PROJECT 4: PINTOS VIRTUAL MEMORY

CS124 – Operating Systems

Spring 2024, Lecture 19

Project 4: Pintos Virtual Memory

- Implement disk-backed virtual memory in Pintos
- Initially, Pintos has very limited support for virtual memory
  - Process address-space isolation, and user program loading
- Several facilities are provided to help:
  - Pintos supports a swap partition for loading and saving virtual memory pages
  - Also supports a bitmap implementation, which may be useful for tracking available slots in the swap partition
  - Should also be able to rely on existing Pintos synchronization mechanisms, etc.
- All Project 3 tests (system calls) should continue to pass
  - Caveat: the "no-vm" tests are not run – this includes multi-oom
- New tests to exercise Project 4 functionality as well

Project 4: Requirements

- Implement demand-paging for loading program binaries
  - This should be pure demand-paging
- Implement demand-paging for stack pages
  - May want to always allocate a frame for the first stack page
- Implement support for memory-mapped files
  - Memory-mapped regions in a process cannot overlap
  - If two processes map the same file into memory, your implementation doesn't need to keep the data consistent

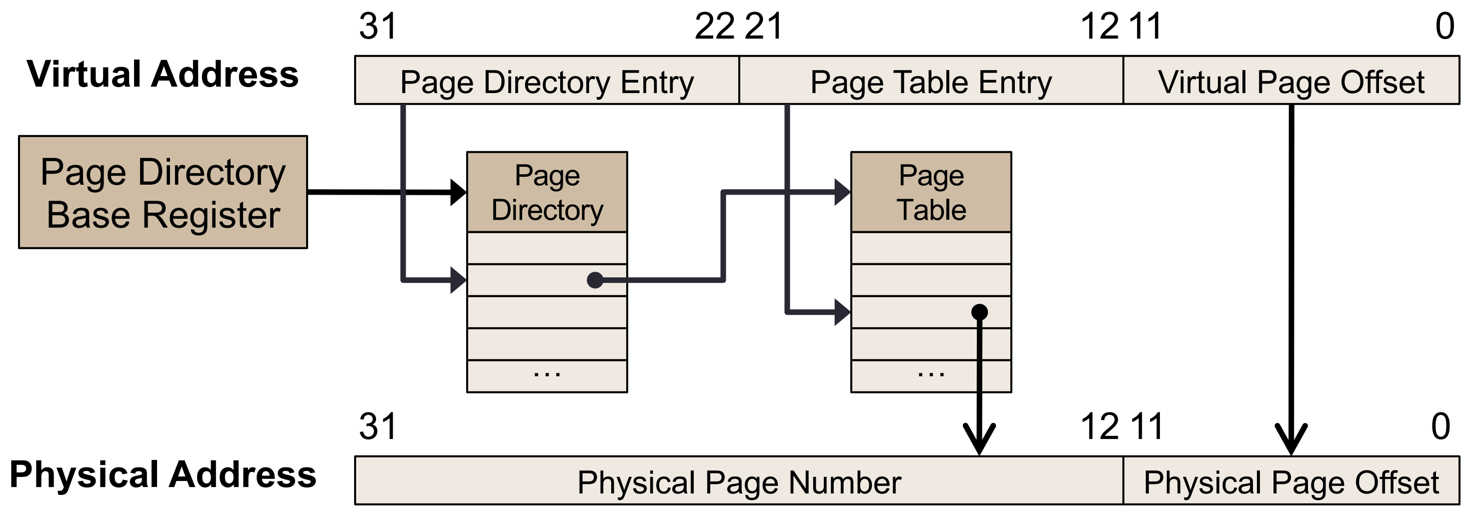
- Pintos doesn't have shared libraries or dynamically resizable data segments (i.e. user-space **`malloc()`**)
  - Only dynamically-resizable memory area in processes is the stack

Project 4: Requirements (2)

