CS 140 Project 3
Virtual Memory

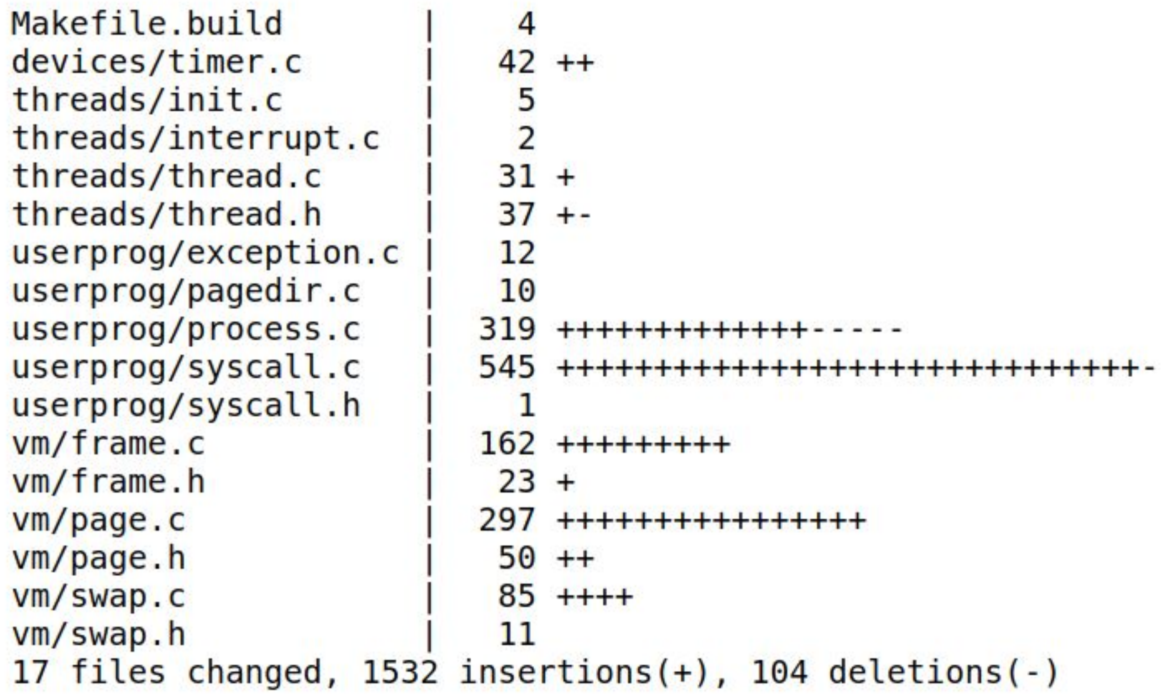
Motivation

After project 2, user processes use virtual addresses but are severely restricted.

- When a process loads, all bytes of process are allocated up front.
- Stack is limited to one page
- No way to release pages that are not in use until a process exits
- Physical memory is not large enough to hold pages of all processes running at one time

By the end of this project your OS will no longer be restricted in this way.

Changes



```
Makefile.build        |    4
devices/timer.c       |   42 ++
threads/init.c        |    5
threads/interrupt.c   |    2
threads/thread.c      |   31 +
threads/thread.h      |   37 +-
userprog/exception.c  |   12
userprog/pagedir.c    |   10
userprog/process.c    |  319 +++++++++++++-----
userprog/syscall.c    |  545 ++++++++++++++++++++++++++++++-
userprog/syscall.h    |    1
vm/frame.c            |  162 +++++++++
vm/frame.h            |   23 +
vm/page.c             |  297 ++++++++++++++++
vm/page.h             |   50 ++
vm/swap.c             |   85 ++++
vm/swap.h             |   11
17 files changed, 1532 insertions(+), 104 deletions(-)
```

