

The Hardware/Software Interface

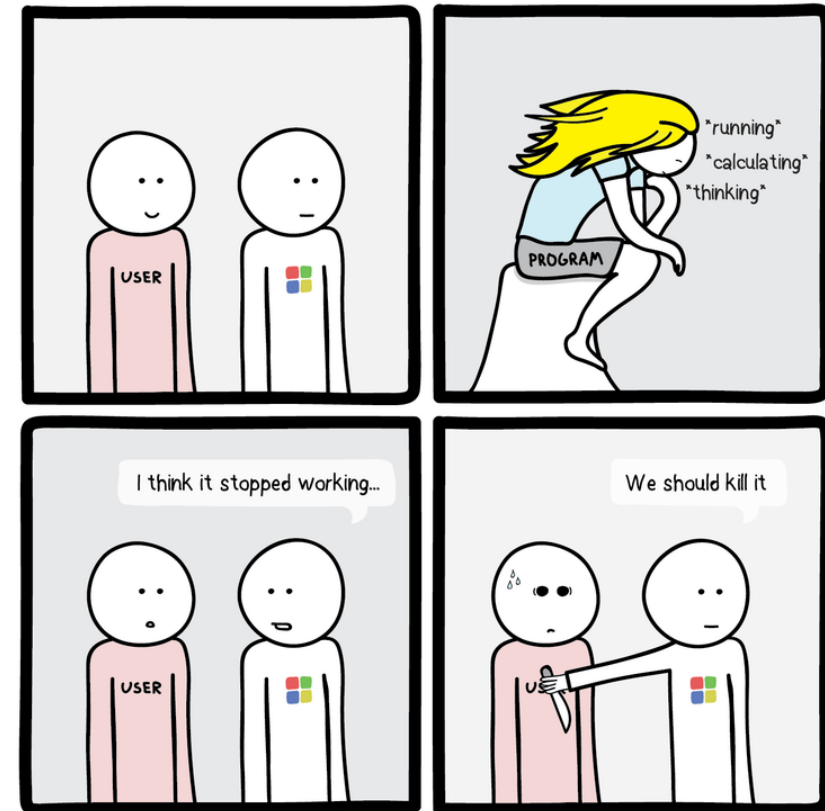
Processes II, Virtual Memory I

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

| | |
|------------------|------------------|
| Anthony Mangus | Divya Ramu |
| Grace Zhou | Jessie Sun |
| Jiuyang Lyu | Kanishka Singh |
| Kurt Gu | Liander Rainbolt |
| Mendel Carroll | Ming Yan |
| Naama Amiel | Pollux Chen |
| Rose Maresh | Soham Bhosale |
| Violet Monserate | |



PRETENDS TO BE DRAWING | PTBD.JWELS.BERLIN

<https://ptbd.jwels.berlin/comic/20/>

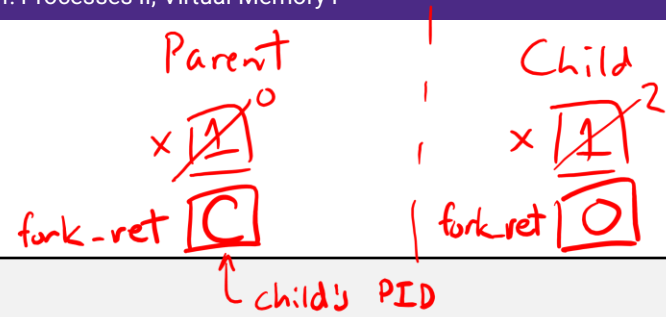
Relevant Course Information

- ❖ HW22 due tonight, HW23 due Monday
- ❖ Lab 4 due tonight, submissions close Monday; Lab 5 due 12/4
- ❖ Final exam: Wednesday, 12/10 @ 12:30 pm
 - Final review section on 12/4, final review session (hybrid) on 12/5
 - *Cumulative*: Questions will be marked “M” (pre-midterm) or “F” (post-midterm)
 - Scores on the “M” questions will be used for [midterm clobber policy](#)
 - TWO double-sided handwritten 8.5×11” cheat sheets
 - Recommended that you reuse or remake your midterm cheat sheet
 - We will distribute copies of the [Final Reference Sheet](#) on Monday

Lecture Outline (1/3)

- ❖ **fork (continued) and exec***
- ❖ Ending a Process
- ❖ Virtual Memory Introduction

fork Example



```
void fork1() {  
    int x = 1;  
    pid_t fork_ret = fork();  
    if (fork_ret == 0)   
        printf("Child has x = %d\n", ++x);  
    else  
        printf("Parent has x = %d\n", --x);  
    printf("Bye from process %d with x = %d\n", getpid(), x);  
}
```

Handwritten annotations on the code:

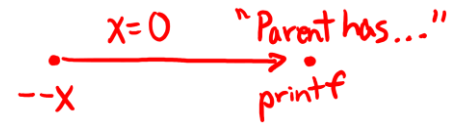
- `fork()` is annotated with "splits here" and an arrow pointing to the function call.
- `++x` is annotated with "child only" and an arrow pointing to the expression.
- `--x` is annotated with "parent only" and an arrow pointing to the expression.
- `getpid()` is annotated with "both" and an arrow pointing to the function call.

❖ Notes/Reminders:

- Both processes continue/start execution after fork
 - Can't predict execution order between **parent** and **child**
- Both processes start with $x = 1$
 - However, subsequent changes to x are independent
- Shared open files: `stdout` is the same in both **parent** and **child**

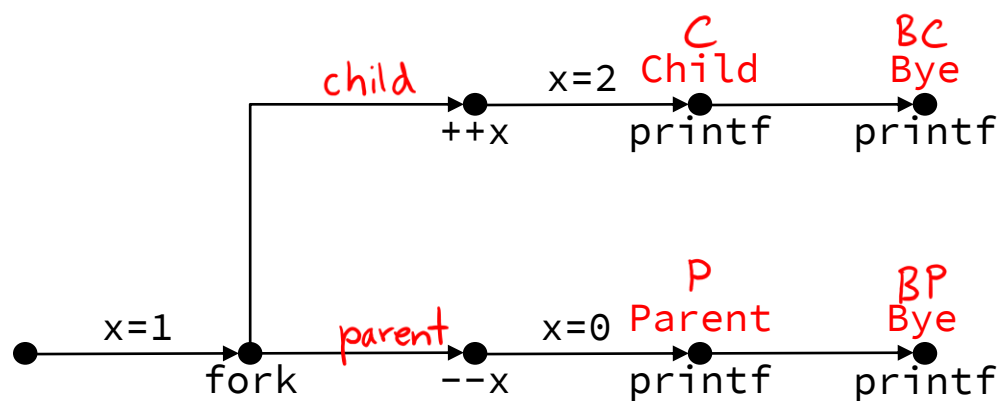
Modeling Concurrency with Process Graphs

