

CSC2/452 Computer Organization Using Virtual Memory

Sreepathi Pai

URCS

November 6, 2019

Outline

Administrivia

Recap

Physical Memory Management

File I/O using mmap

Manipulating pages

Outline

Administrivia

Recap

Physical Memory Management

File I/O using mmap

Manipulating pages

Administrivia

- ▶ Assignment #3 is due Nov 8, 2019 at 7PM
- ▶ Homework #5 is due Monday, Nov 11
- ▶ Assignment #4 will be out Monday, Nov 11
- ▶ Mid-term grades are out
 - ▶ HW+A1+A2+Midterm is 42% of grade
 - ▶ 58% still due!

Outline

Administrivia

Recap

Physical Memory Management

File I/O using mmap

Manipulating pages

Virtual Memory

- ▶ Virtual address space
- ▶ Physical address space
- ▶ All loads/stores use virtual addresses
- ▶ MMU translates virtual addresses to physical addresses
 - ▶ Uses page tables
 - ▶ On 64-bit x86, 4 levels of page tables
- ▶ Page tables allow pages to not be present in memory!
 - ▶ Loads/stores to such pages trigger page faults
 - ▶ OS can handle these page faults

Outline

Administrivia

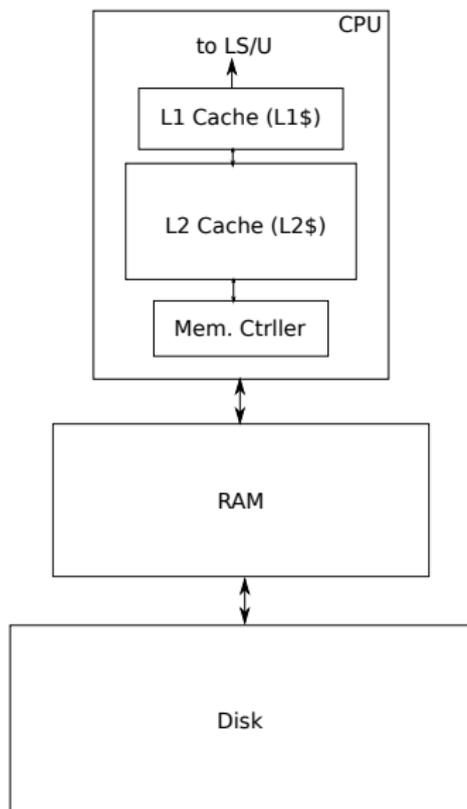
Recap

Physical Memory Management

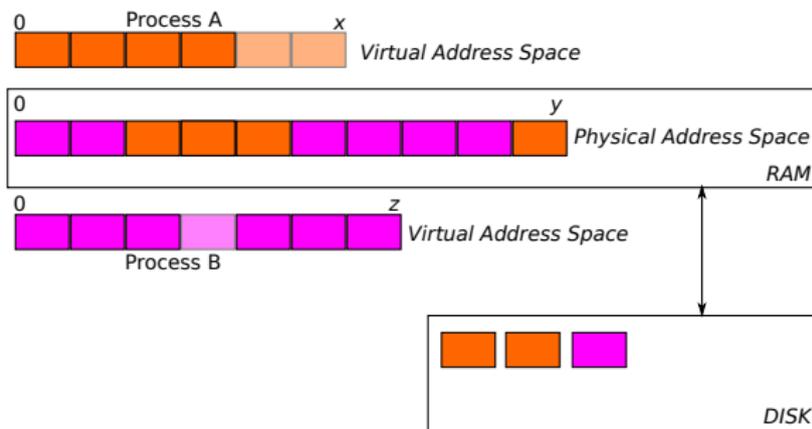
