UiT The Arctic University of Norway

Project 3: Preemptive Multitasking

Managing processes

INF-2201 Staff

UiT

Spring 2024

Project 3

- Project Overview
- Interrupts
- Project Tasks
- Administrative Details

Project 3: Administrative Info (just like Project 2)

- Mandatory assignment
  - This is a mandatory assignment
  - It will be graded as pass/fail by TAs
  - You must pass to gain admission to the exam
- Groups of two
  - You must work in groups of two
- Design review
  - Meet with TA and present your design
  - Informal, but must have some kind of slides / presentation
- Hand-in
  - Hand-in via Canvas
  - Code: whole repository (working tree + `.git/` dir)
  - Report: place in a `report/` directory in your repo
  - Zip up and upload to Canvas

Project 3: Preemptive multitasking

- Project 2 was cooperative multitasking
  - Threads had to call `yield` or `exit` to give up control
- In Project 3: we will *preempt* the threads
  - Timer interrupt will fire every 10 ms
  - Give the kernel a chance to take over and switch threads
- You will:
  - Add timer interrupt handler code
  - Re-implement locks so that they work with preemption
  - Implement other sync abstractions: semaphores, condition variables, barriers
- Extra challenges:
  - Priority scheduling
  - Reimplement the dining philosophers on Linux with POSIX threads

Project 3: Environment / Precode

Same simple environment

- Protected mode, but no active protection
  - CPU in 32-bit Protected Mode
  - No protection active: all runs at kernel level (Ring 0)
  - Flat 32-bit address space
  - Kernel and processes share one address space
- No `malloc` / `free`
  - Statically allocate any globals / arrays you need
  - Allocate single structs from arrays

What's new?

- Precode includes our Project 2 solution, plus…
- Interrupts!
  - Interrupts are turned on
  - Syscall entry via interrupt
- Real time emulation
  - Bochs time syncs with real time
  - OS detects CPU speed with timer

Interrupts

Triggers

- Signal from hardware: timer, keyboard, etc.
- CPU exception: divide by zero, invalid memory access, etc.
- Software trigger: INT instruction

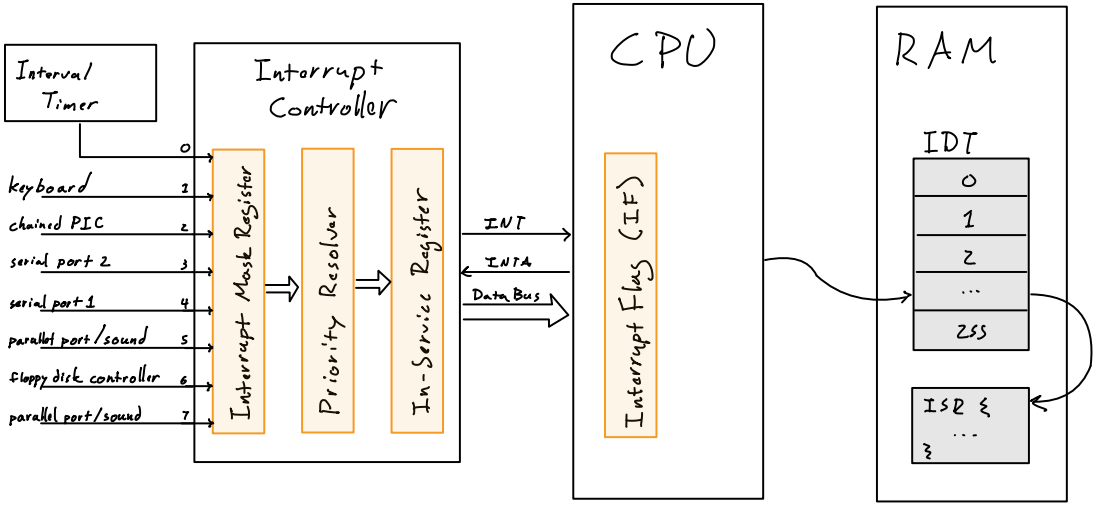
Operation

- CPU saves state to stack and jumps to pre-set address
- Like a function call that can happen at any time
- *Interrupt Handler* aka *Interrupt Service Routine (ISR)*

Saved state

- Not just EIP: EFLAGS, CS, EIP
- EFLAGS Interrupt Flag (IF) is cleared, to disable interrupts during handler
- Special return instruction: `iret`