- ❖ A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
 - Each vertex indicates the execution of a notable statement
 - Edges ($a \rightarrow b$) indicate sequential ordering of statements within a process
 - *i.e.*, a must happen before b
 - Vertices and edges can be labeled with important notes
 - *e.g.*, updated variable values on edges, program output on `printf` vertices
 - Each graph begins with a vertex with no in-edges
- ❖ Any *topological sort* of the graph corresponds to a feasible total ordering
 - An ordering of nodes that contains every node, and only follows edges (lines between nodes) in the direction of the arrows



fork Example: Process Graph

```
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



Possible

| | | | |
|----|----|----|----|
| C | P | C | C |
| BC | BP | P | P |
| P | C | BC | BP |
| BP | BC | BP | BC |

etc...

Not Possible

| | |
|----|----|
| C | P |
| BC | BC |
| BP | C |
| P | BP |

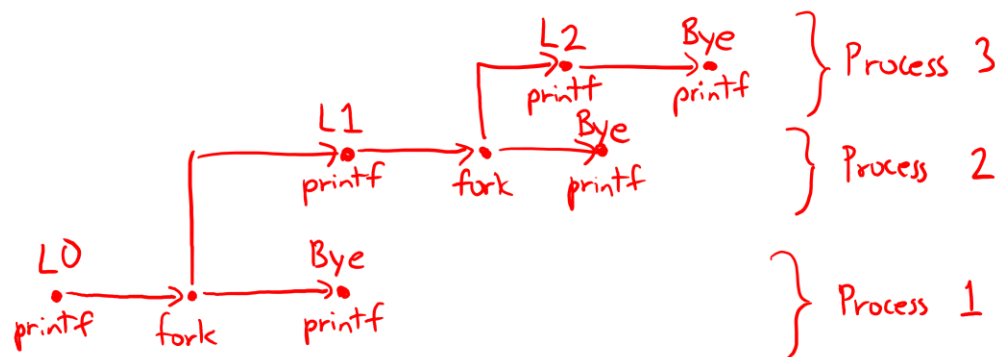
etc...

as long as C comes before BC
and P comes before BP

Polling Questions (1/3)

❖ Are the following sequences of outputs possible?

```
void nestedfork() {
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Seq 1:

L0

L1

Bye

Bye

Bye

L2 !

Seq 2:

L0 ← Process 1

Bye ← Process 1

L1 ← Process 2

L2 ← Process 3

Bye ← Process 2/3

Bye ← Process 3/2

A. No

No

B. No

Yes

C. Yes

No

D. Yes

Yes

E. We're lost...

Fork-Exec

Note: The return values of `fork` and `exec*` should be checked for errors

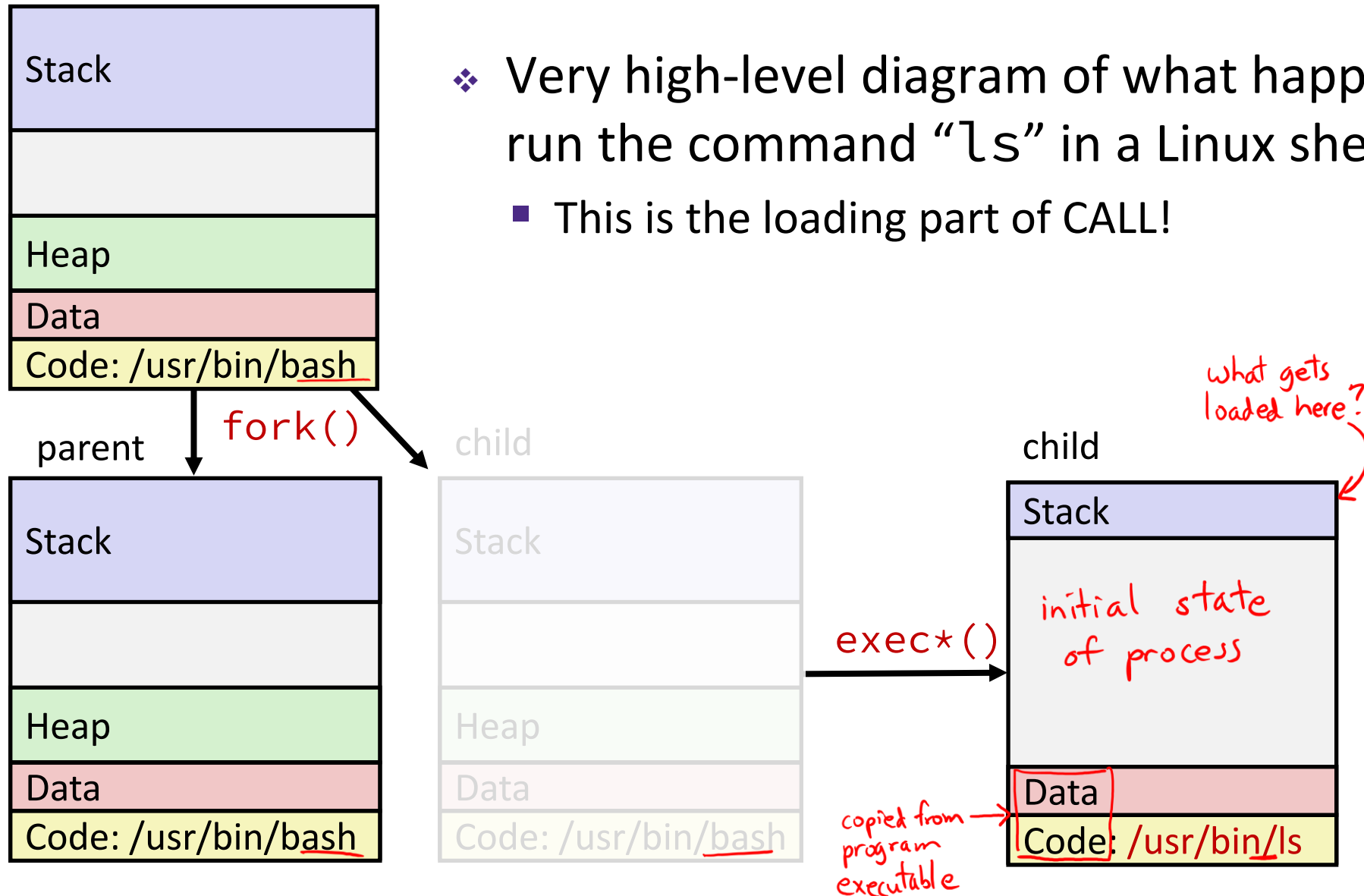
❖ fork-exec model:

