Module R3/R4 Dispatching

Introduction

- Modules R3 and R4 are due at the same time
- Implement context switching and support for multiple processes
- Majority of the code will be written in C, but some Assembly is required (lol)

R3 Goals

- Track particular processes with PCB's
- Support sys_req(IDLE, ...) and sys_req(EXIT, ...)
- Switch out the running process
- Add temporary commands to test above
  - Adding new processes
  - Execute processes until all are complete, then return to comm_hand

R4 Goals

- Treat comm_hand as just another process
- Prevent your polling function from blocking
- Remove temporary commands
- Add an alarm command
- Add an infinitely running process, that must be user suspended to be killed

Review: A Program in Execution

- The CPU has assorted registers
  - Most notably, the Instruction Pointer, the Stack Pointer, and the Base (of the Stack) Pointer
- All of the variables are stored on the stack
  - Heaps are done in R5
- That’s all the values necessary to save, load, fork a running process, and are collectively referred to as the ‘context’

How Interrupts Work

- Similar to a function call
  - A ‘return address’ is pushed to the stack
  - Jump to the interrupt table
  - Execute the function at that spot in the table
  - Jump back to the return address
- The interrupt is responsible for preserving the register values
  - If the interrupt needs to use/change a register value, it must push it at the beginning of the function and pop it at the end of the function

Context Switch

- What is a context switch?
  - An event where one process gives up the processor for another process.
    Overview of a context switch:
    - Save the hardware state (i.e. the values of the CPU registers) of the currently-operating process
    - Reinsert the currently-operating process into the appropriate queue
    - Select the next process to begin or continue executing
- What is a context?
  - The top 60 bytes of the stack
  - Contains the 15 CPU register1 values needed to begin and resume execution

```
struct context {
  u32int gs, fs, es, ds;
  u32int edi, esi, ebp, esp, ebx, edx, ecx, eax;
  u32int eip, cs, eflags;
};
```

- Look here for more info
  - http://int.cs.yale.edu/cs422/doc/pc-arch.html

Context Switching

- How does the system decide when a context switch will occur?
  - In MPX, a process voluntarily gives up control of the CPU by generating a software interrupt
  - Processes will call sys_req(IDLE); to give up control
  - sys_req(IDLE); will generate an interrupt on line 60h
    - Check out mpx_supt.h for more info about sys_req
  - Your interrupt handler will perform the context switch.

Prerequisite Knowledge

- What is the calling convention gcc uses for IA-32 CPUs?
  - Function parameters are pushed onto the stack in reverse-order (that is: the first parameter is the last to be pushed onto the stack)
  - The lower 32 bits of the return value (all we care about for this project) is stored in the EAX register
- What must an interrupt handler do when an interrupt occurs to avoid disturbing the state of a running process?
  - Save the contents of all general purpose registers (i.e. push the values onto the stack)
  - Service the interrupt
  - Restore the computer's state (all CPU register values)

Prerequisite Knowledge

- How can the aforementioned description of an interrupt handler be used to perform a context switch?
  - Simple! - Dedicate a stack to each process and "switch" stacks. For example:
    - Push all general purpose CPU register values onto the stack
    - Switch stacks
    - Pop all general purpose CPU register values o of the stack - Note that this pops different values into the registers than those saved in step 1.
- How much assembly language do I need to know?
  - pusha - Push all general purpose CPU register values onto the stack
  - popa - Pop values o of the stack into all general purpose CPU registers
  - push - Push the value given by the operand onto the stack
  - pop - Pop the value o of the top of the stack into the location
  - described by the operand
  - mov - Move the contents of one location into another
  - call - Call a subroutine
  - iret - Return from interrupt

