EECS 482
Introduction to Operating Systems

Winter 2018

Harsha V. Madhyastha

Recap: Page Replacement

- LRU ≈ OPT for realistic workloads
  - Leverage temporal locality to reduce page faults
- Clock replacement is practical approx. of LRU

- OS can maintain resident, ref, and dirty bits
- Need MMU to only check protection bits
- Trigger faults only when bit changes from 0 to 1

Storing Page Tables

- Two options:
  - In physical memory
  - In kernel's virtual address space
- Difference: Is PTBR a physical or virtual addr?
- Pros and cons?
- Project 3 uses second option
  - Kernel's address space managed by infrastructure

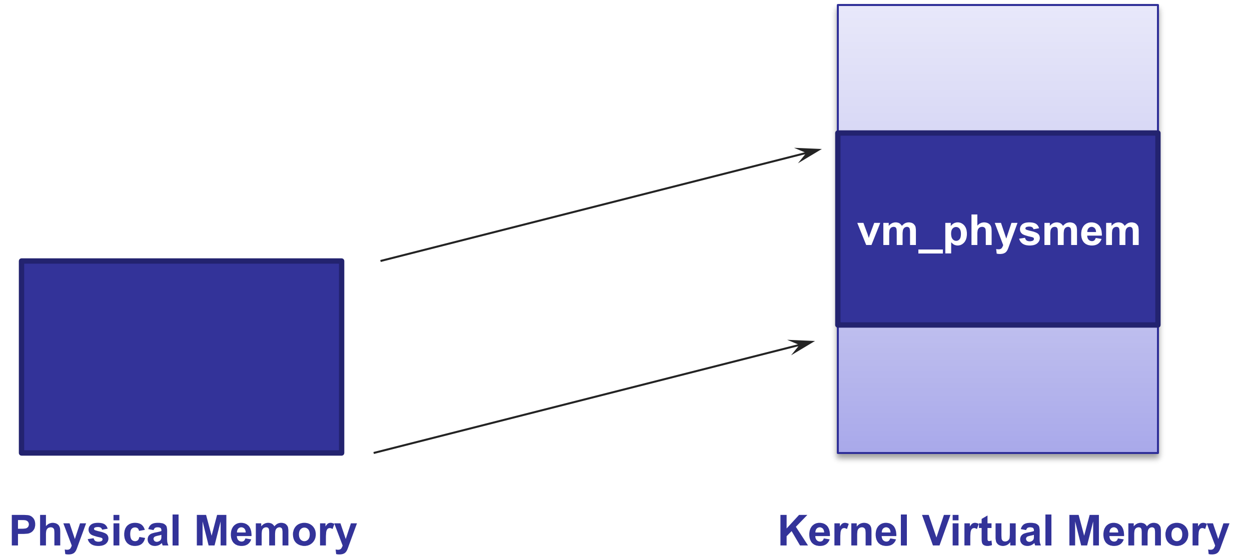
Kernel vs. user address spaces

- Can you evict the kernel’s virtual pages?
  - Yes, except code for handling paging in/out is pinned

- How can kernel access specific physical memory addresses (e.g., to write to page table)?
  - Kernel can issue untranslated address (bypass MMU)
  - Kernel can map physical memory into a portion of its address space (e.g., vm_physmem in Project 3)

Accessing physical memory

- How does kernel access physical memory?
  - Could map physical memory 1-to-1 into window in virtual address space
  - vm_physmem[n]: n^{th} byte of physical memory



Kernel vs. user mode

- How are we protecting a process's address space from other processes?
  - Page table/MMU dynamic translation
  - Must ensure only kernel can modify translation data
- How does CPU know kernel is running?
  - Hardware support: Mode bit

- Recap of protection:
  - Address space → Translation data → Mode bit

Kernel vs. user mode

- How are we protecting a process's address space from other processes?
  - Page table/MMU dynamic translation
  - Must ensure only kernel can modify translation data

In what mode does a root user's process run?

How can a root user reboot the machine?

- Recap of protection:
  - Address space → Translation data → Mode bit

Switching to kernel mode

- Faults and interrupts
  - Timer interrupts
  - Page faults
  - Why are these safe to transfer control to kernel?
- System calls
  - Process management: fork/exec
  - I/O: open, close, read, write
  - System management: reboot
  - …

System calls

- When you call `cin` in your C++ program:
  - `cin` calls **`read()`**, which executes assembly-language instruction **`syscall`**
  - **`syscall`** traps to kernel at pre-specified location
  - kernel's syscall handler calls kernel's `read()`
- To handle trap to kernel, hardware atomically
  - Sets mode bit to kernel
  - Saves registers, PC, SP
  - Changes SP to kernel stack
  - Changes to kernel's address space
  - Jumps to exception handler

Arguments to system calls

- Two options:
  - Store in registers
  - Store in memory (in whose address space?)

- Kernel first checks validity of arguments
  - e.g., `read(int fd, void *buf, size_t size)`
    - » Is fd valid descriptor for open file
    - » Are all addresses in [buf,buf+size) valid
    - » Are all addresses in [buf,buf+size) writable

How does kernel access user's address space?

- Kernel can manually translate a user virtual address to a physical address, then access the physical address

- Can map kernel address space into every process's address space

fffff	
.	
.	operating system
.	
80000	
7ffff	
.	
.	user process
.	
00000	

  - Trap to kernel doesn't change address spaces; it just allows computer to access both OS and user parts of that address space

Protection summary

- Safe to switch from user to kernel mode because control only transferred to certain locations
  - Where are these locations stored?
    - » Interrupt vector table

- Who can modify interrupt vector table?

- Why is it easier to control access to interrupt vector table than mode bit?