- `fork()` creates a copy of the current process
- `exec*()` replaces the current process' code and address space with the code for a different program
 - Whole family of `exec` calls – see [exec\(3\)](#) and [execve\(2\)](#)

```
// Example arguments: path="/usr/bin/ls",
//                   argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char* path, char* argv[]) {
    pid_t fork_ret = fork();
    if (fork_ret != 0) {
        printf("Parent: created a child %d\n", fork_ret);
    } else {
        printf("Child: about to exec a new program\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```


Exec-ing a New Program

- ❖ Very high-level diagram of what happens when you run the command “ls” in a Linux shell
 - This is the loading part of CALL!



Lecture Outline (2/3)

- ❖ fork (continued) and exec*
- ❖ **Ending a Process**
- ❖ Virtual Memory Introduction

Ending a Process (Review)

❖ **void** `exit(int status)`

- Explicitly exits a process
 - Status code: 0 is used for a normal exit, nonzero for abnormal exit

❖ The `return` statement from `main()` also ends a process in C

- The return value is the status code

❖ An *abort* from an exception handler

Zombies! 🧟 🧟 (Review)

- ❖ A terminated process still consumes system resources
 - Various tables maintained by OS
 - Called a “**zombie**” (a living corpse, half alive and half dead)
- ❖ *Reaping* is performed by parent on terminated child
 - Parent is given exit status information and kernel then deletes zombie child process
 - In long-running processes (e.g., shells, servers) we need *explicit* reaping
- ❖ If parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (PID 1)
 - **Note:** on recent Linux systems, `init` has been renamed `systemd`

wait: Synchronizing with Children

❖ `int wait(int* child_status)`

- Suspends current process (*i.e.*, the parent) until one of its children terminates
- Return value is the PID of the child process that terminated
 - *On successful return, the child process is reaped*
- If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
 - Special macros for interpreting this status – see [wait\(2\)](#)

- ❖ **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
 - `waitpid` can be used to wait on a specific child process

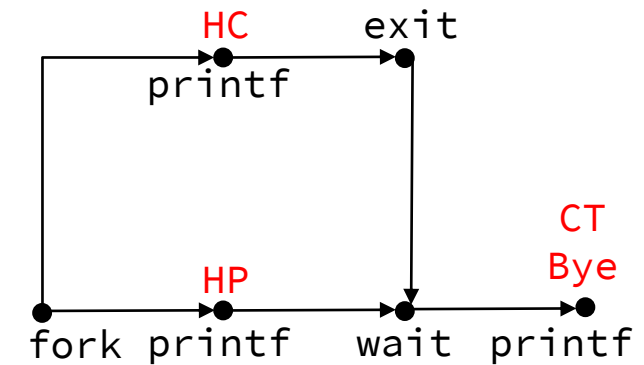
Example: wait

```
void fork_wait() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

child

parent



Feasible
output:

HC HP
HP HC
CT CT
Bye Bye

Infeasible
output:

HP
CT
Bye
HC

Example: Zombie

❖ Parent in infinite loop, terminated child

```
void fork7() {  
    if (fork() == 0) {  
        /* Child */  
        printf("Terminating Child, PID = %d\n",  
               getpid());  
        exit(0);  
    } else {  
        /* Parent */  
        printf("Running Parent, PID = %d\n",  
               getpid());  
        while (1); /* Infinite loop */  
    }  
}
```

- ps shows child process as “defunct”
- Terminating parent allows child to be reaped by init (both are now gone)

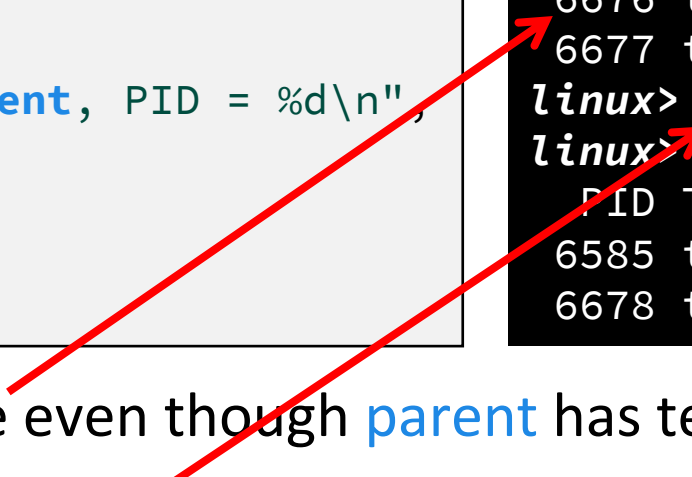
```
linux> ./forks 7 &  
[1] 6639  
Running Parent, PID = 6639  
Terminating Child, PID = 6640  
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9        00:00:00 tcsh  
 6639 ttyp9        00:00:03 forks  
 6640 ttyp9        00:00:00 forks <defunct>  
 6641 ttyp9        00:00:00 ps  
linux> kill 6639  
[1] Terminated  
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9        00:00:00 tcsh  
 6642 ttyp9        00:00:00 ps
```

Example: Non-Terminating Child

❖ Child in infinite loop, terminated parent

```
void fork8() {  
    if (fork() == 0) {  
        /* Child */  
        printf("Running Child, PID = %d\n",  
               getpid());  
        while (1); /* Infinite loop */  
    } else {  
        /* Parent */  
        printf("Terminating Parent, PID = %d\n",  
               getpid());  
        exit(0);  
    }  
}
```

```
linux> ./forks 8  
Terminating Parent, PID = 6675  
Running Child, PID = 6676  
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9        00:00:00 tcsh  
 6676 ttyp9        00:00:06 forks  
 6677 ttyp9        00:00:00 ps  
linux> kill 6676  
linux> ps  
  PID TTY          TIME CMD  
 6585 ttyp9        00:00:00 tcsh  
 6678 ttyp9        00:00:00 ps
```



- Child process still active even though parent has terminated
- Must terminate child explicitly, or else will keep running indefinitely

Polling Questions (2/3)

- ❖ For the following scenarios, what will the outcome be for a **child** process that executes **exit(0)**:

| Scenario | Outcome for child | | |
|--|--------------------------|---------------|---------------|
| Parent is still executing: | Alive | Reaped | Zombie |
| Parent has called wait() : | Alive | Reaped | Zombie |
| Parent has terminated: | Alive | Reaped | Zombie |

waiting for parent
to decide what to do

reaped by
parent

reaped by
init/systemd

Processes Demos (If Time)

