Processes II & Virtual Memory I

CSE 351 Spring 2024

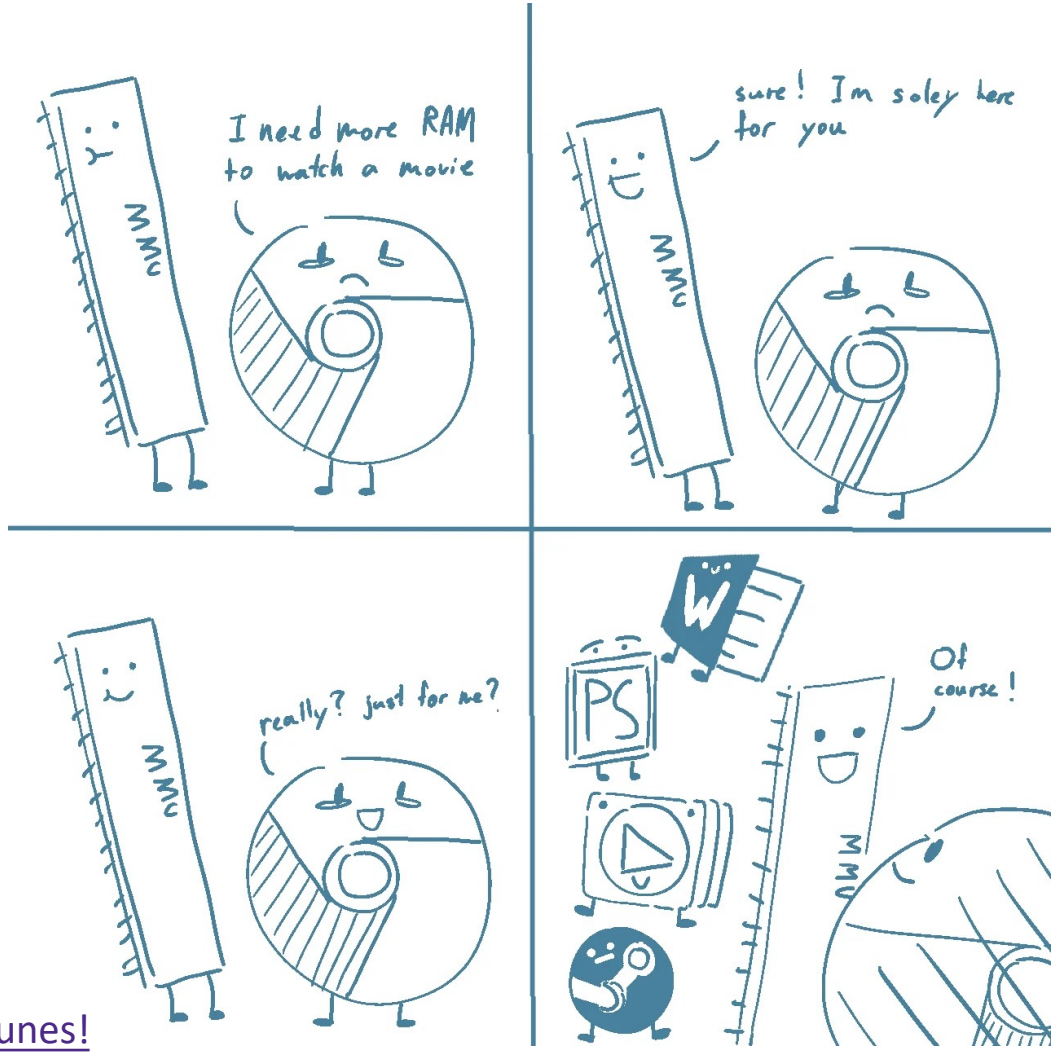
Instructor:

~~Elba Garza~~

Kelly Shaw

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

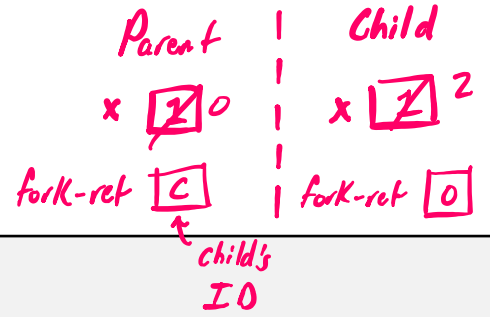


Playlist: CSE 351 24Sp Lecture Tunes!

Announcements, Reminders

- Elba (14 May): Given changing situation, follow any Ed announcements regarding updates to assignments and due dates!