File I/O using mmap

Manipulating pages

The Revised Memory Hierarchy

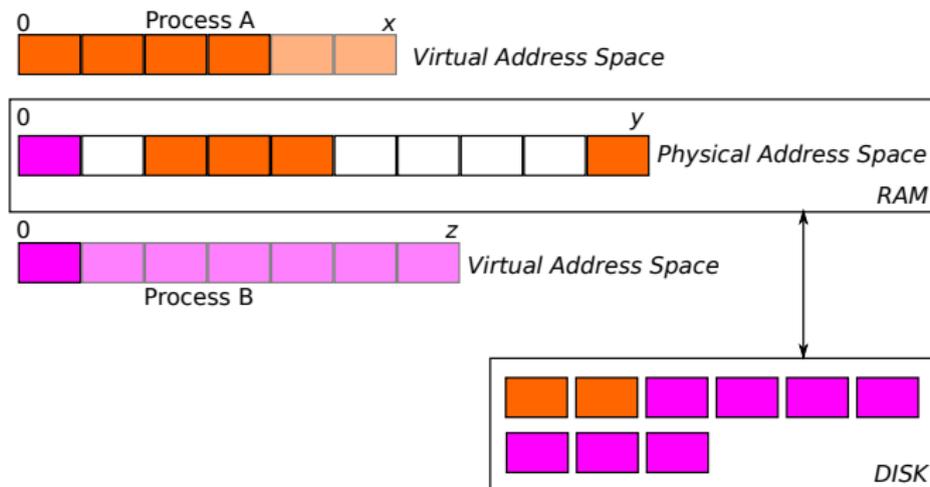


Programs and Virtual Memory



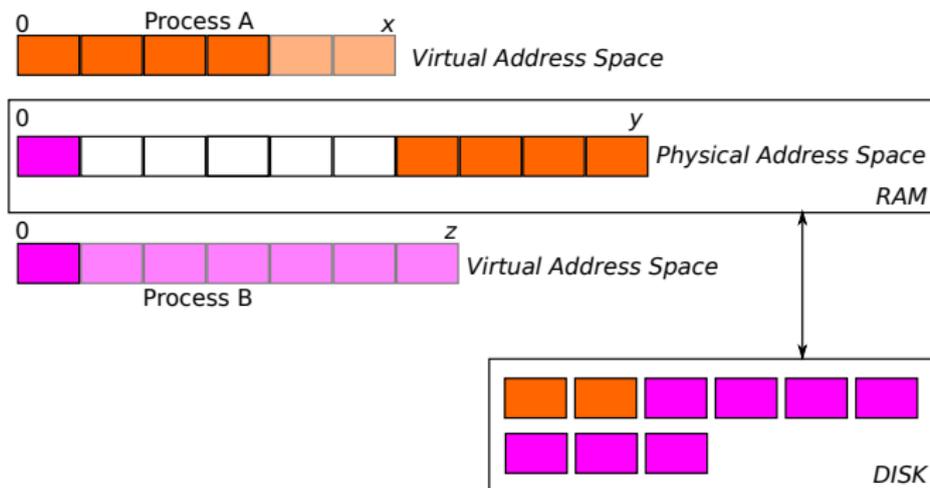
- ▶ Each block corresponds to a page
 - ▶ Process A has 6 pages, Process B has 7 pages
- ▶ Physical memory can accommodate 10 pages
- ▶ Disk can accommodate as many pages as it has space
 - ▶ Depends on OS
- ▶ Shaded blocks indicate pages swapped out

Loading Process B



- ▶ Assume Process B is being loaded for the first time
 - ▶ Or maybe it was completely swapped out
- ▶ Only one page has been loaded
- ▶ From the perspective of the memory hierarchy, all data is initially on disk

Defragmenting Virtual Memory



- ▶ This slide demonstrates that you can change virtual to physical mappings
 - ▶ Here, for physical pages belonging to process A have been moved
- ▶ “Defragmentation” allows larger, physically contiguous chunks to be assigned to the same process
 - ▶ Can improve performance and allocator behaviour

Loading Process B

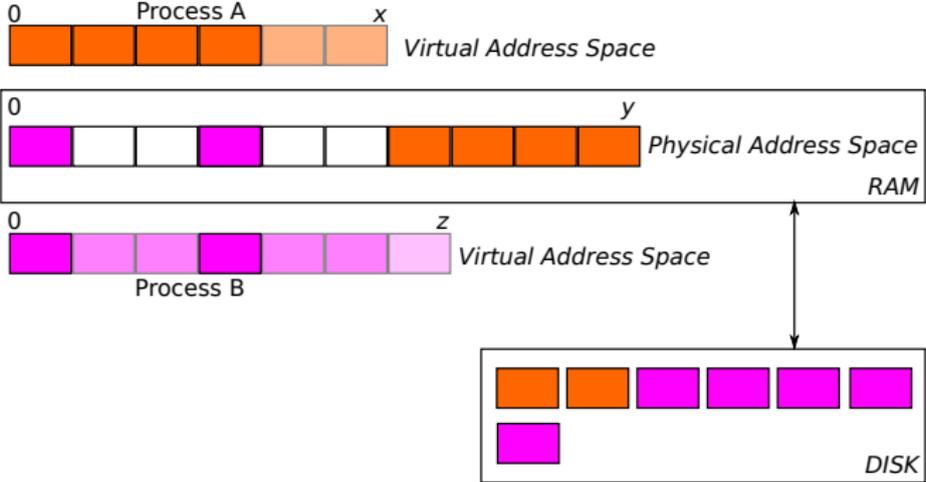
Should we load all pages from disk to memory at once?