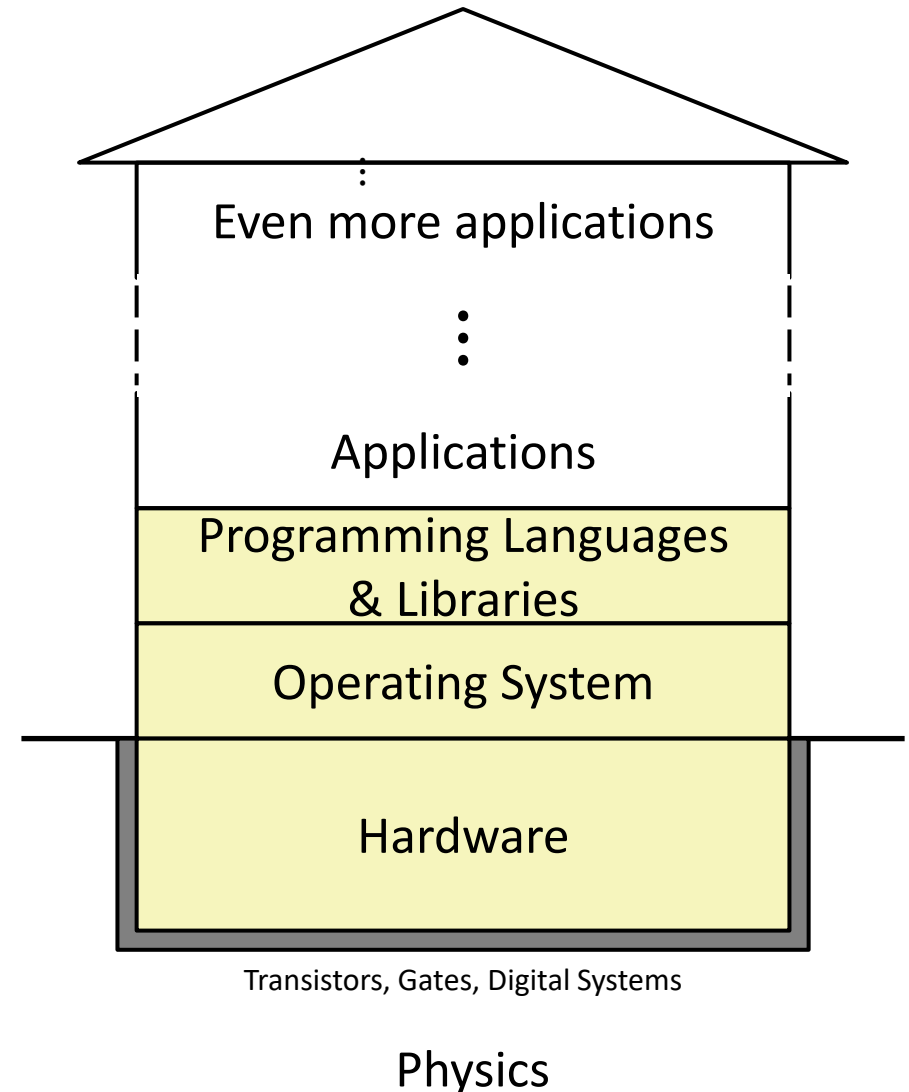
- ❖ How many processes are running on my computer right now?
- ❖ In Linux, the `ps` utility gives a snapshot of currently-running processes and `ps tree` formats these as a tree
 - Can run “`man ps`” and “`man ps tree`” for more info
 - Let’s see a simple `ps tree`
 - Let’s check `at tu` for some 351 zombie processes

Lecture Outline (3/3)

- ❖ fork (continued) and exec*
- ❖ Ending a Process
- ❖ **Virtual Memory Introduction**

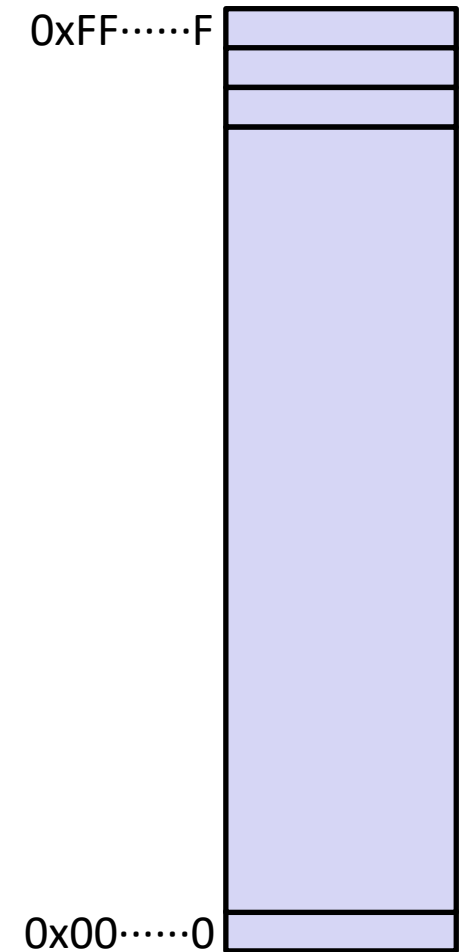
House of Computing Check-in

- ❖ Topic Group 3: **Scale & Coherence**
 - Caches, Memory Allocation, Processes, **Virtual Memory**
- ❖ How do we maintain logical consistency in the face of more data and more processes?
 - How do we support control flow both within many processes and things external to the computer?
 - *How do we support data access, including dynamic requests, across multiple processes?*



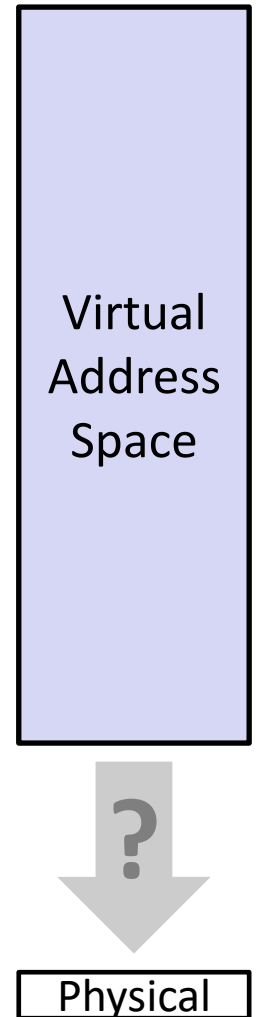
Our View of Memory So Far... is Virtual!

- ❖ Programs refer to virtual memory addresses
 - *Private, virtual* address space (array of bytes) for each process
 - *e.g.*, `movq (%rdi), %rax` # *virtual addresses!*
- ❖ Allocation managed by compiler and run-time system
 - *i.e.*, figure out where different program objects should be stored
- ❖ However, there seem to be some potential issues with this setup...



Problem 1: How Does Everything Fit?

- ❖ *Virtual address space* is set of $N = 2^n$ virtual addresses
 - e.g., 64-bit virtual addresses can address several exabytes ($> 18 \times 10^{18}$ bytes)
- ❖ *Physical address space* is set of $M = 2^m$ physical addresses
 - e.g., 8 GiB main memory offers 8.6×10^9 bytes
 - Note: Not to scale – physical memory would be much smaller than this period: .
- ❖ 1 virtual address space per process, with many processes...



