ECE 391 Discussion Week 8

Announcements

OFFICE HOURS ARE CHANGING!

Kinda

Announcements

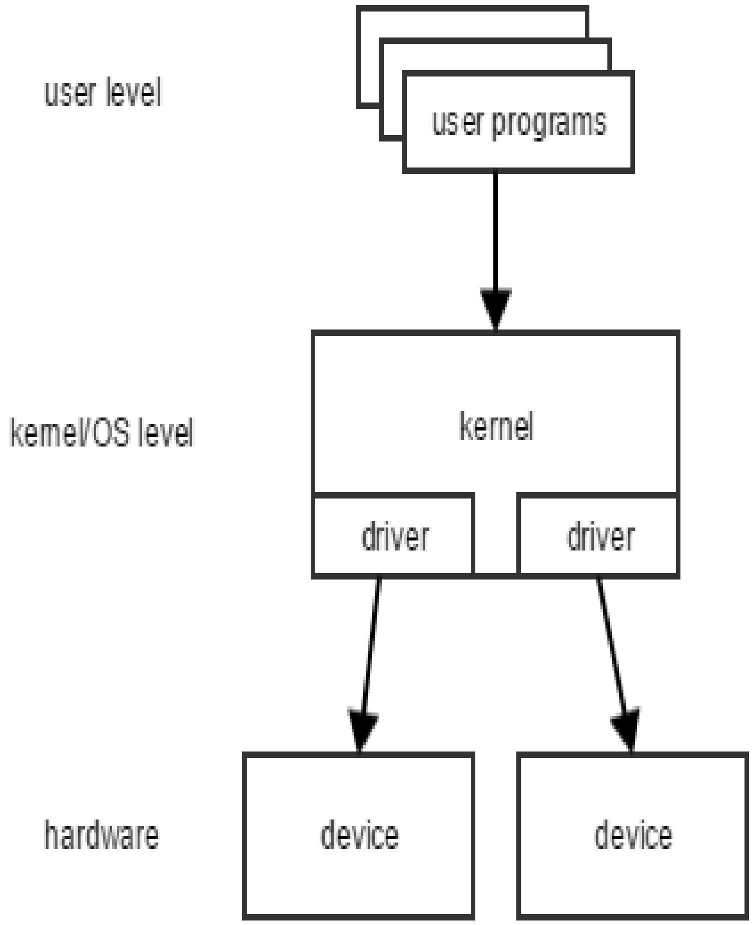
- Students will be dequeued as a group
- TAs will spend a maximum of 30 mins per student at a time
- Students must try to debug their code by themselves before they come to OH
  - TAs will ask you what you have done so far and can ask you to go back and try by yourselves before they help you further
  - TAs will not debug with/for you but instead give you instructions on how to proceed
- Please be mindful of the TAs time and come prepared to Office Hours
- **RTDC x 3**
- Look on Piazza if someone has had similar issues
  - Repeated Piazza questions will be ignored

Announcements & Reminders

- MP3.1 due next Thuesday (October 19st) at 5:59pm in GitLab
  - At least 1 member needs to be at the demo (for checkpoints 1 ~ 5)
  - Everyone must be present for the Final Demos
- Start early!
- Develop a system of working as a team (e.g. work together at the same time, use separate branches for each feature, etc.)
- RTDC

MP3 Overview

- 10/10 to 10/17 – CP1 (1 week) – initializing the kernel
- 10/17 to 10/24 – CP2 (1 week) – device drivers (keyboard/terminal, RTC, filesystem)
  - Normally drivers are separate from kernel, but for simplicity the drivers in this MP will be built into the kernel
- 11/01 – Exam 2!
- 10/24 to 11/7 – CP3 (2 weeks) – starting user program execution (system calls)
- 11/7 to 11/14 – CP4 (1 weeks) – getting all user programs and system calls working
- Thanksgiving Break
- 11/14 to 12/4 – CP5 (3 weeks) – multiple terminals and a scheduler
- Groups of 3 will not be required to do scheduling



MP3 Overview (contd.)

