Office Hours this week:
Tuesday 12-1pm, 2:30-5pm
Wednesday 4-5pm

Virtual Memory Case Studies

MM (programming) due Friday!
Office Hours canceled today

Outline

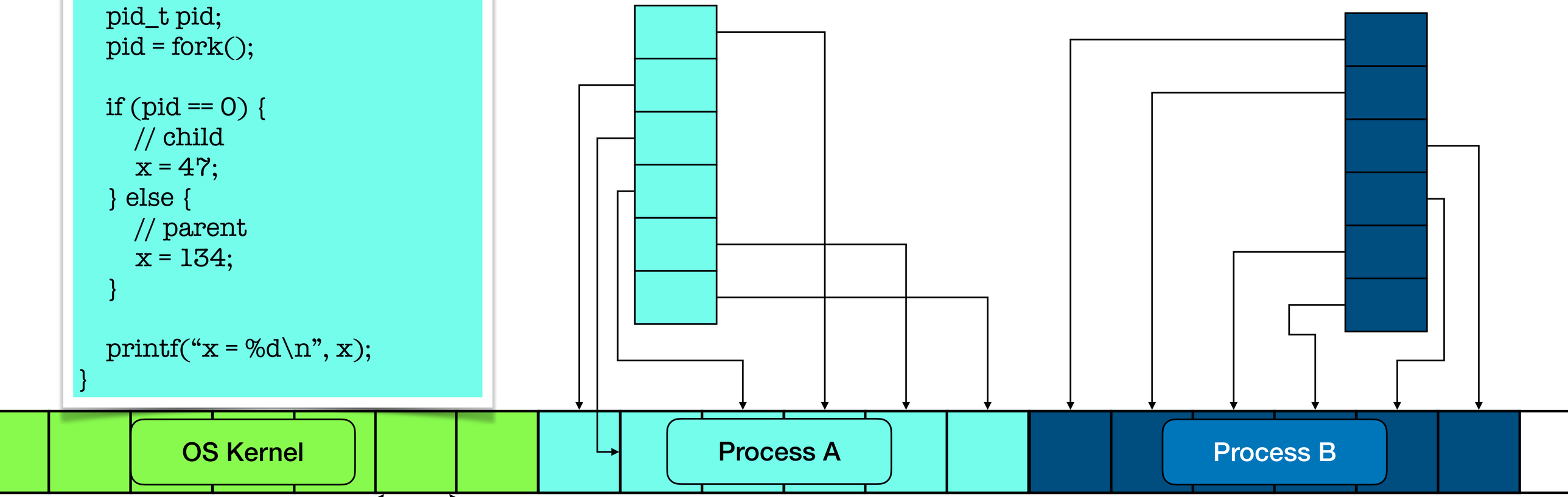
- Copying address spaces via Fork (and CoW)
- Dynamically Loaded Libraries
- Paging optimizations

Naive Implementation of fork/execve

```
void fn() {
    int  x = 0;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        // child
        x = 47;
    } else {
        // parent
        x = 134;
    }

    printf("x = %d\n", x);
}
```



Chat with your neighbor(s)!

When thinking about the naive implementation of `fork`, what inefficiencies exist? Think about runtime and data utilization.

A lot of time is spent to copy over the address space, though very little of it is actually used!

The vfork Variant

This implementation alludes to the idea that tasks, such as calls to `fork`, can be made more efficient if done lazily (as needed)

```
void fn() {
   int  x = 0;
   pid_t pid;
   pid = fork();

   if (pid == 0) {
      // child
      x = 47;
   } else {
      // parent
      x = 134;
   }

   printf("x = %d\n", x);
}
```

Create child context with file descriptors, etc…

```
   movw   $0, %esi
   callq  <fork>
   testq  %rax, %rax
   jne    .L1
   movw   $47, %esi
   jmp    .L2
.L1:
   movw   $134, %esi
.L2:
   movq   0x400400, %rdi
   callq  <printf>
```

Only a small subset of registers change!

Very efficient!

Only works if child returns address space back to the parent

Continue operating in *the same* address as the original parent process

Child shouldn't do anything to the address space that affects the parent!

Introduced in 1979 by Berkeley Software Distribution (BSD), but doesn't provide complete implementation of `fork`…

Lessons Learned from Naive `fork`

- If we try to copy an entire address space every time that we create a new process, then we will (1) spend a lot of time performing the copying procedure, and (2) likely won't use everything that we copy
- By using `vfork`, a child process runs in exactly the same address space as its parent (with a different context), so the child process is able to do a small number of tasks and exit efficiently ➡️ this cannot safely be a complete implementation of `fork`, or else we will disrupt the parent process's execution
- A better approach would be to put off work as long as possible (implement our address space *lazily*) and we might be able to get away with *never* having done the work of copying the addresses

Chat with your neighbor(s)!

