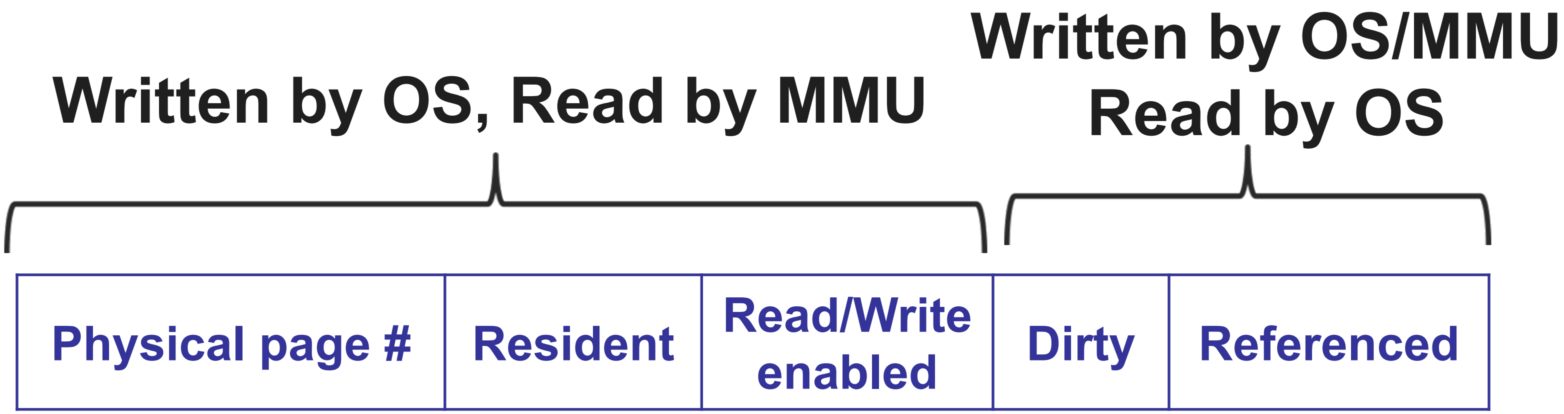
EECS 482
Introduction to Operating Systems

Winter 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Page table contents



Written by OS, Read by MMU

Written by OS/MMU
Read by OS

Physical page #	Resident	Read/Write enabled	Dirty	Referenced

Page table contents

Written by OS, Read by MMU

Physical page #	read_enabled	write_enabled

Address Space Management

- How to manage a process’s accesses to its address space?
  - Kernel sets up page table per process and manages which pages are resident
  - MMU looks up page table to translate any virtual address to a physical memory address

- What about kernel’s address space?
- How does MMU handle kernel’s loads and stores?

Storing Page Tables

- Two options:
  1. In physical memory
  2. In kernel's virtual address space
- Difference: Is PTBR a physical or virtual addr?
- Pros and cons of option 2?
  - Can page out user page tables
  - Kernel page table must be kept in physical memory
- Project 3 uses option 2
  - 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
- How can kernel access specific physical memory addresses (e.g., to refer to translation data)?
  - Kernel can issue untranslated address (bypass MMU)
  - Kernel can map physical memory into a portion of its virtual address space (vm_physmem in Project 3)

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 enables access both OS and user parts of that address space

Kernel vs. user mode

- How are we protecting a process's address space from other processes?
- Must ensure that 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 from user process into kernel

- 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 must check validity of arguments
  - e.g., `read(int fd, void *buf, size_t size)`

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 drawing a state machine for a virtual page first
  - Focus on swap-backed pages first (before file-backed pages)
- Avoid doing unnecessary work

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)
  - Getting this combination right = 21/75

Process creation

- :(){ :|:&};:
  - : ()   -> define a function called :
  - { :|:&} -> the function sends its output to the : function again and runs that in the background.
  - ; is the command separator
  - : runs the function the first time

