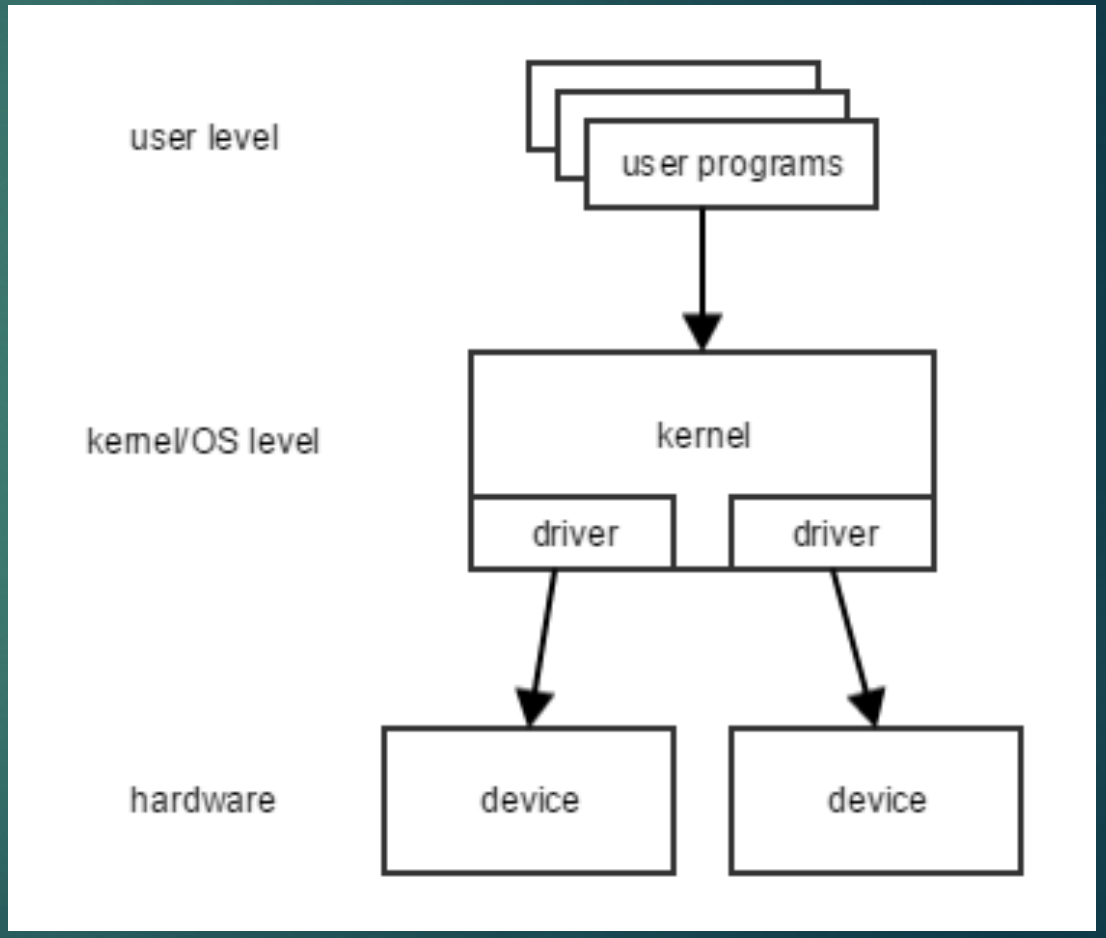
ECE 391 Discussion Week 8

Announcements & Reminders

- Return TUX controllers during MP3.1 handin or before the end of the semester
  - $100 charge if you lose it!
- MP3.1 due next Monday (October 22rd) at 6pm in GitLab
  - At least 1 member needs to be at the demo (for checkpoints 1 ~ 5)
  - Everyone must be present for the Final Demos (Week of 12/10)
- 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

- CP1 (1 week) – initializing the kernel
- 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
- CP3 (2 weeks) – starting user program execution (system calls)
- CP4 (2 weeks) – getting all user programs and system calls working
- CP5 (2 weeks) – multiple terminals and a scheduler



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 fell 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 professors