- MP3 is a group project, include everyone when working on it
- There will be a peer evaluation at the end of the semester
  - Everyone must estimate what percentage of work the others did
  - Your final grade on MP3 will be affected by the peer evaluations
- If you fall behind on one checkpoint, you still have to complete it before the next one because the next checkpoint depends on the functionalities you implement in the current one
- If you have problems with your group, post private posts on Piazza or email the professor **ASAP**
- Half functionality points back for checkpoints **at final demo**

MP3 Checkpoint 1

- Populate GDT/IDT
  - Look at the file x86-desc.S
  - Assembly file which contains GDT/IDT information and initial paging structures
  - Must complete this step or your system won’t boot at all

MP3 Checkpoint 1

After Loading GDT:

- **Q. Why doesn't our OS boot?**
  - The base code initially given won't boot up the system. You have to load the **Global Descriptor Table** (GDT), which is already filled for you! You should take a look at *x86_desc.S* (line 38) and *x86_desc.h* as well as *boot.S* (line 27). It is very similar to how the IDT is done in that same file (*x86_desc.S*). Please don't just blindly copy paste. Understand what's going on in this file.

- **Q. Our OS still isn't booting! (And we loaded the GDT@!@#!@#)**
  - There is a **RUN_TESTS** defined at the top of *kernel.c*. Follow what this macro does through the code. It should lead you to the tests.c file. There is already one test written in here for you. Read what it does in its function header.
  - So it should start to become obvious to you what's happening here. If you have not done anything with setting up the IDT, this test will fail and your system will triple fault on boot. Now you see what to expect when you do something really wrong in your code.
  - You can figure out from here what to do about that...

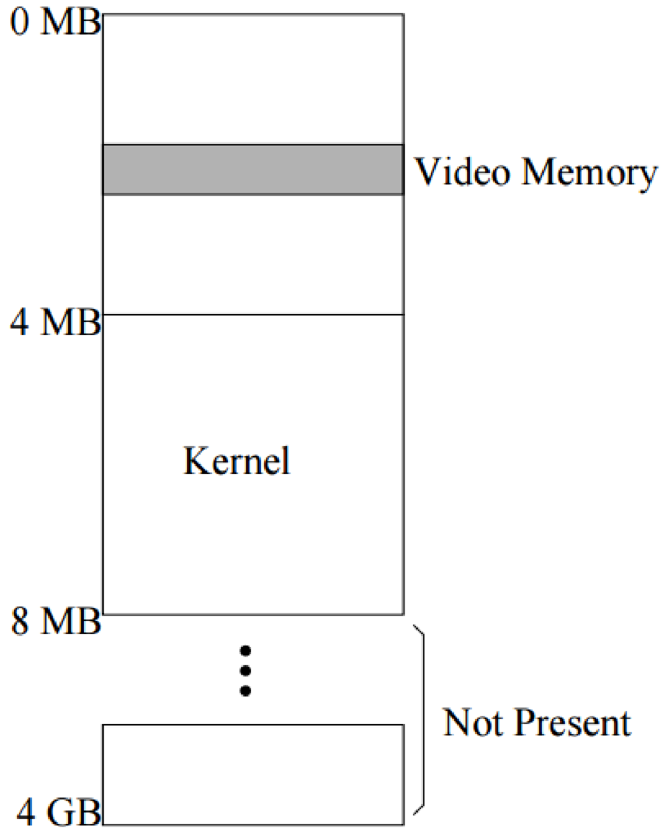
MP3 Checkpoint 1

After step 1……

- Initializing and mapping for exception handling, exception handlers
  - **Assembly linkage** for both exceptions and interrupts is suggested (this will make your life a lot easier in CP3)
- Device initialization (PIC, keyboard, RTC)
  - Remember to **send EOI** for interrupts
  - Know the difference between **trap gate** and **interrupt gate** (which one to use in different situations)
- Paging
  - Last thing to do in CP1
  - Hardest thing to do in CP1

