# CS140 Project 3: Virtual Memory

TA: Diveesh Singh

## High level Project requirements

**High level goal**:  Remove current limitation that the number and size of programs running in Pintos is limited by main memory size.

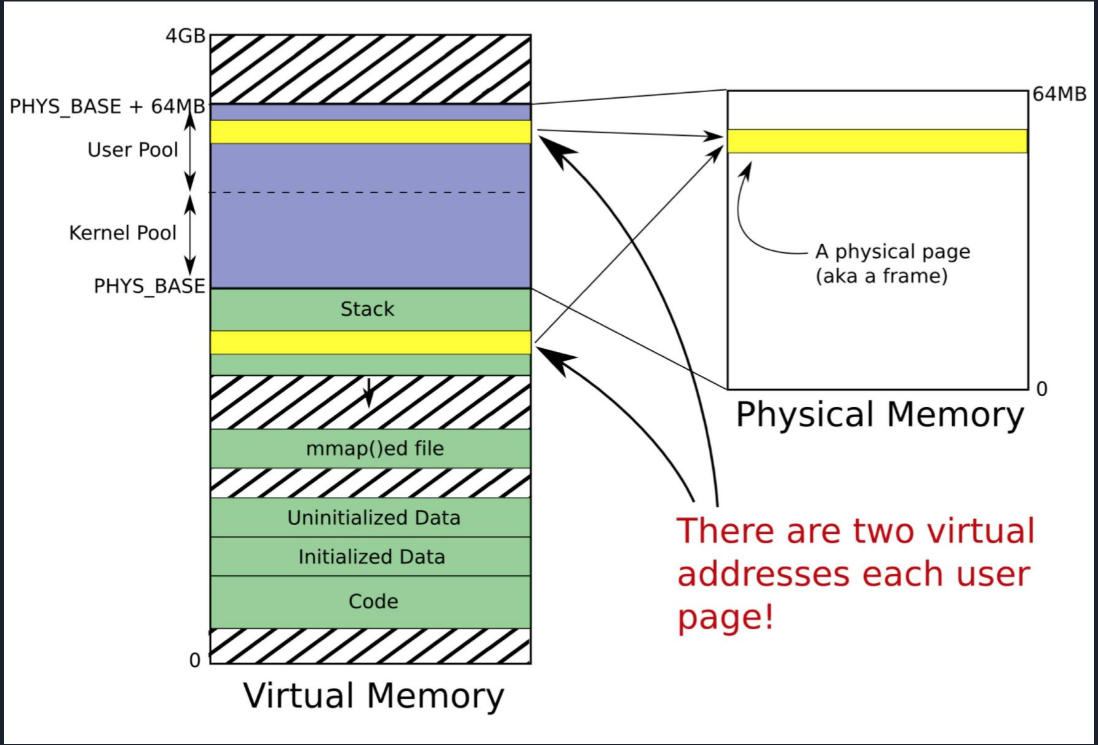
**High level components:**

- Frame table
- Supplemental page table
- Swap table
- Modifying the page fault handler
- Stack growth
- Mmap/Munmap system calls

# Physical memory (Pintos)

- Physical memory is divided up into physical pages or frames
- Divided up into two pools: user pool and kernel pool
- Every kernel virtual address directly maps to a physical address
  - Translated by PHYS_BASE
- Every user virtual address has a corresponding kernel virtual address
  - User virtual address points to kernel virtual address which points to physical address

# Physical memory (Pintos)



## Terminology review

- **Page**: a contiguous region of *virtual* memory
- **Frame**: a contiguous region of *physical* memory
- **Page Table**: data structure to translate a virtual address to a physical address (i.e a page to a frame)
- **Eviction**: removing a page from its frame and potentially writing it to swap/FS
- **Swap slot**: where evicted pages are written to in the swap partition

## What exactly are you doing?

- You have limited physical memory, many processes want to use physical memory
- Physical memory isn’t big enough to house every process’s pages all the time
- If a page isn’t needed in physical memory, gets “paged out”
  - OS Slang: “paged out” = contents of a page in physical memory getting written out to the swap table or file system (more details later)
- When a process needs a page and it’s not in physical memory, it has to get “paged in” (usually “paging out” another page)

## You will need to design...

- Supplemental Page Table
  - Per-process data structure that keeps track of supplemental data for each page, such as location of data (frame/disk/swap), pointer to corresponding kernel virtual address, active vs inactive, etc.
- Frame Table
  - Global data structure that keeps track of physical frames that are allocated/free.
- Swap Table
  - Keeps track of swap slots.
- File Mapping Table
  - Table to keep track of which memory-mapped files are mapped to which pages.