MP3 Checkpoint 1

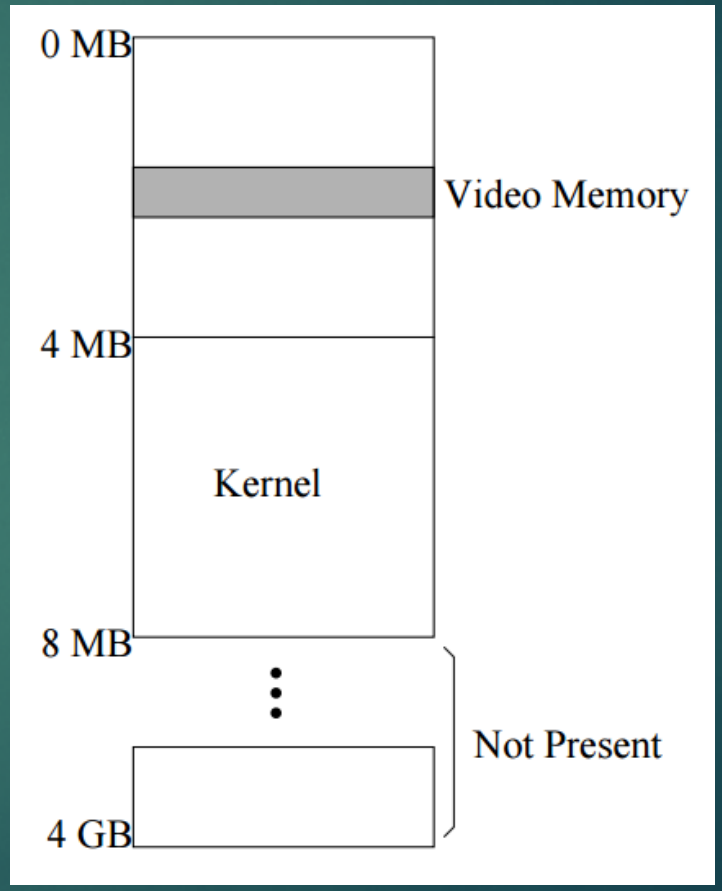
- Populate GDT/IDT
  - Look at the file x86-desc.S
  - Assembly file which contains GDT/IDT information and initial paging structures
  - Must complete this step or your system won't boot at all
- Mapping for exception handling, exception handlers
- Device initialization (PIC, keyboard, RTC)
- Paging
  - Last thing to do in CP1
  - Hardest thing to do in CP1

MP3 Checkpoint 1

- Assembly linkage for both exceptions and interrupts is suggested (this will make your life a lot easier in CP3)
- Remember to send EOI for interrupts
- Know the difference between trap gate and interrupt gate (which one to use in different situations)
- The Interrupt Descriptor Table (IDT)