- ▶ Will Process B access all the pages?
 - ▶ Do you use all the functionality of most programs in a single run?

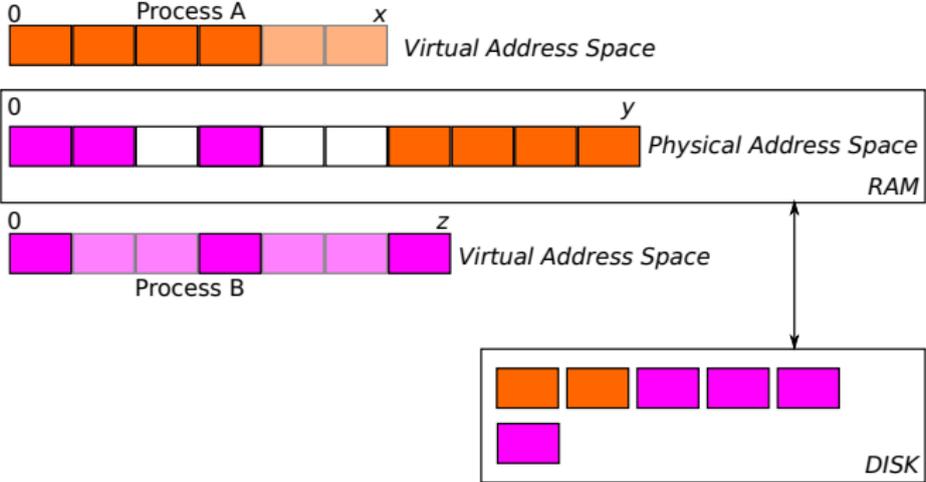
Demand Paging

- ▶ OS simply marks pages as belonging to process B
 - ▶ but marks them all as not present
- ▶ When Process B accesses a page for the first time, it triggers a fault
 - ▶ OS then loads the page from disk

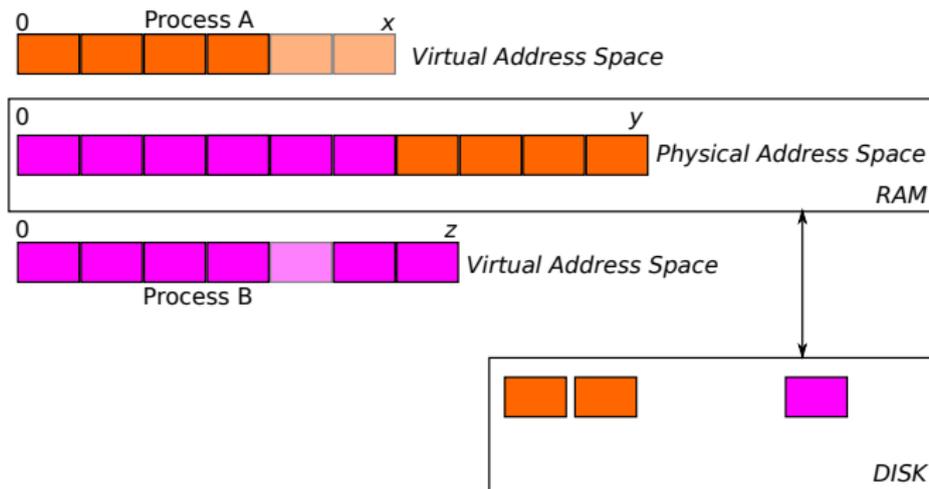
Paging Process B in (2 pages)



Paging Process B in (3 pages)

