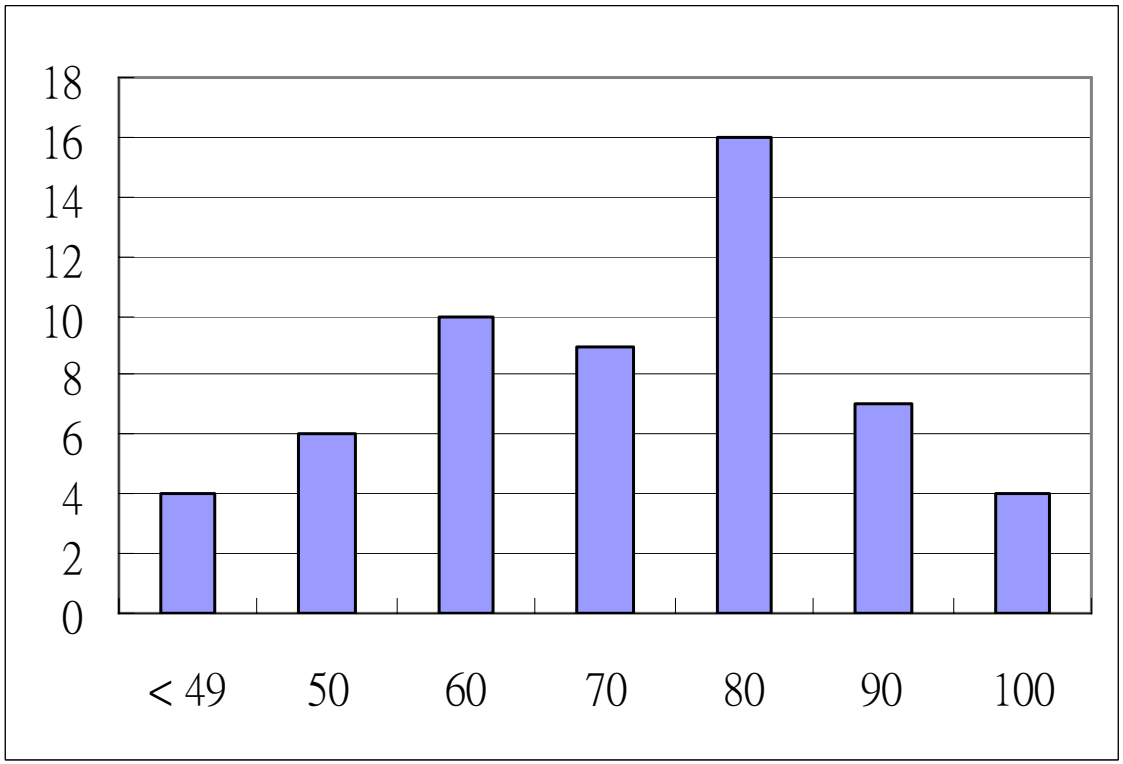
Midterm Statistics



Average: 75

Median: 77

Std Dev:18

Maximum: 100

CS 140 Project 3
Virtual Memory

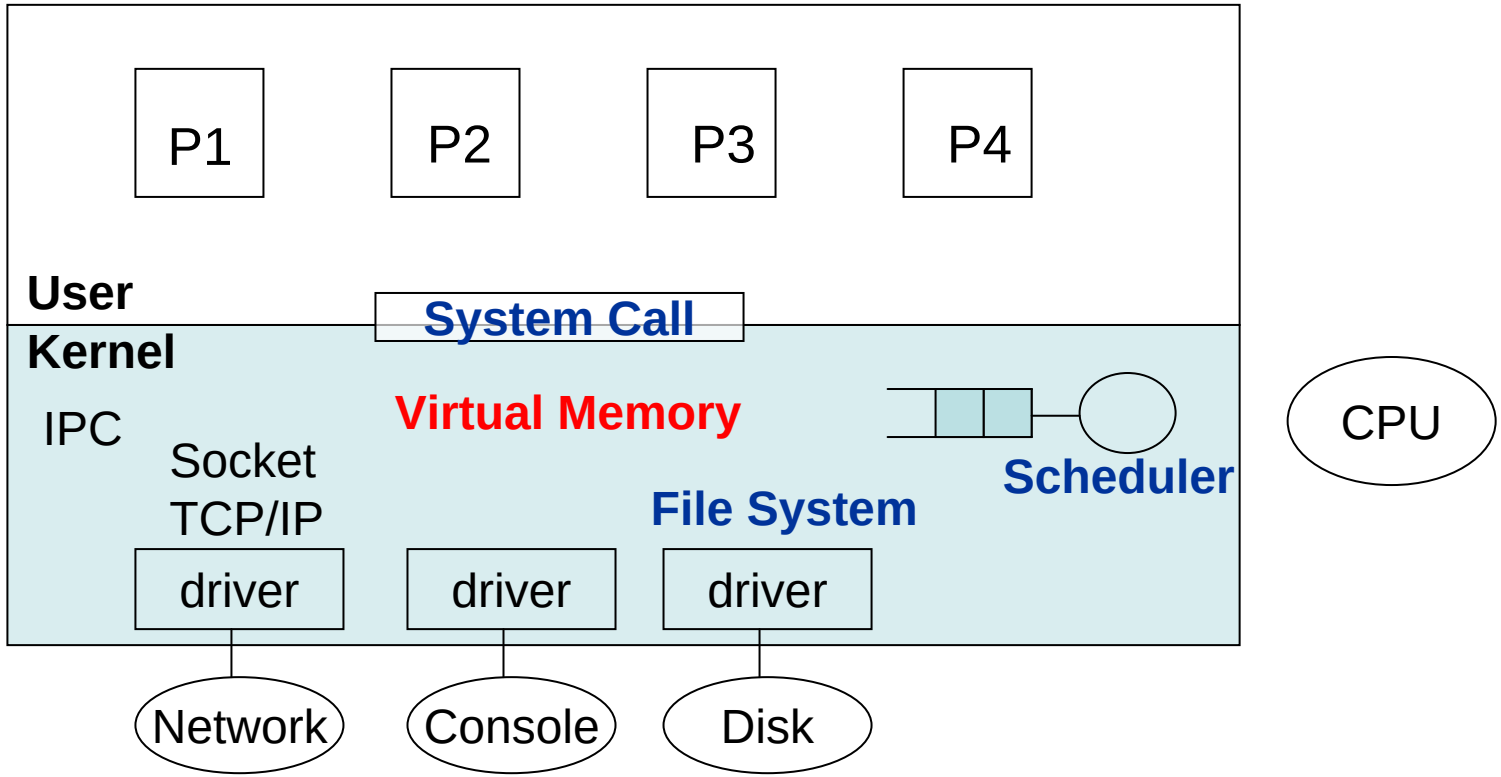
Chia-Hui Tai

CS140 Autumn 07-08
Stanford University

Some slides are adapted from Ben Sapp 2007

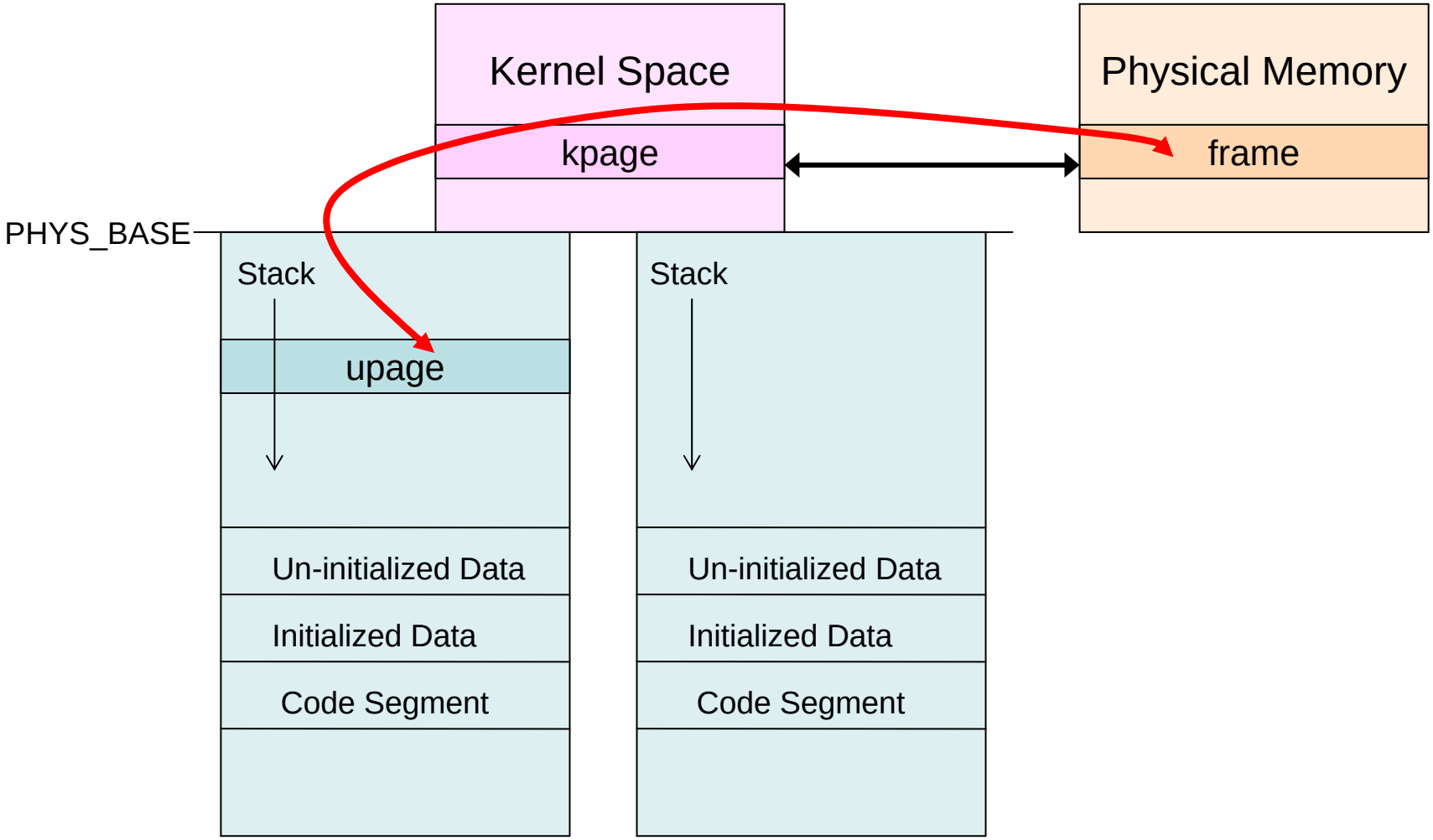
Overview

- Typical OS structure



Adopted from Lecture Notes L1 p.14

Virtual Memory



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'
  - Which one to remove?
  - Where does the removed one go?

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

Page Table (1)

- Original in pintos
  - 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*
  - Is this page accessed before?
    - 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
    - Who uses this table?
      - Page fault handler
      - Process termination

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?
  - No space to bring in the page from swap?
    - Evict others’ page! Which?
  - Or, a request for stack growth?
  - If obtain a new frame, don’t forget to register

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?

Project Requirement

- Page Table Management
  - Page fault handling
  - virtual to physical mapping
- Paging to and from (swap) disk
  - implement eviction policies

- 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 program pages when needed
  - Any benefit?
  - On eviction?
- 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

Questions?