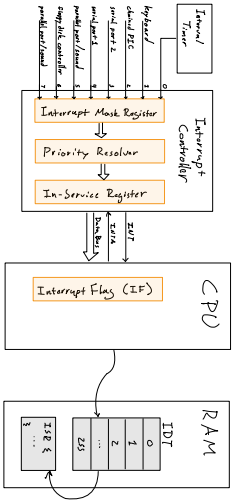
Interrupt Hardware Support



Interrupts, from hardware to CPU

Interrupt Hardware Support

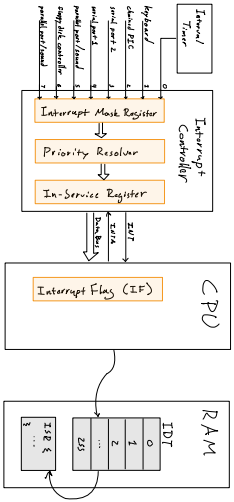
- Devices make *Interrupt Requests (IRQs)*
  - 0 = timer
  - 1 = keyboard
  - …
- to *Programmable Interrupt Controller (PIC)*
  - PIC multiplexes IRQs and sends to CPU
  - Maps IRQ (0–7) to *interrupt vector* (0–255)
  - Typically IRQ 0 maps to interrupt 32
- to CPU
  - Gets signal and vector number
  - Looks in *Interrupt Descriptor Table (IDT)*
  - Descriptor points to ISR
  - CPU Executes ISR
  - ISR `iret` returns to interrupted execution



Interrupts, from hardware to CPU

Three Chances to Block an Interrupt

1. PIC: Interrupt Mask Register
   - CPU can command PIC to ignore specific IRQs
2. PIC: In-Service Register
   - When PIC sends interrupt to CPU, it marks the IRQ as *in service*.
   - Will not send the same interrupt again, until the CPU sends an *End Of Interrupt (EOI)*.
3. CPU: Interrupt Flag (IF)
   - In EFLAGS register
   - Clear to ignore all external interrupts
   - CPU clears flag at interrupt start
   - Flag restored when EFLAGS is restored on `iret`



Interrupts, from hardware to CPU

Interrupt Sequence

1. IRQ comes in to PIC
   - 1.1 Check interrupt mask
   - 1.2 Check in-service
   - 1.3 Mark IRQ as in service (disable this IRQ)
   - 1.4 Send interrupt vector to CPU
2. CPU gets interrupt
   - 2.1 Check Interrupt Flag
   - 2.2 Look up interrupt in IDT
   - 2.3 Push EFLAGS, CS, IP
   - 2.4 Disable Interrupt Flag (disable interrupts)
   - 2.5 Jump to beginning of ISR
3. Interrupt Service Routine
   - 3.1 Save additional context
   - 3.2 Actually handle interrupt
   - 3.3 Restore additional context
   - 3.4 Send EOI (re-enable this IRQ)
   - 3.5 `iret`
4. CPU returns
   - 4.1 Pop and restore EIP, CS, EFLAGS (re-enable interrupts)
   - 4.2 Resume execution

Interrupt on a Timer for Preemptive Scheduling

- *Programmable Interval Timer (PIT)* chip
  - Base clock ticks at 1193180 Hz (1.19 MHz)
  - Output channel can fire at any fraction of that
  - Output channel 0 is connected to PIC IRQ 0
- Precode:
  - Sets up PIT: Fire IRQ 0 every 10 ms
  - Sets up PIC: Map IRQ 0 to interrupt 32, mask others
  - Sets up IDT: Handler for interrupt 32 $\rightarrow$ `timer_isr_entry`
- You: Implement `timer_isr_entry` function (in interrupt_asm.S)
  - Save/restore necessary context
  - Switch to kernel stack if using user stack
  - Call into scheduler to switch to next process
  - Switch back to user stack if necessary
  - Restore necessary context
  - Send EOI

Tasks Overview

1. Implement preemptive scheduling
2. Implement synchronization abstractions
   - 2.1 Re-implement locks to work with preemption
   - 2.2 Implement semaphores
   - 2.3 Implement condition variables
   - 2.4 Implement barriers
3. Ensure that preemption does not break existing `yield` and `exit` calls
4. Fix the dining philosophers implementation

Task: Implement preemptive scheduling

- Need to implement `timer_isr_entry` function (in interrupt_asm.S)
  - Save/restore necessary context
  - Switch to kernel stack if using user stack
  - Call into scheduler to switch to next process
  - Switch back to user stack if necessary
  - Restore necessary context
  - Send EOI
- May need to add/change some code in scheduler as well