- Important concurrency requirement:
  - If a page fault requires I/O to resolve, it shouldn't block other page faults that don't require I/O to resolve
  - Shouldn't be too hard to satisfy this requirement – just don't hold any global lock while performing I/O operations
  - Note: Disk interactions are already serialized in the IDE device implementation (**`src/devices/ide.c`**)
    - The kernel-thread doing the read/write is passively blocked on a lock until the operation completes
- Optional extra credit:
  - Implement shared-memory support for read-only sections of program binaries (this is complicated enough…)
  - Pintos system-call API doesn't support shared data-segments, or shared read-write sections

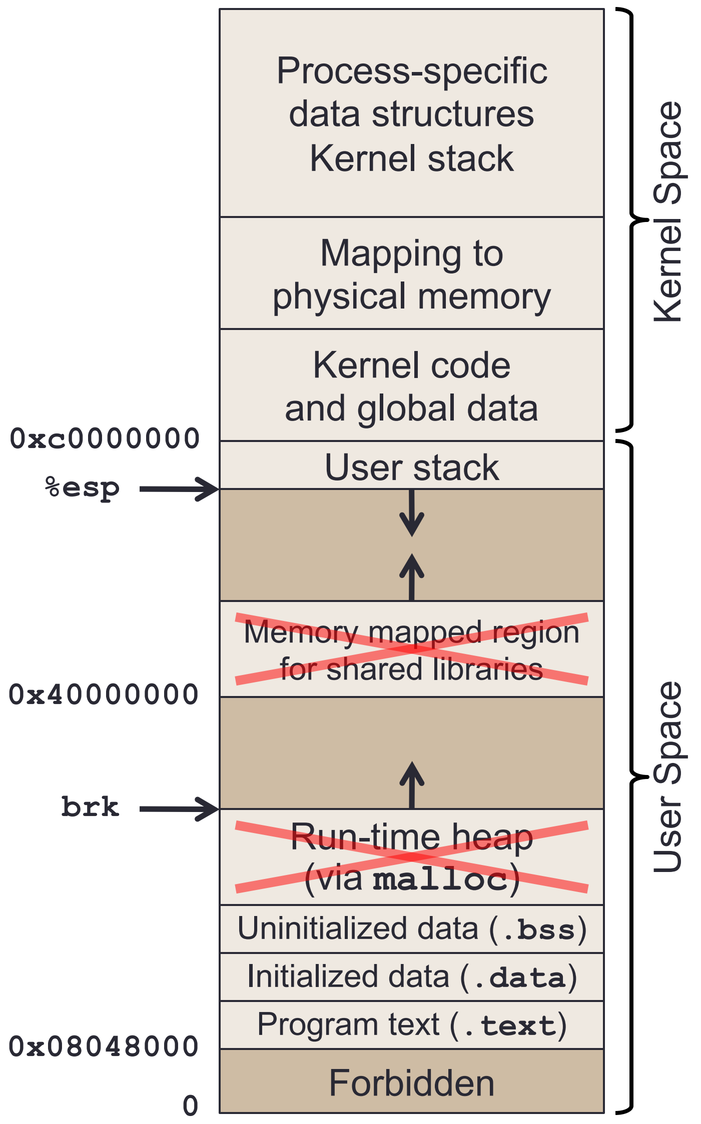
Overview of Pintos Virtual Memory

- **Frames** (aka "physical pages") are contiguous regions of physical memory, with a specific page size and alignment
  - 4KiB on x86 processors
- **Pages** (aka "virtual pages") are contiguous regions of virtual memory, with a specific page size and alignment
- IA32 uses a two-level page-table hierarchy to map pages to frames
- **Swap slots** are contiguous regions of memory on the swap device, for storing the contents of a virtual page when they aren't in physical memory
  - Each slot is 4KiB in size (same as virtual page size)



Pintos Address-Space Layout

- Pintos roughly follows the virtual memory layout used by Linux on IA32
- Boundary between kernel-space and user-space is 0xc0000000 (3GiB)
  - Defined as **`PHYS_BASE`** – use this constant, not a magic number
  - See **`threads/loader.h`** and **`threads/vaddr.h`**
- Several Pintos simplifications:
  - No shared libraries!
  - No dynamically-resizable memory heap!
  - No user-space **`malloc()`** support!