Problem 2: Memory Management

❖ Multiple processes:

- Process 1
- Process 2
- Process 3
- ...
- Process n



❖ Each process has:

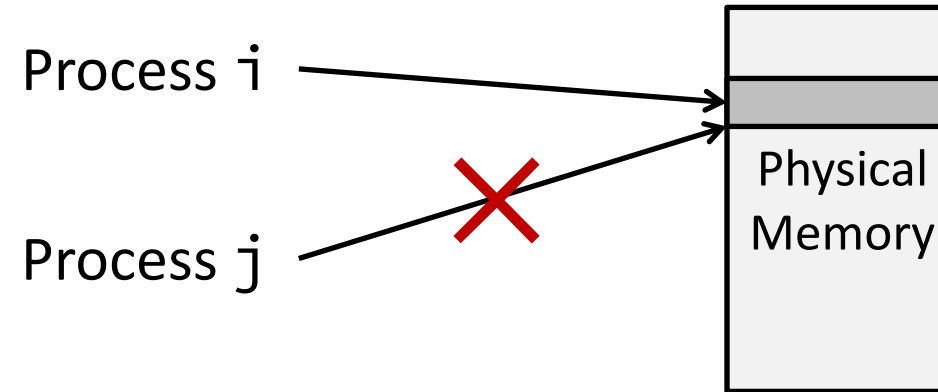
- Stack
- Heap
- .text
- .data
- ...

What goes where?

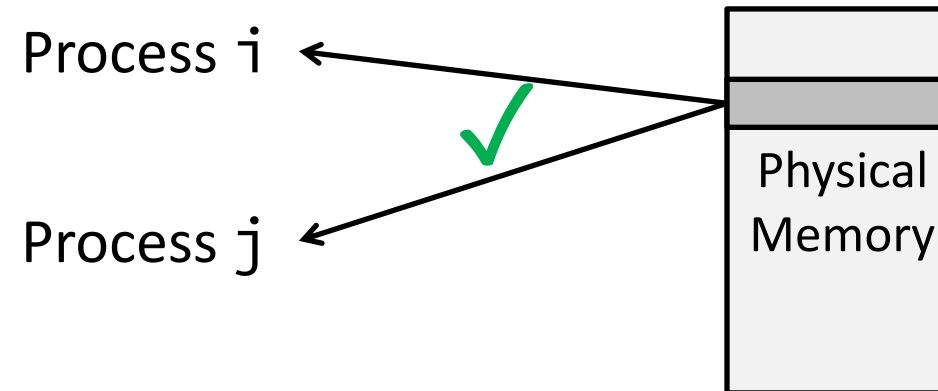
Physical
Memory

Problem 3: Protection and Sharing

❖ Protection:



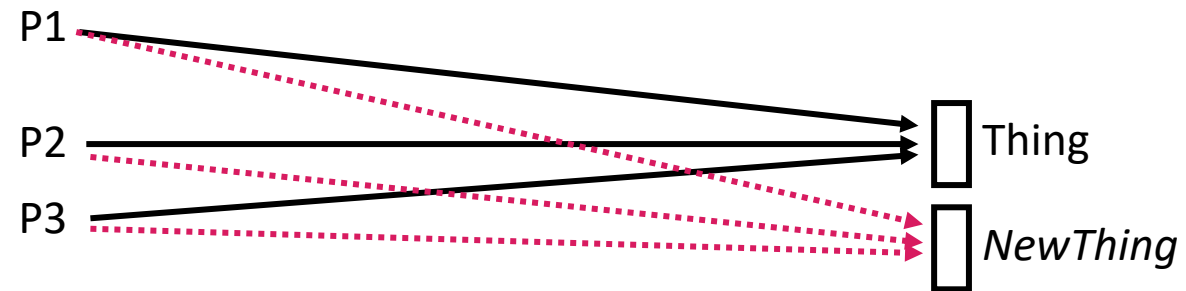
❖ Sharing:



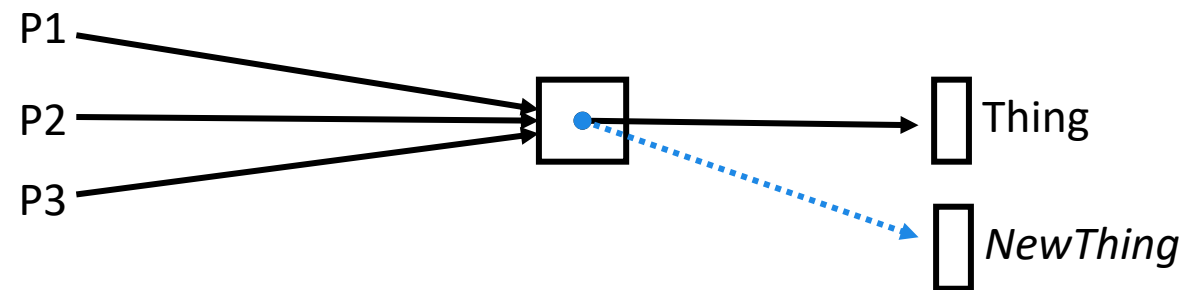
The Solution

- ❖ “Any problem in computer science can be solved by adding another level of **indirection**.” – *David Wheeler, inventor of the subroutine*

- ❖ Without Indirection:



- ❖ With Indirection:



