# Page table contents

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>Resident</th>
<th>Read/Write enabled</th>
<th>Dirty</th>
<th>Referenced</th>
</tr>
</thead>
</table>

- **Written by OS, Read by MMU**
- **Written by OS/MMU, Read by OS**
Page table contents

Written by OS, Read by MMU

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>read_enabled</th>
<th>write_enabled</th>
</tr>
</thead>
</table>

February 27, 2019
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?

- 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?
- How can kernel access specific physical memory addresses (e.g., to refer to translation data)?
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.

  - Trap to kernel doesn’t change address spaces; it just enables access to both OS and user parts of that address space.

<table>
<thead>
<tr>
<th>fffff</th>
<th>operate system</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>80000</td>
<td></td>
</tr>
<tr>
<td>7ffff</td>
<td>user process</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>00000</td>
<td></td>
</tr>
</tbody>
</table>
**Kernel vs. user mode**

- How are we protecting a process’ 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.
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?

- 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?
- How is translation data protected?
- How is mode bit protected?

- Protection boils down to init process which sets up interrupt vector table when system boots up.
Project 3

- Memory management using paging
  - Due March 27\textsuperscript{st}
- 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

February 27, 2019
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?

- Any problems with child being an exact clone of parent?
Unix process creation

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

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

- Buggy code from autograder:

```c
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; set refcnt to one for that page
    » Modify PTE of modifier to point to new page
    » Decrement reference count of old page
Copy-on-write: Example

Parent page table

- 0x00000001
- 0x00000002
- 0x00000003

Physical pages

- (Refcnt: 1)
- (Refcnt: 1)
- (Refcnt: 1)

Parent about to fork()
Copy-on-write: Example

Copy-on-write of parent address space

February 27, 2019
Copy-on-write: Example

Parent page table

Parent page table

Physical pages

Child page table

0x00000001
0x00000002
0x00000003

(Refcnt: 2)

(Refcnt: 1)

(Refcnt: 2)

(Refcnt: 1)

Child modifies 2\textsuperscript{nd} virtual page

February 27, 2019
Copy-on-write: Example

Parent page table
- 0x00000001
- 0x00000002
- 0x00000003

Physical pages
- (Refcnt: 2)

Child page table
- 0x00000001
- 0x00000002
- 0x00000003

Parent modifies 2\textsuperscript{nd} virtual page

February 27, 2019
Copy-on-write: Example

Parent exits

Physical pages

(Refcnt: 1)

Child page table

0x00000001
0x00000002
0x00000003

(Refcnt: 1)

(Refcnt: 1)
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
    }
}
Good luck in the midterm
Have a good spring break