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);
}
```

splits here

// child only

// parent only

// both! :)

- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can't predict execution order of parent and child
- Both processes start with `x` = `1`
  - Subsequent changes to `x` are independent
- Shared open files: `stdout` is the same in both parent and child

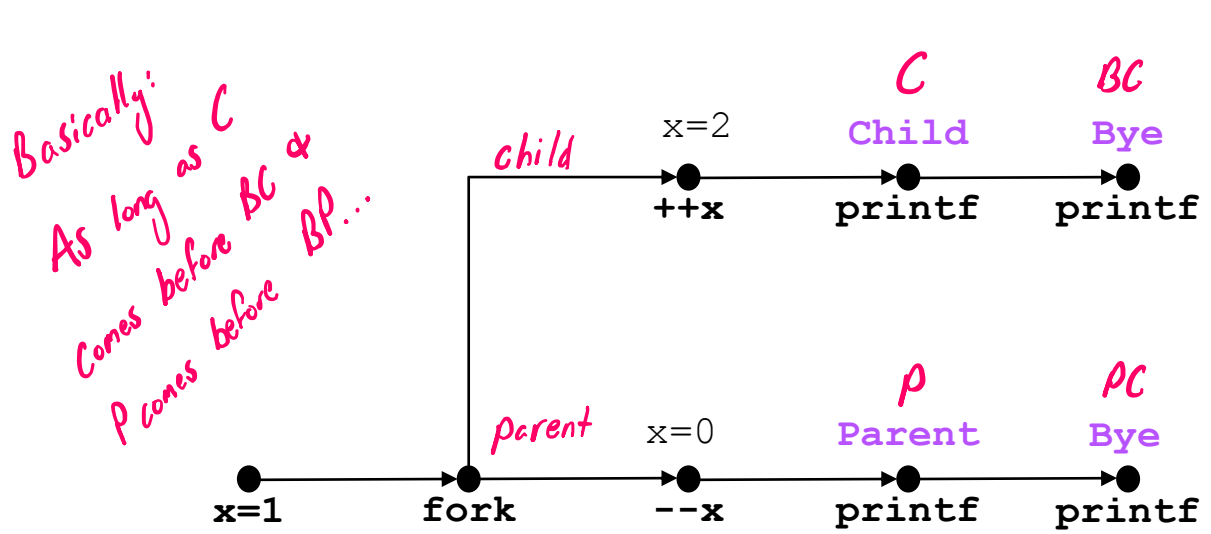
Modeling `fork` with Process Graphs

- A **process graph** is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - `a` → `b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no in-edges

- Any *topological sort* of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right

Fork Example: Possible Output

```
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 etc.
P C BC BP
BP BC BP BC

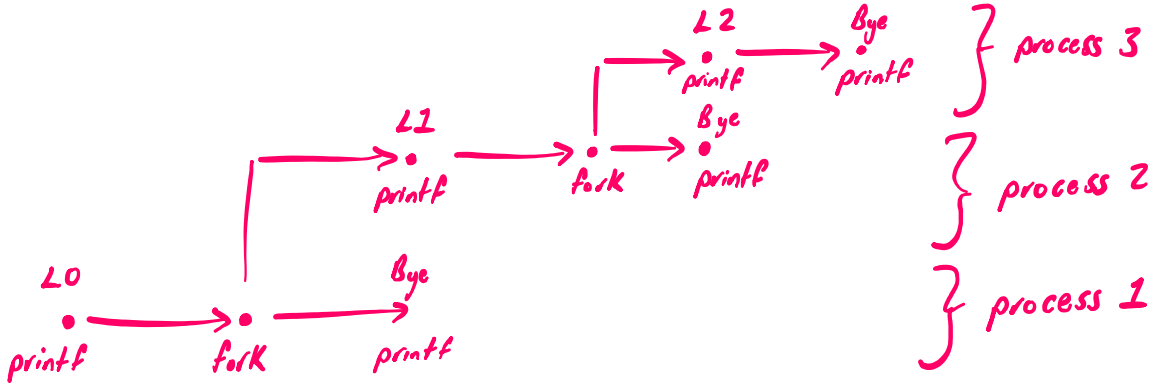
Not Possible

C P
BC BC etc.
BP C
P BP

Polling Question

- 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:	Seq 2:
	L0	L0 ← Process 1
	L1	Bye ← Process 1
	Bye	L1 ← Process 2
	Bye	L2 ← Process 3
	Bye	Bye ← Process 2/3
	L2 !!!!	Bye ← Process 3/2
A.	No	No
B.	No	Yes
C.	Yes	No
D.	Yes	Yes
E.	We're lost…	

Reading Review

- Terminology:
  - `exec*()`, `exit()`, `wait()`, `waitpid()`
  - `init`/`systemd`, reaping, zombie processes
  - Virtual memory: virtual vs. physical addresses and address space, swap space

