Slide No. 2

The presentation includes following topics:
Memory and Access: Under i386 arch., how does Pintos handle the virtual memory, physical memory management and their mappings?
Current Status in Pintos: After a successful implementation of Pintos project 2, what functionalities relevant to virtual memory management does Pintos provide.
Requirements: The requirements of virtual memory management.
Suggestion: some suggestions to how to design and implement project 3.

Slide No. 3

Here is a simple overview of how pintos mount a user program into its RAM. First, user executable was stored on disk. With in the executable, user code uses relative/virtual address (0~3GB in i386 arch.). User virtual address forms a linear virtual address space. On execution, Pintos will install all loadable (in project 2 not 3) segments into its RAM as you have seen in the load, load_segment, and setup_stack function in process.c. During the installation, the mapping between virtual and physical address is setup through MMU components.

Slide No. 4

Here is the animation of the simple idea of what project 3 should think about. A linear user virtual address is divided into three segments: pointer to PDE, pointer to PTE, and offset in the page. The mapping is fulfilled by CPU through the support of MMU component. Pintos MMU and its interface could be find in mmu.h and pagedir.c. Moreover, Pintos allows pages to be stored on disk temporally so as to release the limitation of number of available RAM. There are also other strategies like lazy loading. Project 3 also needs to provide a grow-able stack. Associated with these issues is how Pintos could manage the memory resources efficiently. This is the problem to be addressed in project 3's virtual memory management implementation.

Slide No. 5

After a successful project3 following functions should be available from Pintos without any modification.
Kernel's memory has been well handled. Project 3 manages only user pool RAM.
MMU—PD, PT, pagedir—address maps. These are interface to operate on MMU.
Per-process page table (process.c). Install process's page directory in process_activate and update in install_page.
Preload data and code segment and stack (load, load_segment, and setup stack).
Fixed stack and limited size and number of programs. In setup_stack, it load pages for stack at once and provide no further extension in the rest of project 2.

Slide No. 6:

In project 3 following are specific requirements that you much address.

Page Table Management: you need to implement a “page” table management module so that you can track virtual pages, RAM frames and pages on disk and manage them efficiently. After that you need extend current page_fault handler

Swapping: Your implementation should allow Pintos swap some RAM pages onto swap disk or file so that a process could still go ahead running even there are no too many pages in the user RAM. You need implement some replacement algorithm like a 1-bit clock algorithm (NRU) on deciding which page to swap out. Of cause, you need provide functions to move pages between disk and RAM. You also need provide data structure and function to keep track all these pages.

Lazy loading: In project 2, loader will install all loadable segments into RAM in the very beginning of process running. However, in project 3, you need not to do that. You should allow Pintos to load code and data segment only when they are visited by the process. That is, you need only to setup the stack on initial loading.

Memory mapped files: another function you need to implement is map a file from disk into user address space. We will discuss it later.

Slide No. 7

Page table management is the core function of project3. It need at least provide following functions: It should provide proper data structures to keep track user virtual pages, RAM physical frames, pages on the disk or in files. It also need to provide mechanisms to map between (must be two way) virtual and physical addresses. Given the address, it should be able to locate where is the page: not loaded? In RAM? Or in Swap. If you want to use page directory directly, you will have to understand how pintos currently implement virtual-physical mapping through PD and PT as well as how you implementation is related to PD and PT. However, if you want to implement you own data structure over it, you need only understand the interface provided in pagedir.h and pagedir.c.

Here is some hint of some issues: To execute a program, Pintos need to install user virtual pages into concrete physical frames and keep tracking of such installation. The CPU will use page directory and page table (already implemented in mmu.h and pagedir.h, and pagedir.c). However, although it is convenient for CPU to use MMU, it is hard for you to use it in managing all kinds of pages and provide complex functions. So, we suggest you implement you own page table module on top of current PD & PT.

