Virtual Memory: Anatomy of a memory reference

Questions Answered in this Lecture:
- How do we get illusion of the full address space?
- How do we swap out pages to disk efficiently?
- Which pages do we swap out?
- What is thrashing?
- What does an actual memory reference look like?
- How does the OS detect NULL pointer derefs?
Announcements

• P2a due Thursday! No extensions this time!
• We won’t be making exceptions for bad git commits/stashes/etc. moving forward
Virtualizing Memory

• Remember, we’re giving the illusion of an address space
• This is a great abstraction because we provide that there are bytes named by addresses...
• But from where those bytes come is hidden to the user
• Recall:
  • phys addr space: bytes can come from RAM, ROM, memory controller, PCI device, SCSI device, etc.
Where do the bytes come from?

• **Default case**: a physical page of RAM
• What if we’re running low on RAM?
Use Disk!

page directory

page table

disk

stashed physical page

“paged out”

“swapped”
What does this mean?

• We need some way to tie PTEs to disk (we’ll come back to this)
• Can’t just use a physical address!
• Need to integrate paging code with block (disk) driver
Swapping

• Now if we’re running low on memory, we pick a victim process, and throw some of its pages out to disk
• We *stash a pointer to the disk blocks*, make a record of it
• Then invalidate the old PTE
this page can now be allocated for something else  
e.g. kernel data, another process, etc.
What if we need the page?

• Process tries to access the old VA again. What happens?
page directory

page table

physical page

free physical page

disk

stashed physical page

“paged out”

“swapped”
This free physical page doesn’t have to be the same one as before!

What happens if we’re really low on memory? E.g. there is no free physical page?
Thrashing

• When there are no free pages, we’re constantly swapping out to disk
• E.g., take a page from one process, give it to another, and so on
• Very bad place to be. Cache won’t help here.
• Buy more RAM!
Page replacement (policy)

• Which page to replace?
• FIFO (oldest mapped page is the target)
• LRU (least recently used. how to keep track?)
• Random
Disk is slow

• Spinning disks especially are very slow!
• We want to minimize how much we go off to disk
• What do we do?
The *buffer cache*

![Diagram of buffer cache and disk]

- **RAM**
- **buffer cache**
- **hot disk blocks**
- **disk**
swap_in (block_no) {
    blk = block_lookup(block_no, buffer_cache)
    if (blk == NULL) { // MISS
        blk = disk_read(block_no);
    }
    page = page_alloc();
    copy(page, blk);
    return page;
}
Anatomy of a memory reference

Or, how does mapping work?
include <stdio.h>
#include <stdlib.h>

int main (int argc, char ** argv) {
    unsigned long * a_ptr = (unsigned long*)0xdeadbeefULL;
    *a_ptr = 0x1234;
    return 0;
}
This is our memory reference
why?
how?
subutai.cs.iit.edu → 450 cat good_ptr.c

#include <stdlib.h>
#include <stdio.h>

#define PAGE_SIZE 4096

int main (int argc, char ** argv) {

    unsigned long * good_ptr = (unsigned long*)malloc(PAGE_SIZE);
    *good_ptr = 0x1234;

    printf("%lu\n", *good_ptr);

    return 0;
}
```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char ** argv) {
    unsigned long * clever = (unsigned long*) 0x405000ULL;
    void * a = malloc(10); // make sure we have a heap

    *clever = 0x1234;
    printf("Gotcha!\n");
    return 0;
}
```
subutai.cs.iit.edu → 450 ./evil
Gotcha!

Hale | CS 450
```
subutai.cs.iit.edu -> 450  cat /proc/11152/maps

00400000-00401000 r--p 00000000 08:11 49285371  /home-remote/kyle/450/evil
00401000-00402000 r-xp 00001000 08:11 49285371  /home-remote/kyle/450/evil
00402000-00403000 r--p 00002000 08:11 49285371  /home-remote/kyle/450/evil
00403000-00404000 r--p 00002000 08:11 49285371  /home-remote/kyle/450/evil
00404000-00405000 r-wp 00003000 08:11 49285371  /home-remote/kyle/450/evil
00405000-00426000 r-wp 00000000 00:00 0 [heap]
7fffffffde5000-7fffffff7e07000 r--p 00000000 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7e0700-7fffffff7f54000 r-xp 00002200 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7f5400-7fffffff7fa0000 r--p 0016f000 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7fa0000-7fffffff7fa1000 ---p 001bb000 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7fa1000-7fffffff7fa5000 r--p 001bb000 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7fa5000-7fffffff7fa7000 rw-p 001bf000 fd:00 8395967  /usr/lib64/libc-2.28.so
7fffffff7fa7000-7fffffff7fad000 rw-p 00000000 00:00 0 [vvar]
7fffffff7fcd000-7fffffff7fd0000 r--p 00000000 00:00 0 [vvar]
7fffffff7fd0000-7fffffff7fd2000 r-xp 00000000 00:00 0 [vvar]
7fffffff7fd2000-7fffffff7fd3000 r--p 00000000 fd:00 8395883  /usr/lib64/ld-2.28.so
7fffffff7fd3000-7fffffff7ff3000 r-xp 00001000 fd:00 8395883  /usr/lib64/ld-2.28.so
7fffffff7ff3000-7fffffff7ffbb000 r--p 00021000 fd:00 8395883  /usr/lib64/ld-2.28.so
7fffffff7ffbb000-7fffffff7ffdc000 r--p 00029000 fd:00 8395883  /usr/lib64/ld-2.28.so
7fffffff7ffdc000-7fffffff7ffe0000 rw-p 0002a000 fd:00 8395883  /usr/lib64/ld-2.28.so
7fffffff7ffe000-7fffffff7fff0000 rw-p 00000000 00:00 0 [stack]
7fffffffde000-7fffffff7fffffff0000 r-wp 00000000 00:00 0 [vsyscall]

```
Representing address space *regions*

```c
struct mem_region {
    unsigned long start;
    unsigned long len;
    int type;
    int present;
    int paged_out;
    ...
}
```
Starting a process

• Kernel constructs memory regions for initial regions (*stack, heap, kernel*)
• All other portions of the address space are *unmapped*
• New regions must be created *by request* from userspace (*mmap()*)
What happens on a page fault?

• Lookup faulting address in the *region map*
  • Some kind of search data structure: hash table, binary search tree, linked list, etc.

• Hit? Something special (like swapped page) is going on

• Miss? This is an address that isn’t mapped. SEGFAULT
char *map = mmap(0, textsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

how does this work?
Summary

- Disk allows us to better support illusion of full address space (swapping)
- Kernel backs address space regions with metadata (mechanism)
- Page faults drive the whole thing