Lecture 12 Scribe Notes

Dan Zangri & Myles Novick

Pre-emptive Multitasking is an alternative to *Cooperative Multitasking*, which is a way to force a process to give up the CPU. This solves an infinite loop attack, because then no process can have the CPU forever.

To do this, we need special features from the CPU, including *interrupts*. An *interrupt* or *exception* is an involuntary control transfer. In contrast, a *jump* is a voluntary control transfer. This means an interrupt causes the CPU to change its program counter from one location to another, but not via a jump. This means the CPU jumps to a new PC (%esp) due to some "external" event and signal (e.g. a printer dies or a key is touched). This allows the CPU to handle the hardware requirements.

There are also *traps*, which are called by software. *Faults* are another type that are called by software errors.

So how can we implement this? Suggestion 1: a ticking clock that interrupts a process periodically and automatically. This means any infinite loop will always be interrupted. This is called a *timer interrupt*.

QEMU's timer interrupt:

```
  // All other processes' special registers can be copied from the
  // first process
  segments_init();
  interrupt_init();
  paged_virtual_memory_init();
  timer_init(1000)
  //Erase the console, and initialize the cursor position shared
  // variable to polar to its upper left.
  console_clear();
}
```

Looking at timer_init() :

```
void timer_init(int rate) {
  // if the clock interrupt is enabled, initialize the clock
  if (rate > 0) {
    outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
    outb(IO_TIMER1, TIMER_DIV(rate) % 256);
    outb(IO_TIMER1, TIMER_DIV(rate) / 256);
    interrupts_enabled |= 1 << (INT_CLOCK - INT_HARDWARE);
} else
  interrupts_enabled &= ~(1 << (INT_CLOCK - INT_HARDWARE));
  interrupt_mask();
}
```

outb is a programmed I/O instruction. The timer interrupt is hardware, and therefore it needs programmed I/O or memory-mapped I/O instructions.

```
void interrupt(struct registers *reg) {
  // The processor responds to an interrupt by saving some of the
  // application's state on the kernel's stack, then jumping to
  // kernel assembly code (in os01-int.S). That code saves more
  // registers on the kernel's stack, then calls interrupt(). The
  // first thing we must do is copy the saved registers into the
  // 'current' process descriptor.
  current->p_registers = *reg;

  switch (reg->reg_intno) {

  case INT_SYS_GETPID:
  current->p_registers.reg_eax = current->p_pid;
  run(current);

  case INT_SYS_YIELD:
    schedule();

  case INT_TIMER:
    console_moveto(console_printf(cursorpos, 0xC00, "."));
    run(current);

  default:
  console_printf(cursorpos, 0x0C00, "\nUnexpected interrupt %d!\n",
  reg->reg_intno);
  loop: goto loop;
  }
}
```

In this function, different things happen if an interrupt occurs. It gets control of the CPU when one of these interrupt signals are sent. The highlighted code is the code for adding the timer interrupt. This allows the kernel to periodically (pun!) take control from the infinite loop.

However, we need to make the process pass off to another (like sys_yield). Here, we have schedule:

```
void schedule(void) {
  pid_t pid = current->p_pid;
  while (1) {
    pid = (pid + 1) % NPROCS;
    if (processes[pid].p_state == P_RUNNABLE)
      run(&processes[pid]);
  }
}
```

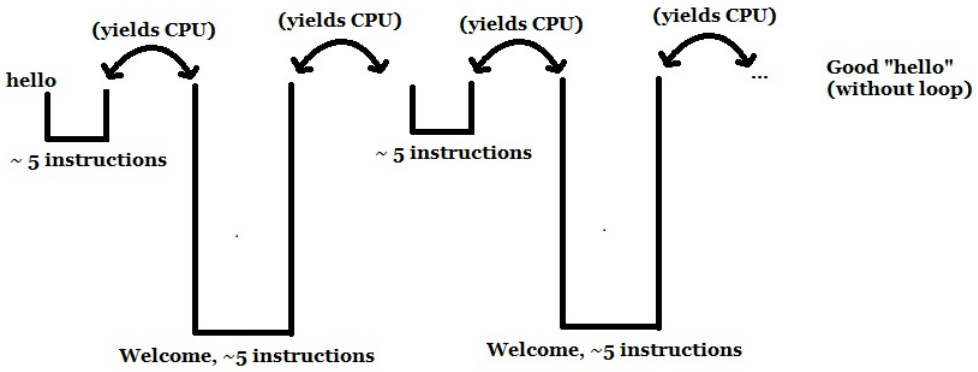
Which is a lot like an array search in an array of processes. It searches for and then runs a runnable process. So now we go back to our interrupt code.

