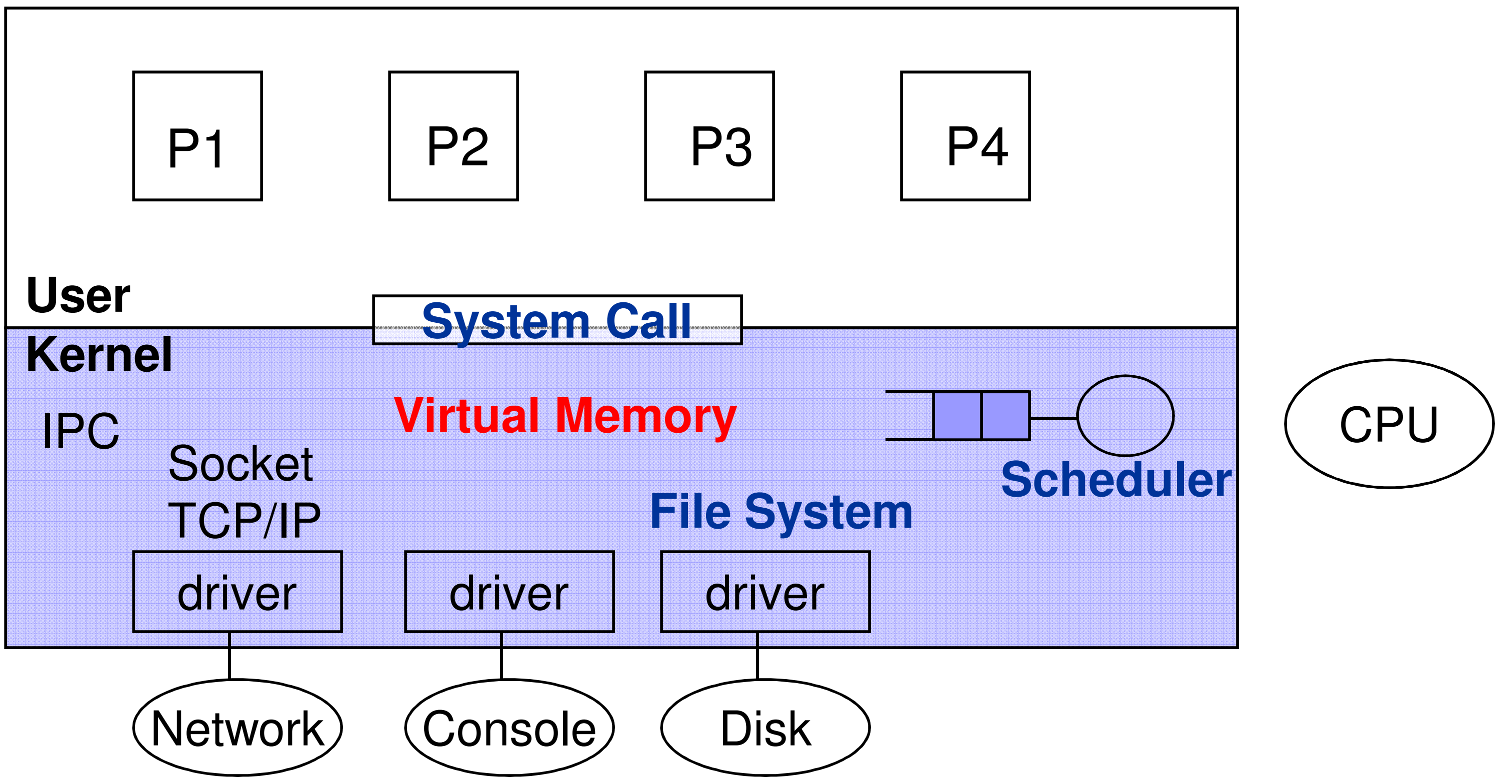
CS140 Project Session 3
Virtual Memory

Syed Akbar Mehdi

Based on slides from Derrick Issacson's (Win '08) and Ben Sapp's (Win '07)