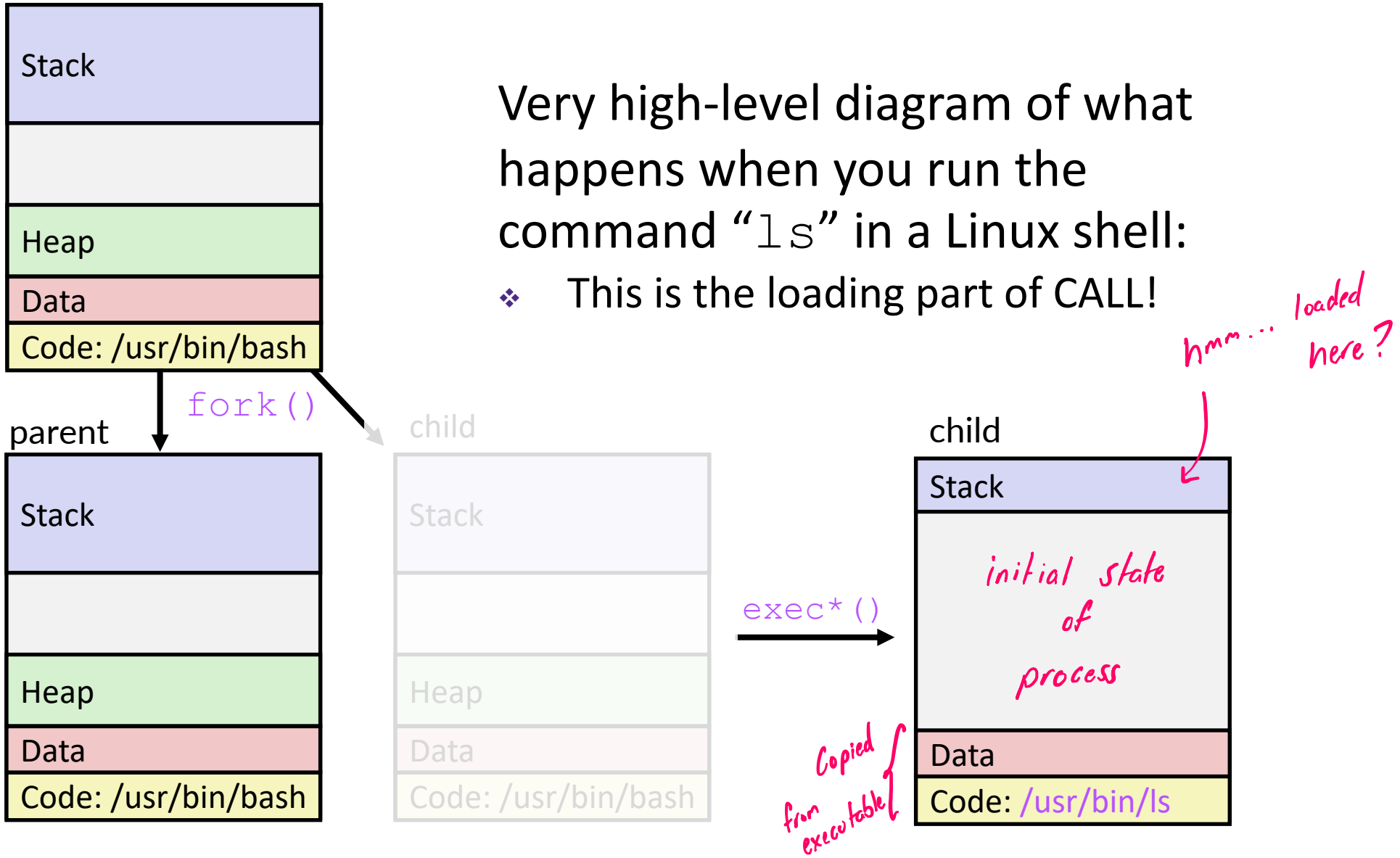
Fork-Exec

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!

Processes

- Processes and context switching
- Creating new processes
  - `fork()` and `exec*()`
- **Ending a process**
  - **`exit()`, `wait()`, `waitpid()`**
  - **Zombies**

`exit`: Ending a process

- **`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

Zombies

- 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
    - if `NULL`, that means the status was ignored
    - Special macros for interpreting this status – see **`man 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

wait: Synchronizing with Children

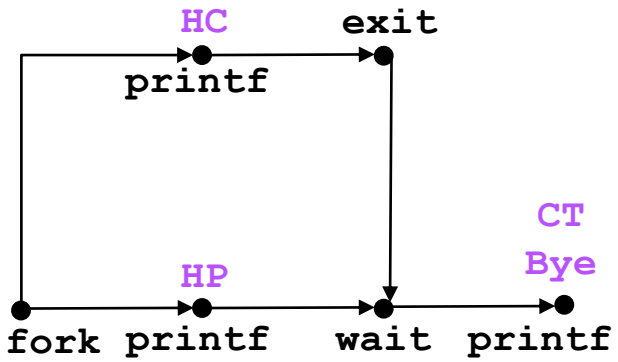
forks.c

```
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

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

parent sticks around

forks.c

