

Pintos Project #3

Virtual Memory

CS3204: Operating System

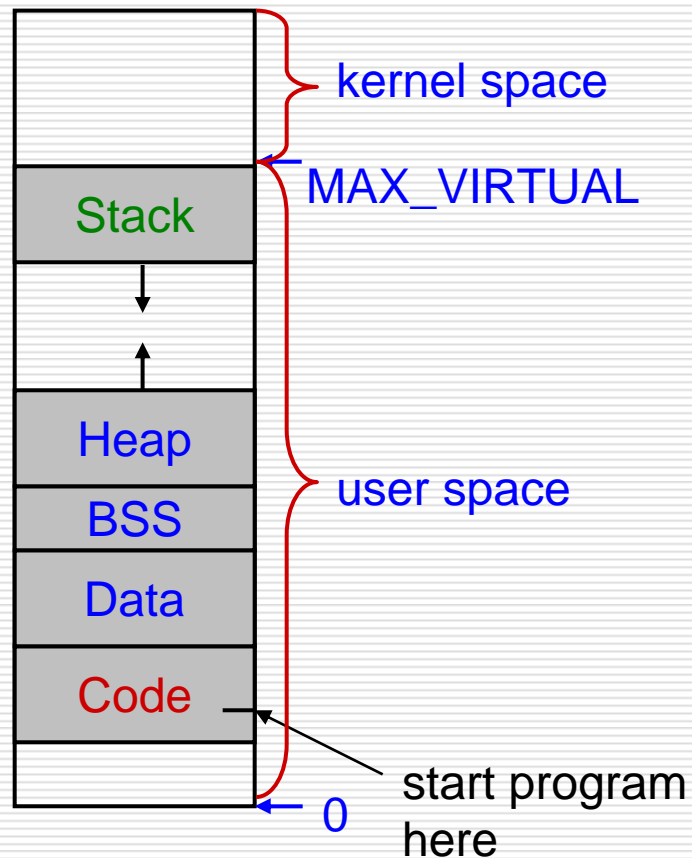
Xiaomo Liu (slightly edited by W McQuain)

Spring 2008

Outline

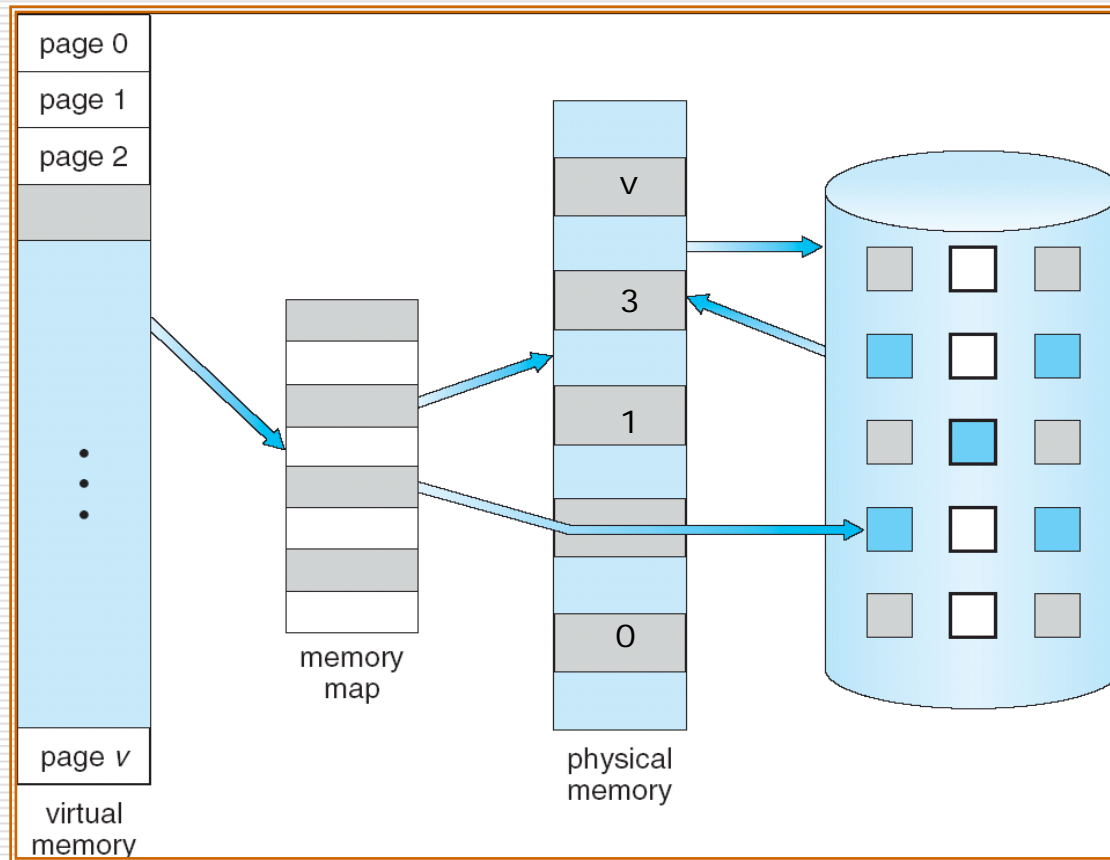
- Virtual memory concept
- Current pintos memory management
- Task
 - Lazy load
 - Stack growth
 - File memory mapping
 - Swapping
- Suggestion
 - How to start
 - Implementation order