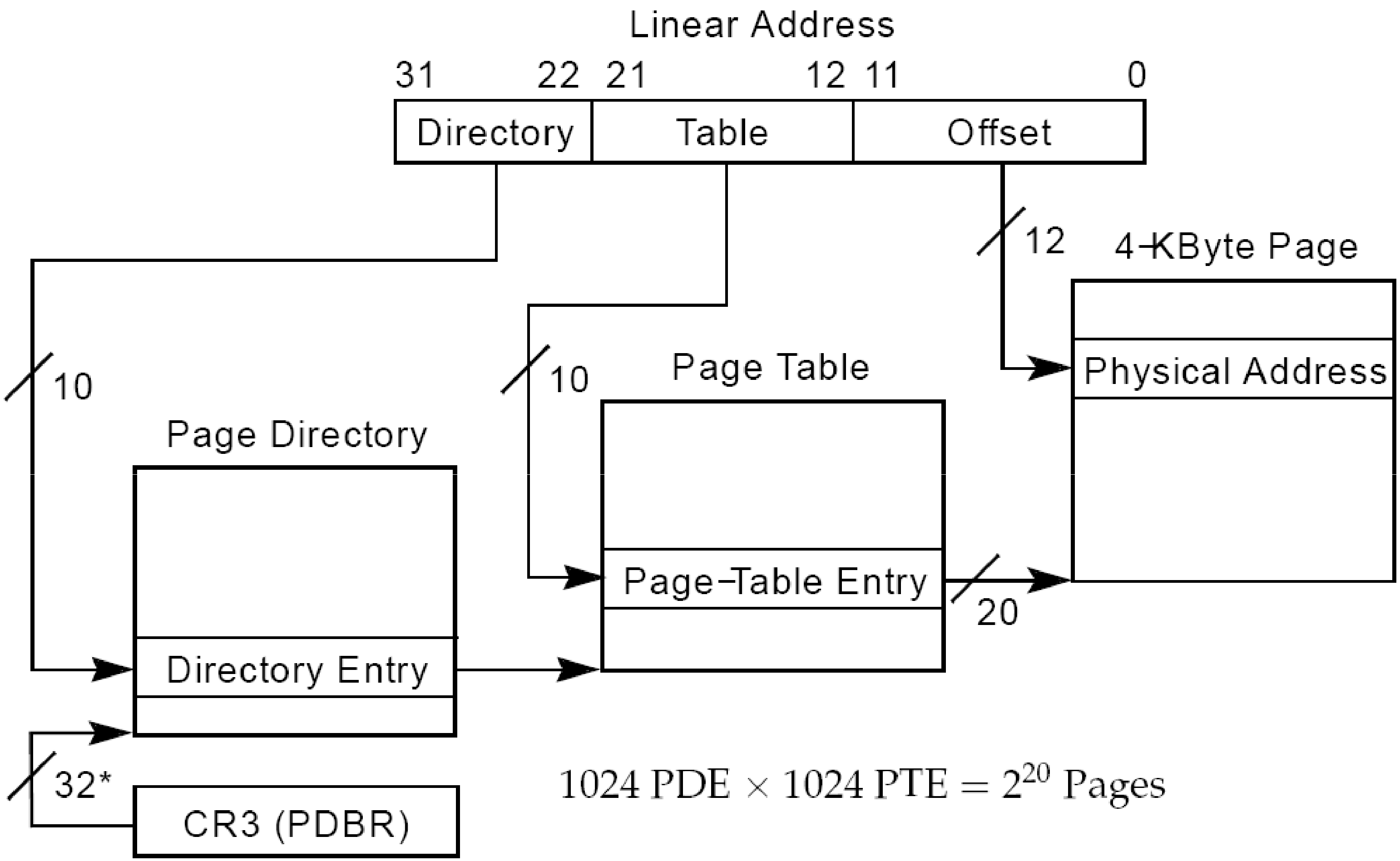
Overview

- Typical OS structure



Adopted from Lecture Notes L1 p.14

x86 page translation