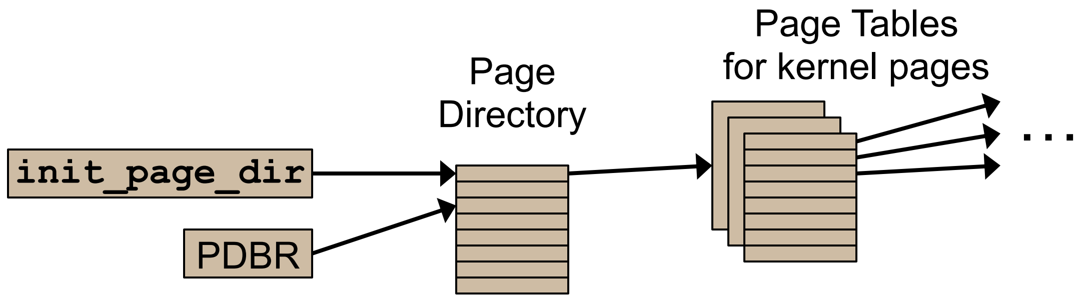

Process-specific data structures Kernel stack
Mapping to physical memory
Kernel code and global data
Kernel Space
0xc0000000
User stack
%esp
~~Memory mapped region for shared libraries~~
0x40000000
brk
~~Run-time heap (via malloc)~~
Uninitialized data (.bss)
Initialized data (.data)
Program text (.text)
0x08048000
Forbidden
0
User Space

Pintos Address-Space Layout (2)

- Pintos uses up to a maximum of 64MiB physical memory
  - 64MiB physical memory / 4KiB page size = 16384 page frames maximum
  - **`init_ram_pages`** stores the actual number of frames (set up in **`threads/start.S`**)
- A major simplification: Pintos always keeps all page-frames mapped into kernel space
  - Frame at physical address 0x0000 is mapped to kernel-space address PHYS_BASE + 0x0000
  - Frame at physical address 0x1000 is mapped to kernel-space address PHYS_BASE + 0x1000
  - *Real operating systems don't do this!!*
- Can easily manipulate the contents of any frame from within kernel
- Can easily compute any frame's physical address or kernel-virtual address

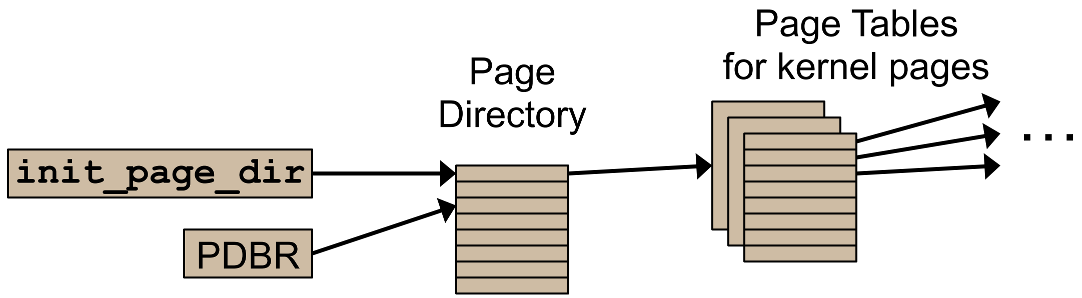
Pintos Process Page Directories

- Every Pintos process has its own page directory, initialized when the process is started
  - **`pagedir_create()`** in **`userprog/pagedir.c`**
- Process' page directory is copied from an "initial page directory" created when virtual memory is initialized
  - **`paging_init()`** in **`threads/init.c`**
- Only includes virtual memory mappings for kernel pages
- No user-process pages, yet…