```
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
```

parent

child

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`

Example: Non-terminating Child

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

Child sticks around

forks.c

```
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
```

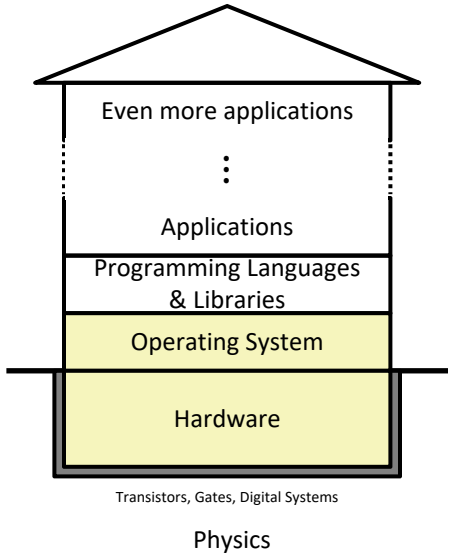
- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Process Management Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Two-process program:
    - First `fork()`
    - **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }
  - Two different programs:
    - First `fork()`
    - **if** (pid == 0) { execv(…) } **else** { */* parent code */* }
- `exit` or `return` from `main` to end a process
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

The Hardware/Software Interface

- Topic Group 3: **Scale & Coherence**