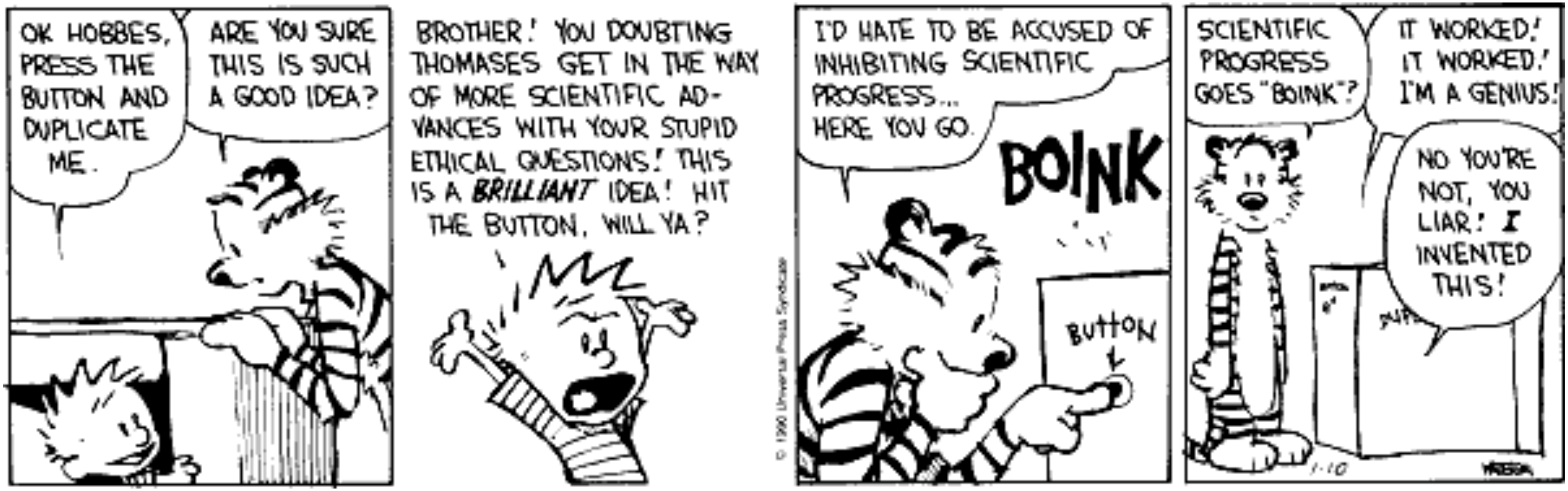
Unix process creation

- 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?
  - Linux: Share code, file descriptors, etc
  - Windows: CreateProcess(program, args) uses a different mode of creating from scratch

- 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?
© 1990 Universal Press Syndicate
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!
1-10
WATTERSON