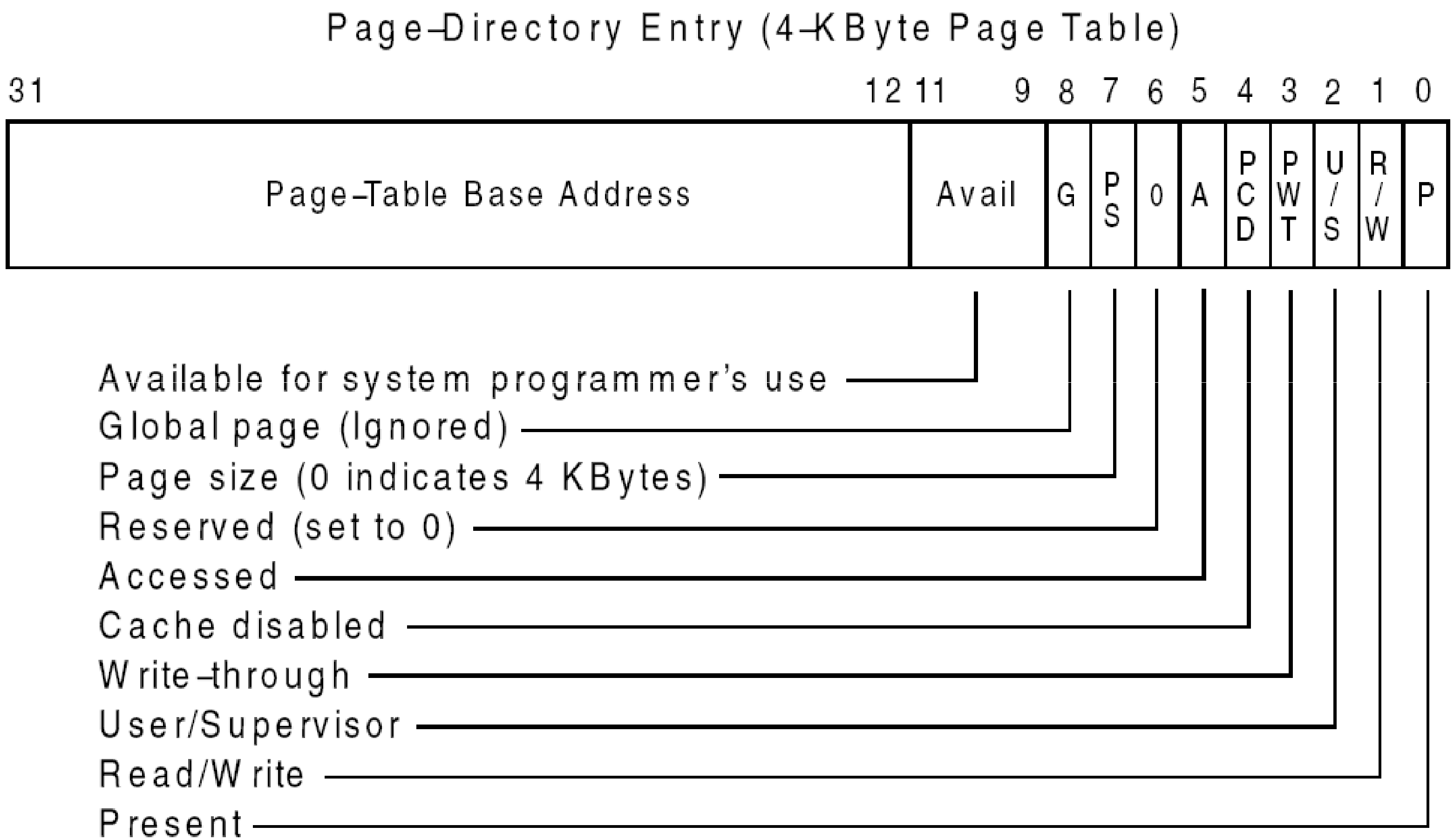


$1024 \text{ PDE} \times 1024 \text{ PTE} = 2^{20} \text{ Pages}$