Paging Layout

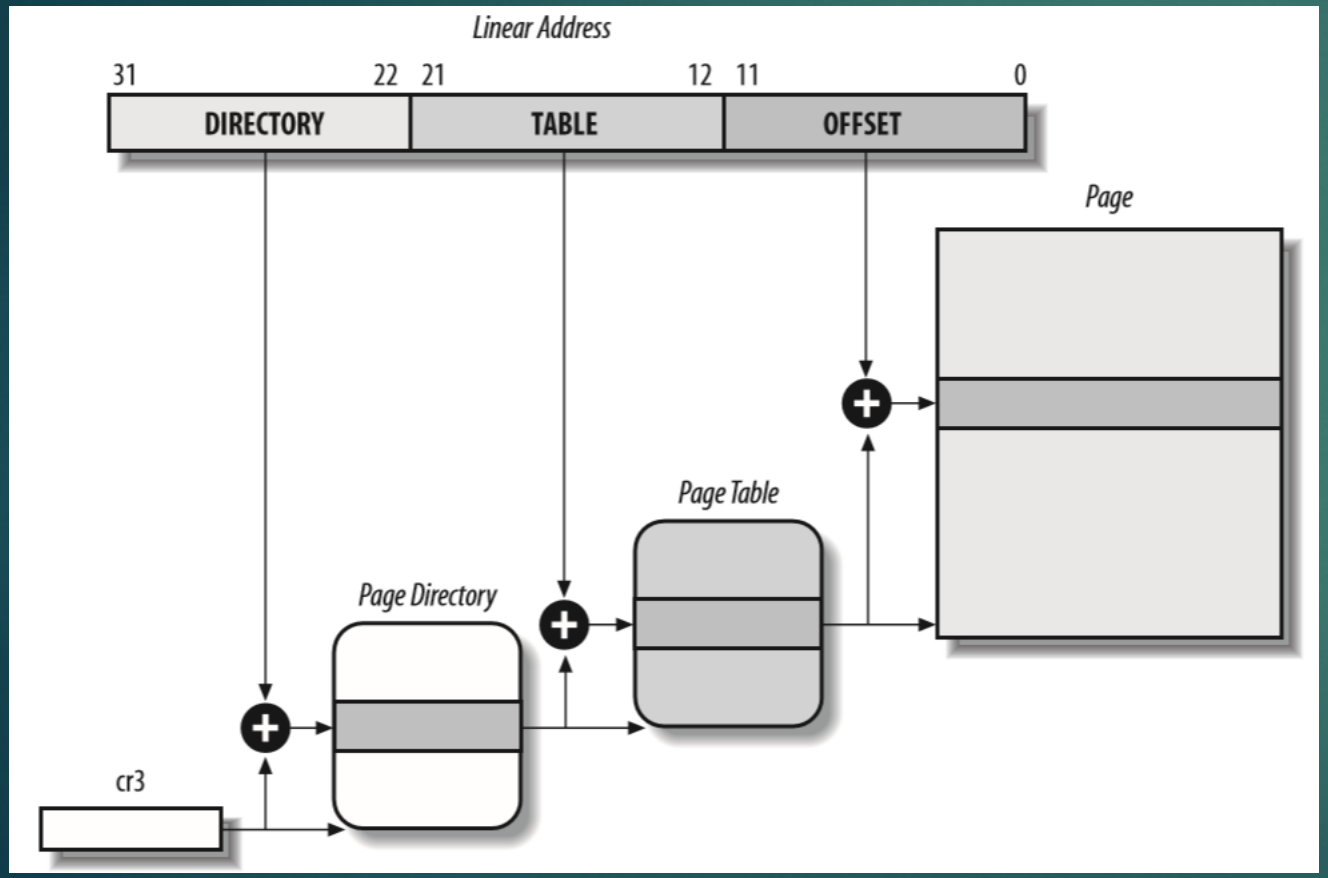
- 0 – 4 MB (virtual and physical)
  - 4KB page (why?)
  - Video memory
- 4 – 8 MB (virtual and physical)
  - 4MB page (why?)
  - Kernel