Terminology

- **Page**: a contiguous region of virtual memory (virtual page)
- **Frame**: a contiguous region of physical memory (physical page)
- **Page Table**: data structure to translate a virtual address to a physical address (i.e a page to a frame)
- **Page Directory**: a level of indirection to page tables
- **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

Memory in Pintos

Physical memory divided into physical pages (frames)

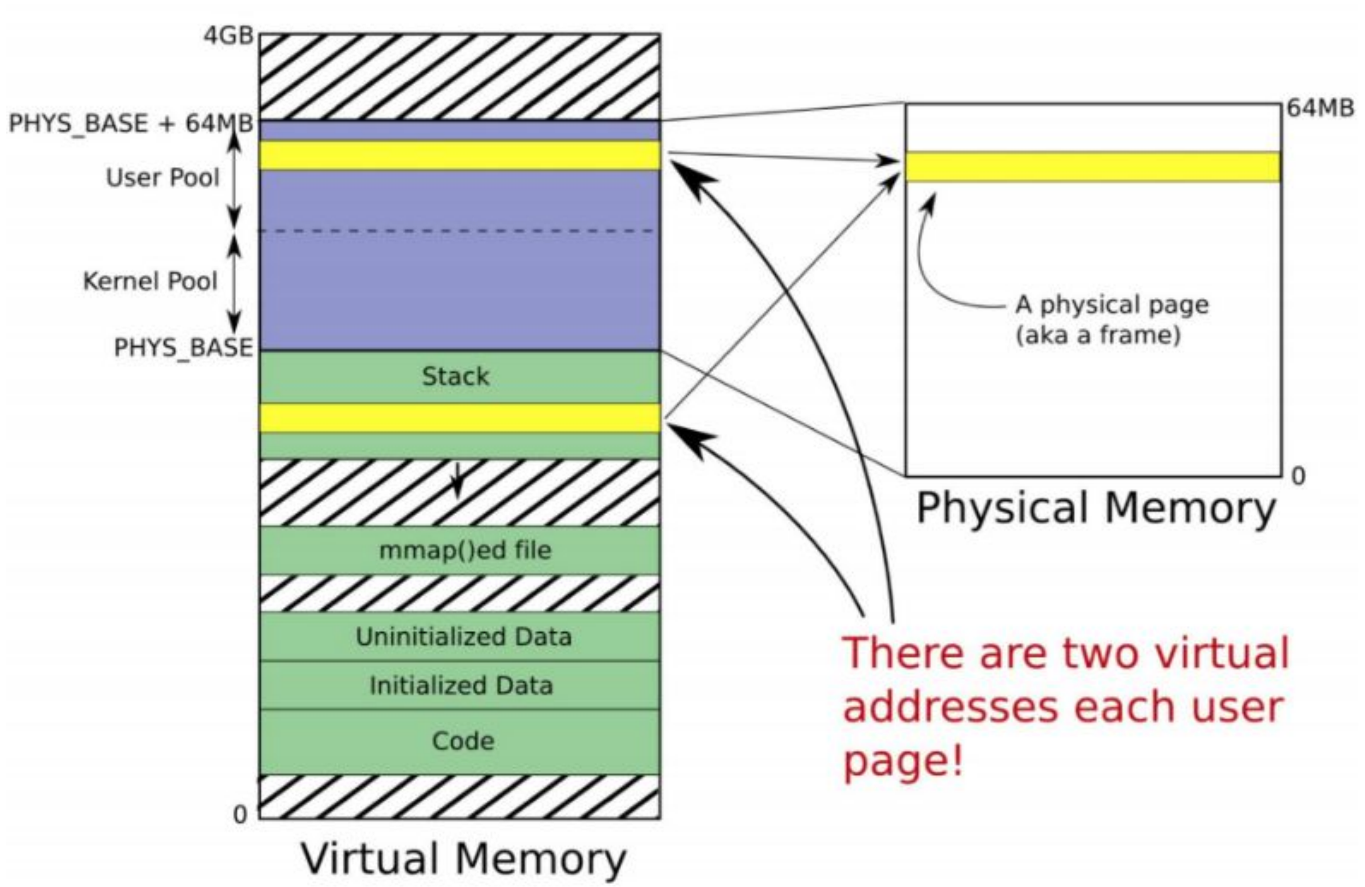
Two pools - user pool and kernel pool

```
    palloc_get_page (PAL_USER)
```