*32 bits aligned onto a 4–KByte boundary

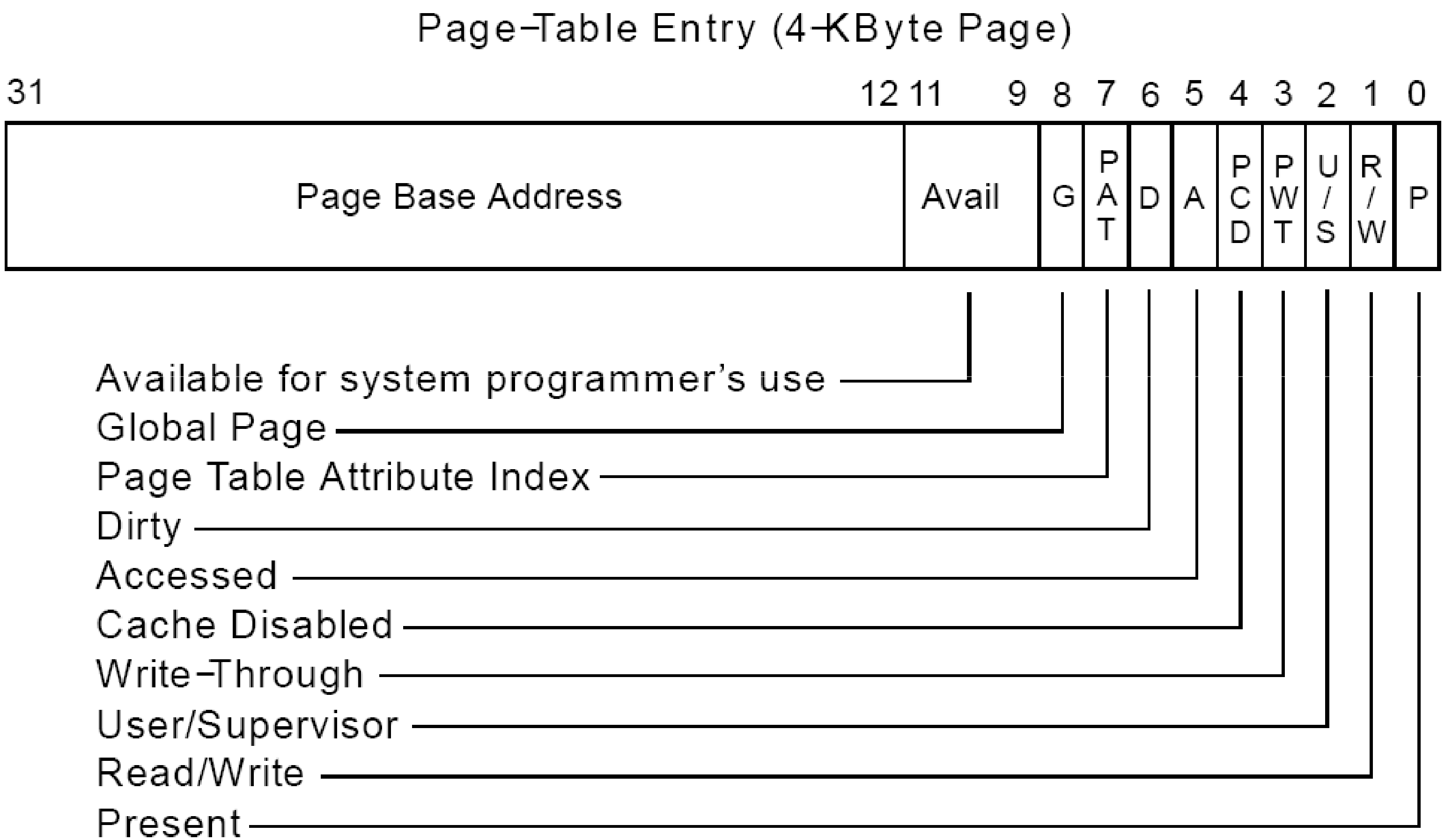
Taken from David Mazieres's Slides from Fall '07