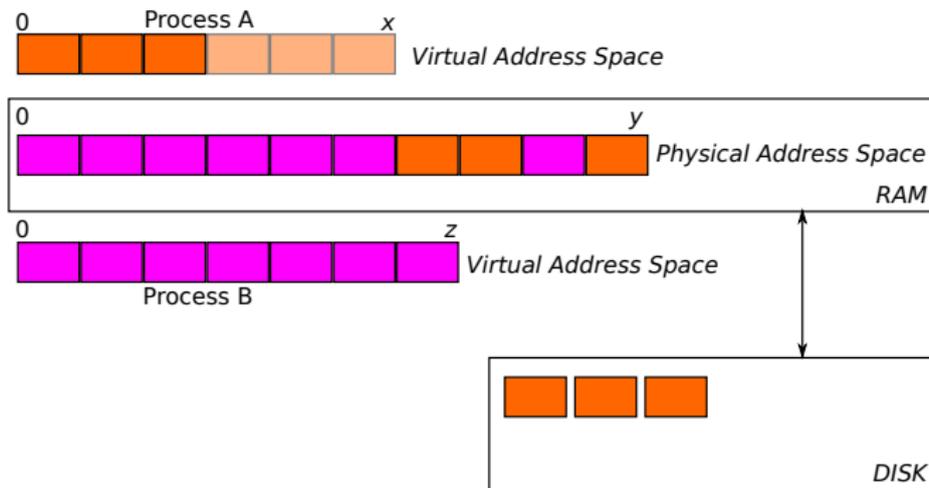


Paging Process B in (6 pages)



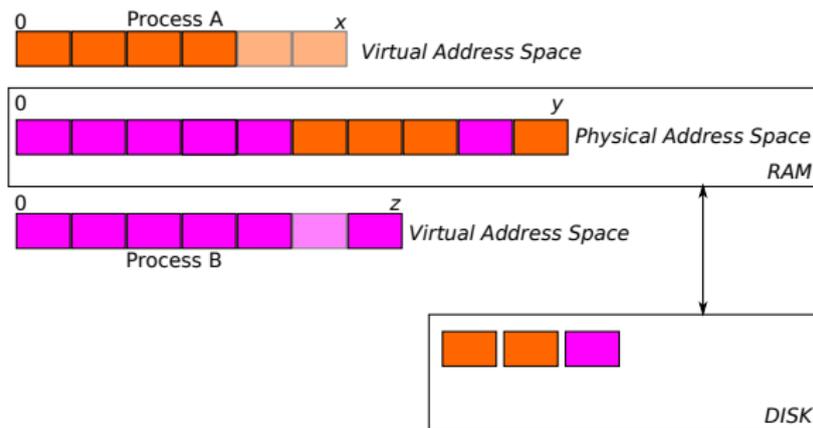
- ▶ In this case, B is accessing all pages
- ▶ No more free physical pages
 - ▶ Which page should we kick out?

Which page to throw out?



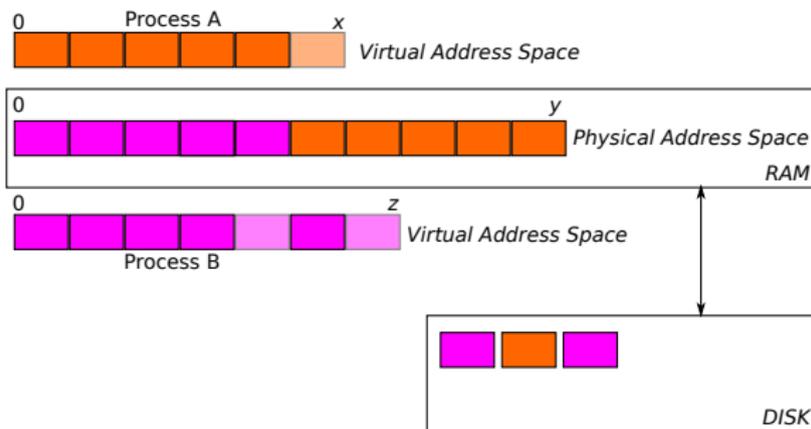
- ▶ Same problem as cache replacement
- ▶ Similar solutions
 - ▶ LRU is often used

Context switch back to Process A



- ▶ Process A is now active and brought the recently kicked out page back in
 - ▶ Bad luck on replacement!
- ▶ We throw out a page from Process B

Context switch again



- ▶ Process B is now active and brought its recently kicked out page back in
 - ▶ More bad luck on replacement!
- ▶ We throw out a page from Process A
- ▶ Ad infinitum?