## Lots of options for data structures!

- Arrays
  - Simplest approach
  - Sparsely populated array wastes memory
- Lists
  - Also pretty simple
  - Traversing a list can take lots of time
- Bitmaps
  - Array of bits, each of which can be true or false
  - Used to track usage in a set of identical resources
  - Supported by Pintos (check out lib/kernel/bitmap.c and lib/kernel/bitmap.h)
- Hash Tables
  - Also supported by Pintos (check out lib/kernel/hash.c and lib/kernel/hash.h)

## Frame table: `vm/frame.[ch]`

- Frame = one chunk of physical memory
- Data structure that keeps track of  which user page occupies which slot of physical memory (frame)
  - Which thread owns which slot/frame?
- Obtain pointers to frames via `palloc_get_page(PAL_USER)`
  - returns a kernel vaddr, directly corresponding a slot of physical memory
- Think of frames like actual picture frames, user pages as pictures

## Frame table: `vm/frame.[ch]`

- Frame = one chunk of physical memory
- Data structure that keeps track of which user page occupies which slot of physical memory (frame)
    - Which thread owns which slot/frame?
- Obtain pointers to frames via `palloc_get_page(PAL_USER)`
    - returns a kernel vaddr, directly corresponding a slot of physical memory
- Think of frames like actual picture frames, user pages as pictures