x86 page directory entry



Taken from David Mazieres's Slides from Fall '07

x86 page table entry



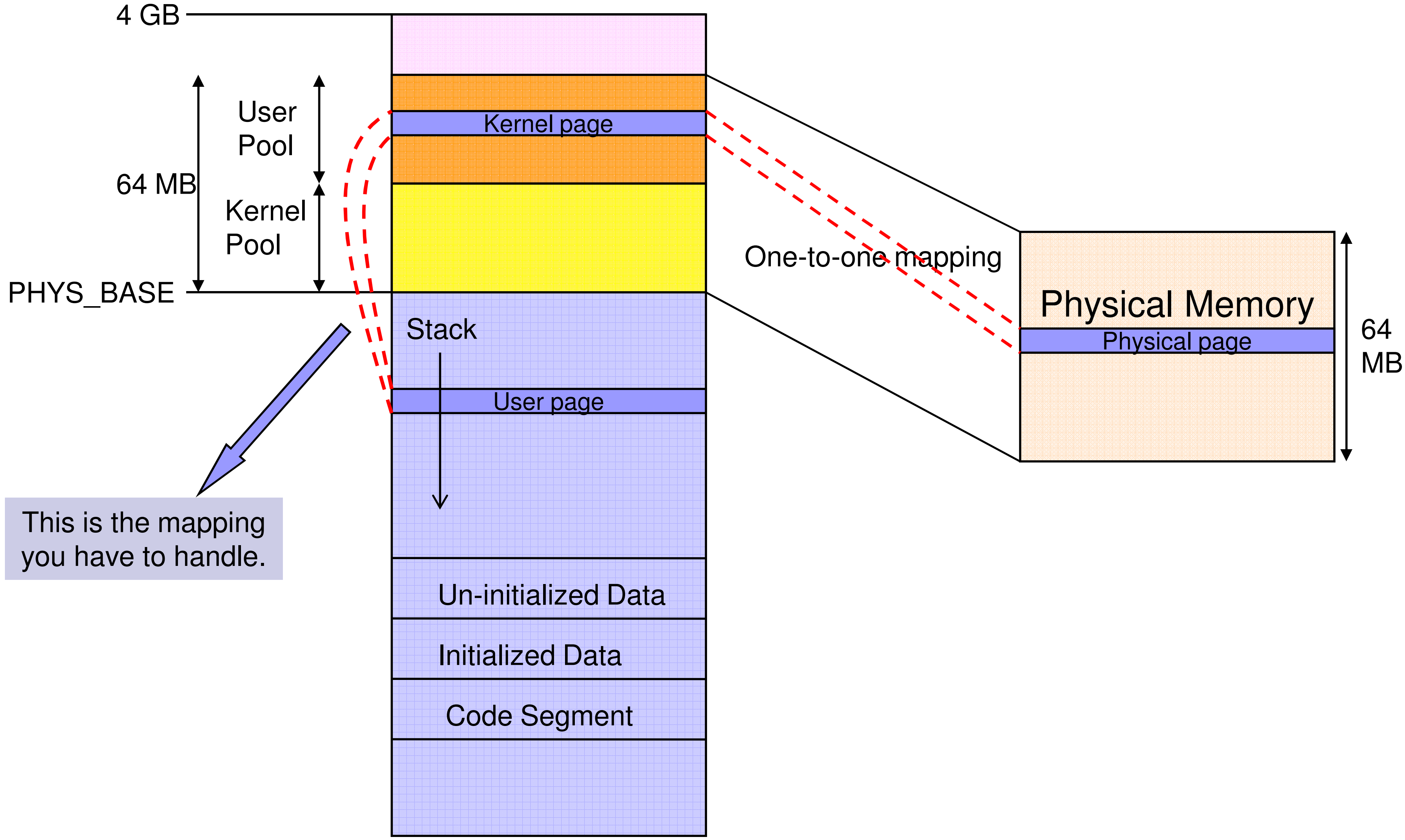
Taken from David Mazieres's Slides from Fall '07

Paging

- Users think that they have the whole memory space.
  - Let them think that way.
  - Load in their stuff only when they need it.
- Not enough space? Remove others' stuff.
  - Which page should we remove?
  - Where does the removed page go?

Manage a "Frame Table" and a "Swap Table" for that…

Page Mapping in Pintos



Page Table (1)

- Original in pintos
  - Pintos base implementation already creates basic page directory & page table structure mappings. (Look at paging_init( ) in init.c)

  - From upage to frame / kpage

    e.g. user wants a new upage at *vaddr*:
    - palloc_get_page(PAL_USER) returns a *kpage*
    - Register *vaddr* ⇔ *kpage* with pagedir_set_page()
    - User accesses the mem space at *kpage* via *vaddr*

  - Has this page been accessed/written to?
    - read => pagedir_is_accessed() == true
    - written => pagedir_is_dirty() == true

  - pagedir.c, Ref manual A.6, A.7

Page Table (2)

- Now with paging
  - upage might not be in physical memory
    - How do we know if it is or not?
    - If not, then where is the user page?
  - Supplemental Page Table
    - Data structure ? (hash, list ??)
    - Who uses this table?
      1. Page fault handler
      2. Process termination handler

Page Faults

- Previously
  - After every context switch, each process installed its own page table into the machine which contained all valid virtual to physical address space mappings.
  - In this scheme, a page fault only occurred when the process accessed an invalid virtual address.
- Now
  - A page fault is no longer necessarily an error, since it might only indicate that the page must be brought in from a disk file or from swap
  - Now, virtual address to physical address mappings are only done as and when needed.

Page Fault!