Pintos Process Page Directories (2)

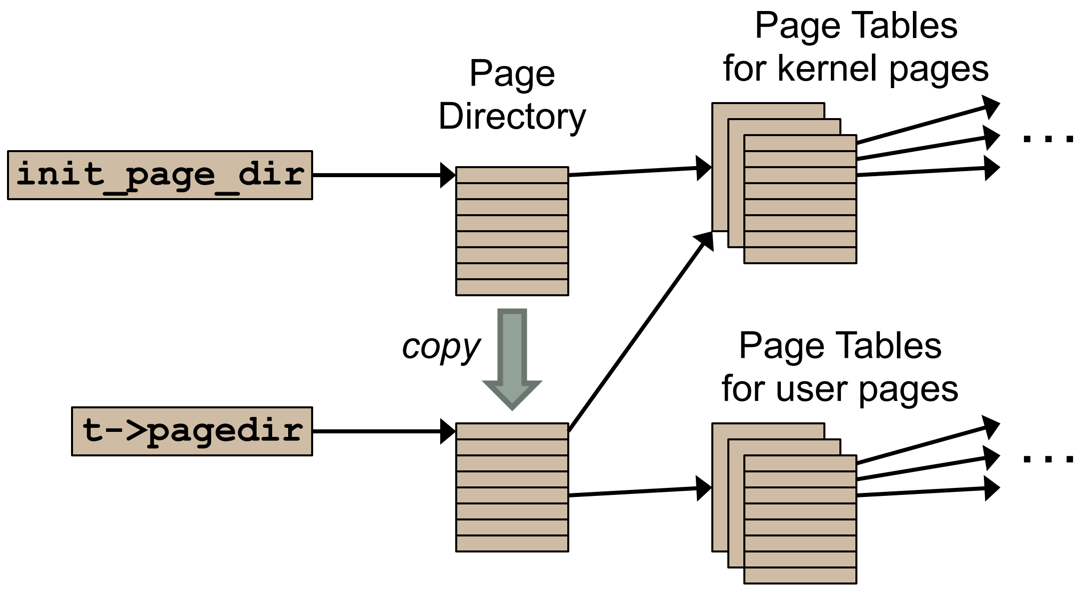
- On IA32, there are 1024 entries in the page directory…
  - Page directories are 4KiB, each entry is 4 bytes → 1024 entries
  - The IA32 address-space is 32 bits, or 4GiB addressable memory
  - 4GiB / 1024 entries → each page-directory entry corresponds to 4MiB of the process' virtual address space
- The kernel-space boundary starts at 3GiB…
  - The top 256 entries of the "initial page directory" correspond to kernel-space



- Since Pintos only supports a maximum of 64MiB of memory, only 16 page-directory entries are required for kernel-space
  - 16 entries × 4MiB = 64MiB

Pintos Process Page Directories (3)