Task: Re-Implement Locks

1. Re-implement locks to work with preemption
   - Remember: timer interrupt can happen between any two instructions
   - Need some kind of atomicity
- Provided helper functions for “no-interrupt” sections
  - Call `nointerrupt_enter` to disable interrupts
  - Call `nointerrupt_leave` to reenable interrupts
  - Can be nested, will only re-enable on last leave
- Precode includes skeletons for multiple implementations:
  - Project 2 version: `_coop` suffix (don’t bother)
  - Project 3 version: `_nointerrupt` suffix (impement these)

Task: Implement Other Sync Abstractions

2. Implement semaphores
   - Test code: dining philosophers (philosophers.c)
3. Implement condition variables (monitors)
   - Test code: lock test threads (th2.c)
     upgraded to also use condvars
4. Implement barriers
   - Test code: barrier test threads (barrier_test.c)

Barriers

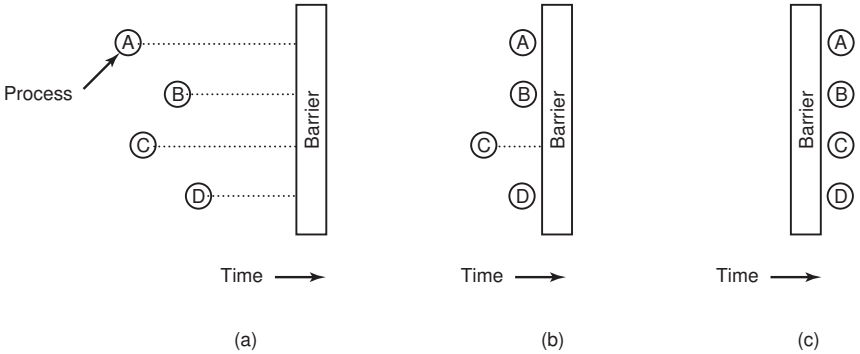


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Barriers (Tannenbaum, *Modern Operating Systems*, 4th ed)

Task: Ensure that preemption does not break `yield` and `exit`

- Precode includes syscall implementations
- New implementation uses interrupt (INT 48) instead of fixed address
- Precode includes scheduler implementation
- You may need to make some changes to ensure that `yield` and `exit` still work with preemption active
- Or they're fine and you just need to make sure you don't break them

Task: Improve the Dining Philosophers implementation

- Precode includes an implementation of the Dining Philosophers problem
- Display on screen and also keyboard LEDs: num lock, caps lock, scroll lock
- Our solution is not fair
  - The middle philosohper, Caps Lock, as a slight advantage
  - Over time, he gets to eat more than the others
  - Why is this?
- Analyze this in your report. Answer these questions:
  - Can this solution deadlock?
  - What makes this solution unfair?
  - How does this relate to the concept of *starvation?*
  - How would you go about making it fair?
- Update the existing solution: try to make it more fair

Extra Challenges

1. Implement priority scheduling
   - The plane process (process1.c) uses the new `setpriority` syscall
   - Implement priority scheduling: should see the plane's speed change
2. Re-implement dining philosophers on Linux using POSIX threads (pthreads)
   - There is no precode for this
   - Set up a simple C project and write your own dining philosphers simulation

Where to start

- Implement the preemptive scheduling first
- Then the lock implementation
- Then semaphores, then condition variables, then barriers
- Remember: all sync abstractions should work with preemption
- Finally, fix the philosophers

Procedure is the same as Project 2

- Design reviews
- Code
- Report
- Hand in via canvas: zip up entire repository plus report

Design Reviews

- Design reviews start next week
  - For some, this means Monday (!!)
- Remember: informal presentation
  - Don't need code
  - Need some kind of visual
  - Need to show that you understand the theory and that you've got a plan
- Keep it at the design level
  - You don't need to go deep into implementation details
  - But we want to see that you have a ideas for implementation
- Notes to individual groups:
  - Group 1 with Øyvind: Most DRs will be online (Discord)
  - Group 2 with Ilya: Friday will be a presentation about how to write a good report

Report

- Should be around 4 pages
- Give an overview of how you solved each task (or extra challenge)
- Describe how you tested your code
  - Point out any known bugs/issues
  - Describe how you would try to fix the bugs if you had more time
- Describe the methodology, results, and conclusions of your performance measurements
- We are working on a “How to Write a Report” guide and report template
  - Expect those late this week or early next week

Hand In via Canvas

- Put your report in your repository, under a `report/` dir
  - Report shoud be a PDF format
  - If you write in Word or other WYSIWYG word processor, export to PDF
- Zip up your entire repository (code tree + report + `.git/` dir)
- Submit via Canvas