Virtual Memory Concept



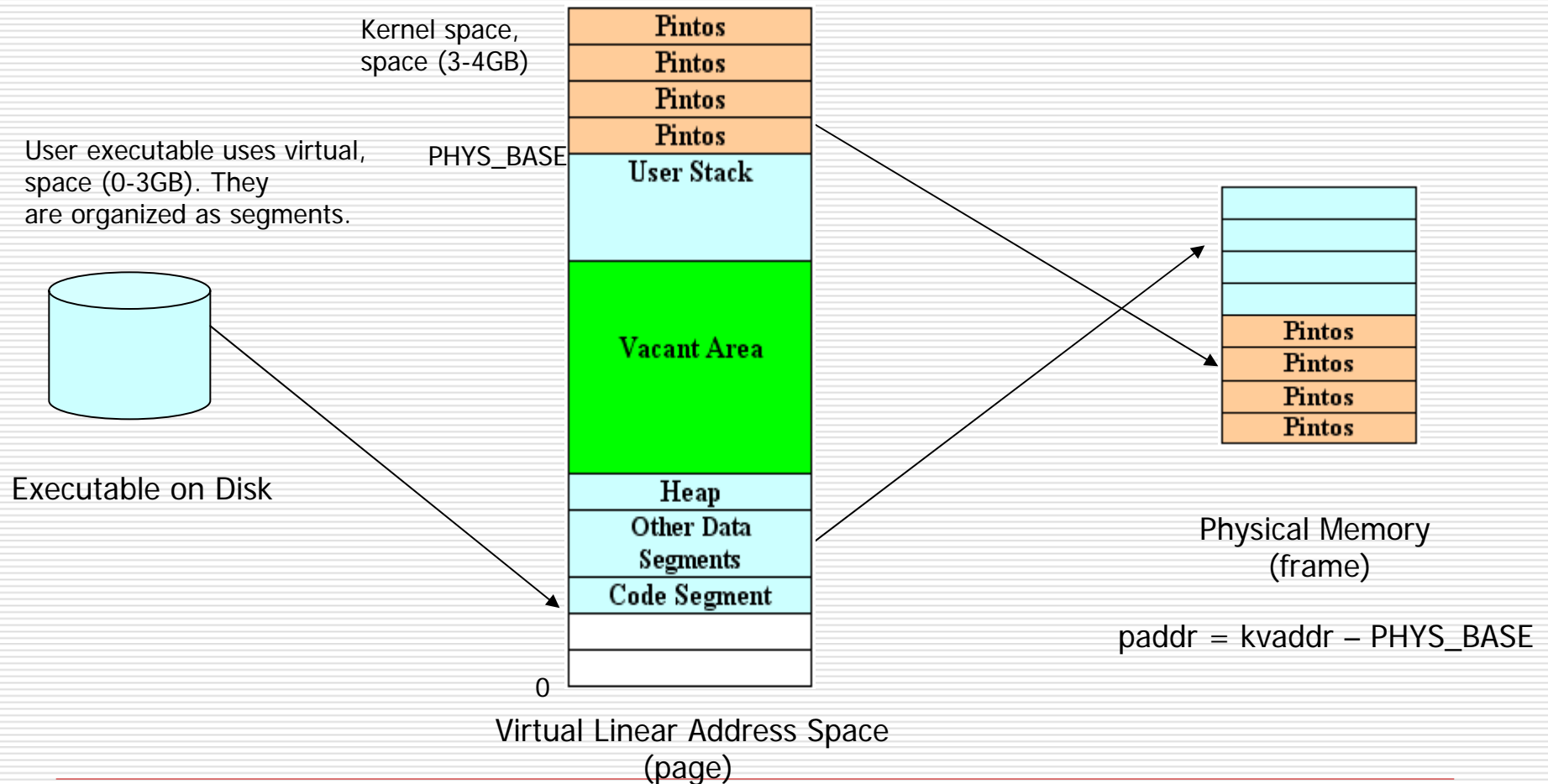
- VM is the logical memory layout for every process
 - It is divided into kernel space and user space
 - Kernel space is global (shared)
 - User space is local (individual)
- Different from physical memory
- Map to the physical memory
- How to do it? Paging!
 - Divide the VM of a process into small pieces (pages)– 4KB
 - “Randomly” permute their orders in PM