Unix process creation

- 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 */
}
```

Subtleties in handling fork

- Buggy code from autograder:

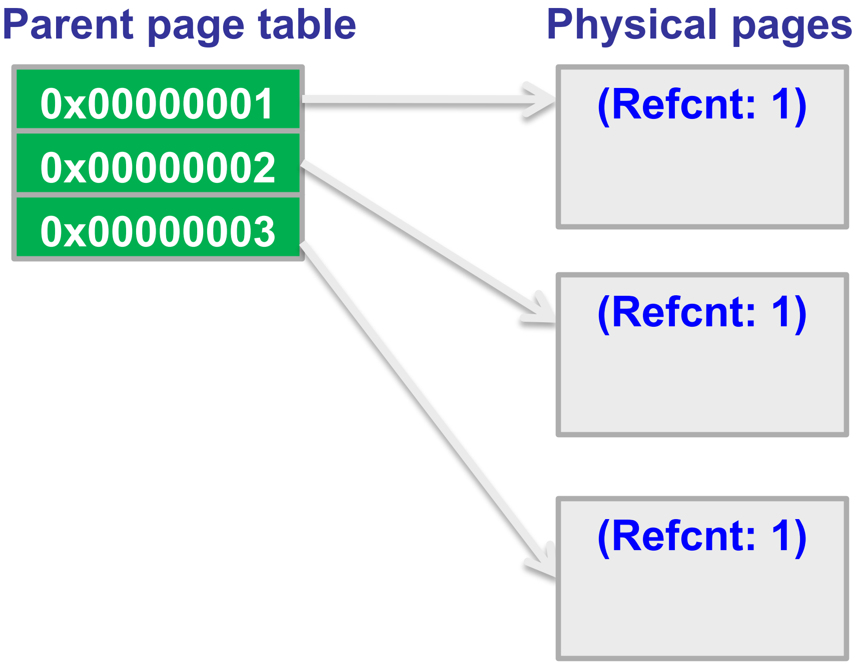
```
if (!fork()) {
    exec(command);
}
while(child is alive) {
    if (size of child address space > max) {
        print "process took too much
memory”;
        kill child;
        break;
    }
}
```

- What is the bug here?

Avoiding work on fork

