CS 450 Module R3 Dispatching

West Virginia University

Fall 2025

Goals

Goals

- Support cooperative multitasking via `sys_req(IDLE)` and `sys_req(EXIT)`
- Implement commands to demonstrate processes loading, executing, and exiting properly
- Some Assembly required ;)

Background

Review: A Process is a Program in Execution

- The CPU has various registers – Most relevant here are the Instruction Pointer and the Stack Pointer
- Variables and the function call stack (list of future Instruction Pointers) are stored on the stack (on x86 – other architectures differ)
- Heaps come in R5
- Registers, stack, and heap are everything necessary to save, load, or fork a running process – collectively called the ***context***

Context Switch

- An event where one process gives up the CPU to another
  - Save the process *context* of the currently running process (we'll use the PCB stack space for this)
  - Move the pausing current process to the appropriate queue
  - Select the next process to execute
  - Load the new process's context
- Minimal Context on x86 (a `struct` should hold all of these registers):
  - Segment Registers: CS, DS, ES, FS, GS, SS
  - Status Control Registers: EIP, EFLAGS
  - General Purpose Registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- More info: http://flint.cs.yale.edu/cs422/doc/pc-arch.html

When to Context Switch in MPX

- MPX is *cooperatively* multitasking – processes willingly give up the CPU
- An MPX process yields the CPU by calling `sys_req(IDLE)`
- An MPX process terminates by calling `sys_req(EXIT)`
- You will implement support for both of these operations in this module

How C Functions Work (on 32-bit x86)

- Function parameters are *pushed* onto the stack, from right-to-left (i.e. the first parameter is pushed last)
- The compiler issues a `call` instruction with a function's address as its parameter
- The CPU pushes the address of the next instruction (the *return address*) onto the stack
- The CPU jumps to the address provided and executes until a `ret` instruction
- The CPU *pops* the return address from the stack and resumes execution there
- The ***return value*** of the previous function is in EAX

How Interrupts Work (on 32-bit x86)

- An `int` instruction is issued with a one-byte IRQ parameter
- The CPU *pushes* (in order) EFLAGS, CS, and EIP (the *return address*) onto the stack
- CPU examines the interrupt table and jumps to the Interrupt Service Routine (ISR) referenced by the IRQ in the interrupt table
- Execution continues until an `iret` instruction
- The CPU *pops* the EIP(return address), CS, and EFLAGS from the stack
- Ordinary execution resumes at the return address

Writing ISRs

ISR-> Interrupt Service Routine

- Requirements placed on an ISR
  - Save the contents of general purpose registers (e.g. `push` them onto the stack)
  - Service the interrupt
  - Restore the contents of the general purpose registers (e.g. `pop` them from the stack
  - Return (with the `iret` instruction, **not** `ret`)
- Your ISR can use the stack space in each PCB for context switching
  - Reserve enough space at the top for register contents
  - `YOUR ISR will push` and `pop` from that space when switching

Intel Assembly

Look at file sys_call_isr.s for a ISR stub

- `push` – Push the specified value or register onto the stack
- `pusha` – Push all general purpose registers onto the stack
- `pop` – Pop the last value from the stack to the specified register
- `popa` – Pop all the general purpose registers from the stack
- `mov` – Copy data from one memory address or register to another memory address or register
- `call` – Call a function
- `iret` – Return from an ISR

Kernel Functions

ISR `sys_call_isr` (A stub is provided in `sys_call_isr.s`

- Clang cannot directly generate complex ISRs, so at least some of the ISR must be written in Assembly
- You will probably find things easier to implement most of it in C (referred to here as `sys_call()`), and call that from the Assembly stub

ISR Assembly Outline

- Remember that the CPU pushes EFLAGS, CS, and EIP automatically (in that order)
- Push all general purpose registers and remaining segment registers onto the stack in the opposite order as they are defined in your context struct
- Push ESP – At this point, ESP is a pointer to a manually created context struct, and pushing it makes it a C function parameter
- Call your C function
- Set ESP based on the return value of your function (EAX)
- Pop registers in order of your `struct`
- Return from ISR

struct context *sys call(struct context *)

- Take one parameter that is a pointer to a struct representing the context of the current process
- Return a pointer to the context of the process to be loaded
- You'll need a global PCB pointer representing the currently executing process – it should initially be `NULL`
- You'll also need a global or static context pointer representing the context from the first time `sys_call()` is entered
  - This should initially be `NULL`
  - If it is `NULL`, set it to point to the context provided as a parameter
  - When the last process issues an `EXIT` request, this is the context you will load

System Call Parameters and Return Value

- How do the parameters to `sys_req()` (e.g. `IDLE`) get to `sys_call()`?
- The interrupt convention doesn't place them on the stack, so MPX places them in registers EAX, EBX, ECX, and EDX (in that order)
- This is done by sys_req
- So, in the parameter to your C function, the EAX member will have the operation code (e.g. `IDLE`)
- Similarly, when your C function returns, it should set the EAX member to the return value that `sys_req()` will see and return
- **If the operation code is anything but `IDLE` or `EXIT`, do not load any new context and set the return value to `-1`**

System Call Operation: IDLE

- If there are any ready, non-suspended PCBs in the queue, remove the first from the queue and store it in a temporary variable as the next process
- Save the context of the current PCB by updating its stack pointer
- Add the current PCB back to the queue
- Return the context of the next process
- If the PCB queue is empty, or only consists of blocked or suspended PCBs, continue with the current process
- In all cases, ensure that the return value seen by `sys_req()` is 0

System Call Operation: EXIT

- Delete the currently running PCB
- If there are any ready, non-suspended PCBs in the queue, load the first as in `IDLE`
- If the PCB queue is empty, or only consists of blocked or suspended PCBs, load the original context
- In all cases, ensure that the return value seen by `sys_req()` is 0

User Commands

Yield

Pausing your command handler

- Cause Command Handler to yield the CPU
- If any processes are in queue, they will execute
- Literally a one-liner: `sys_req(IDLE);`

Load R3 -- there will be 3 versions

- LoadR3 -- Loads the R3 test processes from `<processes.h>`
- Each process (one per function) is loaded and queued in a non-suspended ready state, with a name and priority of your choosing
- Initialize and save the context for each process at the top of the PCB stack:
  - CS must be `0x08`, all other segments `0x10`
  - EBP must be the bottom of the PCB stack
  - ESP must be the top of the PCB stack
  - EIP must be a pointer to the function (the name of the function, without parenthesis, is a pointer to that function)
  - EFLAGS must be `0x0202`
  - All other registers should be `0`

- LoadR3Suspended –
  - Loads all R3 processes but sets the status to ready suspended
- LoadProcess <name> --
  - Loads a single process giving one of the process names proc1 -> proc5
  - The process will be loaded as a ready non-suspended process
- Resume all -- resumes all ready suspended processes

Remove Command: Create PCB

- Disable the Create PCB Command completely

Final Notes

Final Notes

- Remember to update:
  - Commands: Version and Help
  - Documentation: User's and Programmer's Manuals; Contributions
- Remember to set aside space for context in PCB stack – you may need to modify `setup_pcb()` to account for this
- Pay attention to compiler output to fix errors and warnings
- Use the debugger
- Don't confuse the members of your context struct with their use outside the struct
- This is the most difficult module – procrastination *will* negatively affect you