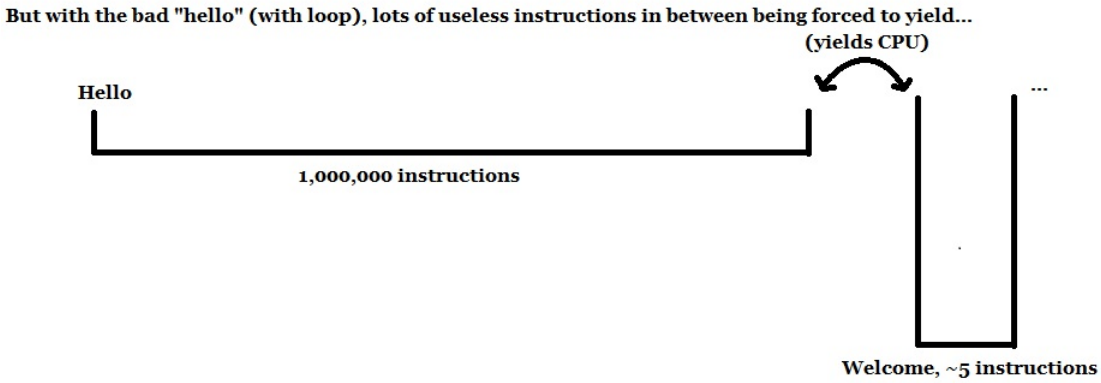
```
void interrupt(struct registers *reg) {
  // The processor responds to an interrupt by saving some of the
  // application's state on the kernel's stack, then jumping to
  // kernel assembly code (in os01-int.S). That code saves more
  // registers on the kernel's stack, then calls interrupt(). The
  // first thing we must do is copy the saved registers into the
  // 'current' process descriptor.
  current->p_registers = *reg;

  switch (reg->reg_intno) {

  case INT_SYS_GETPID:
  current->p_registers.reg_eax = current->p_pid;
  run(current);

  case INT_SYS_YIELD:
    schedule();

  case INT_TIMER:
    console_moveto(console_printf(cursorpos, 0xC00, "."));
    schedule();
```

```
22
23      default:
24      console_printf(cursorpos, 0x0C00, "\nUnexpected interrupt %d!\n",
25      reg->reg_intno);
26      loop: goto loop;
27      }
28  }
```

Which allows our "Welcome!" message to appear. Infinite loop solved (!?!?)

When we take away the loop, the two message resume at the pace that they originally were. This means our "Welcome" message is executed much more quickly than with the loop. Relative to how often the processes normally yield the CPU, we're doing few timer interrupts.





The *kernel* provides access to hardware, and ensuring the the division of access is "fair". The kernel defines fairness, though it ensures things like infinite loops do not happen. As long as processes are not allowed to define fairness, then the kernel is successful.

The timer interrupt allows "Welcome" to run. But "Hello" can call a command *cli* that disables interrupts. This is a dangerous instruction.

Every CPU has *safe* and *dangerous* instructions. In general safe instructions may not violate process isolation ("fairness"). Dangerous instructions can violate process violation. Therefore, dangeous instructions should only be allowed to be called by the kernel.

Processes and kernels, to the CPU, are not distinguishable by themselves. Instead, there are special flags in registers for the CPU that indicate to the CPU if a process has special privileges attributed only to the kernel.

A dumb kernel will give processes the ability to call dangerous instructions. *cli* is an example.

How can this happen? Because of this fragment:

```
void process_init(proc *p) {
  memset(&p->p_registers, 0, sizeof(p->p_registers));
  p->p_registers.reg_cs = SEGSEL_APP_CODE | 3;
  p->p_registers.reg_ds = SEGSEL_APP_CODE | 3;
  p->p_registers.reg_es = SEGSEL_APP_CODE | 3;
  p->p_registers.reg_ss = SEGSEL_APP_CODE | 3;
  // Enable interrupts
  p->p_registers.reg_eflags = EFLAGS_IF | EFLAGS_IOPL_3;
```

Remove this, and the *cli* can no longer turn off interrupts (that is, the process can no longer call *cli*)!

Removing permissions to call *cli* causes the process to jump (involuntarily) to the kernel if it attempts to call *cli* (an involuntary control transfer!). In this cause, it is a *fault*. Executing a "dangerous" instruction caused this (and it is instead turned into an interrupt). This is called a *general protection fault*.

We can give instructions on what to do if we see a *general protection fault*, we can give instructions to halt the program (like if we notice something wrong with a process like a seg fault) and instead run a different process (with schedule()).

To do this, we add some new code. In the interrupt function:

```
void interrupt(struct registers *reg) {
  // The processor responds to an interrupt by saving some of the
  // application's state on the kernel's stack, then jumping to
  // kernel assembly code (in os01-int.S). That code saves more
  // registers on the kernel's stack, then calls interrupt(). The
  // first thing we must do is copy the saved registers into the
  // 'current' process descriptor.
  current->p_registers = *reg;

  switch (reg->reg_intno) {

  case INT_SYS_GETPID:
  current->p_registers.reg_eax = current->p_pid;
  run(current);

  case INT_SYS_YIELD:
    schedule();

  case INT_TIMER:
    console_moveto(console_printf(cursorpos, 0xC00, "."));
    schedule();

  case INT_GPF:
   current-> p_state = P_BLOCKED;
   schedule();

  default:
  console_printf(cursorpos, 0x0C00, "\nUnexpected interrupt %d!\n",
  reg->reg_intno);
  loop: goto loop;
  }
}
```

So no more evil commands!