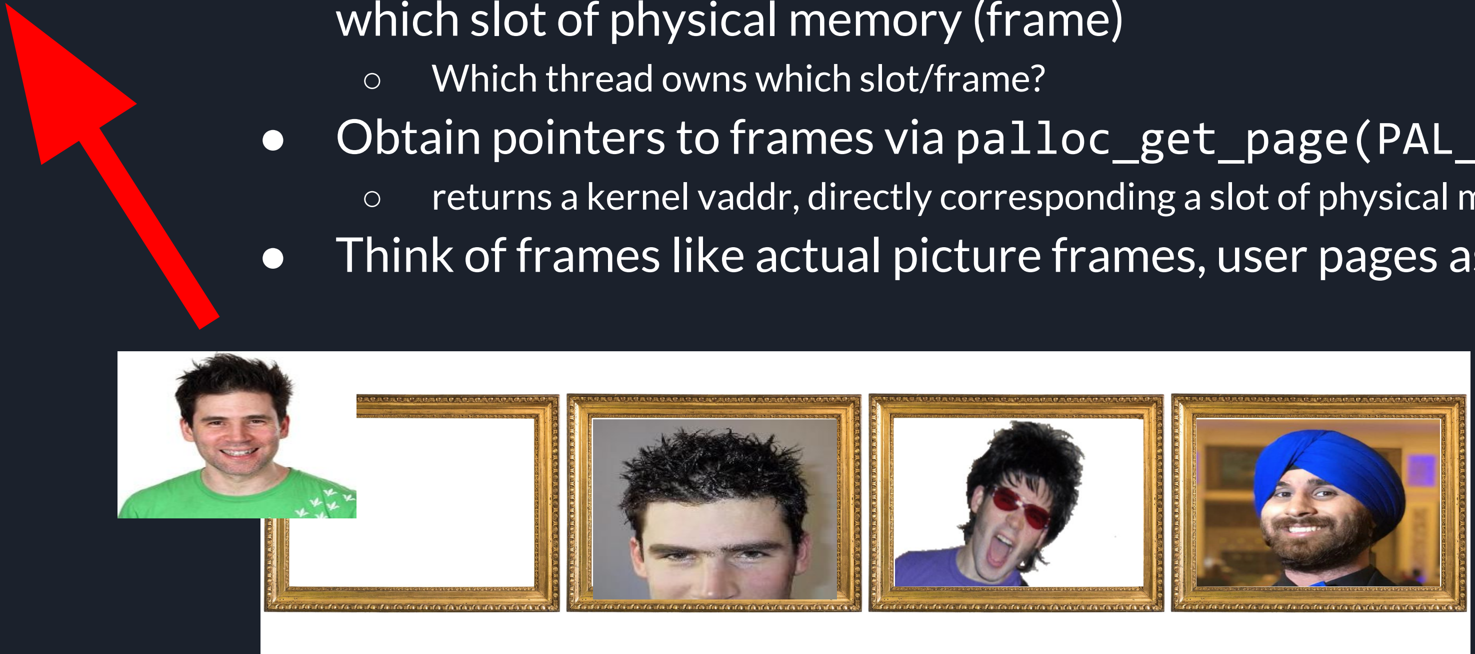
## Frame table: `vm/frame.[ch]`

- Frame = one chunk of physical memory
- Data structure that keeps track of which user page occupies which slot of physical memory (frame)
  - Which thread owns which slot/frame?
- Obtain pointers to frames via `palloc_get_page(PAL_USER)`
  - returns a kernel vaddr, directly corresponding a slot of physical memory

Think of frames like actual picture frames, user pages as pictures

## Frame table: `vm/frame.[ch]`

- Frame = one chunk of physical memory
- Data structure that keeps track of which user page occupies which slot of physical memory (frame)
    - Which thread owns which slot/frame?
- Obtain pointers to frames via `palloc_get_page(PAL_USER)`
    - returns a kernel vaddr, directly corresponding a slot of physical memory
- Think of frames like actual picture frames, user pages as pictures



## Frame table: `vm/frame.[ch]`

- Frame = one chunk of physical memory
- Data structure that keeps track of which user page occupies which slot of physical memory (frame)
    - Which thread owns which slot/frame?
- Obtain pointers to frames via `palloc_get_page(PAL_USER)`
    - returns a kernel vaddr, directly corresponding a slot of physical memory

Think of frames like actual picture frames, user pages as pictures

## Frame table: Eviction

- If there's a free frame, simply return it
- In most cases, there's no free frame i.e. you must evict another page from a frame
- Eviction is done lazily i.e whenever you need a frame, evict page
- Steps
    - Use algorithm to choose a frame to evict,  just make sure it is at least as good as the "clock" algorithm
        - Accessed and dirty bits will be extremely useful here!
    - Remove references to the frame from any page table that refers to it (supplemental page table AND pagedir)
    - If necessary, write the page to the file system or to swap.

## Accessed and dirty bits

- On any read/write to a page, the hardware sets accessed bit to 1
- On any write, hardware sets dirty bit to 1
- OS (i.e the code you are writing) can set these back to 0 (likely during your eviction code)
- Two virtual addresses can refer to the same frame => CPU only updates the accessed/dirty bits of the page used to access
    - I.e if you access a page in physical memory via the user vaddr, it won't update the bits for the corresponding kernel vaddr and vice versa

# Frame table: General Tips

- Synchronization!
    - Multiple threads will be accessing your frame table and modifying frames while trying to page in/page out
    - Eviction algorithm will require you to synchronize
- Frame table manages frames for user pages ONLY (PAL_USER)
    - Another way to think about it:
        - Frame table acts as a layer of abstraction over `palloc_get_page ()`
    - Modify calls to `palloc_get_page(PAL_USER)` in process.c
- `threads/init.c:` good place to call your frame init function
- You're allowed to use `calloc/malloc` to create your structs that store info about each frame
    - JUST MAKE SURE TO CHECK THE RETURN VALUE

## Supplemental page table: `vm/page.[ch]`

- Not every page for a user process will be in physical memory all the time (could be in swap or FS)
- Existing page directory only keeps info regarding pages that are in physical memory (used by the hardware)
- One supplemental page table per process that keeps track of its user pages
  - Most important is to keep track of page’s current location (memory, swap, FS)
  - Will need to add other fields

