CS61 Scribe Notes
Process isolation; Multiprocessing
Michelle Ran, Scott Zhuge
10/11/2012

<u>Virtual machine</u>

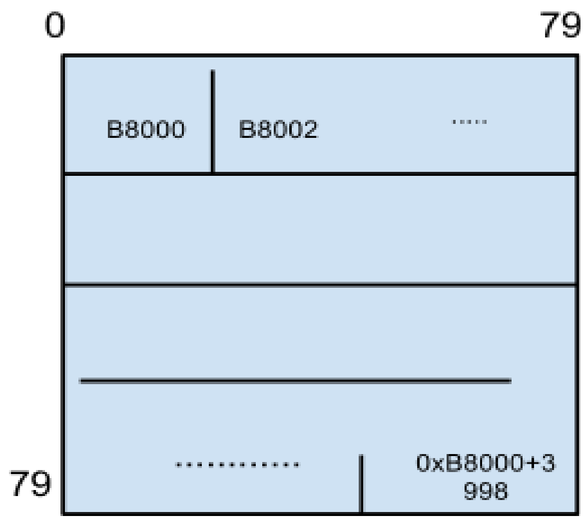
- Software implementation of a computer
- Analogous to a CPU with a physical chemical screen “dreaming” of a CPU with another virtual screen which translates instructions to the physical screen

The following code creates a colored @ sign:

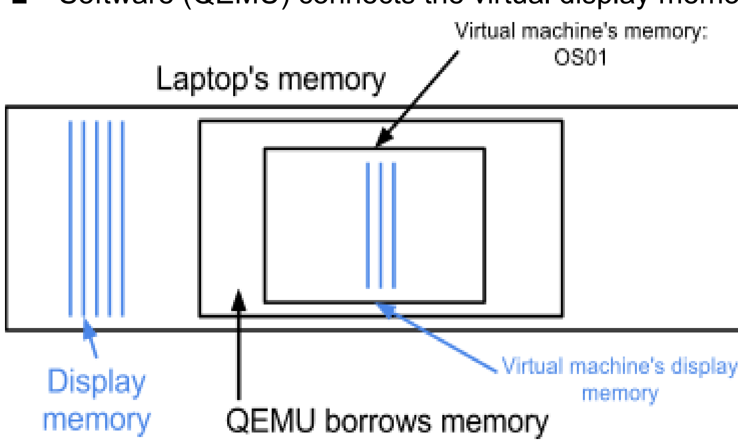
```
#include "os01-app.h"
#include "lib.h"
#include "x86.h"

void process_main(void) {
    unsigned i = 0;
    uint16_t *console = (uint16_t*) 0xB8000;
    *console = 0x8A00 | '@';
    while (1)
        ;
    while (1) {
      ++i;
      if (i % (1 << 10) == 0)
          app_printf(0, "Hello #%x!\n", i);
      sys_yield();
    }
}
```

<u>CPUs and hardware</u>

- Programmed I/O
    - Special instructions to interface with hardware devices
    - E.g. `inb`, `inw`, `outb`, `outl`, etc.
- Memory mapped I/O
    - Region of memory is used to interact with device
    - What region of memory is used to interact with the virtual VGA console?
        - VGA console on x86 hardware is mapped as an array of 16-bit ints @0xB8000



            - Upperleft corner of console screen stored at 0xB8000, next is 0xB8002, down to the bottomright corner ar 0xB8000 + 3998
            - As CPU puts data into memory, the data is interpreted by the graphics card to be put on the screen
            - Question: where does 3998 come from?
              3998 = 2 (80 * 25 - 1) (80x25 dimensions for the screen, each is 2 bytes wide)
- How does a virtual machine work?
    - Inside virtual machine is an operating system
    - QEMU processor emulator borrows memory from the laptop
        - QEMU is the virtual version of hardware
    - Inside the QEMU memory borrowed, it chooses a region for the virtual machine's memory: the OS01 memory
    - QEMU code is stored in QEMU memory, which is outside the virtual machine's memory
    - Inside laptop memory, there is a region for display
    - Inside virtual machine memory (inside QEMU memory) also has display memory, implemented identically with a normal display memory
        - Hardware connects the normal display memory and screen
        - Software (QEMU) connects the virtual display memory with screen



- What makes virtual machines possible?
    - Information is bits + context

- You think of instructions as something a CPU can execute
  - E.g. two bytes 0xEB 0xFE correspond to L2: jmp .L2
    ```
    char *pc =  …;
      if (pc[0] == 0xEB && pc[1] == 0xFE)
        infinite loop;
    ```
  - Not only can a processor interpret those instructions as a loop, because you can write a different program to interpret those instructions differently
- Representation of programs and data as memory allows us to do virtual machines
- Stored program computers (store instructions in memory), allow for virtual machines

```
#include "os01-app.h"
#include "lib.h"
#include "x86.h"

void process_main(void) {
    unsigned i = 0;
    uint16_t *console = (uint16_t*) 0xB8000;

    while (1) {
      ++i;
      if (i % (1 << 10) == 0)
          app_printf(0, "Hello #%x!\n", i);
    while (1) {
        ;
    }
      sys_yield();
    }
}
```

- Above code hogs all memory, and doesn't let the other operating system run
- With QEMU, can debug entire computer with GDB since it is just a program

Examine process_main, which is the first thing executed when the machine boots up

- Single-stepping through the gdb for the virtual machine shows up characters on the screen (in this case "HA HA HA HA" in yellow)
- How to fix infinite loop?