Suppose we wanted to implement the address space copying procedure of a call to fork in as lazy a manner as possible. At what point would you *need* to copy a page from the parent virtual address space into a fresh version?

Just because an address is used doesn't mean that we need to copy it!

If we change data in the parent's address space, then we will need a fresh copy of it

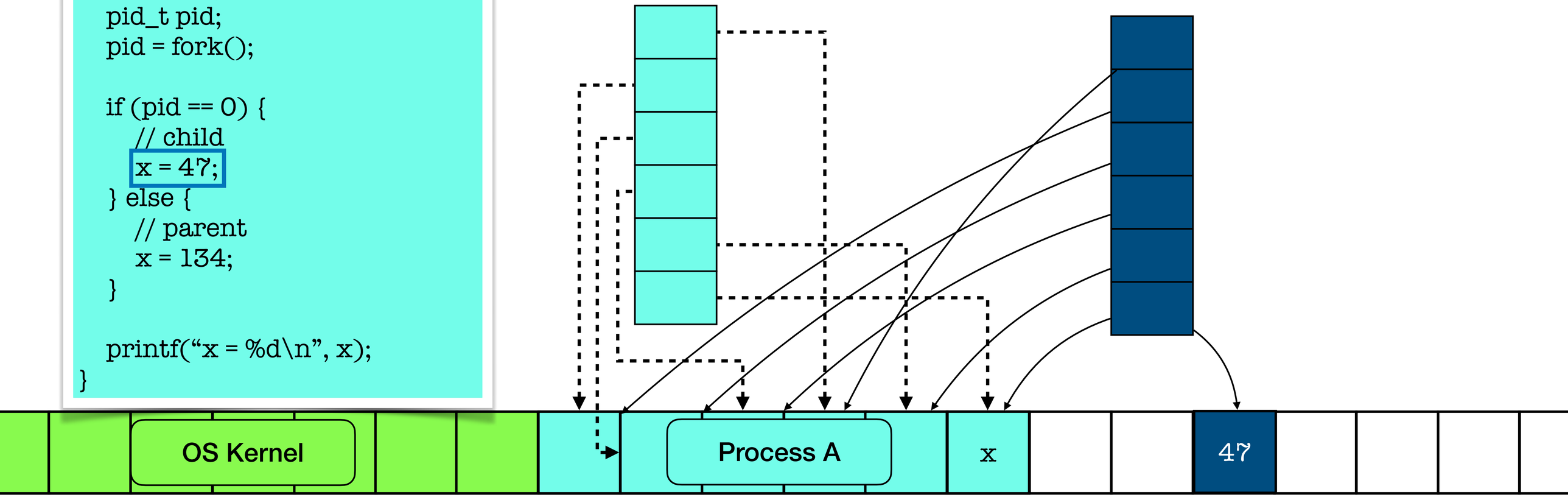
To do so, we are describing a procedure described as *Copy-on-Write (CoW)*

Lazy Implementation of fork/execve

```
void fn() {
    int  x = 0;
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        // child
        x = 47;
    } else {
        // parent
        x = 134;
    }

    printf("x = %d\n", x);
}
```