  - Caches, Processes, **Virtual Memory**, Memory Allocation



- 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?

Virtual Memory (VM*)

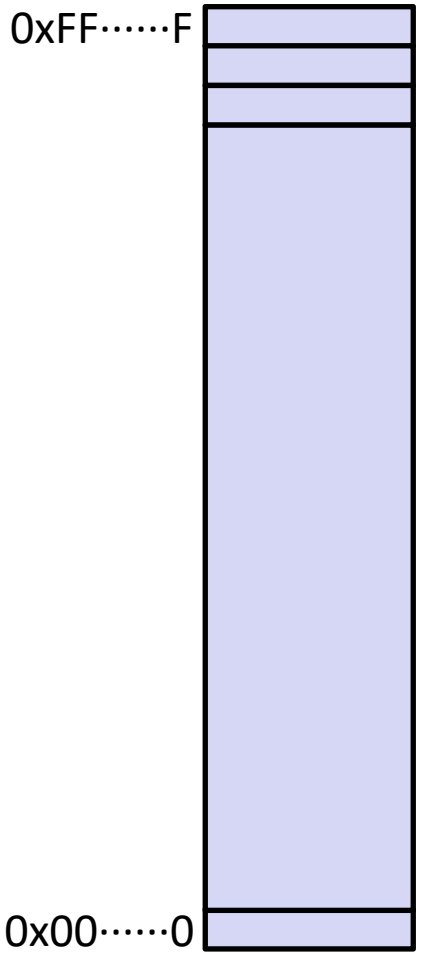
- **Overview and motivation**
- **VM as a tool for caching**
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection

Warning: Virtual memory is pretty complex, but crucial for understanding how processes work and for debugging performance

**Not to be confused with "Virtual Machine" which is a whole other thing.*

Memory as we know it so far… is *virtual!*

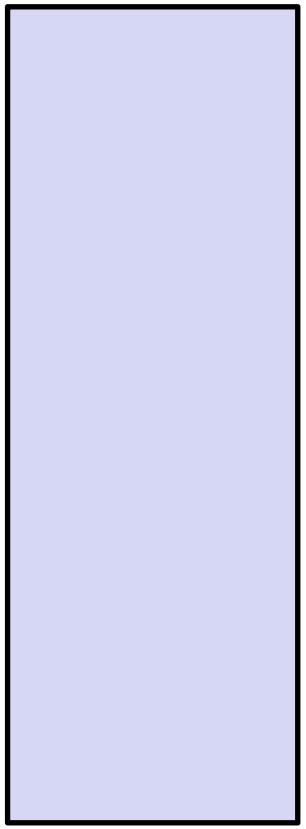
- Programs refer to virtual memory addresses
  - `movq (%rdi),%rax`
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process
- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space
- But…
  - We *probably* don't have 2^w bytes of physical memory
  - We *certainly* don't have 2^w bytes of physical memory ***for every process***