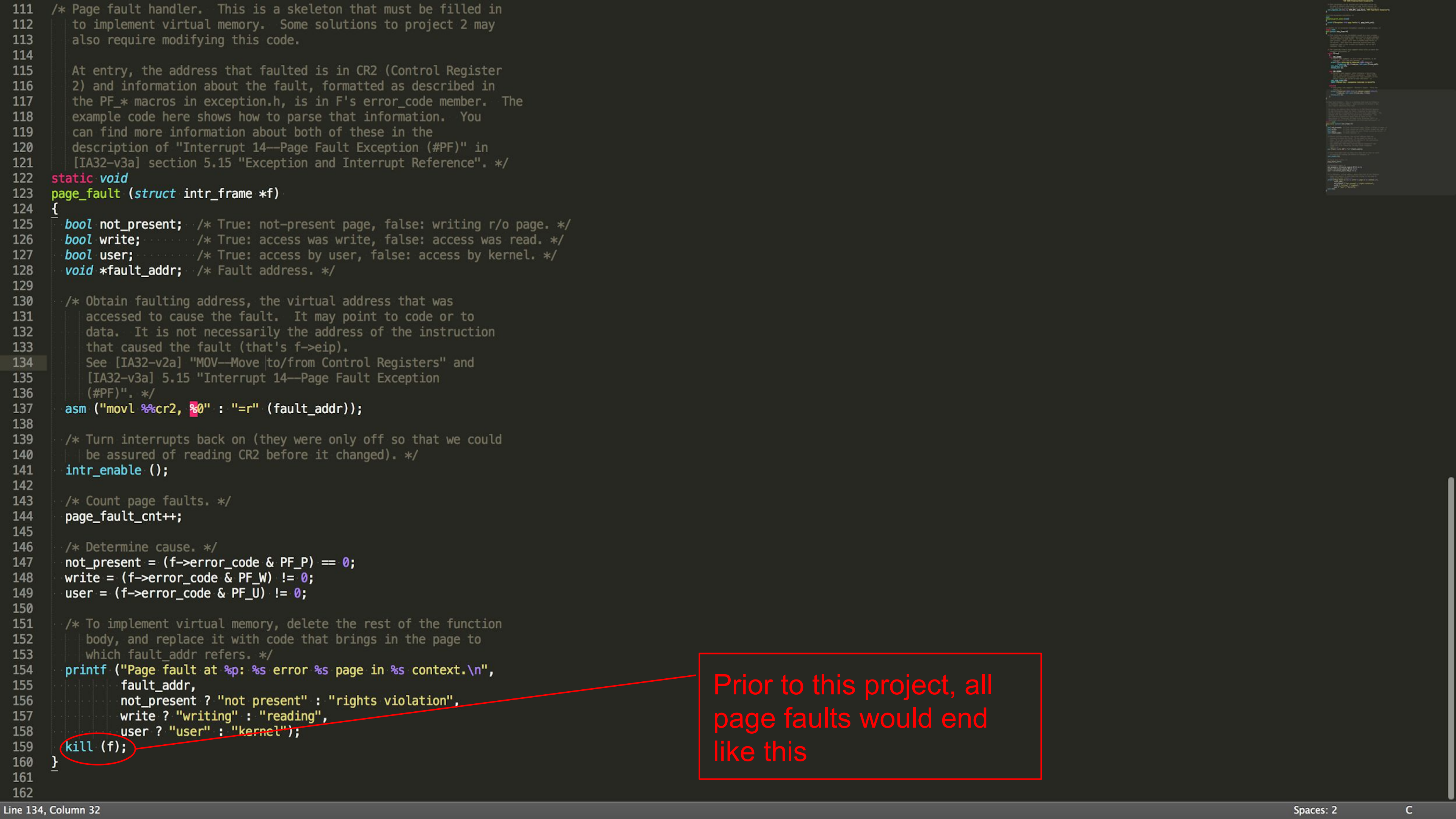
## Swap table: `vm/swap.[ch]`

- **Swap**: Special disk partition used for managing evicted pages
- *Swap table* should allow  picking an unused swap slot for evicting and a page from its frame to the swap partition
- Keep track of which swap slots are occupied (data structure!)
- Make sure accesses to the swap partition are synchronized
- Look into `devices/block.h` to figure out how to get access to swap partition