Address Space Protection

- How are address spaces protected?
  - Separation of translation data
- How is translation data protected?
  - Can update translation data only if mode bit set
- How is mode bit protected?
  - Sets/reset mode bit when transitioning from user-level to kernel-level code and back
  - Transitions limited by interrupt vector table
- Protection boils down to init process which sets up interrupt vector table when system boots up

Project 3

- Memory management using paging
  - Due March 21st
- By the end of this lecture, we will cover all the material you need to know to do the project
- Begin by drawing state machine for a virtual page
  - Focus on swap-backed pages to start

Project 3

- Incremental development critical
  - Swap-backed pages with a single process
  - File-backed pages
  - Fork

- Minimum amount of functionality to test
  - vm_init
  - vm_create (with parent process unknown)
  - vm_map (with filename = NULL)
  - vm_fault
  - Getting this combination right = 21/75

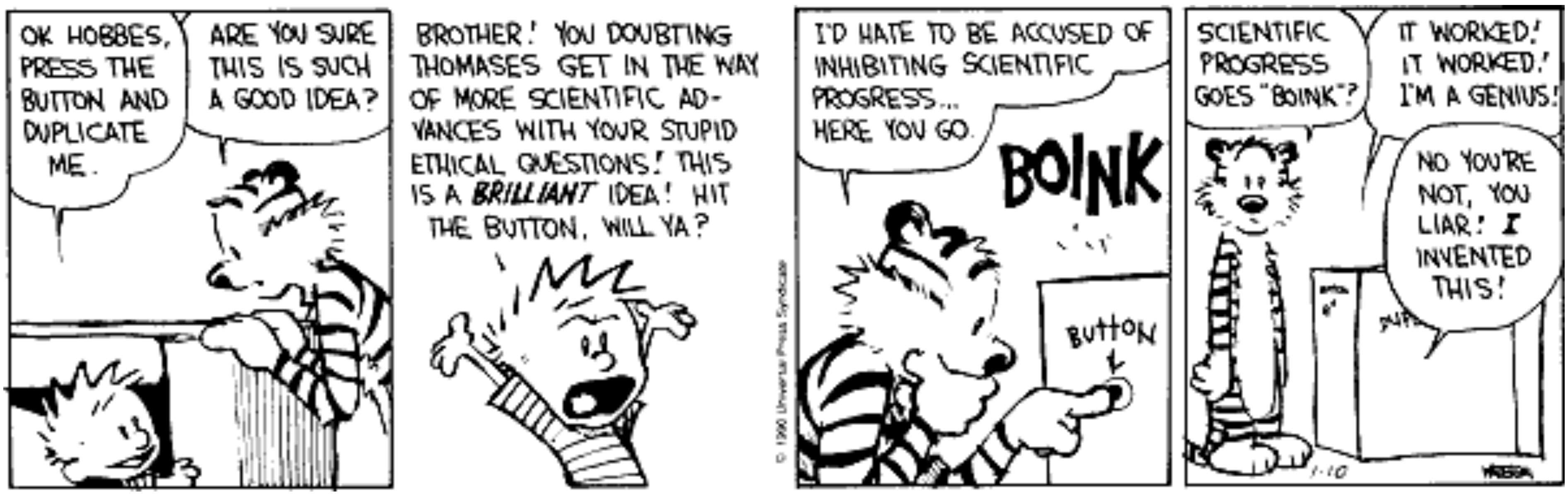
Process creation

- Steps
  - Allocate process control block
  - Initialize translation data for new address space
  - Read program image from executable into memory
  - Initialize registers
  - Set mode bit to “user”
  - Jump to start of program
- Need hardware support for last few steps
  - Similar to switching from kernel to user process after system call

Unix process creation

- System calls to start a process:
  1. Fork() creates a copy of current process
  2. Exec(program, args) replaces current address space with specified program
- Why first copy and then overwrite?
  - Windows: CreateProcess(program, args)
- Any problems with child being an **exact** clone of parent?

Cloning


OK HOBBES, PRESS THE BUTTON AND DUPLICATE ME.
ARE YOU SURE THIS IS SUCH A GOOD IDEA?
BROTHER! YOU DOUBTING THOMASES GET IN THE WAY OF MORE SCIENTIFIC ADVANCES WITH YOUR STUPID ETHICAL QUESTIONS! THIS IS A BRILLIANT IDEA! HIT THE BUTTON, WILL YA?
I'D HATE TO BE ACCUSED OF INHIBITING SCIENTIFIC PROGRESS... HERE YOU GO.
BOINK
BUTTON
SCIENTIFIC PROGRESS GOES "BOINK"?
IT WORKED! IT WORKED! I'M A GENIUS!
NO YOU'RE NOT, YOU LIAR! I INVENTED THIS!

Fork and exec

- Fork uses return code to differentiate
  - Child gets return code 0
  - Parent gets child's unique process id (pid)

```
If (fork() == 0) {
    exec ();        /* child */
} else {
    /* parent */
}
```

Implementing a shell

```
while (1) {
   print prompt
   ask user for input (cin)
   parse input //split into command and args
   fork a copy of current process (the shell prog.)
   if (child) {
      redirect output to a file/pipe, if requested
      exec new program with arguments
   } else { //parent
      wait for child to finish, or
      run child in the background and ask for
another command
   }
}
```

Subtleties in handling fork

- Buggy code from autograder:

```
if (!fork()) {
        exec(command);
}
while(child is alive) {
        if (size of child address space > max) {
                print "process took too much mem";
                kill child;
                break;
        }
}
```

- What is the race condition here?

- Go to lab section on Friday for run down on project 3
- Have a good spring break!