(Disk/VM) Thrashing

- ▶ When physical memory is full and
- ▶ OS spends more time swapping pages in and out than actually running programs
 - ▶ Replacement policy is not picking the right pages
- ▶ Happens because of oversubscription
 - ▶ More virtual pages than physical RAM can accommodate

Outline

Administrivia

Recap

Physical Memory Management

File I/O using mmap

Manipulating pages

Traditional File I/O

```
f = fopen(argv[1], "r");
if(f) {
    while(!feof(f)) {
        c = fgetc(f);
        if(c == EOF) break;

        if(c >= 'A' && c <= 'Z') {
            counter[c - 'A']++;
        }
    }
}
```

- ▶ Open file using `fopen`
- ▶ Read data using `fgetc` (or `fread`, `fscanf`, etc.)

Memory Mapped I/O for Files

- ▶ Recall that disk is part of the virtual memory system
- ▶ Files are on disk
- ▶ Can we use the virtual memory subsystem to read/write files?

Functionality needed

- ▶ Ask OS (i.e. virtual memory subsystem) to read a file into a bunch of pages
- ▶ Use pointers to read/write the data in the pages directly
- ▶ Save changed data back to disk

The `mmap` function

```
int fd = fileno(f);
struct stat st;
char *data;

if(fstat(fd, &st) == 0) {
    data = mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if(data) {
        int i;
        for(i = 0; i < st.st_size; i++) {
            if(*data >= 'A' && *data <= 'Z') {
                counter[*data - 'A']++;
            }
            data++;
        }
    }
}
```

- ▶ The `mmap` function allows you to do map files into memory
 - ▶ Also supports other functionality such as allocation
- ▶ Use `fopen` as usual, but convert `FILE` to a file descriptor understood by OS (`fileno`)
 - ▶ Or use `open` to get a file descriptor in the first place
- ▶ Use `fstat` to obtain the size of the file
- ▶ Call `mmap` to map data from the file into memory

Examining `mmap`

```
data = mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);
```

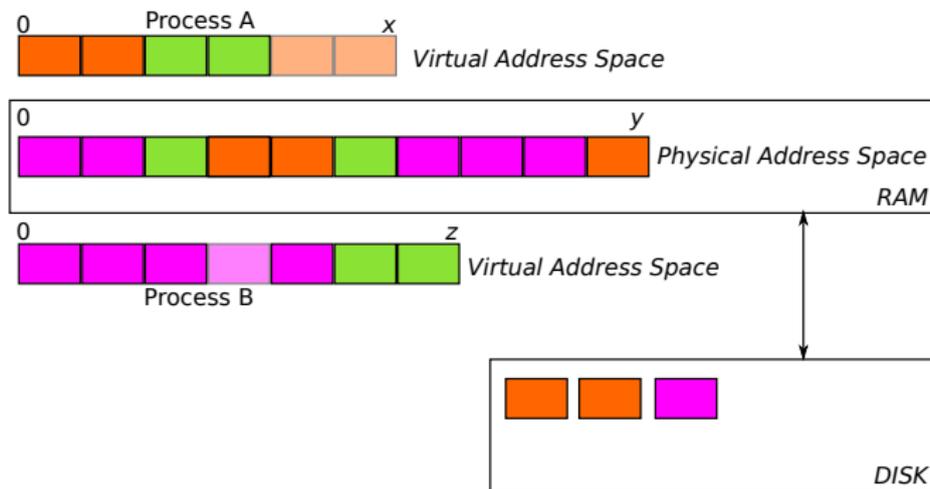
- ▶ First argument (`NULL`), which virtual address to map file to
 - ▶ `NULL` specifies any address is okay
- ▶ Second argument (`st.st_size`) is size of file
 - ▶ This will be rounded up to page size
- ▶ Third argument (`PROT_READ`) opens the data in read-only mode
 - ▶ Attempting to write will cause a page fault
- ▶ Fourth argument (`MAP_SHARED`) makes all writes visible immediately to other processes who have opened this file using `mmap`
 - ▶ In this case, writes by other processes will be visible to this program
- ▶ Fifth argument (`fd`) gives the file descriptor
- ▶ Sixth argument (`0`) gives the offset from which the data must be loaded
 - ▶ `0` is the first byte of the file

Private copies vs shared copies

```
data = mmap(NULL, st.st_size, PROT_READ | PROT_WRITE,  
            MAP_PRIVATE, fd, 0);
```

- ▶ A MAP_PRIVATE flag maps a file privately
- ▶ Changes are not visible to other processes
 - ▶ And are also not written through to the file being mapped

How shared mmap works



- ▶ Green blocks are shared pages
 - ▶ Same file mmap'ed by two different processes
- ▶ Same physical pages
- ▶ Different virtual addresses per process
 - ▶ But could be the same