Paging Layout

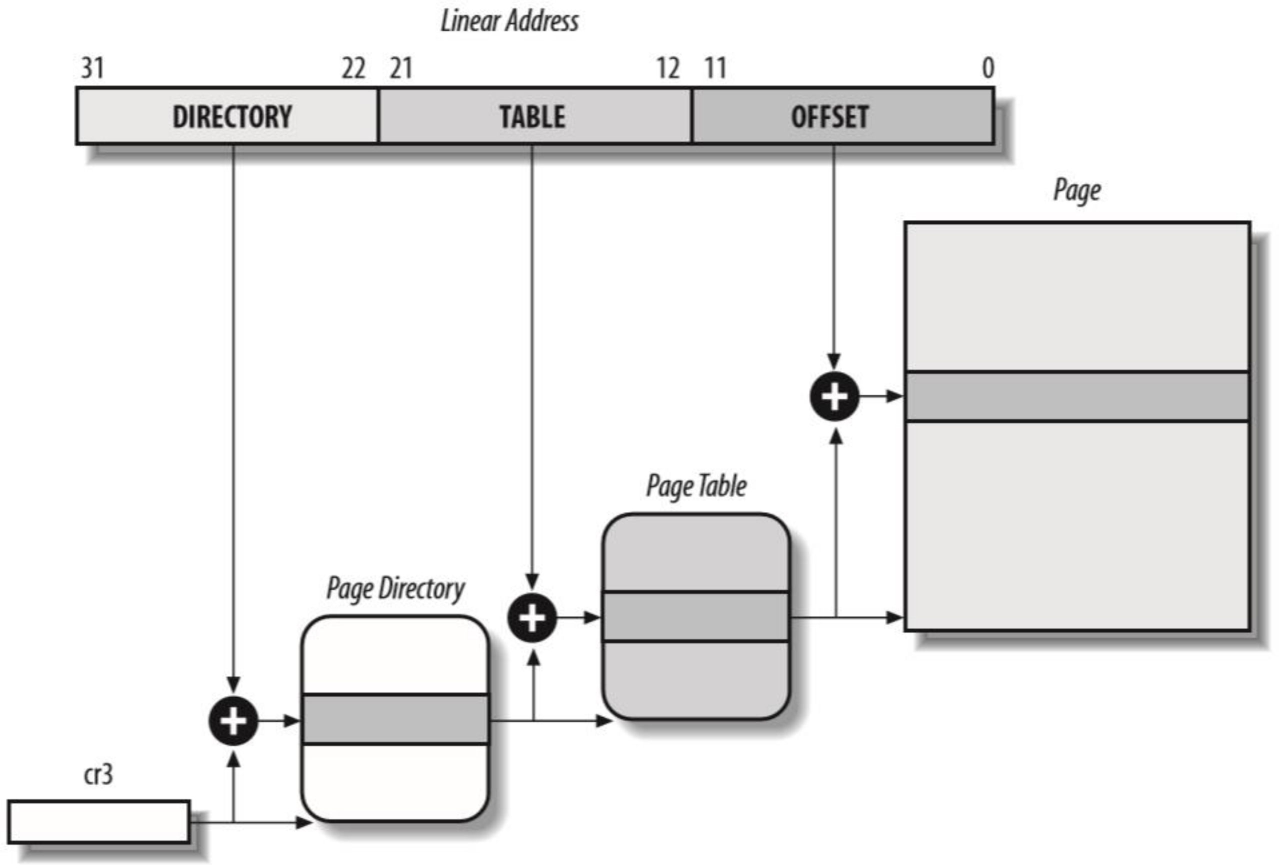
- 0 – 4 MB (virtual and physical)
  - 4KB page (why?)
  - Video memory
- 4 – 8 MB (virtual and physical)
  - 4MB page (why?)
  - Kernel

