight, I would argue the conclusion here with confidence.

2 Assumptions

1. Everything run prior to main entry point runs normally and successfully.
2. All CPUs are setup to recieve the interrupts mentioned in the first code listing above
3. Nothing within the kernel can disable interrupts without a re-enabling them.