Virtual Memory Mapping

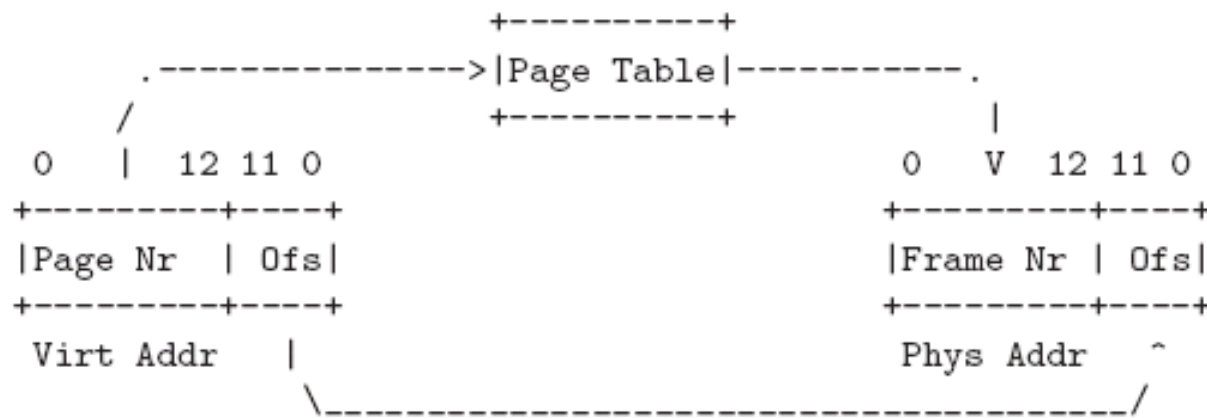


- Page
 - 4KB in VM
- Frame
 - 4KB in PM
- One to one mapping

Pintos Virtual Memory Management



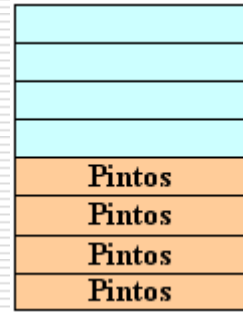
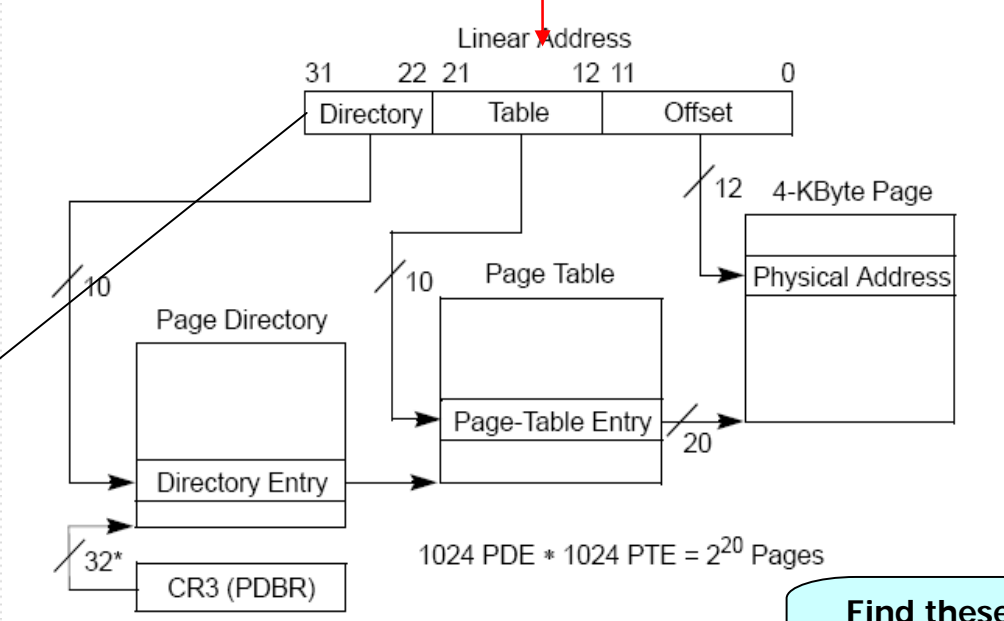
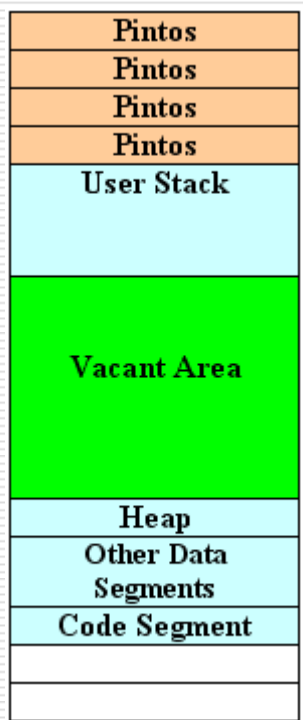
Pintos Virtual Memory Mapping



- ❑ Virtual address (31–12: page number, 11–0: offset)
- ❑ Physical address (31-12: frame number, 11-0: offset)
- ❑ Two-level mapping
 - Page number finds to the corresponding frame
 - Page offset finds to the corresponding byte in the frame

Pintos Virtual Memory Mapping...

Virtual Memory Mapping



RAM Frames

Find these vaddr.h and pagedir.h/c for its interface.

Three-level mapping

Current Status (Before project 3)

- Support multiprogramming
- Load the entire data, code and stack segments into memory before executing a program (see `load()` in `process.c`)
- Fixed size of stack (1 page) to each process
- A restricted design!

Project 3 Requirement

- Lazy load
 - Do not load any page initially
 - Load one page from executable when necessary
- Stack growth
 - Allocate additional page for stack when necessary
- File memory mapping
 - Keep one copy of opened file in memory
 - Keep track of which memory maps to which file
- Swapping
 - If run out of frames, select one using frame table
 - Swap it out to the swap disk
 - Return it as a free frame

Step 1: Frame “Table”

- Functionalities
 - Keep track all the frames of physical memory used by the **user** processes
 - Record the statuses of each frame, such as
 - Thread it belongs to (if any!)
 - Page table entry it corresponds to (if any!)
 - ... (can be more)
- Implementations (two possible approaches)
 - 1. Modify current frame allocator “palloc_get_page(PAL_USER)”
 - 2. Implement your own frame allocator on top of “palloc_get_page(PAL_USER)” without modifying it. (Recommended)
 - Have a look at “init.c” and “palloc.c” to understand how they work
 - Not necessary to use hash table (need figure out by yourself)
- Usage
 - Frame table is necessary for physical memory allocation and is used to select victim when swapping.

