Task Scheduling and Programming Tips for FIFOS
Agenda

- Today:
  - Organization of task scheduler
  - Non-preemptive task switching
  - Setup of GDT

- Next lab:
  - Preemptive task switching
  - Interrupt handling (PIC)
  - Setup of the system timer (PIT)
Overview

Kernel Initialization

- **GDT** w/ at least kernel code and data descriptor
- (*): **IDT**: to handle hardware exceptions and IRQs
- (*): **PIC**: to deliver timer interrupts to the scheduler
- (*): **PIT**: to set preemption points
- Initialize a pool of (up to constant N) tasks
- Start the scheduler to launch the first task

(*): Preemption support requirements

Scheduler Functionalities

- **Scheduler’s Public Interface**
  - thread_create(func, stack)
  - thread_yield()
- **Scheduler’s Private Interface**
  - current_thread()
  - find_next_thread()
  - switch_thread(from, to)
  - launch_thread(t)
  - exit_thread()
  - (*): preempt_thread()
TCBs: Task Control Block

- A thread is a function with a private stack
- What information do we keep in TCB
  - State: New, Ready, Active, Dead, etc.
    - Affects the behavior of the scheduler and dispatcher
    - E.g. Switching to a newly created task w/o an initial state to restore
  - Next Instruction to run: EIP
    - call addr;
    - pushl addr; ret;
  - Stack top: ESP
  - Machine State (minimally the following)
    - General registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP (pushl/popl, pushal/popal)
    - Flags: EFLAGS (pushf/ popf)
Organization

- Functionalities
  - Add/Remove tasks
  - Find the next task to run
  - Handle state transitions
  - Context switching

- Main components
  - Task pool
  - Run Queue
  - Dispatcher
Non-preemptive Context Switch

- A context switch happens when:
  - The current running task finishes execution
  - Explicitly yields execution

- What should happen?
  - The current task goes to the scheduler’s code
  - The scheduler finds the next task to run
  - Pushes the machine state on the stack
  - Updates the TCB of the current (ESP, EIP, State)
  - Switches to the stack of the next thread (mov next->esp, %esp)
  - Pops the machine state from the new stack
  - Returns to the new current task
Example (T1 -> T2 -> T1)

Different colors show whose stack is active

- T1
  - yield()
- Scheduler
  - Finds T2
  - Stores T1’s state
  - Switches to T2’s stack
  - Restores T2’s state
  - Returns
  - ret
- T2
  - yield()
  - Finds T1
  - Stores T2’s state
  - Switches to T1’s stack
  - Restores T1’s state
  - Returns
  - ret

*1
Example (T1 -> T2 -> T1)

Let's take a look at the stack of T1 at different times.
Example – Before T1 yields

Before T1 yields:

- T1
  - yield()
  - %esp
  - T1's user data
  - *1
  - ret

- Scheduler
  - Finds T2
  - Stores T1's state
  - Switches to T2's stack
  - Restores T2's state
  - Returns

- T2
  - ret
  - yield()
  - Finds T1
  - Stores T2's state
  - Switches to T1's stack
  - Restores T1's state
  - Returns
Example - After T1 yields

T1

yield()

Scheduler

Finds T2
Stores T2's state
Switches to T2's stack
Restores T2's state
Returns

ret

T2

yield()

Finds T1
Stores T1's state
Switches to T1's stack
Restores T1's state
Returns

T1's user data

Addr of *1

%esp

*1

ret

Addr of *1
Example – T1’s executing the sched. code

T1

yield()

%esp

T1's user data
Addr of *1
Sched. frame

*1

Scheduler

Finds T2
Stores T1’s state
Switches to T2’s stack
Restores T2’s state
Returns

ret

yield()

T2

Finds T1
Stores T2’s state
Switches to T1’s stack
Restores T1’s state
Returns

ret

addr of *1
Sched. frame

%esp
Example – Before switching to T2’s stack

- **T1**
  - yield()
  - T1’s user data
  - Addr of *1
  - Sched. frame
  - Machine Registers
  - %esp

- **Scheduler**
  - Finds T2
  - Stores T1’s state
  - Switches to T2’s stack
  - Restores T2’s state
  - Returns

- **T2**
  - ret
  - yield()
  - Finds T1
  -Stores T2’s state
  - Switches to T1’s stack
  - Restores T1’s state
  - Returns