- What's going on?
  - What's the faulting virtual addr? Is it valid?
  - Which page contains this addr?
  - Is the data in swap or filesys? Where exactly?
    - If there is no space to bring in the page from swap then evict a page.
  - Or, do we need to grow the stack?
  - If we obtain a new frame, don't forget to register it.
    - We need to call pagedir_set_page() to create the vaddr ⇔ kpage mapping.

How to Handle a Page Fault

- In page fault handler, you start with fault_addr, the virtual address the user faulted on.

- Find which virtual page it's in, (using pg_round_down( ) in vaddr.h)

- Check Supp Page Table to see if the faulting page is in it.
  - If not, is it an illegal access or attempt to grow stack ?

- Copy page in either from fs or swap (you'll need to track where it is somehow)
  - If there's no free frame in memory to stick it, you'll have to evict a page.

Frames and Eviction

- You need to keep track of all the possible places to put pages in user memory, i.e., frames.
  - Frame table to track which physical frames are **occupied** and by whom.
  - Palloc( ) with USER_POOL to get **available** frames

- If no frame is free, must evict from the frame table using clock algorithm. (use access/dirty bits of PTE)

- Evicted page sent to disk…

Swap Disk

- You may use the disk on interface hd1:1 as the swap disk (see devices/disk.h)

- From vm/build
  - *pintos-mkdisk swap.dsk n*, to create an *n* MB swap disk named swap.dsk
  - Alternatively, create a temporary n-MB swap disk for a single run with *--swap-disk=n*.

- Disk interface easy to use, for example:

```
struct disk* swap_disk = disk_get(1,1);
disk_read(swap_disk,sector,buffer);
disk_write(swap_disk,sector,buffer);
```

- Maintain free slots in swap disk. Data structure needed.

Project Requirements

- Page Table Management
  - Page fault handling
  - Virtual to physical mapping
- Paging to and from (swap) disk
  - Implement eviction policies – some LRU approximation
- Lazy Loading of Executables
- Stack Growth
- Memory Mapped Files

(Easy extensions once have paging infrastructure)

More at Page Fault

- Lazy Loading of Exec
  - Only bring in pages of a program as needed.
  - Instead of using the swap disk as a backing store for the executable, you should use it's file location, since it never changes
  - Should be easy extension of previous work: just treat this like a page fault, but read from the file location on disk, instead of the swap disk.
- Stack Growth
  - Page faults on an address that "appears" to be a stack access, allocate another stack page
    - How to you tell if it is a stack access?
  - First stack page can still be loaded at process load time (in order to get arguments, etc.)

Memory Mapped Files

- Example (of a user program)
  - Map a file called foo into your address space at address 0x10000000

```
void *addr = (void *)0x10000000;
int fd = open("foo");
mapid_t map = mmap(fd, addr);
addr[0] = 'b';
write(addr, 64, STDOUT_FILENO)
```

- The entire file is mapped into consecutive virtual pages starting at *addr*.

- Make sure addr not yet mapped, no overlap

To pass more tests…

- Synchronization
  - Paging Parallelism
    - Handle multiple page faults at the same time
    - Synchronize the disk
- Resource Deallocation
  - Free allocated resources on termination
    - Pages
    - Locks
    - Your various tables
    - Others

Useful Functions

- uintptr_t **pd_no** (const void **va*)
- uintptr_t **pt_no** (const void **va*)
- unsigned **pg_ofs** (const void **va*)
- void ***pg_round_down** (const void **va*)
- void ***pg_round_up** (const void **va*)
- bool **pagedir_is_dirty** (uint32_t **pd*, const void **vpage*)
- bool **pagedir_is_accessed** (uint32_t **pd*, const void **vpage*)
- void **pagedir_set_dirty** (uint32_t **pd*, const void **vpage*, bool *value*)
- void **pagedir_set_accessed** (uint32_t **pd*, const void **vpage*, bool *value*)
- What are they for?
  - Read Ref manual A5 – A8

Suggested order of implementation.

1. Frame table: change process.c to use your frame table allocator.
    1. Layer of abstraction on top of palloc_get_page().
    2. Don’t do swapping yet. Fail when you run out of frames.
2. Supplemental page table and page fault handler: change process.c to record necessary info in table when loading executable and setting up its stack.
    1. Add loading of code and data segments in page fault handler.
    2. For now only consider valid accesses.
    3. You should now pass all proj. 2 functionality tests, and only some of robustness tests.
3. Next implement eviction, etc.
    1. Think about things like synchronization, aliasing, and eviction algorithm.