Slide No. 8

After you have implemented you page table management modules, you need to extend you page_fault handler so as to provide more efficient memory utilization.

After project 2, you loader will install new PD when switching so that multiple process could share RAM user pool. However, it setup all the mapping on loading the executable and provide no further extension and flexibility afterward. For example, it will simply kill a process when it use an invalid address even it should cause Pintos add new pages to the stack.

However, after project 3, an page fault is not always an error. It might cause swap in or out a page, add a new page to the stack, or load pages from executable file.

The problem is how to implement these functions. Here are some necessary functions you need to have: get the fault address (already implemented), find the page (either in file or on swap), let these pages in/out RAM, and update the PTE so that CPU will use you updated address mapping.
After the you have processed the page fault, it should return from the page_fault handler and resume the user process.

Slide No. 9

Here is graph that might help you understand the process of causing and handling page_fault. It is a part of Dr. Back's slides, I omitted the TLB here.

Slide No. 10

Now you might be a little confusing. There seems to be too many pages and page tables. In the above discussion we mentioned several concepts: PageDirectory & PageTable; Frame page; Virtual page; and Swapped page. You have to understand the relation and difference of these concepts before you can design and implement you project 3. Then, the major task of the project is to keep track and update their relations (or mapping).

Slide No. 11

Here frame, physical page, and RAM page all mean to the memory page in RAM's user pool. You DO NOT handle kernel page in virtual memory management.

Remember that, Pintos support upto 64MB main memory and half (32MB) are assigned to user. However, the actual available RAM might be less than 64M. There is a global variable "ram_pages" in init.c. Its value is provided by load.s. You need to figure out what does this variable mean and if you could use it in judging how many frames are there in the user pool. The current allocator of frames is provided through palloc.c. You need to figure out how you get a page from user pool and kernel pool and where the current allocator is utilized in current implementation.

We suggested that there are two approaches to handle frames in your implementation: utilize current allocator (palloc, mmu.h, and pagedir). These interface is not convenient for future complex memory management. The other one is our suggested approach: implement your own frame table and allocator on top of current one.

Slide No. 12

Virtual page table (or simply page table) management is the core in the project 3. It is the hub of tracking mapping information among virtual page, frames, and pages on disk. You have to provide a delicate data structure to manage page table. First, you need to understand virtual pages relation to those segments in linear user address space which is used by in organizing executable file. Then , when you install a virtual page into frame, you need to keep track the mapping between page in virtual address space and frame in physical address space. You also need update PT so that CPU will be aware of the new built mapping. Whenever a page is swapped from RAM to disk, you need keep track the relation between virtual page and swapped page otherwise you would not be able to swap

them in during later page_fault. As to the lazy loading, you also need to keep track that a virtual page is related to which part of the executable.

Slide No. 13

When there is no free pages and a process is requesting more memory, Pintos should be able to swap out some currently frames on to disk so that the process could go ahead running. The page is stored on the swap disk temporally. Some issues need to be understood. The memory (both virtual page and frame) is of size 4kB while disk is divided into sectors of size of 512B. Thus, you need to consider how to organize the sectors on swap disk so that you could store pages on it. One thing should be mentioned is that there should be no specific ordering of pages to be stored on disk.

Slide No. 14

After introducing the above concepts, we could use several question to give some hints on what information you have to handle at least.
What is the start/end address of the page/frame? (Here the end address is only important for partial pages) What is the flag of the page? Where is the page located, in RAM, on Swap disk, or in file? How to keep track the relations among these “pages”? When and what to update to keep the track?