## exception.c

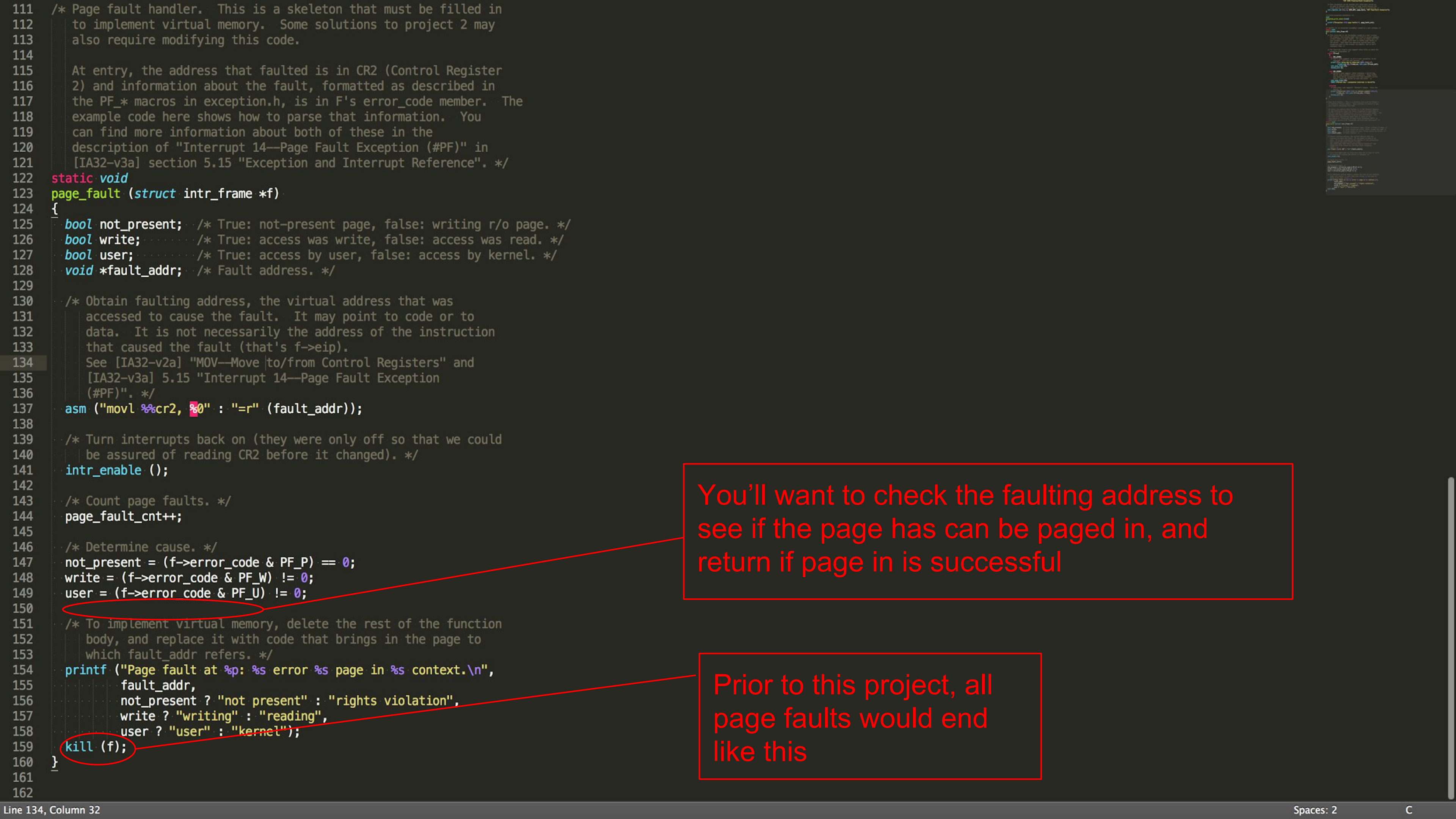
```c
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

Line 134, Column 32 Spaces: 2 C



```c
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

Prior to this project, all page faults would end like this



```c
/* Page fault handler.  This is a skeleton that must be filled in
   to implement virtual memory.  Some solutions to project 2 may
   also require modifying this code.

   At entry, the address that faulted is in CR2 (Control Register
   2) and information about the fault, formatted as described in
   the PF_* macros in exception.h, is in F's error_code member.  The
   example code here shows how to parse that information.  You
   can find more information about both of these in the
   description of "Interrupt 14--Page Fault Exception (#PF)" in
   [IA32-v3a] section 5.15 "Exception and Interrupt Reference". */
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
          fault_addr,
          not_present ? "not present" : "rights violation",
          write ? "writing" : "reading",
          user ? "user" : "kernel");
  kill (f);
}
```