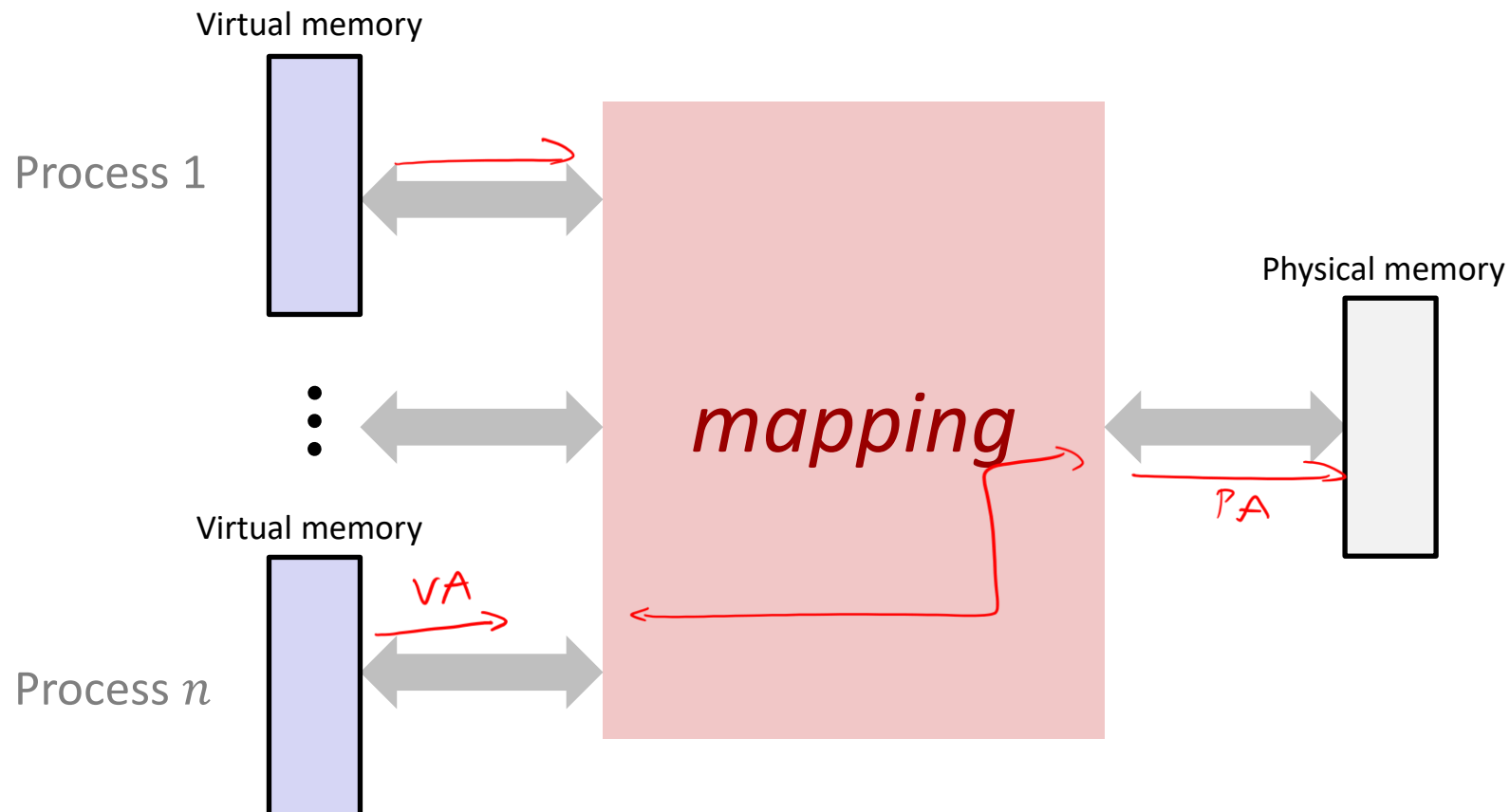
What if I want to move Thing?

Indirection

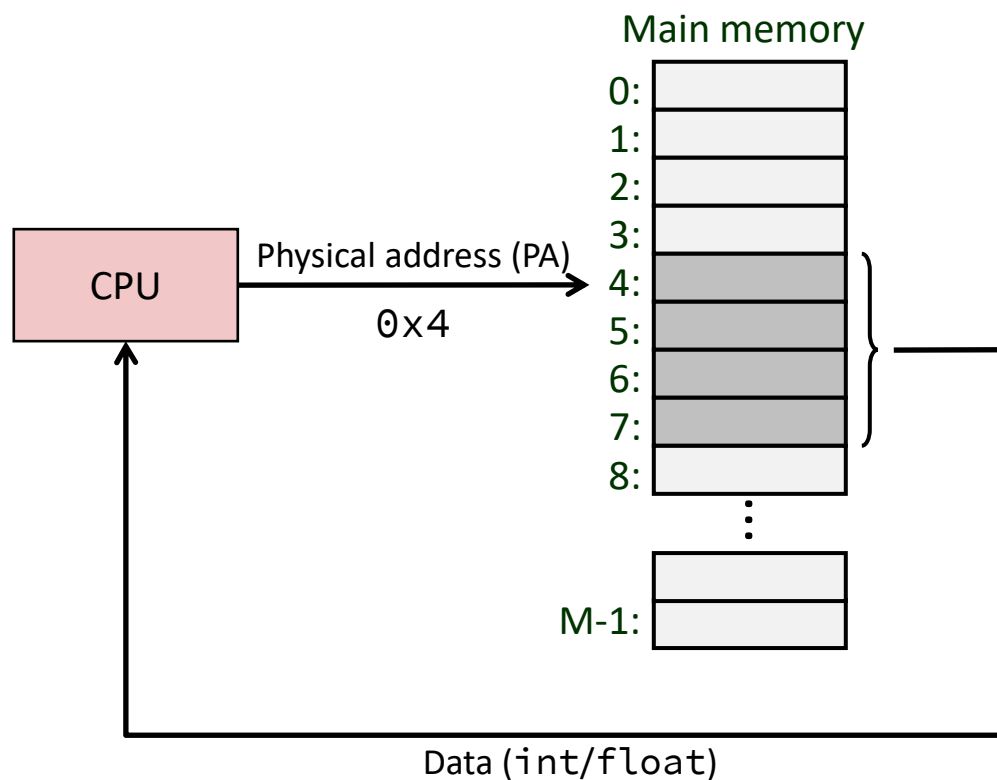
- ❖ The ability to reference something using a name, reference, or container instead of the value itself. A **flexible mapping** between a name and a thing allows **changing the thing without notifying holders of the name**.
 - ■ Adds some work (now we must look up 2 things instead of 1)
 - + ■ But don't have to track all uses of name/address (single source!)
- ❖ Examples:
 - **Domain Name Service (DNS):** Translation from name to IP address
 - **Call centers:** Route calls to available operators, etc.
 - **Dynamic Host Configuration Protocol (DHCP):** Local network address assignment

Indirection in Virtual Memory

- ❖ Each process gets its own private virtual address space
 - Solves the previous problems!