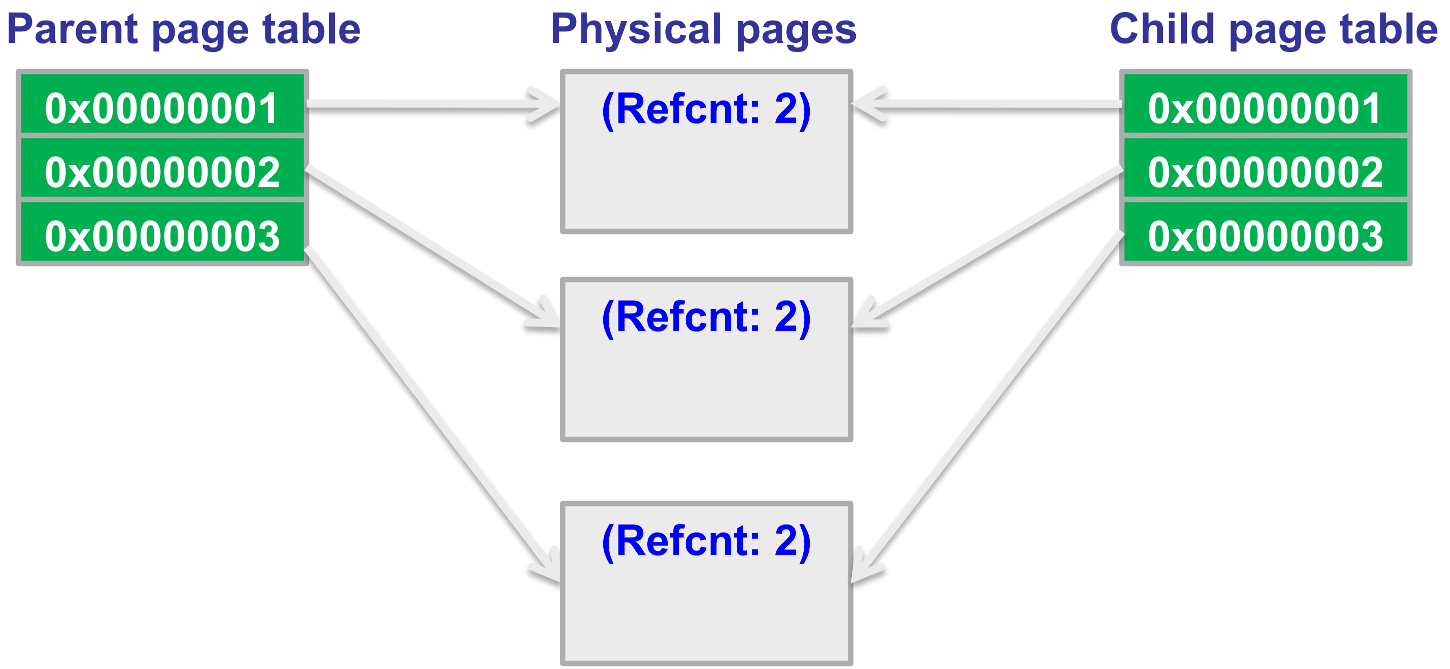
- Copying entire address space is expensive
- Instead, Unix uses **copy-on-write**
  - Assign reference count to each physical page
  - On fork(), copy only the page table of parent
    - » Increment reference count by one
  - On store by parent or child to page with refcnt > 1:
    - » Make a copy of the page with refcnt of one
    - » Modify PTE of modifier to point to new page