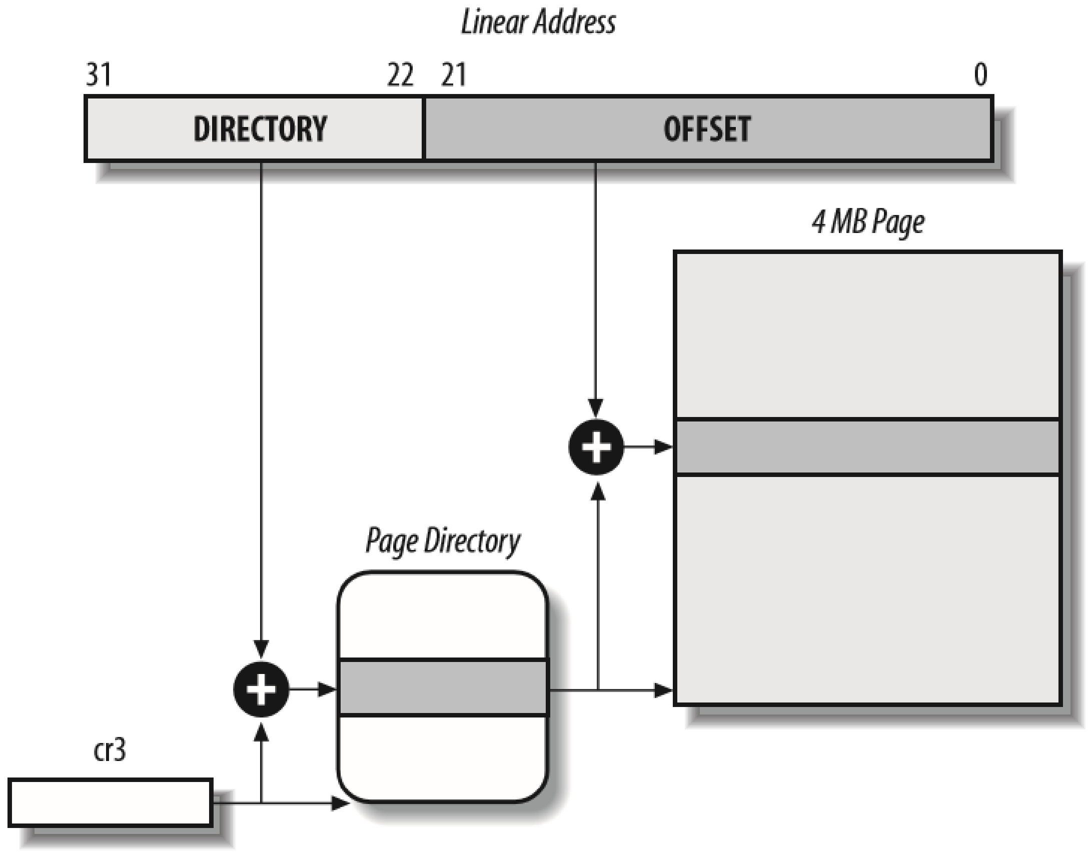


Paging

- Page Directories (PD) and Page Tables (PT) takes up space
  - Do not assign some random address to pointers
- PDs and PTs can be allocated either in assembly or C
- PDs and PTs must be aligned
  - Alignment spec is in MP3 doc
  - For CP1 no address after 8MB should be mapped
- Example bugs
  - System crashes after turning on paging
  - Printing debugging information crashes system (with paging enabled)

Paging (contd.)