Slide No. 15

Stack growth is a function you need to provide in project 3. Here is some hints of the problem:
The first page allocated in setup_stack in project 2. According to the i386 memory layout: there is a vacant between data segment and stack, and the stack grow downwards from PHYS_BASE. In i386 arch., it use push instruction to put something onto the stack. The instruction could put at most 32B on top of the stack at once. When your are push something larger than what stack can hold, it will cause a page_fault and the top of the stack is stored in esp. Thus, if fault_address-32 is less than esp, you may say there is a need to allocate new page for the stack. Finally, you need to limit the size of stack reasonably

Slide No. 16

Swapping is another function you need to provide. The function is specified as following: When there is no free memory and a process request one, some current loaded pages need to be evicted—swap out onto disk. Then after a while, the swapped file might be needed by its owner process, then it is reloaded—swap into the some free frame. Both of them involves causing page_fault. Whenever you swap, you need to keep track and let CPU know the new mapping. For efficient swapping, you need to manage the swap disk which should not indicate any ordering of pages. You should allow parallel access to swap.

Slide No. 17

As to evicting a page, here are some hints that might be helpful when you are implementing. You implementation need to keep track where is the free frames, who are evict-able frames. Then you need to pick a replacement algorithm like LRU. To keep

track and evict a page, you may need to use the Accessed/dirty bit in PTE. Then you need to provide functions to move pages between memory and swap.

Slide No. 18

Project 3 requires implementing lazy loading. Here are some hints on the problem.
In lazy loading (demand loading), you don't have to load all loadable pages into RAM at the very beginning. Instead, you load a page only when it is needed by the process. Thus, you need to change current load functions (load and load_segment in process.c). Remember, you still need to setup the stack when creating a new process. Lazy loading need not use swap disk, it use the executable file directly. Thus , you need to keep track the location of a page in the file.

Slide No. 19

Finally, you need to implement memory mapped file in project 3. The main idea is to keep a copy of an open file in memory (user's virtual space, not necessarily in frame any time). The mmapped file need to be kept in continuous virtual pages. If file size is not multiple of PGSIZE, you will still need a whole page for the tail. However, the rest of last page is filled with zero which causes a partial page. The zeroed partial page is discarded when the file is unmapped. You will need to assign each mmapped file a unique ID (within the process). There are some cases that you should not memory map a file like: zero address or length, overlap, or console file. You need to provide two system calls: mmap(fd, addr), munmap(ID).

Slide No. 20

Finally, you need take care you virtual memory management on process termination. You need to implement at least following tasks: Clean you page table, Free your frames, Free your SWAP, and Close all files.

Slide No. 21

There several issues you need to take care during you design and implementation. First, in project 3, sometimes you need actually access the content of user memory instead of just verify user address as what you have done in project 2. Following actions must be taken to guarantee your access to user memory won't cause inconsistency: check address, lock frame, read/write, and unlock the frame so that others can use.
Synchronization is an important issue, since you have to allow parallelism of multiple processes. For example, there might be page fault form multiple processes.
Say, A's page fault need I/O (swap, lazy load); B's page fault need not, then B should go ahead. Finally, select a proper container will affect your design significantly. There are several containers you could utilize in Pintos: Bit map, hash, list, and array. Moreover, you need to consider how many copies of a container you will need (as to virtual page table, frame, swapped page), should it be a per-process data structure or globally available? Make sure you consider it carefully before you start you implementation. Also, be sure to make your data structure simple to handle. For example, it is fine to use a linked list to keep your mmap IDs.

Slide No. 22

Here is a suggestion that you do read the design milestone document provided by Dr. Back carefully before you start action on project 3, it will save you significant time and effort. Following is a suggested implementation order for project 3, details please refer to the following link:

- Frame table management: your own “palloc”—modify or design new
- Switch to your allocator—modify process.c
- Virtual page table management
- Page Fault
- Stack growth, mmap, and reclamation
- Swap: accessed/dirty bits, alias, parallelism, and eviction algorithm

Slide No. 23

There are still some other suggestions:

- Go over your lecture notes, Dr. Back’s lecture 19 through 25 and Professor McQuain’s note on memory management and virtual memory are all helpful. Make sure you understand them.
- Design before start
- Keep an eye on Dr. Back’s project pages as provided in the previous slide.