Welcome code:

```
#include "os01-app.h"
#include "lib.h"

void process_main(void) {
    unsigned i = 0;

    while (1) {
      ++i;
      if (i % (1 << 10) == 0)
          app_printf(1, "Welcome #%x!\n", i);
      sys_yield();
```

```
    }
}
```

- `sys_yield` is a system call which allows other programs to run; implements something called <u>cooperative multitasking</u>
  - This means that processes voluntarily give up CPU (cooperative)
  - Advantages: efficient
  - Disadvantages: vulnerable, because processes can just enter into infinite loops
- Alternative is <u>preemptive multitasking</u>
  - A process can be forced to give up the CPU involuntarily
  - Solves infinite loop attack, because processes can be forced to give up CPU
  - Requires special features from the CPU

<u>Interrupts and exceptional control flow</u>

What is an interrupt (exception)?

- Involuntary control transfer
  - Jump instruction is an example of a *voluntary* control transfer
  - CPU changes program counter (`%eip`) from one memory location to another due to an external event
    - Interrupts -> caused by hardware (e.g. printer dies)
      - Signals are sent to CPU so that the CPU can handle the hardware's requirements
    - Traps -> caused by software (e.g. system call)
    - Faults -> software error (e.g. accessing memory that doesn't exist)
- To prevent infinite loops, have a "ticking clock" that periodically interrupts the CPU so that another piece of software can run something else, called a timer interrupt

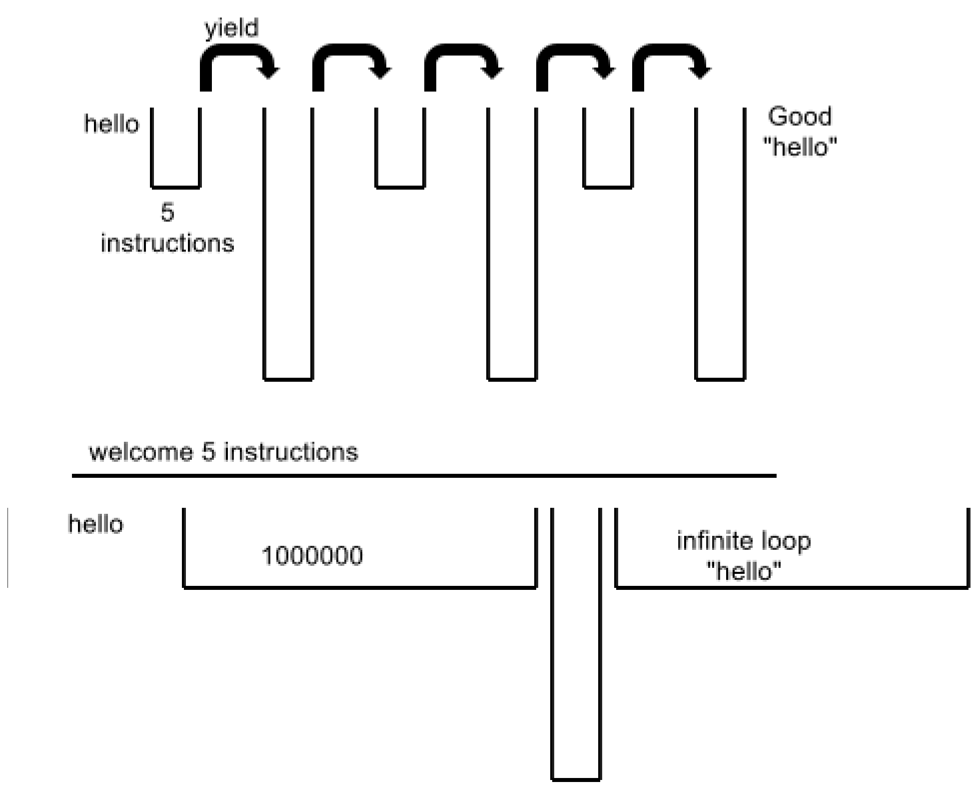
```
void timer_init(int rate) {
    // if the clock interrupt is enabled, initialize the clock
    if (rate > 0) {
      outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
      outb(IO_TIMER1, TIMER_DIV(rate) % 256);
      outb(IO_TIMER1, TIMER_DIV(rate) / 256);
      interrupts_enabled |=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

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();

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

    }
}
```

- Function `interrupt` gets control of the CPU whenever an interrupt happens
- Can use a timer interrupt to stop a program from running too long... but this doesn't allow us to run the other process
    - Need to force it to run another process, system yield

```
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]);
    }
}
```

    - Call `schedule`, which simply searches an array of processes for one to run
- Not doing enough timer interrupts compared to how often hello yielded the CPU



    - `hello` (not infinite loop version) executes five instructions, then yields the CPU

    - `welcome` executes five instructions, then yields the CPU
    - `hello` (infinite loop) executes as many instructions as possible until the timer interrupts it (it gets away with 1 million instructions), then yields the CPU
- Kernel divides fair access to hardware resources among the processes
    - Allowing a single process to monopolize hardware is unfair
    - Successful kernels prevent processes from monopolizing resources
- However, the function `cli()` can disable timer interrupts
    - `cli()` is a dangerous instruction

Safe instructions vs dangerous instructions

Safe Instruction	Dangerous Instruction ⚠
CANNOT violate process isolation (fairness property) One process cannot isolate CPU/kill another process unless it has permission	Dangerous instructions can violate process isolation ● Should be kernel-only ● Set of flags loaded into special registers determine if the program running as kernel or application privilege

General protection fault

- Interrupt, involuntary control transfer into the kernel