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

The principal aim of this proof centers around showing that any arbitrary process eventually reliquishes the CPU it is running on by calling yield, in `trap()` function - line 106 of `trap.c` (Listing 1). In order to arrive at this conclusion, my argument shows that:

1. Interupts and interupt vectors are all setup correctly and that all CPUs are setup to recieve and acknowledge timer interrupts, e.g. "*the lapic eventually delivers a timer interrupt to each core*".
2. A timer interrupt will cause a trapframe to be built.
3. No process can disable interrupts without a re-enabling them.

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

Listing 1: yield() call in trap.c

---

2 Assumptions

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

---

3 Proof

- 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`.
- At this point, using Assumption 2, we can claim that all interrupt vectors are setup and that "... *the lapic eventually delivers a timer interrupt to each core*".
- 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 a process (in the run queue we implemented) and setting its state to `RUNNABLE` (Listing 2).

```
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);
}
```

Listing 2: trap.c switch case and wakeup() calls

- Yield is eventually called for a non-killed process, which in turn calls sched, which calls `swtch` to save the current context in proc− >context and switch to the scheduler context previously saved in cpu− >scheduler (Listing 3)

```
  // Force process exit if it has been killed and is in user space.
  // (If it is still executing in the kernel, let it keep running
  // until it gets to the regular system call return.)
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();

  // Check if the process has been killed since we yielded
  if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
    exit();
}
```

Listing 3: trap.c yield() and exit() calls

- Because of the round-robin scheduling, every process is then guaranteed to be run eventually via this mechanism.

---

4 Conclusion

- Using Assumption 3, we can now claim that *"in all circumstances xv6 eventually time-slices away from a CPU-bound process"*. Though I did not perform an exhaustive search, it looks like nowhere in the kernel are interrupts disabled and then not re-enabled, e.g. a `pushcli()` without a corresponding `popcli()`. Clearly some handwaving here, however should this portion be airtight, I would argue the conclusion here with more confidence.