Physical addresses accessible through kernel virtual address with 1-1 mapping

Reference A.6

PHYS_BASE + 0x1234 accesses physical address 0x1234



There are two virtual addresses each user page!

Aliasing

Two virtual addresses refer to the same frame

CPU only updates the accessed/dirty bits in the page table entry of address used

Example: if you access a page in physical memory via the user vaddr, it won't update the dirty bits for the corresponding kernel vaddr

User stack

0xbffffff0 -> 0x00004000

Kernel

0xc0004000 - > 0x00004000

PTE for 0xbffffff0 and 0xc0004000 have different page table entries

Changes to this project

Page not needed in physical memory gets “paged out”

- Contents written out to swap

When a process needs a page not in physical memory, gets “paged in”

- Bytes read from file on disk or swap

Physical frames given to processes **lazily** after process load **(Demand Paging)**

- Current process.c **eagerly** loads executable into pages

Support new **mmap** system call for memory mapping

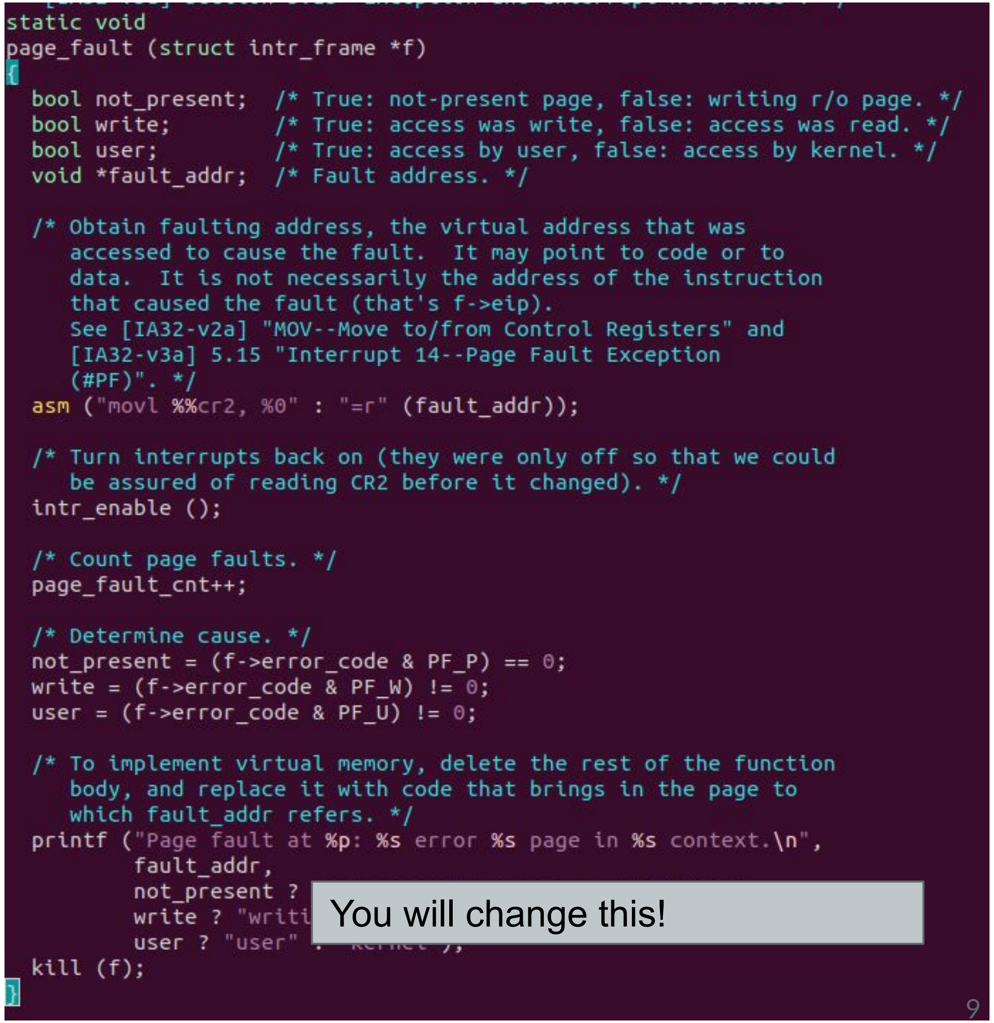
- See `man mmap` in linux

Page Fault

In assignment 2 a page fault meant the page was invalid and killed the process

Page fault now happens whenever a user accesses a **valid non-present** address

Returning from the page fault will allow the process to continue as if nothing happened



```
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 ?
          write ? "writi
          user ? "user" : "kernel");
  kill (f);
}
```

You will change this!

Required Metadata

Supplemental Page Table

- Keeps track of virtual page state (location, active/inactive, frame)

Frame Table

- Indicates if frame is free or in use, used for paging-in and eviction

Swap Table

- Swap slots on disk

File Mappings

- Which files are mapped to which pages