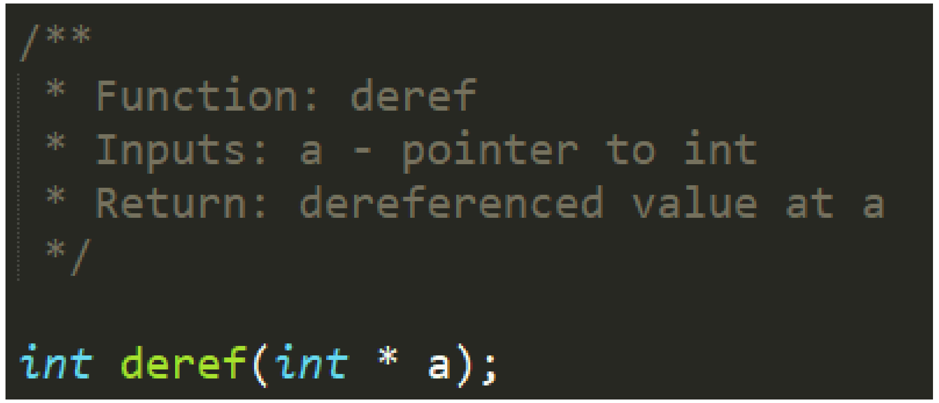
Testing

- Write your own tests to show functionality for all checkpoints
- For Checkpoint 1:
  - This table is not comprehensive
  - Also look at the hints document on the website

Functionality	Test should show
GDT Loaded Correctly	System does not boot loop
RTC	Able to receive RTC interrupts
Paging	Cannot dereference regions that should not be allocated

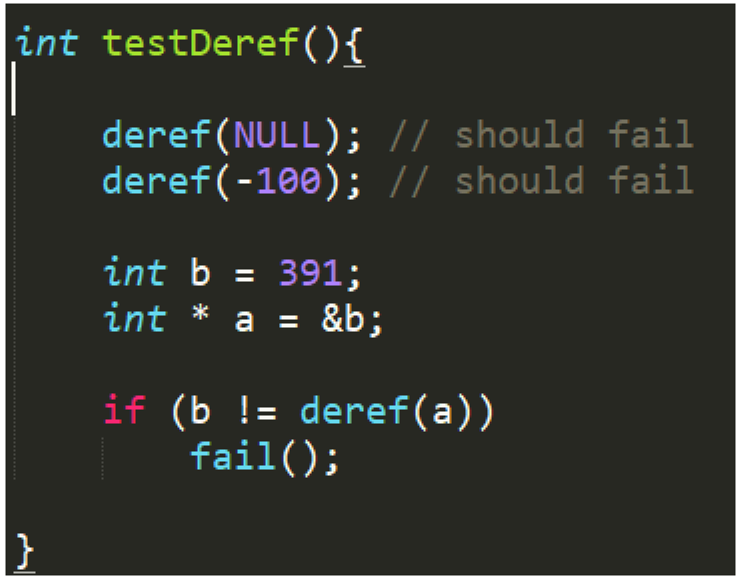
Dealing with pointers

- What about pointers?
  - Always check NULL
  - Check known values (a & b in example)
  - Check values on stack
  - Check values on heap (malloc'd)
  - In MP3
    - Check userspace pointers
    - Check kernel pointers
    - Check non-paged areas (unallocated pointers)



```c
/**
 * Function: deref
 * Inputs: a - pointer to int
 * Return: dereferenced value at a
 */

int deref(int * a);
```



```c
int testDeref(){

    deref(NULL); // should fail
    deref(-100); // should fail

    int b = 391;
    int * a = &b;

    if (b != deref(a))
        fail();

}
```

….but why?

- Write more code to test existing code? Sounds like a bad idea…
  - Test code is usually separated from production code.
  - If your tests are broken, production code isn’t affected
- But what about GDB? I’ve tested my code there and it works!!!
  - Sure, but what if you come back and add something?
  - Easier to re-run tests than debug with GDB all over again
  - If you test functions against their specification, adding more code shouldn’t require new test cases
- How do I know I’ve covered all cases?
  - Usually not possible to cover every possible execution
  - But unit testing is not meant to – only to eliminate basic design and code bugs
  - Other methods exist to validate code to a higher level of assurance, but require more time and effort
- How is this useful to writing an OS?
  - Working in teams
  - Sanity check your teammates work
  - Get to blame others when things fail