  - Processes should not interfere with one another
    - Except in certain cases where they want to share code or data

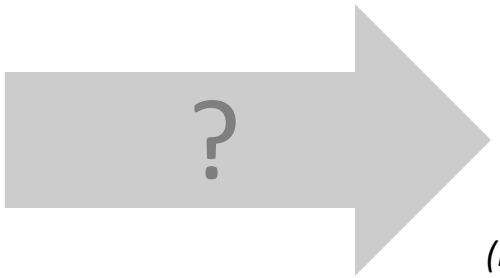


Problem 1: How Does Everything Fit?

64-bit virtual addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

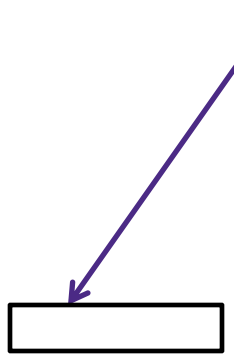
16 EiB!!!





Physical main memory offers
a few gigabytes
(*e.g.*, 8,589,934,592 bytes)

8 GiB

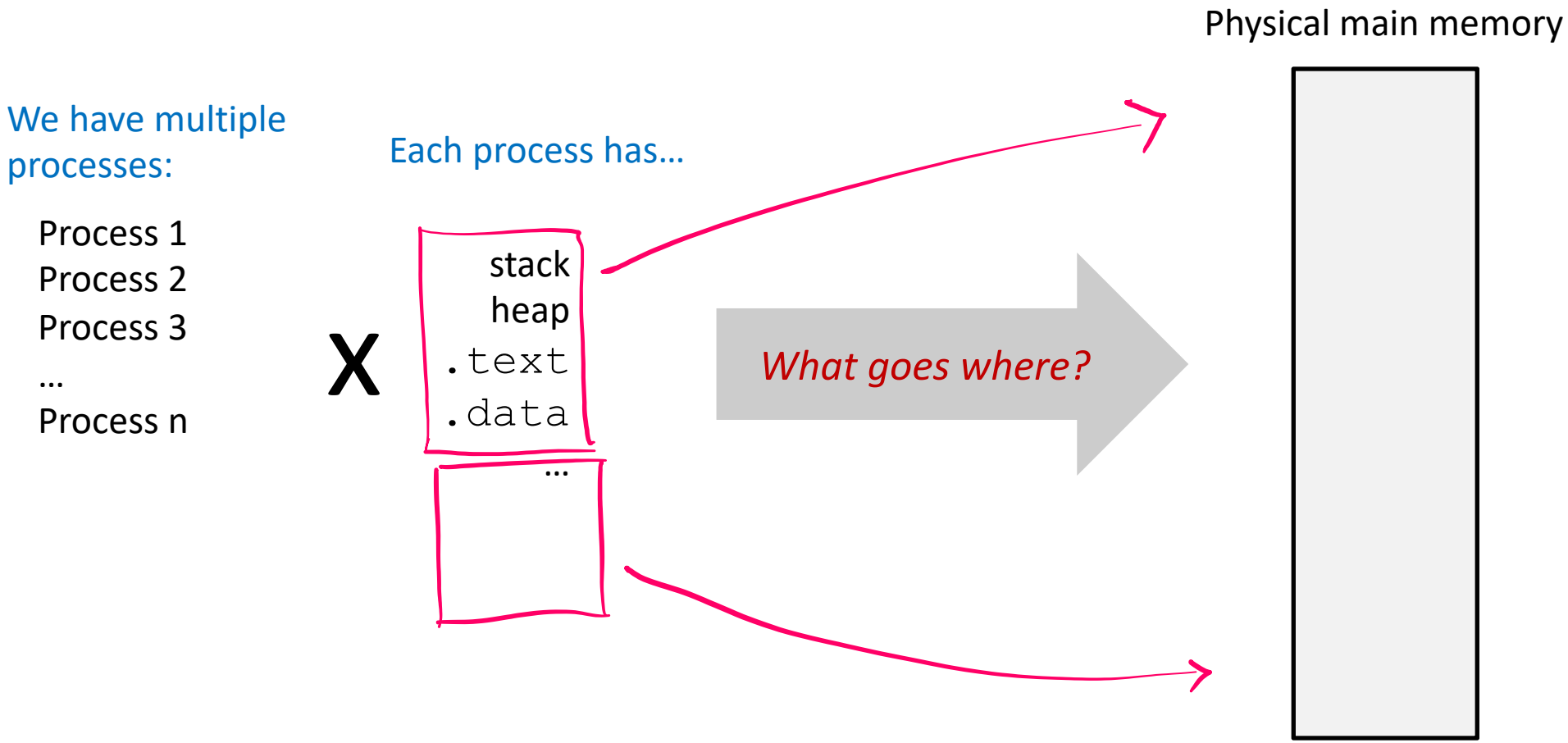


(Not to scale; physical memory would be smaller than the period at the end of this sentence compared to the virtual address space.)

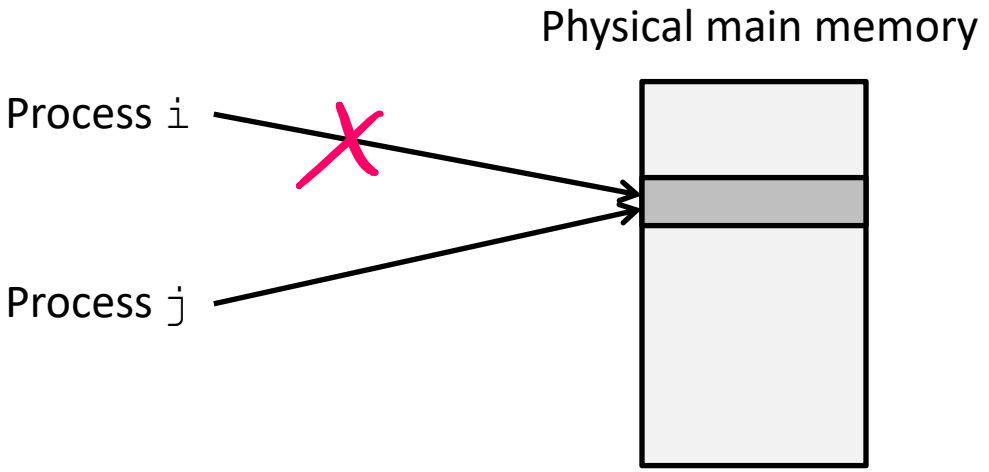
actually, smaller...

1 virtual address space per process, with many processes…

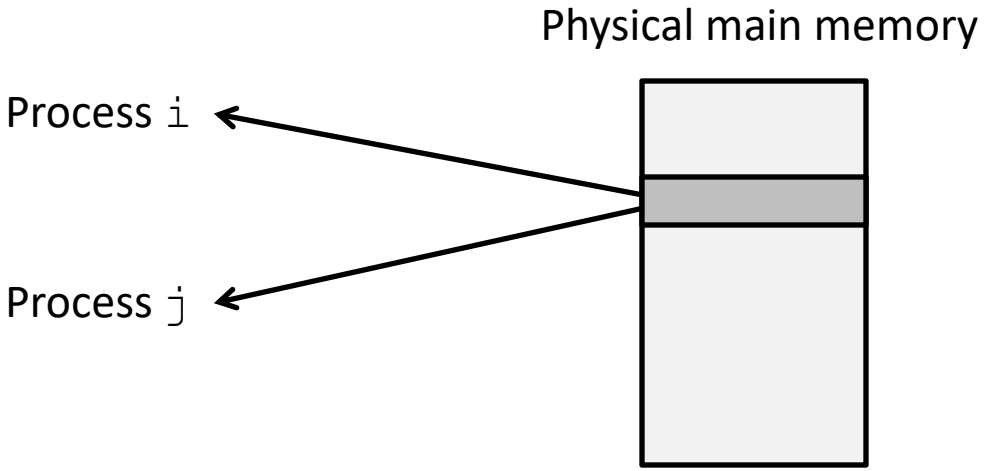
Problem 2: Memory Management



Problem 3: How To Protect



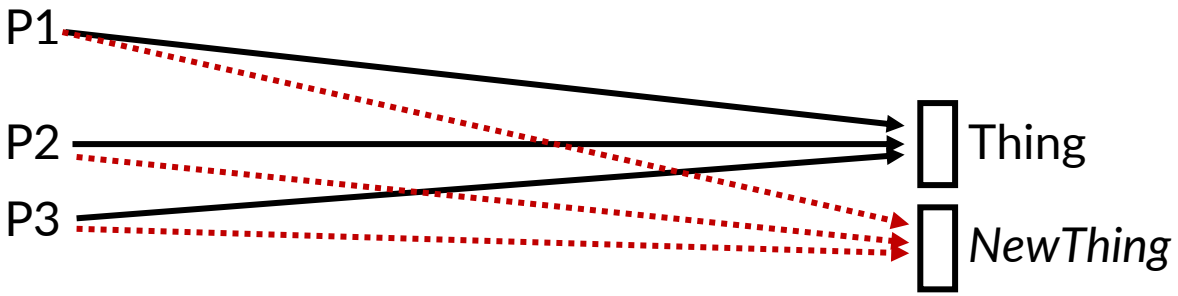