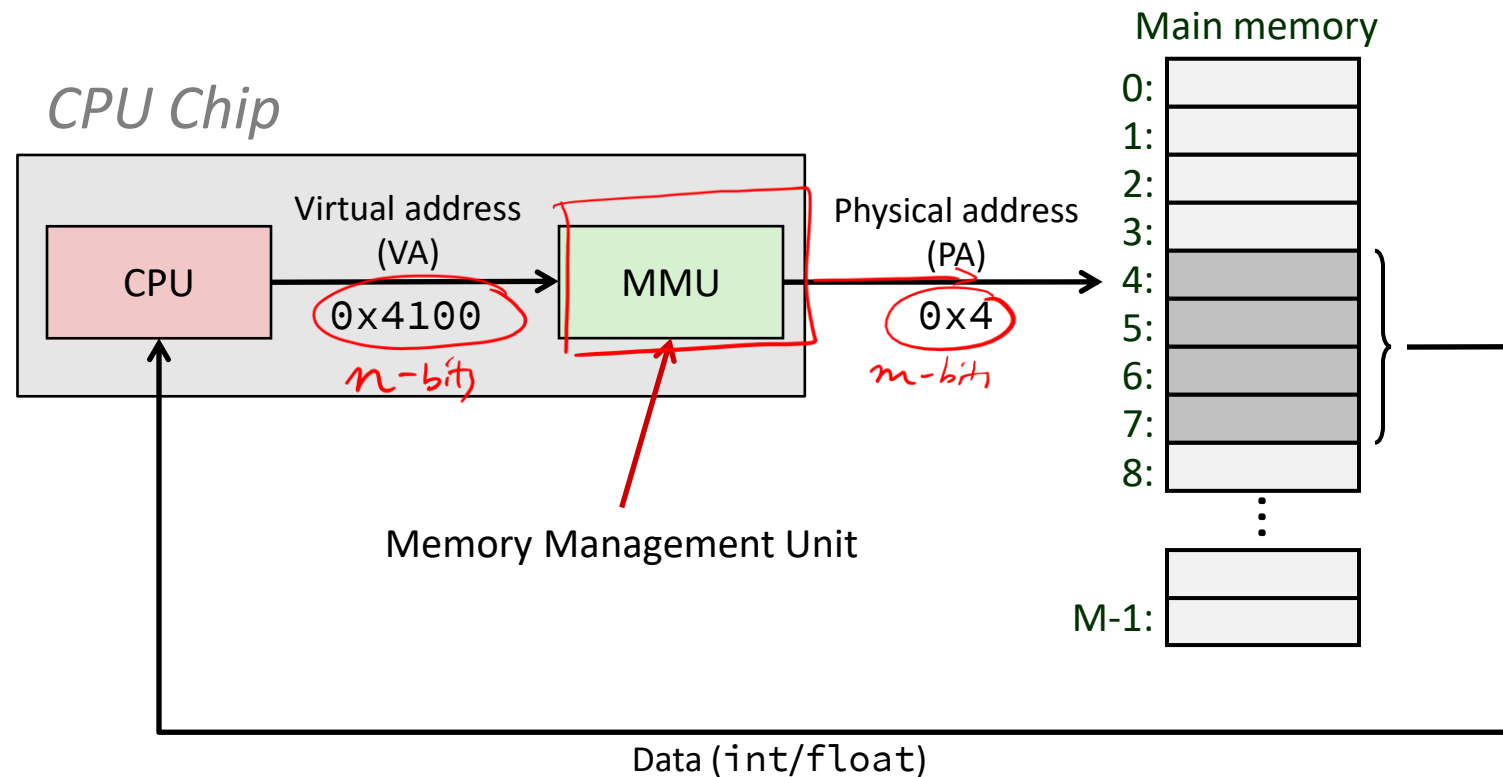


A System Using Physical Addressing



- ❖ Used in “simple” systems with (usually) just one process:
 - Embedded microcontrollers in devices like cars, elevators, and digital picture frames

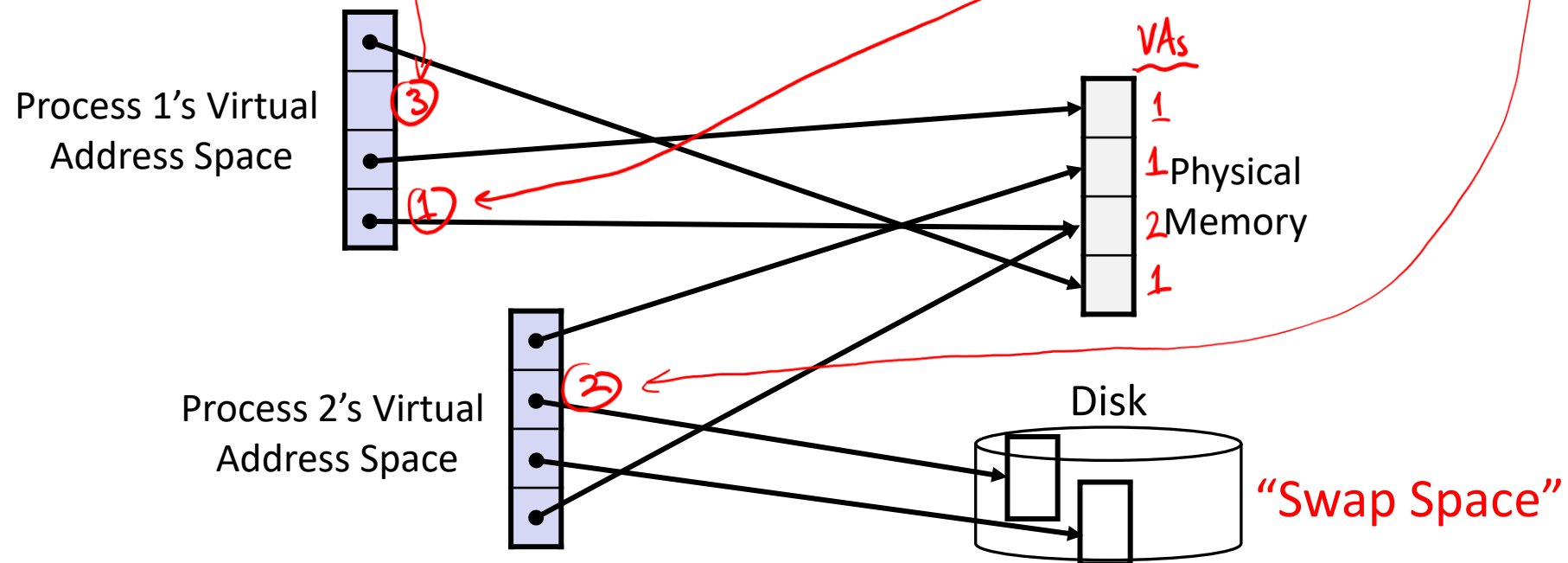
A System Using Virtual Addressing



- ❖ Physical addresses are *completely invisible to programs*
 - Used in all modern desktops, laptops, servers, smartphones...
 - One of the great ideas in computer science

Address Mapping

- ❖ A *virtual address (VA)* can map to (1) physical memory, (2) the *swap space* on disk, or (3) nothing (*i.e.*, unused VA)
 - Virtual addresses from *different* processes may map to same location
 - Every byte in main memory has 1 *physical address (PA)* and 0+ VAs



Polling Questions (3/3)

- ❖ On a 64-bit machine currently running 8 processes, how much virtual memory is currently available?

word size is 64 bits, so $n = 64$ and $N = 2^{64}$ bytes per process.

$$2^{64} \times 8 = \boxed{2^{67} \text{ bytes}} \text{ of virtual memory}$$

- ❖ True or False: A 32-bit machine with 8 GiB of RAM installed would never use all of it (in theory).

word size is 32 bits, so each process has 2^{32} bytes = 4 GiB of virtual memory

however, we have more than 1 process, so we can easily use up all 8 GiB of physical memory

note: there are other limitations, (e.g., motherboard, OS) that restrict the maximum amount of usable RAM in practice