These data structures may be global, per process or combined

Data Structures to Consider

Arrays - Simple but wasteful if sparse

Lists - Simple but inefficient traversal

Hash Tables - Efficient lookup and sparse

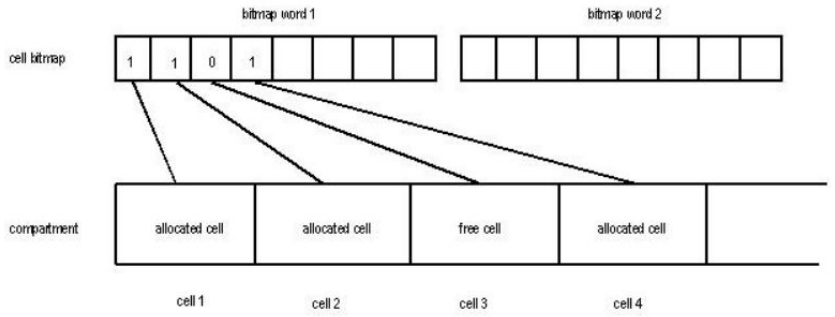
    lib/kernel/hash.c

    See ref A.8 for example use in code

Bitmaps - Compact

    1 bit can represent 4096 bytes

    lib/kernel/bitmap.c



Supplemental Page Table

On page fault, consult supplemental page table to find where data resides

Is it on swap, not loaded from disk, or zero memory?

When process exits, consult supplemental page table for resources to free.

Page fault

1. Lookup page that faulted in SPT.
    a. If valid find where the data lives, otherwise kill the process and free it's resources as before
2. Grab an available frame from your frame table
3. Load the data onto the physical frame given to you
4. Update processes pagedir and page table entry to point to the frame you loaded (see pagedir.c)

Additional Notes

When loading a process you should not use frames right away

Wait until an access triggers page fault handler and consult supplemental page table

One exception: first frame for the user stack

Other stack pages lazily faulted in

You will want to change process.c in load() to bookmark which pages are valid and what data should be paged in later

Frame Table

Hands out frames during page fault and determines which pages to evict

All frames are from the user pool (PAL_USER): Kernel frames are not candidates for eviction)

Initially most frames are free so just return first frame available

1. When none are free, choose a frame to evict and save the data if necessary
2. Use a page replacement algorithm checking accessed/dirty bits in the page table
    a. See userprog/pagedir.c
3. Update references to the evicted frame in the page table that previous held this frame

