CMSC412

Project 0
The usual

- Info:

- Recitation:
  - Wed 9:00-9:50
  - Wed 10:00-10:50

- TAs:
  - Robert Grove, Nick Frangiadakis
Why are we here

• To get you started on the project and answer your questions
• Give you background material
• Show you how the concepts you learn apply to GeekOS.
Why are we here

• _not_ to tell you where and what to code!
• Pointers:
  • The lab is about GeekOS: so read the source
  • You’ll be implementing major functionality into the base kernel.
  • So start early…
  • Challenging but fun!
Start Early
Project 0

• Setup:
  • QEMU
  • GeekOS base setup

• Project requirements:
  • Resource restrictions on GeekOS processes:
    • # of active processes
    • # of syscalls by a single process
Project0 lessons...

- You have learned:
  - The QEMU simulation setup is:

- Alternatives:

<table>
<thead>
<tr>
<th>GeekOS</th>
<th>VMware Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native OS</td>
<td></td>
</tr>
<tr>
<td>HW</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java VM</td>
</tr>
<tr>
<td>Native OS</td>
</tr>
<tr>
<td>HW</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GeekOS</th>
<th>HW</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>QEMU (HW Emulation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native OS</td>
</tr>
<tr>
<td>HW</td>
</tr>
</tbody>
</table>
Project0 Lessons...

• You have learned:
  • About GeekOS:
    • Reading the source code is good and works!
    • OS split in two: *User-level* and a *Kernel-level*
    • Connected by the *System call* boundary
    • GeekOS user processes are just kernel threads with a special *User_Context* structure
    • grep is your friend
In more detail: System Calls

- Software interrupt
  - The only interrupt callable from user level idt.c #Init_IDT
  - SYSCALL_INT: 0x90
- Operation: syscall.h; syscall.c; libc/process.c
  - Put args in registers on user side; raise INT
  - Recover them on kernel side
  - Call the appropriate Sys_XXX
  - Return result/error code in appropriate register
- Use g_CurrentThread for information about who raised it
In more detail: **Thread System**

- In the kernel
  - Each thread is a Kernel_Thread object: \texttt{kthread.h}
- Current thread: \texttt{g\_CurrentThread} global
- User mode threads
  - Kernel_Thread objects with a populated User_Context
- User mode -> kernel mode execution: \texttt{syscall}
- Kernel vs user memory
  - Distinct views: one from the user and one from the kernel
  - Kernel needs to access user memory
  - Use \texttt{Copy\_From\_User/Copy\_To\_User}
In more detail: The system queues

- **Thread_Queue structure**
- **Run queue:**
  - Threads which are ready to run, but not currently running
  - GeekOS has a single run queue, as of the moment
- **Wait queues:**
  - Threads that are waiting for a specific event or on a specific device; eg Keyboard IO, network IO, other threads: `geekos/kthread.c#Join()`
  - Spend 2 mins: follow the Get_Key syscall to see how the thread eventually gets to the keyboard wait queue
In more detail: **Interrupts**

- **Types:**
  - Illegal operations: *result in kills*
  - Faults: page faults etc: *not of concern right now*
  - h/w interrupts
  - s/w interrupts: *syscall int*

- **Interrupt handlers**
  - *src/geekos/int.c*
  - On completion -> control returns to thread that was interrupted
Interrupts

- When you don’t want to receive them:
  - When you are modifying global data structures; queues etc
  - When you want to make some operation atomic
- `Disable_Interrupts()` / `Enable_Interrupts()`:
  - Can use `Disable_Interrupts()`: `include/geekos/int.h`
  - Extreme caution
  - `Enable_Interrupts()` when atomic operation finished
  - See places where this has been done: eg `src/geekos/user.c#Attach_User_Context()` and `src/geekos/kthread.c#Reaper()`
- `Begin_Int_Atomic()` / `End_Int_Atomic()`
  - Oblivious way of saving and restoring interrupt state.
  - `include/geekos/int.h`