Step 2: Lazy Loading

- How does pintos load executables?
 - Allocate a frame and load a page of executable from file disk into memory
- Before project 3
 - Pintos will initially load all pages of executable into physical memory
- After project 3
 - Load nothing except setup the stack at the beginning
 - When executing the process, a page fault occurs and the page fault handler checks where the expected page is: in executable file (i.e. hasn't loaded yet)? in swap disk (i.e. swapped out already)?
 - If in executable, you need to load the **corresponding** page from executable
 - If in swap disk, you need to load the **corresponding** page from swap disk
 - Page fault handler needs to resume the execution of the process after loading the page

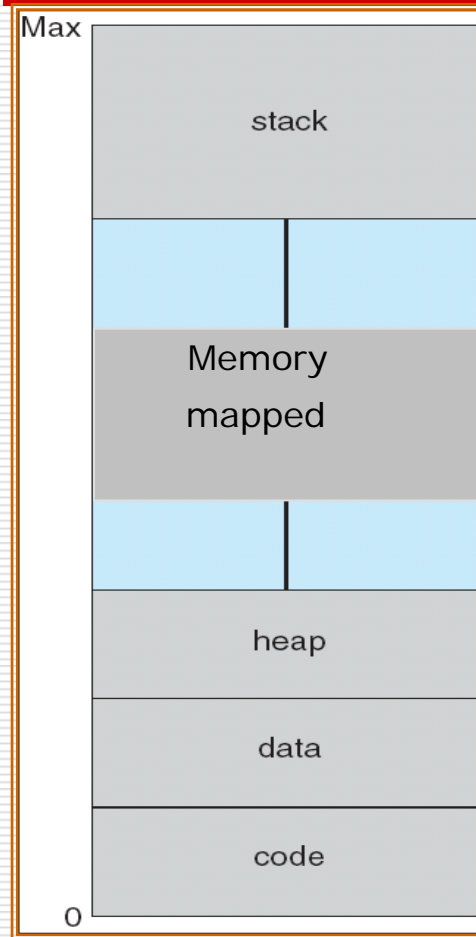
Step 3: Supplemental Page Table

- Functionalities
 - Your “s-page table” must be able to decide where to load executable and which **corresponding** page of executable to load
 - Your “s-page table ” must be able to decide how to get swap disk and which part (in sector) of swap disk stores the **corresponding** page
- Implementation
 - Use hash table (recommended)
- Usage
 - Rewrite load_segment() (in process.c) to populate s-page table without loading pages into memory
 - Page fault handler then loads pages after consulting s-page table

Step 4: Stack Growth

- Functionalities
 - Before project 3: user stack is fixed with size of 1 page, i.e. 4KB
 - After project 3: user stack is allowed to allocate additional pages as necessary
- Implementation
 - If the user program exceeds the stack size, a page fault will occur
 - Catch the stack pointer, `esp`, from the interrupt frame
 - In page fault handler, you need to determine whether the faulted address is “right below” the current end of the stack
 - Whether page fault is for lazy load or stack growth
 - Don't consider fault addresses less than `esp - 32`
 - Calculate how many additional pages need to be allocated for stack; or just allocated faulting page.
 - You must impose an absolute limit on stack size, `STACK_SIZE`
 - Consider potential for stack/heap collisions

Step 5: File Memory Mapping



Functionalities

- Make open files accessible via direct memory access – “map” them
 - Storing data will write to file
 - Read data must come from file
- If file size is not multiple of PGSIZE—sticks-out, may cause partial page – handle this correctly
- Reject mmap when: zero address or length, overlap, or console file (tell by fd)

Step 5: File Memory Mapping...