You'll want to check the faulting address to see if the page has can be paged in, and return if page in is successful

Prior to this project, all page faults would end like this

## Page fault handler: `exception.c`

- Verify whether the read/write access was legal
- If legal, and the page has been evicted, load back in from FS or swap
    - Use your supplemental page table to figure out where the page is
- Obtain a frame to store the page
- Fetch data into the frame via FS or swap
- Update your supplemental page table accordingly
- Make sure that the *original* pagedir implementation is aware of the new page in physical memory
    - See function `process_install_page()` or `pagedir_set_page()`

## Page fault handler: General tips

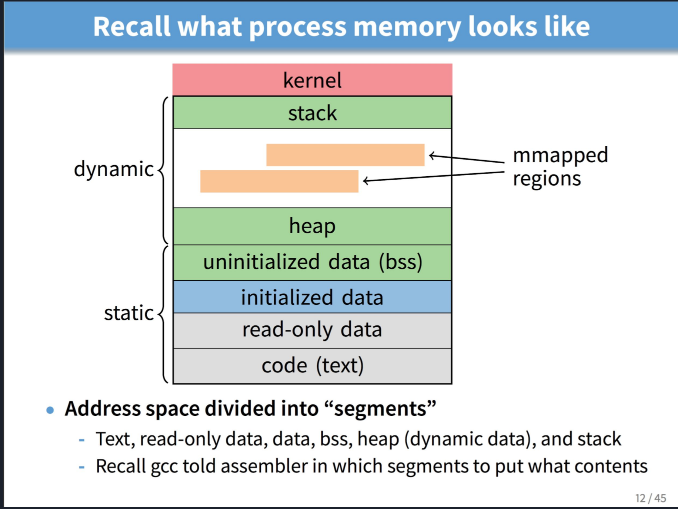
- Can still make calls to `thread_current()` from inside the page fault handler
- After paging in the appropriate page, call `return` to continue process execution
- Page faults also occur if a process tries to write to a read-only page (even if it’s in physical memory)
    - So not EVERY page fault is going to end up paging something into physical memory

## Stack growth

- Your implementation currently pre-allocates space for a user process's stack (remember Project 2 `process.c`)
- Now need to dynamically allocate more pages for a process stack as needed
- Valid stack accesses may cause the following page faults
  - PUSH: 4 bytes below ESP
  - PUSHA: 32 bytes below ESP
- Allocate new page in page fault handler if valid stack access
- Read spec carefully for details of how to get access to the current esp at the time of the page fault

# Memory mapped files

- `mmap()` and `munmap()`
- Processes may map files into their address space
- Memory mapped pages must be loaded from the disk lazily
- mmap() will return error status if
  - The size of the file is 0 bytes
  - File will overlap with another already mapped page
  - `addr` is not page aligned
- When you evict an mmap'd page, write changes to it back to the original file
- All mappings are implicitly unmapped on process exit



**Recall what process memory looks like**

- **Address space divided into "segments"**
  - Text, read-only data, data, bss, heap (dynamic data), and stack
  - Recall gcc told assembler in which segments to put what contents

12 / 45

## Where to page out/evict to?

- There are different “types” of pages that can be paged out
- User stack pages => page out to swap
- File pages (i.e mmap’d files) => page out to file system
  - If it’s dirty, write changes out to the corresponding file
  - If it’s not dirty, simply deallocate because you can reload from file system
- What about the loaded executable?
  - If it’s dirty, you must page out to swap (treat it like another user stack page)
  - If it’s not dirty, simply deallocate because you can reload from file system (treat it like any other clean file page)

## Suggested implementation order

- Fix any project 2 errors
- Frame table:
    - Change process.c to use frame table allocator instead of directly calling `palloc_get_page (PAL_USER)`
    - Kernel should still pass Project 2 tests
- Supplemental page table
- Stack growth, memory mapped files (mmap), and page reclamation on process exit
- Eviction/paging in and out

## General tips

- Spend a LOT of time designing your locking scheme
  - Ensure that locks are ALWAYS acquired and released in the same order
  - Page fault handler can be fired at *ANY* point
- START EARLY!!!
- This is (in most people's opinion) the most time-consuming project of the class