    - » Decrement reference count of old page

Copy-on-write: Example



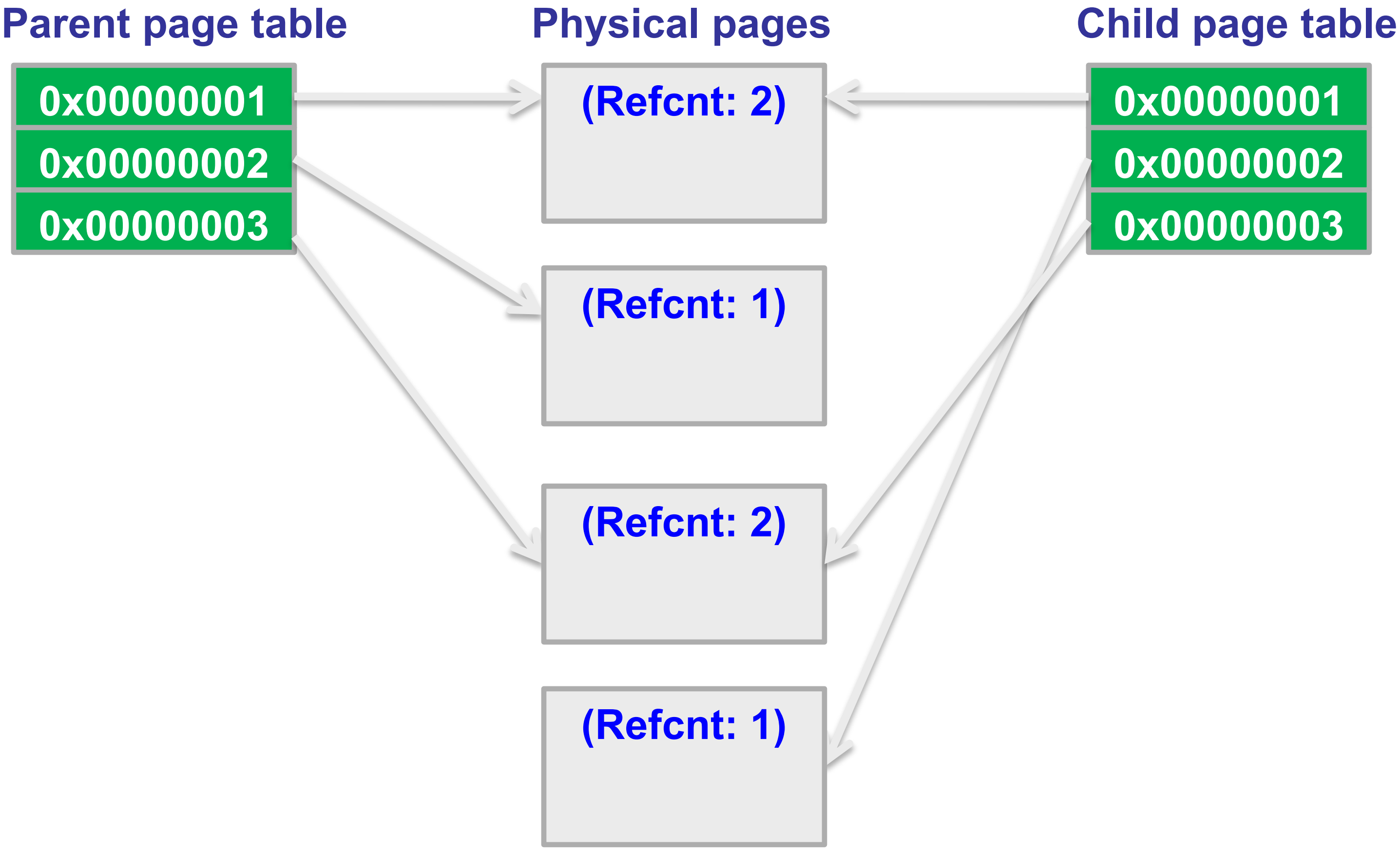
Parent about to fork()

Copy-on-write: Example



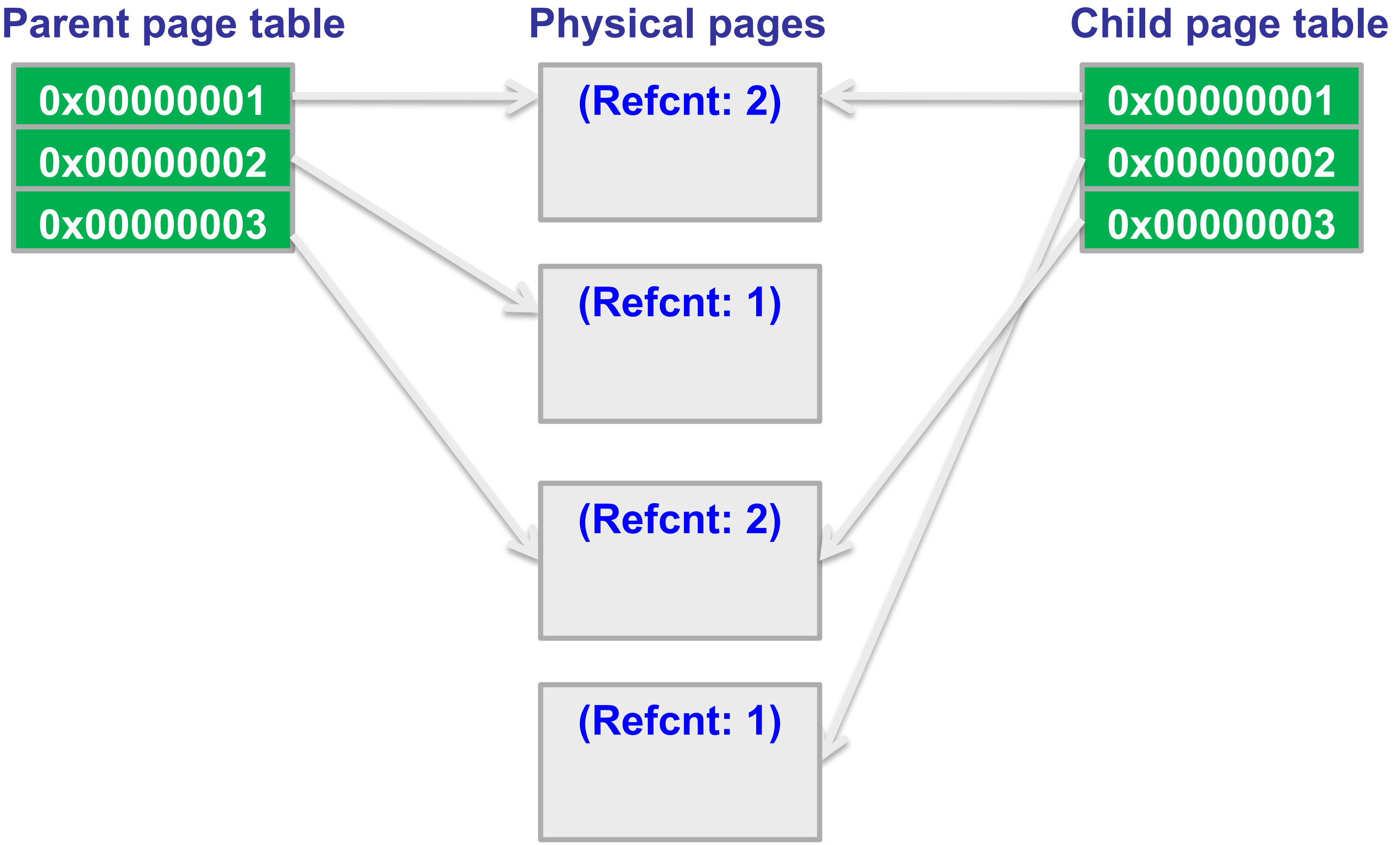
Copy-on-write of parent address space

Copy-on-write: Example



Child modifies 2nd virtual page

Copy-on-write: Example



Parent page table

0x00000001

0x00000002

0x00000003

Physical pages

(Refcnt: 2)

(Refcnt: 1)

(Refcnt: 2)

(Refcnt: 1)