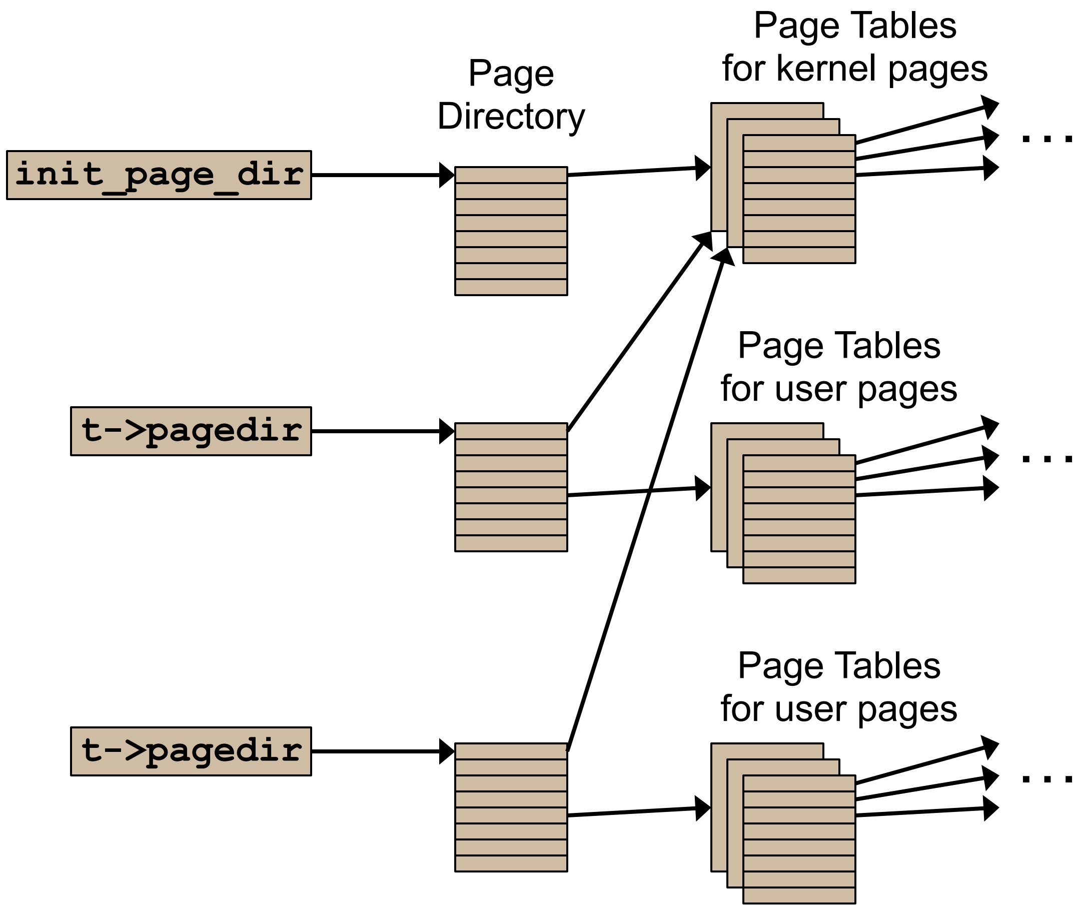
- The process' page table is copied from **`init_page_dir`**
  - **`pagedir_create()`** in **`userprog/pagedir.c`**
  - This is a *shallow* copy – only the Page Directory node is copied
- Subsequent user-process memory-map operations only affect the process' page directory, but not the contents of **`init_page_dir`**



Pintos Process Page Directories (4)

Consequences:

- All processes share the same Page Table nodes specifying the kernel-space memory map
- If a kernel-space mapping is changed, or if kernel data is changed, while running one process, all processes see the change
  - The same physical pages are mapped into all processes' page tables
- If a user-space mapping is changed, or if user data is changed while running one process, only that process sees the change


Page Directory
Page Tables for kernel pages
init_page_dir
…
t->pagedir
Page Tables for user pages
…
t->pagedir
Page Tables for user pages
…

Implementing Virtual Memory

- Most of the virtual-memory system design falls out of answering the question: **What does the page-fault handler need to do?**

1. Is the faulting address valid?
   - If so, what data *should* exist at the faulting address?
   - (If address isn't valid, just terminate the process)
2. Allocate a page-frame to the faulting process
   - May require evicting another process' page to free up a frame
3. Load the necessary data into the page-frame
4. Install the frame at the required virtual address

Implementing Virtual Memory (2)

- Most of the virtual-memory system design falls out of answering the question: **What does the page-fault handler need to do?**

- Note: The page-fault handler will often block on IO
- May seem strange for an interrupt handler to perform a blocking operation…
- Page faults are caused by user processes, so they actually run in process context, not in interrupt context
  - i.e. the kernel is resolving a fault on behalf of a specific process
  - You can think of it as a second way a user application can trap into the kernel to get some work done – but without its knowledge
- While the fault is being resolved, other processes can run

Is the Faulting Address Valid?