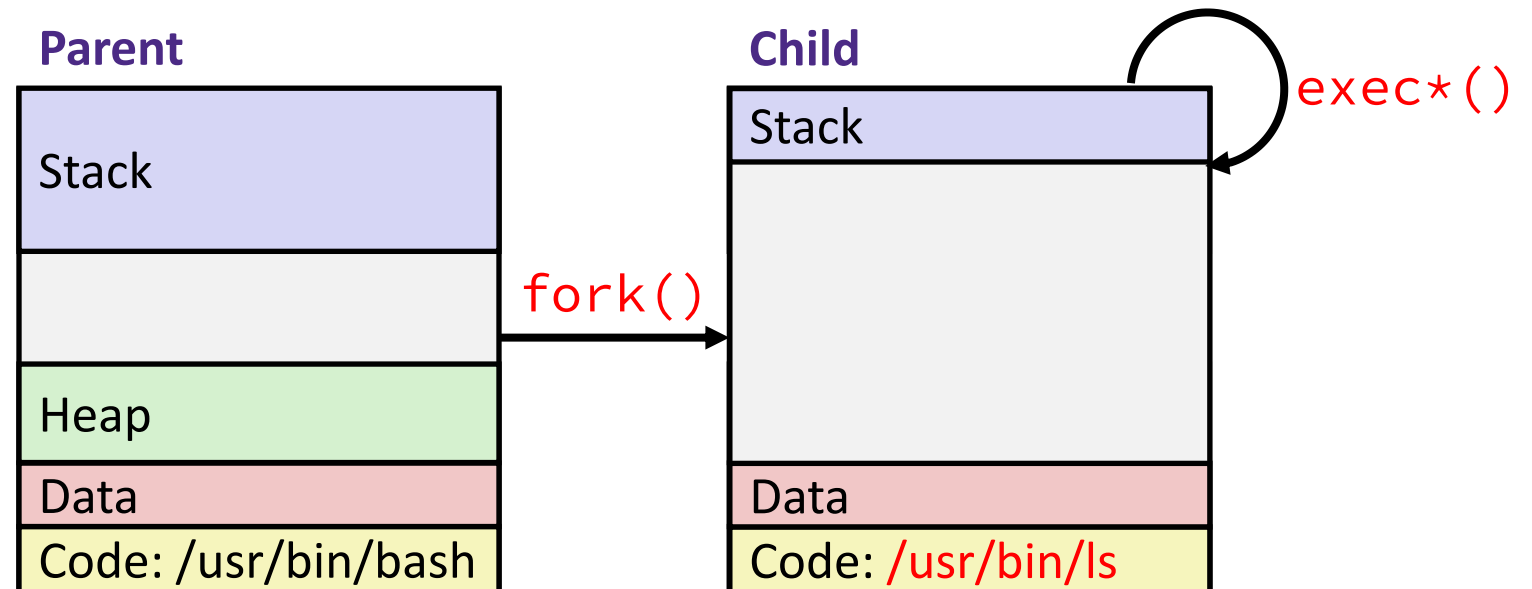
Why Virtual Memory (VM)?

- ❖ Efficient use of limited main memory (RAM)
 - Use RAM as a cache for the parts of a virtual address space
 - Some non-cached parts stored on disk, some (unallocated) non-cached parts stored nowhere
 - Keep only active areas of virtual address space in memory
 - Transfer data back and forth as needed
- ❖ Simplifies memory management for programmers
 - Each process “gets” the same full, private linear address space
- ❖ Isolates address spaces (*i.e.*, provides protection)
 - A process can’t interfere with another’s memory – *different address spaces*
 - User process cannot access privileged information – implements memory permissions (*i.e.*, different memory layout segments)

Summary (1/4)

❖ The *fork-exec model*

- Every process is assigned a unique **process ID** (pid)
- Every process has a parent process except for `init`/system (pid 1)
- `fork()` returns 0 to child, child's PID to parent
- `exec()` replaces the current process' code and address space with the code for a different program



Summary (2/4)

❖ Terminating a process

- Return from `main()` or explicit call to `exit(status)`
- Passes a ***status code*** (`main`'s return value or `exit`'s argument) to parent process
 - 0 for normal exit, nonzero for abnormal exit

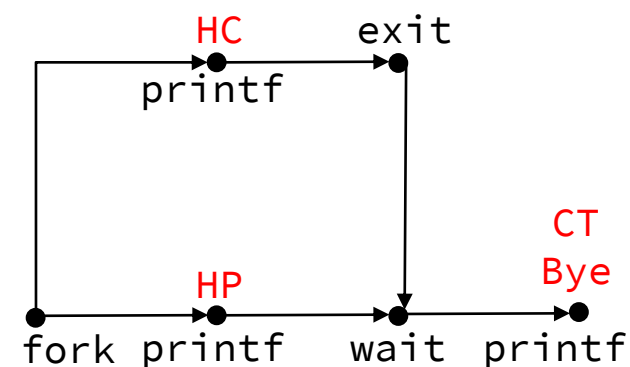
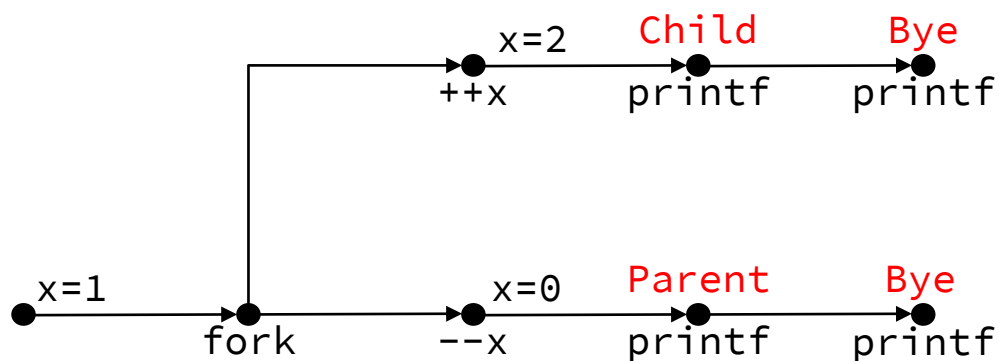
❖ Processes and resources

- A terminated (***zombie***) process still consumes system resources until ***reaped***
- Child is reaped when parent process terminates or explicitly calls `wait/waitpid`
- Orphaned children reaped by `init/systemd`

Summary (3/4)

❖ Concurrency and *process diagrams*

- Concurrently executing processes are scheduled non-deterministically by the operating system
- A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program
 - Vertices are program statements, directed edges capture sequencing *within a process*
 - Flexible visualization tool:



Summary (4/4)

- ❖ **Virtual memory** is software's perspective (e.g., memory layout), **physical memory** is hardware's perspective (e.g., memory hierarchy)
- ❖ Virtual memory manages the memory for multiple concurrently running processes
 - Each process has its own virtual address space that gets mapped into parts of the physical address space
 - When run out of physical address space, put least recently used data in disk