Child page table

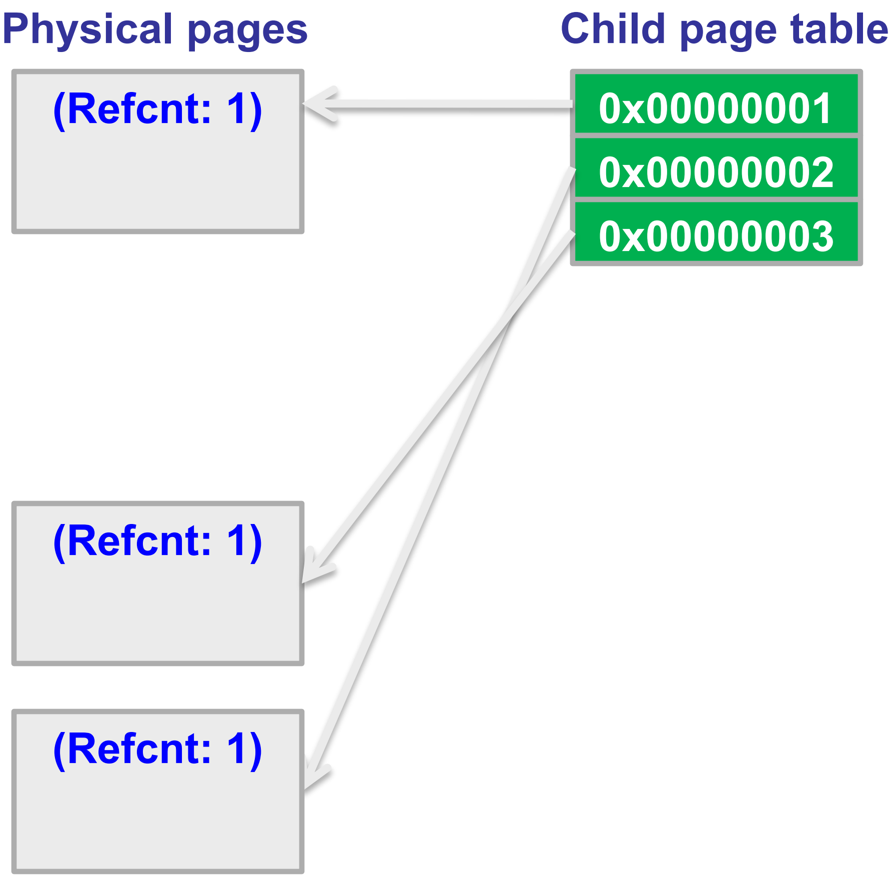
0x00000001

0x00000002

0x00000003

Parent modifies 2nd virtual page

Copy-on-write: Example


Physical pages

Child page table

(Refcnt: 1)

(Refcnt: 1)

(Refcnt: 1)

0x00000001

0x00000002

0x00000003

Parent exits

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
    }
}
```

- Go to the lab section on Friday for a run down of project 3

- Have a good spring break