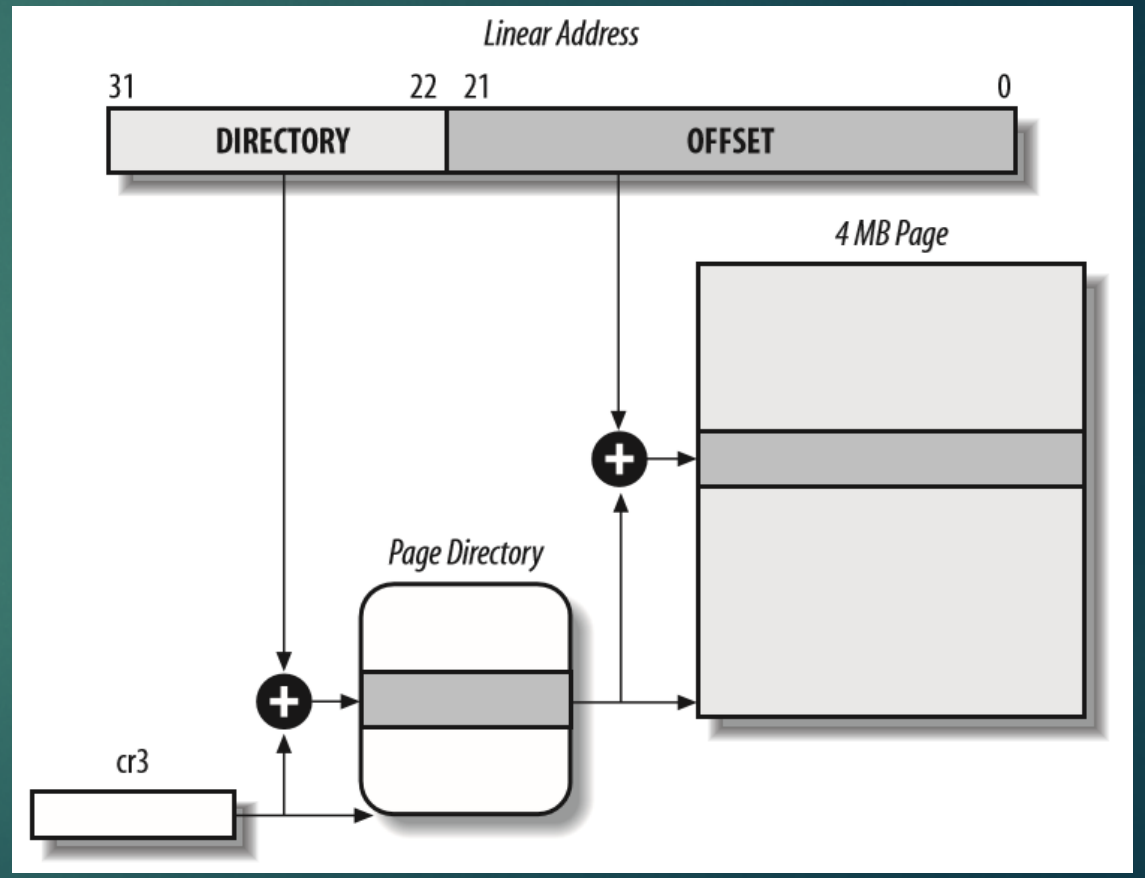


Paging

- 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.)





Intro to Unit Testing

Unit Testing

- Goal: Validate individual correctness of functional code units

- Test individual units of code – functions, subroutines or procedure
- Find problems early in development cycle
  - Could be implementation flaw within function
  - Larger design issue with module
- Test code at interface level
  - validate what function claims to do vs. internal implementation details
- Parametrize tests with different cases

Simple Example

- Given just the interface, how would you test this function?
  - What arguments do you test with?
  - a = +ve, b = +ve
  - a = -ve, b = -ve
  - a = +ve, b = -ve
  - Zeros? Large values? Overflows?
- What should your test function look like?

```
/**
 * Function: sum
 * Inputs: a - int, b - int
 * Return: sum of a and b
 */

int sum(int a, int b);
```

```
int testSum(){

    if (!(sum(4,5) == 9))
        testFail();

    if (!(sum(-4,5) == 1))
        testFail();

    if (!(sum(-10,-10) == -20))
        testFail();

    // this should fail
    if (!(sum(4294967295, 1) == 4294967296))
        testFail();

}
```

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)

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

int deref(int * a);

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

Testing Frameworks

- Various testing frameworks
  - C – cUnit, Check, Autounit
  - Python – pytest
  - Java – JUnit, NGUnit
- Automatically run tests against your code
- Provide coverage information – can tell you what parts of functions are not tested
- Usually integrated into IDEs & CI/CD platforms (look these up if you're curious)

Uh…so what framework do we use?

- Unfortunately, none!
- In MP3, your code runs directly on the VM's emulated hardware
  - No standard libraries available
  - Cannot run additional programs (like frameworks) to unit test your OS

- Solution: Run tests directly within your kernel
  - Makeshift solution
  - Write test functions for each module (in a separate file)
  - Launch your tests once your OS is stable
  - Include/Exclude tests from compilation with a Macro (see example in kernel.c)

What tests do I write?

- May be hard to test some functionality
  - Exceptions may cause blue screens, never return to your test functions
  - System calls may change state of system
- But…test smaller units of code
  - Some portions of your OS use hardcoded values – addresses, offsets, etc
  - Check for hardcoded values
  - Check for non-NULL values in crucial structures (see given example)
  - Minimum expected coverage in MP3 spec, may be expanded on Chara
- For CP1: RTC, Paging, Exception handler