R3

- For R3 and R4, you will need to add two new commands (more on this later), fill in the body of the system call interrupt service routine (in irq.s) and write a system call function which prepares MPX for the next ready process to begin/resume execution.
- You may write your system call function in C or assembly
  - C is advised
  - u32int* sys call(context *registers)
- GCC is incapable of creating "naked functions", so the Interrupt Service Routine stub must be written in assembly language

sys_call_isr

- Push all general purpose register values to the stack
- Push the segment register values to the stack (in the order: ds, es, fs, gs)
- Push esp, a register serving as an indirect memory operand pointing to the top of the stack at any time, this will be the parameter of your sys_call function
- Call sys_call
- Set a new stack pointer (returned by your sys_call function)
  - Recall that the return value of your sys_call function will be stored in eax
- Pop values into your segment registers in reverse order (from step 2)
- Pop values into general purpose registers
- Return from interrupt
- This is one of the easiest parts of R3/R4, come to me if you find yourself struggling, I don't want you to be stuck here

sys_call

- Declare a PCB* cop as a global variable, representing the currently operating process
- Use the prototype u32int* sys_call(context* registers)
- Check to see if sys call has been called before (i.e. if the currently-operating process (cop) is NULL, sys call has not been called).
  - If sys_call has not been called, save a reference to old (the caller's) context in a global variable.
  - If sys_call has been called check params.op code
    - If params.op code == IDLE, save the context (reassign cop's stack top)
    - If params.op code == EXIT, free cop
- If there is a ready process:
  - Remove it from the ready queue, set state to running
  - Assign cop
  - Return cop's stack top
- Otherwise, return the context saved in step 1

User Commands

- yield
  - The yield command (temporary – only in R3) will cause commhand to yield to other processes (i.e. voluntarily give up CPU time)
  - If any processes are in the ready queue, they will be executed
  - Yield is literally a one-liner:
  - asm volatile ("int $60");
- loadr3
  - loadr3 will load all r3 ”processes” (proc3.c file eCampus) into memory in a suspended ready state at any priority of your choosing

```
pcb * new_pcb = create_pcb ( name , 1 , 1 , 1 , stack_size );
context * cp = ( context *)( new_pcb -> stack_top );
memset ( cp , 0, sizeof ( context ));
cp - > fs = 0 x10 ;
cp - > gs = 0 x10 ;
cp - > ds = 0 x10 ;
cp - > es = 0 x10 ;
cp - > cs = 0 x8 ;
cp - > ebp = ( u32int )( new_pcb -> stack );
cp - > esp = ( u32int )( new_pcb -> stack_top );
cp - > eip = ( u32int ) func ;// The function correlating to the process , ie. Proc1
cp - > eflags = 0 x202 ;
return new_pcb ;
```

VERY IMPORTANT

- In order for your interrupt handler to hook an interrupt request on line 60, you need to enter the appropriate interrupt handler in the interrupt descriptor table (kernel/core/interrupts.c)
  - idt_set_gate(60, (u32int)sys_call_isr, 0x08, 0x8e);
- In setupPCB() from R2, be sure to set your stacktop to:
  - stackbase + 1024 - sizeof(struct context)

R4

- R4 is to demonstrate continuous dispatching and to allow commhand to compete for resources (it becomes a process)
- The deletePCB() command is no longer temporary due to new changes, and will be used in this module
- Inside kmain(), the following will need to happen:
  - Remove the call to commhand
  - Add commhand and idle to your ready queues (see the previous slide concerning loading R3 processes).
  - Trigger a software interrupt on line 60
- Remove the yield command from the user

R4 - Alarm

- Create an alarm command/process with 2 components
  - Message – Some user specified string
  - Time – When message should be printed
  - Process idles when a time has not yet occurred
  - Process prints the message and exits at or after that time
- System should support multiple concurrent alarms
- It would be wise to keep the alarm PCB parameterless, and upon its creation/deletion, update a list of times for it to check

R4 – Infinite Command/Process

- Should forever call sys_req(IDLE...)
  - If not suspended, the process can not be deleted
  - If suspended, it may be deleted
- Should be implemented as a process
  - Process idles when not yet suspended
  - Process prints a message every time it runs, to show it is properly running

Questions?