- Implementations
 - Use “fd” to keep track of the open files of a process
 - Design two new system calls: `mapid_t mmap(fd, addr)` and `void munmap(mapid_t)`
 - `Mmap()` system call also populates the s-page table
 - Design a data structure to keep track of these mappings (need figure out by yourself)
 - We don't require that two processes that map the same file see the same data
 - We do require that `mmap()`'ed pages are
 - Loaded lazily
 - Written back only if dirty
 - Subject to eviction if physical memory gets scarce

Step 6: Swap “table”

- Functionalities
 - When out of free frames, evict a page from its frame and put a copy of into swap disk, if necessary, to get a free frame — “swap out”
 - When page fault handler finds a page is not memory but in swap disk, allocate a new frame and move it to memory — “swap in”
- Implementation
 - Need a method to keep track of whether a page has been swapped and in which part of swap disk a page has been stored if so
 - Not necessary to use hash table (need figure out by yourself)
 - Key insights: (1) only owning process will ever page-in a page from swap; (2) owning process must free used swap slots on exit

Step 7: Frame Eviction

- Implementations
 - The main purpose of maintaining frame table is to efficiently find a victim frame for swapping
 - Choose a suitable page replacement algorithm, i.e. eviction algorithm, such as second chance algorithm, additional reference bit algorithm etc. (See 9.4 of textbook)
 - Select a frame to swap out from frame table
 - **Unfortunately, frame table entry doesn't store access bits**
 - Refer frame table entry back to the page table entry (PTE)
 - Use accessed/dirty bit in PTE (must use `pagedir_*` function here to get hardware bit.)
 - Send the frame to swap disk
 - Prevent changes to the frame during swapping first
 - Update page tables (both s-page table and hardware page table via `pagedir_*` functions) as needed

Step 8: On Process Termination

Resource Management

- Destroy your supplemental page table
- Free your frames, freeing the corresponding entries in the frame table
- Free your swap slots (if any) and delete the corresponding entries in the swap table
- Close all files: if a file is mmaped + dirty, write the dirty mmaped pages from memory back to the file disk

Important Issues

□ Synchronization

- Allow parallelism of multiple processes
- Page fault handling from multiple processes must be possible in parallel
- For example, if process A's page fault needs I/O (swapping or lazy load); and if process B's page fault does not need I/O (stack growth or all '0' page), then B should go ahead without having to wait for A.

Implementation Order Suggestions

- Pre-study
 - Understand memory & virtual memory (Lecture slides and Ch 8 & 9 of the textbook)
 - Understand project specification (including Appendix A.6, A.7 and A.8)
 - Understand the important pieces of source code (process.c: `load_segment()`, exception.c: `page_fault()`)
- Try to pass all the test cases of project 2
 - At least, argument passing and system call framework should work
- Frame table management

Implementation Order Suggestions...

- ❑ Supplemental page table management
- ❑ Run regression test cases from project 2
 - They are already integrated in the P3 test cases
 - Your kernel with lazy load should pass all the regression test cases at this point
- ❑ Implement stack growth and file memory mapping in parallel
- ❑ Swapping
 - Implement the page replacement algorithm
 - Implement "swap out" & "swap in" functionality

Other Suggestions

- ❑ Working the VM directory
 - Create your page.h, frame.h, swap.h as well as page.c, frame.c, swap.c in VM directory
 - Add your additional files to the makefile: Makefile.build
- ❑ Keep an eye on the project forum
- ❑ Start the design document early
 - It counts 50% of your project scores!
 - Its questions can enlighten your design!
 - Is shared this time (1 per group)

Design Milestone

- Decide on the data structures
 - Data structures for s-page table entry, frame table entry, swap table entry
 - Data structures for the “tables” (not necessary a table) such as hash table? array? list? Or bitmap?
 - Should your “tables” be global or per-process?
- Decide the operations for the data structures
 - How to populate the entries of your data structures
 - How to access the entries of your data structures
 - How many entries your data structure should have
 - When & how to free or destroy your data structure
- Deadline
 - March 19th 11:59pm, no extensions

End

- Questions?
- Good luck!