Don’t want previous frame holder to think it still holds the evicted frame

Eviction Details

Dirty frames should be written to swap

- Evicted stack frames with dirty bit set

Clean pages and read-only pages do not need to be written back

- text section of a program can be read directly from the file/ shouldn't be duplicated in swap

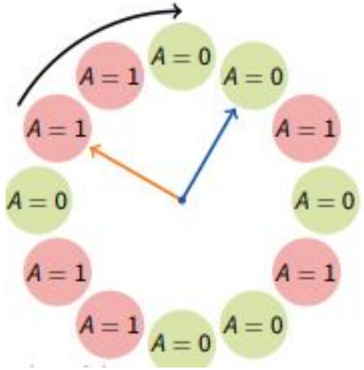
Your eviction algorithm should approximate LRU and be as good as the second chance clock algorithm.

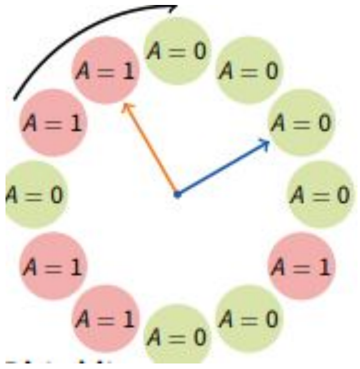
Most groups choose to go with clock algorithm from lecture.

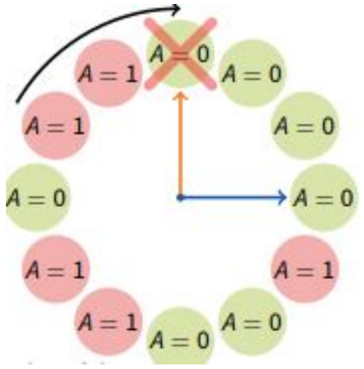
Clock Algorithm (Lecture 6)

First hand clears bits

Second hand evicts frames







Synchronization

You need to avoid race conditions during frame eviction

Multiple threads access the frame table and modify frames concurrently

Certain frames need to be “pinned” when loading data back from swap or a file

Pinned frames should not be evicted so avoid keeping a frame pinned longer than necessary

Memory Mapped Files

You will support an additional system call

- `mapid_t mmap (int fd, void *addr)`
- `void munmap (mapid_t mapping)`

Eviction writes contents back to the file instead of swap if dirty

File should be paged in lazily like other pages

Check file to be mapped with addr does not overlap existing data

Generally can be done in parallel after frame table and SPT implemented

Stack Growth

Supplemental page table will not map entire stack mapped when a process loads

Allocate additional pages as the stack grows

Devise a heuristic to determine if page fault caused by valid PUSH

Instructions can push 32 bytes below stack pointer (PUSHA)

struct intr_frame in page_fault () holds user ESP if valid accesses checked manually

Implementation Order

You are building off project 2 so fix any remaining bugs first. You might choose to use the timer you implemented in project 1 for eviction.

1. Start by implementing a frame table so loading a process uses frames instead of palloc. All of your project 2 tests should pass still.
2. Implement the SPT and again replace loading a process for demand paging. You will want to now have all pages fault in. Verify this works before moving on.
3. Mmap can be done in parallel with stack growth
4. If you thought ahead early about synchronization, eviction shouldn't add too many lines but this can get tricky with deadlocks and race conditions.