Unmapping files

```
munmap(data, st.st_size);
```

- ▶ `munmap` removes the page mapping
 - ▶ Here, all pages in addresses from `data` to `data + st.st_size`
 - ▶ Future accesses to the page will cause segfaults
 - ▶ All changes are written to disk
- ▶ You can remove individual page from the mapping
 - ▶ Just change the pointer and the size
- ▶ All mapped regions are automatically unmapped when the process ends

Writing to pages

- ▶ When you `munmap`, all changed pages must be flushed to disk
 - ▶ OS can also periodically flush changes to disk
- ▶ This makes writes to `mmap`'ed files visible to processes using traditional file-based I/O
 - ▶ Note, writes are always immediately visible to other processes using `mmap`d I/O
- ▶ Two issues:
 - ▶ How does the OS track changed pages?
 - ▶ Can we control flushing to disk?

Dirty pages

- ▶ A page with changed data is called a “dirty” page
- ▶ This page must be written to disk eventually
- ▶ How can the OS track dirty pages efficiently?

Mechanism

- ▶ Assume a file is mapped with `PROT_WRITE`
- ▶ When data is first mapped, its protection is set to read only
 - ▶ Regardless of `PROT_WRITE`
- ▶ On first write, page fault occurs
- ▶ OS detects the page fault, and marks page as dirty
 - ▶ Is there a bit in the PTE it could use?
 - ▶ Also enables write permission for page
 - ▶ Future writes will not cause segfaults
 - ▶ Note on x86 – the dirty bit is set by hardware, but cleared by software
- ▶ Ultimately, pages that are marked dirty need to be flushed to disk
 - ▶ Dirty bit is reset and write permission removed
- ▶ What about unchanged pages?

Flushing data to disk

```
msync(data, st.st_size, MS_SYNC);
```

- ▶ `msync` forces the OS to write changed data back to disk
 - ▶ Note only changed pages are written
- ▶ The flag `MS_SYNC` causes it to wait until all data is actually on disk

Prefetching

```
madvise(data, st.st_size, MADV_SEQUENTIAL);
```

- ▶ Provides hints to the OS on the order in which you will read/write data
- ▶ This affects OS-level prefetching of pages
- ▶ It also affects page replacement policy
- ▶ For example, `MADV_SEQUENTIAL` will:
 - ▶ prefetch pages in sequential order
 - ▶ throw away pages (since there will be no reuse)

Anonymous Pages

```
data = mmap(NULL, 1048576, PROT_READ | PROT_WRITE,  
            MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

- ▶ `MAP_ANONYMOUS` allocates pages
 - ▶ Like `malloc`
 - ▶ Actually used by `malloc` under the hood
- ▶ Here, we're allocating 1MB of memory *not* backed by a file
- ▶ Note, for anonymous mappings:
 - ▶ `fd` should be `-1`
 - ▶ `offset` should be `0`

Fixed Address Pages

```
data = mmap(0x7ffff0001000, 1048576, PROT_READ | PROT_WRITE,  
            MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

- ▶ You can provide a fixed address to mmap (both anonymous and non-anonymous)
- ▶ This allows you to control which addresses you use
 - ▶ If this call succeeds, data will contain 0x7ffff0001000
- ▶ Can fail

Outline

Administrivia

Recap

Physical Memory Management

File I/O using mmap

Manipulating pages

Change page protections

```
data = mmap(..., PROT_READ | PROT_WRITE, ...);  
...  
mprotect(data, st.st_size, PROT_READ)
```

- ▶ `mprotect` allows you to change permissions for pages after you've loaded them.
 - ▶ Here, all pages in address range `data` to `data + st.st_size`
- ▶ You can:
 - ▶ remove all permissions (`PROT_NONE`)
 - ▶ change some pages to executable (`PROT_EXEC`)

Locking pages

What if you want to prevent pages from being swapped out?

- ▶ Maybe you want good performance?
- ▶ Maybe the page contains secret data that you don't want written to disk?

The `mlock/munlock` functions

```
mlock(data, 4096);
```

- ▶ If successful, `mlock` prevents the pages in the address range `data` to `data + 4096`.
 - ▶ Why could this fail?

References and Acknowledgements

- ▶ The GNU libc Manual
 - ▶ Portions of Chapter 3 (3.1: Process Memory Concepts)
 - ▶ Portions of Chapter 13 (13.8: Memory-mapped I/O)
 - ▶ See References on the course website for the manual
- ▶ Chapter 9 of the textbook