Option 3: refer to A's address space in a unique page table

Can We Get Even Lazier?

```c
/* Process descriptor */
typedef struct proc {
    pid_t pid;                    // Process ID
    char p_name[MAX_STRING_LEN];  // Process name

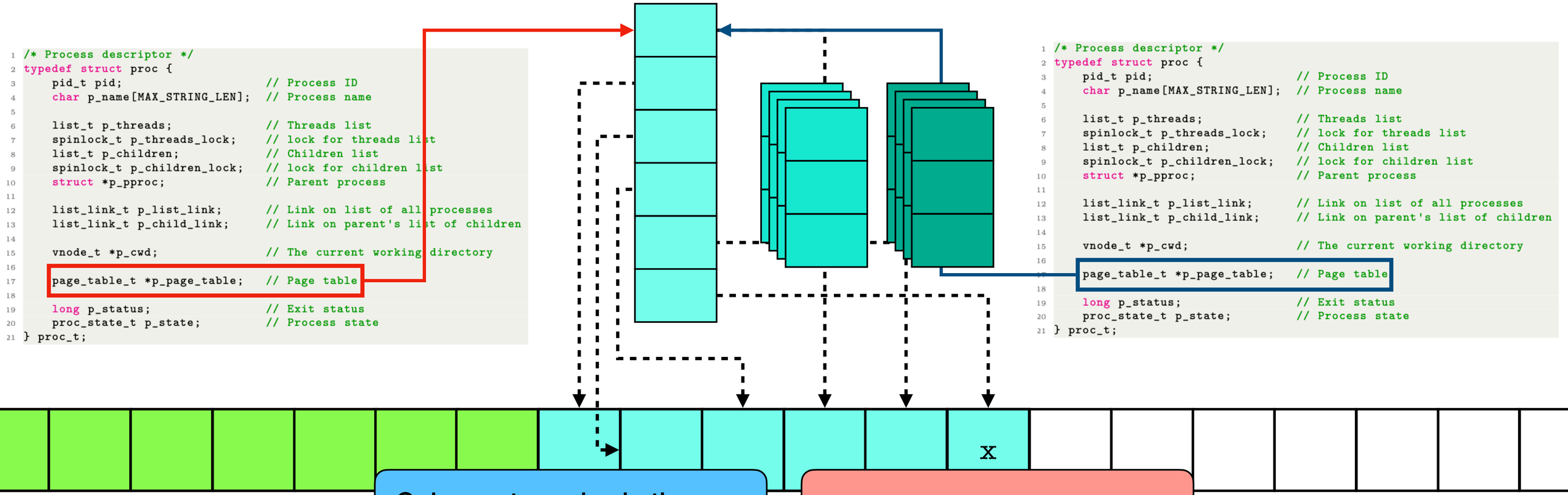
    list_t p_threads;             // Threads list
    spinlock_t p_threads_lock;    // lock for threads list
    list_t p_children;            // Children list
    spinlock_t p_children_lock;   // lock for children list
    struct *p_pproc;              // Parent process

    list_link_t p_list_link;      // Link on list of all processes
    list_link_t p_child_link;     // Link on parent's list of children

    vnode_t *p_cwd;               // The current working directory

    page_table_t *p_page_table;   // Page table

    long p_status;                // Exit status
    proc_state_t p_state;         // Process state
} proc_t;
```




```c
/* Process descriptor */
typedef struct proc {
    pid_t pid;                    // Process ID
    char p_name[MAX_STRING_LEN];  // Process name

    list_t p_threads;             // Threads list
    spinlock_t p_threads_lock;    // lock for threads list
    list_t p_children;            // Children list
    spinlock_t p_children_lock;   // lock for children list
    struct *p_pproc;              // Parent process

    list_link_t p_list_link;      // Link on list of all processes
    list_link_t p_child_link;     // Link on parent's list of children

    vnode_t *p_cwd;               // The current working directory

    page_table_t *p_page_table;   // Page table

    long p_status;                // Exit status
    proc_state_t p_state;         // Process state
} proc_t;
```

Only create nodes in the new page table when needed! Then, just change pointers!

Violates process isolation…

Lazily Copying Virtual Memory

- To avoid preemptively copying over the entire parent address space to a newly forked child, initialize the child page table to exclusively refer to the parent's page table entries
- Whenever a process *writes to a location in memory*, the child process should create a new mapping from the virtual address in its address space to the physical address in the parent's address space (copy-on-write)
- We cannot implement address space isolation and do CoW on the page table itself… every process needs its own page table on process creation
- Page table entries themselves need to be updated! The MMU needs to be able to distinguish between private pages and shared pages so that an update to a shared page triggers a page fault and a new mapping from virtual address to physical address can be constructed

Chat with your neighbor(s)!

Suppose we want to support dynamically loaded libraries in our system. This means that the operating system will add a library to the virtual address space when starting up a process. Where should the OS put the library in the virtual address space?

The OS needs to build the layout, but the assembly deals in virtual addresses…

The compiler needs to know where the function addresses are so that we can call them from the assembly instructions

How do the compiler and the OS know where the other expects the shared library to go?

Demo!

```
Disassembly of section .plt:

0000000000400480 <.plt>:
  400480:       ff 35 82 0b 20 00       pushq  0x200b82(%rip)        # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  400486:       ff 25 84 0b 20 00       jmpq   *0x200b84(%rip)        # 601010 <_GLOBAL_OFFSET_TABLE_+0x1
0>
  40048c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000400490 <puts@plt>:
  400490:       ff 25 82 0b 20 00       jmpq   *0x200b82(%rip)        # 601018 <puts@GLIBC_2.2.5>
  400496:       68 00 00 00 00          pushq  $0x0
  40049b:       e9 e0 ff ff ff          jmpq   400480 <.plt>
```

Dynamically Loaded Libraries

- When using libraries, we make function calls to code that exists in the file system (and potentially in memory as well) that we would like to refer to in the process' address space
- On process creation, the operating system creates a mapping between the virtual addresses and physical addresses so that the library is accessible from the virtual address space ➡️ this means creating a mapping to a known location where the library lives
- When the binary is compiled, the process does not exist ➡️ the "process lookup table" (PLT) allows the operating system to know which virtual addresses should be destinations to which it should link the physical library files

Summary

- When making a call to fork, we need the created process to have its own independent address space, but copying over an entire address space is expensive
- Lazily copying data on write is important for performance, and our data structures need to have a mechanism to know that a page is private or shared
- Dynamically loaded libraries require the operating system and the compiler to be aware of where the shared library is going to be placed