Project 3: Virtual Memory
Project 3

- Background
- Goals
- Data structures
- Functionality
- Getting started
- Hints
Due **Wednesday** May 19 at 10:00 PM

Builds on project 2 – you need to have a working project 2 to get all the tests to pass

- Especially system calls
- Many of the tests reused from project 2
- Test multi-oom not reused, you should be ok if you got everything but this test to pass
You should have noticed in project 2 that the virtual memory system was very limited
- Processes limited to one page of stack
- System limited to the amount of physical memory
- Read-only code segments not supported
- No support for memory-mapped files

We’re going to fix these in project 3
Each process has its own page directory
- \*pagedir on struct thread

Switching user processes switches page tables
- pagedir_active(*pd)

Since 32-bit, 2-level page directory (PML2)

All processes share the same kernel mappings
The overall goal: give user processes the illusion they have all the memory they want

Run out of physical memory? Kick other pages out!
- What page will you remove and where does it go?

Be as lazy about using physical memory
- Only load pages into physical memory when they are actually needed
Goals - Specifics

- **Paging**
  - Data structures to record the usage of physical frames, virtual pages, swap space
  - Lazy loading of executable files
  - Page eviction algorithm
  - Support swapping to disk

- **Stack growth**
  - Allocate additional pages on demand

- **Memory mapped files**
  - System calls to map and unmap files
Data Structures

- Three main data structures for paging: frame table, supplemental page table, swap table

- We will look for these three data structures, but for each data structure, many designs are possible
  - Specifically left open-ended
  - First, you should decide:
    - Implement with arrays, lists, bitmaps, hashtables, etc.
    - Are data structures per-process or global?
Frame table – records info about physical frames of memory being used as pages in user processes

Confusing terminology – by “frames” we really mean pages from `palloc_get_page(PAL_USER)`
- Virtual pages divided into a user pool and kernel pool
- User pages mapped 1-1 with actual physical memory, thus called “frames”

Isn’t this what the page table does?
- Need a global (not per-process) table to support eviction
- Layer over functionality of `palloc_get_page(PAL_USER)` so you can record which frames are in use (design up to you)
Supplemental page table – record information about where pages live

Now pages can be in frames, in swap, or on disk and not yet loaded (lazy loading of executables and memory mapped files)

During a page fault, you need to be able to look up the location of an existing page to bring it into memory, or grow the stack on demand for new pages

Clean up the pages for a process when it exits
Think about what this is doing: you should be moving frames in and out of the current page table (*pagedir)

You will want to layer over `pagedir_set_page()` and `pagedir_clear_page()`

You will additionally use `pagedir_is_dirty()` and `pagedir_is_accessed()`, more about this later
- Make a swap disk. From vm/build:
  - Create an n-MB swap disk: `pintos-mkdisk swap.dsk --swap-size=n`
  - Create a temporary n-MB swap disk for a single run with `--swap-size=n`

- There is a block device specifically for swap (see devices/block.h)
  - `swap_device = block_get_role (BLOCK_SWAP);`
Fairly easy to work with a block device:

- `void block_read (struct block *, block_sector_t, ...);`
- `void block_write (struct block *, block_sector_t, ...);`
- You will need to determine the number of swap blocks per page

Swap table – track in-use and free swap slots

- Swap slot – page sized region of swap blocks
- Choose an unused swap slot when you evict a page from its frame to swap

Free a swap slot when its contents are read back into a frame
In project 2: you need to know when you need a new page

- Say the user runs out of stack, he needs a new *upage at vaddr
- `palloc_get_page(PAL_USER)` returns a *kpage
- Register vaddr to *kpage with `pagedir_set_page()`
- User accesses memory at *kpage via user vaddr
In project 3: Do work until a page fault

Page fault handler: What’s the faulting virtual address? Is it valid?

Data structures: Which virtual page contains this address? Is the data in swap or in a file?
  - Do we need to evict a page to swap for this page?
  - Or, do we need to grow the stack?
  - Once you get a frame, register it with `pagedir_set_page()` to create the vaddr mapping

Return from `page_fault()` retries faulting instruction
Lazy loading of executables

- Modify `load_segment()` in `process.c` so you record how to load segments from the file – don’t actually read them into frames yet
- Wait until a page fault to actually read from the file and load into a frame

Upgrade your page fault handler

- Figure out if fault was from a bad address, a reference to a page that needs to be swapped in, a page that needs to be read from disk, or if the stack needs to grow
Grow the stack on demand

- Add a new stack page automatically
- Still setup the first stack page and arguments immediately
- To know that you want to grow the stack, you need to check the user’s stack pointer
- If you used technique #2 for user memory access, you can’t get the stack pointer from `intr_frame` in `page_fault()`
Add system calls for memory mapped files

- `mmap(fd, *addr)` lazily maps the file `fd` into consecutive virtual pages starting at `addr`

- When unmapped, any dirty pages are written back to the actual file (also upon eviction)
Functionality – Eviction

- When a page needs to be loaded into a frame but none are free, evict a frame

- Use an eviction algorithm to choose a frame to evict
  - Suggest the clock algorithm – use functions from `pagedir.c` to check accessed and dirty bits
  - Upon eviction, frames go to swap, get written back to disk, or go away (i.e. read only pages)
  - If swapping out and no more slots, panic kernel
Suggested order of implementation

- Frame table, be able to allocate frames – change `process.c` to use your frame table allocator. Don’t swap yet; panic when you run out of frames
- Enough of the supplemental page table and page fault handler to get lazy loading of executables
- Then can move on to stack growth, memory mapped files, page reclamation on process exit in parallel
- Finally eviction and swapping – you can actually do the swap table at the end
Hints

- Getting started – you’re going to want to modify anything in `process.c` that calls `palloc_get_page(PAL_USER)` to use your frame table allocator

- Modify `/src/Makefile.build` when you add VM files

- Don’t try to do the whole thing at once – design and develop in stages, even if you have to backtrack some
  - Suggest updating design document after each stage
You will end up needing to synchronize a lot of code
- In general, anytime you are in the middle of reading or writing to a frame, you need to pin it in memory
- What could go wrong if you don’t pin a frame?

You will end up needing to change your system calls.
- You need to make sure pages passed to devices are in memory and pinned during the call

You will need to be able to handle concurrent page faults from different processes
- In general, you need to synchronize pages and frames

Try to keep eviction algorithm fairly simple
Extra credit: implement sharing of read-only pages for executables loaded from the same file
   - Not worth a lot (+5)

I want to re-emphasize – don’t try to do it all at once!

Near the end, have stable code to fall back upon so you can submit something working if your latest code breaks

Update your design doc as you go along so it’s good by the end, even if you don’t finish all the features / tests