- The IA32/x86-64 page-table structure doesn't hold sufficient information to implement the Pintos virtual memory abstraction
- OSes usually implement a **supplemental page table** that holds the required OS-level details
  - Each process has a supplemental page table along with an MMU page table
  - Implementation should support all details necessary for the OS virtual-memory abstraction being provided
  - Typically, the supplemental page-table describes the entire address-space of the process, not just what is not currently in physical memory
- Should be fast to determine whether a virtual address is valid, and if so, what data resides in that virtual page

Sources of Page Data

- **Where does page data initially come from?**
- Binary program data
  - The **`.text`** / **`.data`** / **`.rodata`** sections in ELF binary file
  - Some of these are read only, others are read/write
  - Changes to read/write data must not be saved back to original file!
- The "anonymous file" – zero-initialized memory
  - Stack pages should initially be all zeros
  - The program's **`.bss`** segment should initially be all zeros
  - **`.bss`** is described in ELF binary (starting address + size is given), but the file doesn't contain any actual data for **`.bss`**
- Memory-mapped files
  - Loaded on request of user applications
  - In Pintos, changes to data must be saved back to original file

Sources of Page Data (2)

- **Where is page data evicted to?**
- Binary program data
  - Read-only sections (`.text` / `.rodata`) can be discarded, since they can be reloaded from the original binary
  - Read-write sections (`.data`) cannot be saved back to original binary – must use swap storage
- The "anonymous file" – zero-initialized memory
  - Stack pages and `.bss` pages have no backing file data – must use swap storage
- Memory-mapped files
  - Upon eviction, may be saved to the backing data file
  - Pintos has no concept of memory-mapping files as "read only"
- If a page is saved to swap, must also store what slot it was saved to

Sources of Page Data (3)

- The supplemental page table must record and track all of these details for managing virtual pages
- Should make it easy to determine if a faulting address is valid, and if so, where to get the data from
- Similarly, when a page is being evicted, should make it easy to decide where the page data should be saved to

Allocating Page Frames

- Page-fault handler must find an available frame to hold the page being loaded
- Implication: The OS needs to know how many frames the hardware provides, and which frames are available / in use
- Also: If no frame is available, and the OS must evict a page to make one available, it must know which process was previously using the frame
    - Perform proper page-out operations based on the page's details (see earlier slides)
    - Update process' MMU page table to record page is not in memory

- This information is recorded in the **frame table**

Suggested Order of Implementation

- Implement the frame table first, and update `process.c` to use your frame-table allocator
  - e.g. populate the frame table with repeated calls to `palloc_get_page(PAL_USER)`
- Idea: need to know what frames are in use, so the virtual memory system can make allocation/eviction decisions
  - What frames are available for use?
  - What frames are currently in use, and by which process?
- Swapping and eviction won't work yet, so you can panic the kernel if you run out of frames

- Make sure all the Project 3 tests pass!!!
  - (That is, Project 3 tests that are also included in Project 4's tests; no-vm tests are not incl.)

Suggested Order of Implementation (2)

- Implement the supplemental page table and page-fault handler next, and update `process.c` to use your supplemental page table
- Idea: Instead of allocating pages immediately, just record the information necessary for loading each page in the supplemental page table
- When a virtual page is accessed, it will generate a page fault
  - Allocate a frame to hold the page's contents using your frame allocator from previous step
  - Install the frame into the process' address space at the appropriate address
  - Load the page's contents from disk file or swap, if necessary
  - Info for where to fetch the page's data from will be in your supplemental page table
- Still don't have eviction yet, so don't worry about evicting pages! ☺

- Make sure all the Project 3 tests pass!!!

Suggested Order of Implementation (3)

- From this point, can implement various things in parallel
  - Stack growth, mapped files, reclaiming pages/swap at exit
- Finally, need to implement page eviction
  - Technically, you can implement this once you have the previous operations completed, but it may be difficult to debug
- Initially you can implement a very simple paging policy
- **For the love of all that is good, <u>do not</u> use "Always evict frame 0" policy!**
  - Tests will run *extremely slowly*. You will be sad.
  - Even a random eviction policy would be better than this.