Misc

Setting up a swap partition requires one additional pintos command

--swap-size=n

You can look over “devices/block.c” to access the swap

Read/write is already provided but you must keep track of swap info

src/vm is empty

You can add source files in src/Makefile.build

General Tips

Spend a lot of time thinking about your data structures and synchronization before writing code

This assignment typically has the most deadlocks

Avoid adding synchronization at the very end: **(Warning)** It may require you to change a lot of code if done late

Try to think through all of the information needed in your frame table and supplemental page table early on after writing detailed demand paging and eviction pseudocode

Read the docs thoroughly including the reference materials in section A

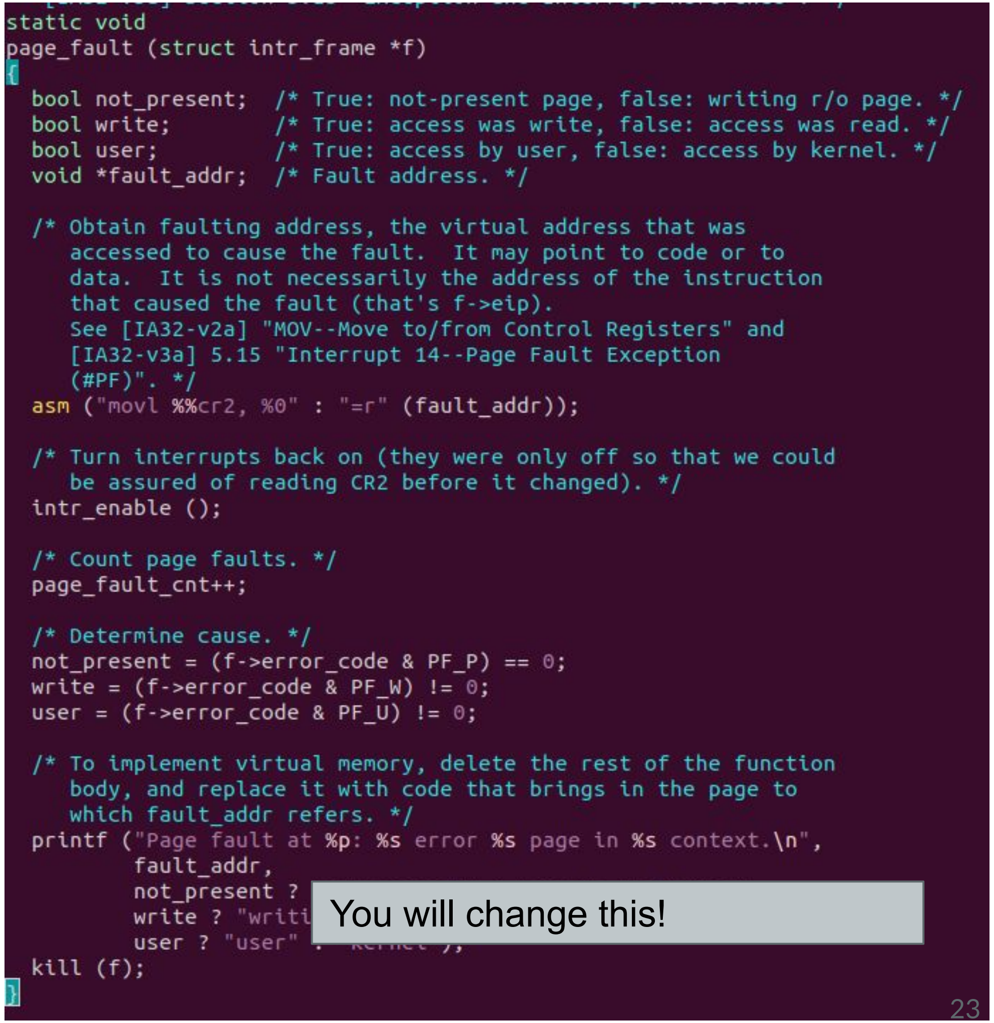
Most groups find either assignment 3 and 4 the most challenging so start early!

Style Tip

Don’t cram everything in here

Just make a good interface in src/vm

More readable and easy to debug if decomposed well



```
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 ?
          write ? "writi
          user ? "user" :
  kill (f);
}
```

You will change this!

Last Remarks

You should feel proud when you finish this lab!

All modern systems use demand paging virtual memory

Your pintos programs will now have the illusion of using a full 3GB address space while only using 64MB of RAM.