*1 T1’s user data
Addr of *1
Sched. frame
Machine Registers
%esp
Example – Running in T’2 context

T2’s running and %eip is pointing to somewhere in T2’s stack until it yields/exits and we get to
Example – After switching to T1’s stack

T1

yield()

T1’s user data
Addr of *1
Sched. frame
Machine Registers

%esp

*1

ret

Scheduler

Finds T2
Stores T1’s state
Switches to T2’s stack
Restores T2’s state
Returns

ret

yield()

T2

Finds T1
Stores T2’s state
Switches to T1’s stack
Restores T1’s state
Returns

*1

T1’s user data
Addr of *1
Sched. frame
Machine Registers

%esp
Example – After restoring T1’s state

T1

yield()

%esp

%1

Scheduler

T1’s user data
Addr of %1
Sched. frame

Finds T2
Stores T1’s state
Switches to T2’s stack
Restores T2’s state
Returns

ret

yield()

Finds T1
Stores T2’s state
Switches to T1’s stack
Restores T1’s state
Returns

T2

ret

Addr of %1
Example – At the end of sched.'s code

T1

yield()

%esp

T1’s user data

Addr of *1

*1

ret

Scheduler

Finds T2
Stores T1’s state
Switches to T2’s stack
Restores T2’s state
Returns

ret

yield()

T2

Finds T1
Stores T2’s state
Switches to T1’s stack
Restores T1’s state
Returns

*1

ret

addr of *1

%esp
Example – After the scheduler returns

Scheduler

Finds T2
Stores T1’s state
Switches to T2’s stack
Restores T2’s state
Returns

ret

yield()

Finds T1
Stores T2’s state
Switches to T1’s stack
Restores T1’s state
Returns

yield()

T1

T1’s user data

%esp

*1

ret

*1

T2
Setting up a GDT for your OS!

- GRUB sets up a default GDT and hands over control to us after setting the CPU mode to Protected Mode.
- Can we rely on that default table?...No since we don’t know the base address of the table itself!
- Set up our own GDT since we need it to refer to memory segments
- GDT
  - Each GDT table entry is 8 byte. It decides the accessible memory range.
  - GDT is too complex! Just use the very basic feature of it!
  - Setting up the GDT first: at least three entries: one empty, one for code, one for data
- GDT Tutorial
  - Tell CPU where GDT is: length of GDT - 1 and the linear address of the GDT
    - The lgdt instruction and a GDT pointer structure
  - Reload all the segment registers to point to the GDT entry
  - Neither POP nor MOV can place a value in the code-segment register CS; only the far control-transfer instructions can change CS.
## Format of GDT entries

- An array of 64-bit entries – Look [here](#) for definitions
  - In Assembly: Check out `.byte`, `.short` and `.long` directives [here](#)
  - In C: Check out *packed data structures* and GNU inline assembly

- Format of each GDT entry:

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base[15:0]</td>
<td>Limit[15:0]</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>56</td>
<td>55</td>
<td>51</td>
</tr>
</tbody>
</table>
Format of GDT Entries

- **Base**: A 32-bit value indicating the linear address where the segment begins.
- **Limit**: A 20-bit value indicating size of the segment with a granularity specified by the flags field, bit 55 of the entry
- **Flags.Granularity (Bit 55)**:
  - 0: 1-byte granularity -> W/ a limit of 0xFFFFF can address up to 1MB after the base
  - 1: 4-KB granularity -> W/ a limit of 0xFFFFF can address up to 4GB
- **Flags.CodeSize (Bit 54)**:
  - 0: 16-bit code in Protected Mode (you won’t need it)
  - 1: 32-bit code in Protected Mode
- **Flags (Bits 52 to 53)**: Reserved, must be Zero
Example: Setting up your GDT in assembly

# Somewhere in your assembly code:
lgdt gdt_pointer

# Somewhere your assembly data:
gdt_base:
### Null descriptor
.long 0x0
.long 0x0
### Flat 4 GB code segment descriptor (ring 0)
... bit definitions for your kernel's code segment
### Flat 4 GB data segment descriptor
... bit definitions for your kernel's data segment
### End of my GDT
gdt_pointer:
.short gdt_pointer - gdt_base - 1
.long gdt_base