Page Eviction Policy

- Project 4 requirement is to approximate LRU somehow
- Assignment write-up suggests the CLOCK policy
  - An efficient implementation of the Second Chance policy, which does not require a timer interrupt-handler
  - Should be pretty straightforward to implement
  - Use an index into your frame-table for the clock hand

- Feel free to do something more sophisticated. (Or, feel free to keep your life as simple as possible. It's CS124 after all.)
- More interesting policies will get bonus credit
  - Anything with a timer tick will get a few points
  - Anything adaptive will get more points

Concurrency and Synchronization

- Your implementation will have several long-running operations that can interrupt each other
  - Scanning to find an available page-frame to use
  - Scanning to find an available swap slot to use
  - Evicting a page from a frame (includes I/O)
  - Loading a page into a frame (includes I/O)
- Important concurrency requirement (from earlier):
  - If a page fault requires I/O to resolve, it shouldn't block other page faults that don't require I/O to resolve
  - Shouldn't be too hard to satisfy this requirement – just don't have your fault handler hold one global lock for the entire fault operation!

Concurrency and Synchronization (2)

- Can achieve the required concurrency goal entirely using locks; shouldn't have to disable interrupts anywhere
  - Note: If you implement a timer-based page-replacement policy, interrupts will likely need to be disabled in a few critical places
- Feel free to have locks guarding larger operations, e.g.
  - Scanning to find an available page-frame to use
  - Scanning to find an available swap slot to use
  - Will limit the concurrency of the virtual memory system, but if it isn't required by the assignment, why make your life harder?
- Main issue: don't hold a lock guarding a larger operation, and then perform I/O while holding it
  - If holding that lock will block other page-faults that don't require I/O, then it will violate the concurrency requirement

Concurrency and Synchronization (3)

- Operations on frames need to be synchronized carefully
- Example:
    - Process A page-faults on its virtual-page 25. Some page must be evicted to make a frame available.
    - The kernel picks frame 19, which is currently being used by Process B for its virtual-page 38. It starts writing this page to swap. Process A is suspended, and kernel switches to another process.
    - Process B starts to run; it tries to access data in its virtual-page 38.
- Does your pager fault? If so, how is the fault handled?
    - When is the MMU page-table entry for Process B's page 38 updated? Before or after the page has been written to swap?
    - If Process B faults on a page that is being evicted, do you make B wait until the eviction is completed, before B's page is reloaded? If so, how?

Concurrency and Synchronization (4)

- You may find it helpful to incorporate locks into individual page-frames, to coordinate operations on frames
- Can use locks to implement pinning, as well as blocking other kernel threads from accessing the locked data
  - `void lock_acquire(struct lock *)` – blocks the thread trying to acquire the lock
  - `bool try_lock_acquire(struct lock *)` – attempts to acquire the lock; returns false immediately if lock is unavailable
  - `bool lock_held_by_current_thread(struct lock *)` – reports whether the current thread holds the specified lock
- Example: finding an available frame for page-in
  - Can try to acquire each frame's lock in sequence
  - If the lock is already held by some other thread, the lock-attempt will fail. Just go on to the next frame.

Final Notes

- Follow a modular approach in your implementation
  - Separation of concerns
  - Provide functions to encapsulate important operations, e.g. “find a frame,” “evict page,” “find swap slot,” “load page,” etc.
  - Should make it much easier to follow the logic of your page-fault handler.
- Develop incrementally, and test your implementation after each new feature is added
  - This project is horrible to debug if you don’t test along the way.
  - Should be quite reasonable to debug if you add features one at a time, and exercise your code completely before moving forward.
- If your team develops components in parallel, integrate your work often
  - Don’t incur extra merge-conflict overhead for your team
  - Communication within the team is of *utmost importance* in these situations