Problem 4: How To Share?



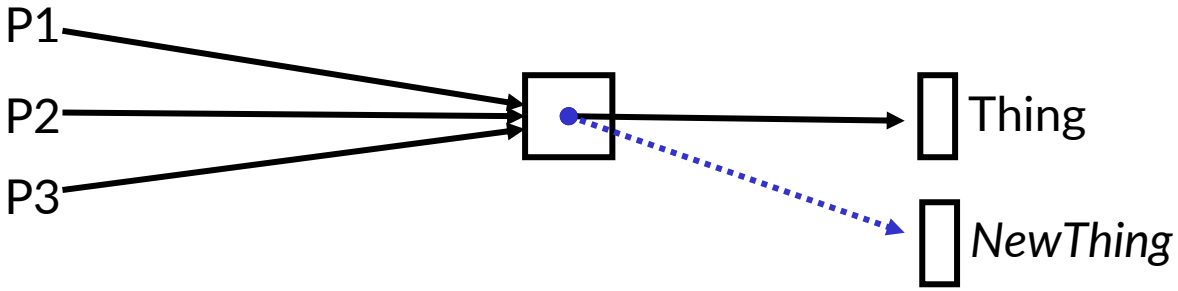
How can we solve these problems?

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

- Without Indirection



- With Indirection

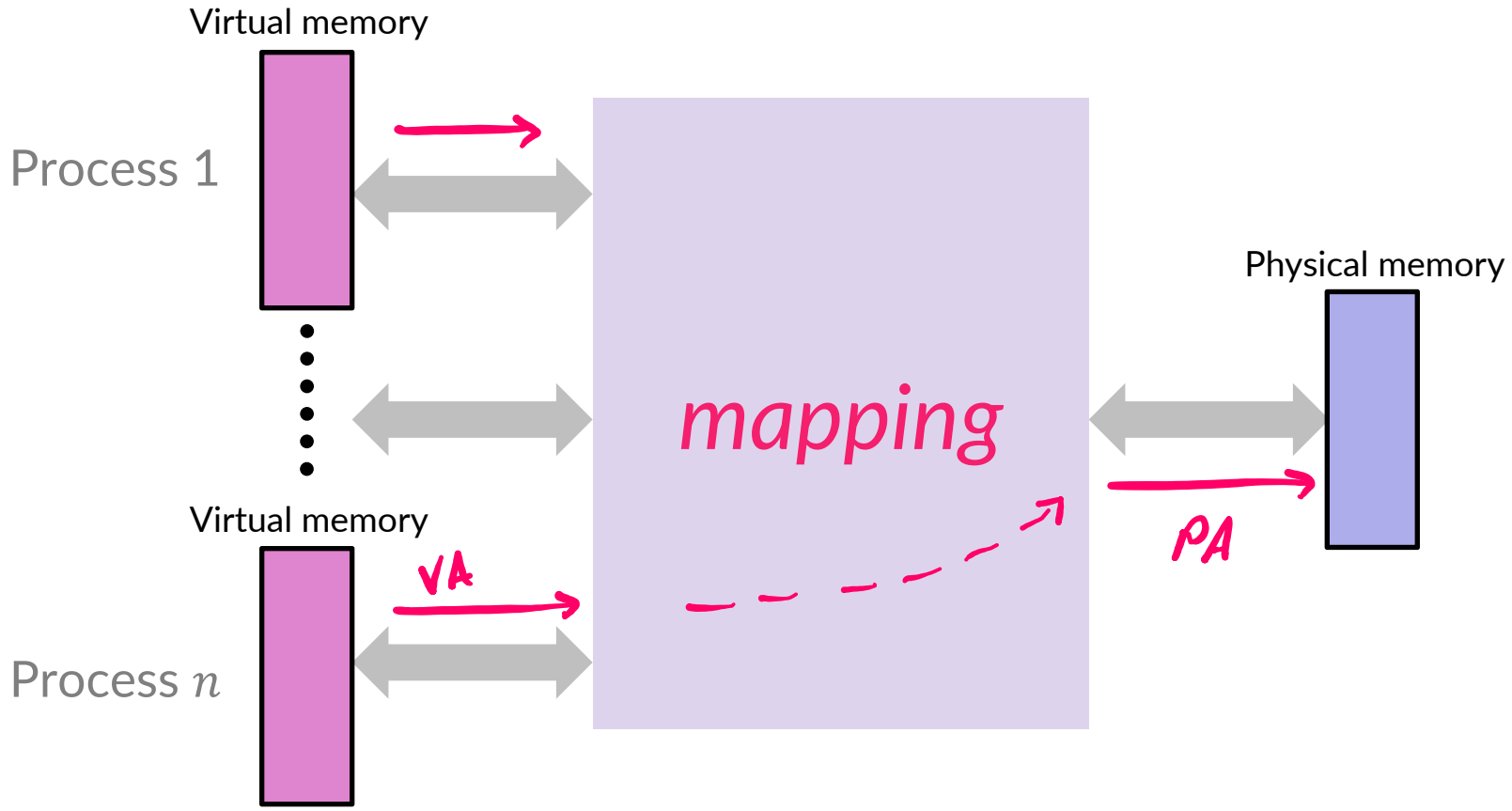


What if I want to move Thing?

Indirection

- ***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.
  - Con: Adds some work (now have to look up 2 things instead of 1)
  - Pro: But don't have to track all uses of name/address (single source!)
- Examples:
  - **Phone system:** cell phone number portability
  - **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

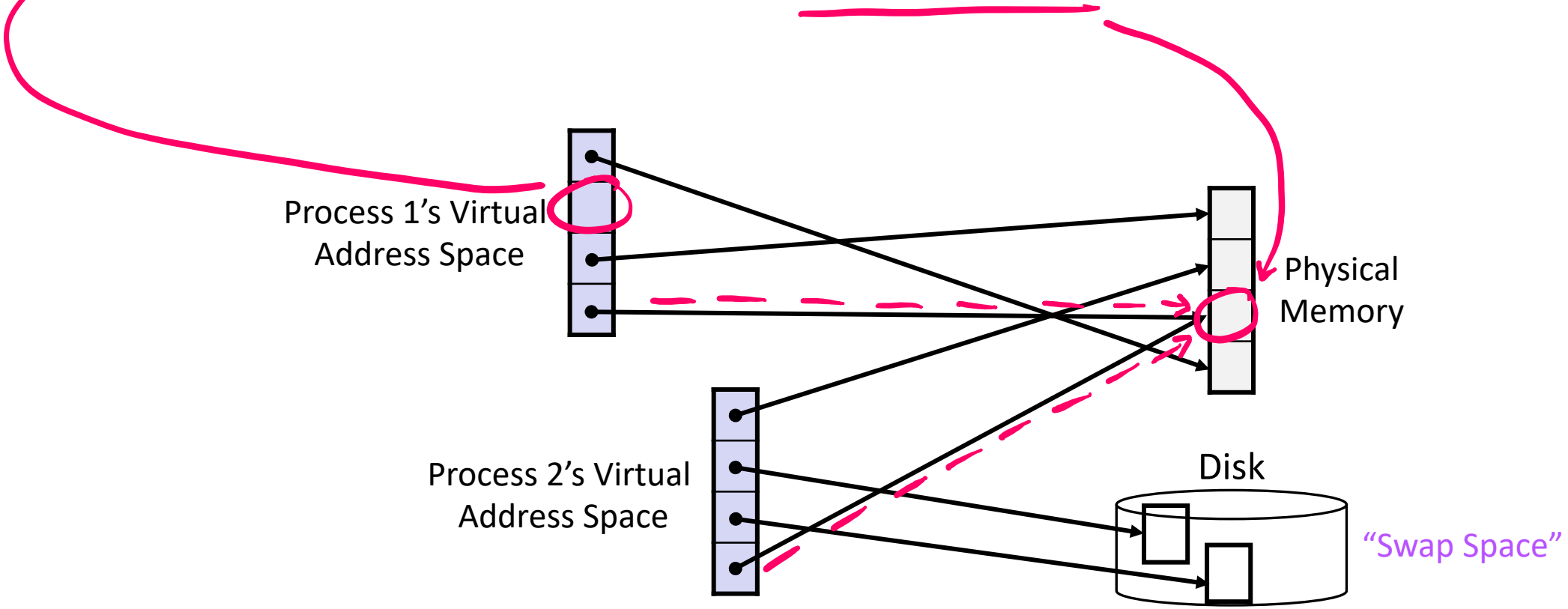
Indirection in Virtual Memory



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

Mapping

- A virtual address (VA) can be mapped to either physical memory or disk
  - Unused VAs may not have a mapping
  - VAs from *different* processes may map to same location in memory/disk



Address Spaces

- Virtual address space: Set of $N = 2^n$ virtual addr
  - {0, 1, 2, 3, …, N-1}
- Physical address space: Set of $M = 2^m$ physical addr
  - {0, 1, 2, 3, …, M-1}

$n = \lceil \log_2 N \rceil$ ← ceiling function aka round up!

$m = \lceil \log_2 M \rceil$

- Every byte in main memory has:
  - one physical address (PA)
  - zero, one, *or more* virtual addresses (VAs)

unused

used by a process

used by many processes

Polling Questions

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

word size = 64 bits = n $\Rightarrow N = 2^{64}$ bytes per process.

$\therefore 2^{64} \times 8 = 2^{67}$ bytes

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

word size = 32 bits = n $\Rightarrow N = 2^{32}$ bytes per process.

= 4 GiB per process.

Just having more than 2 processes means we're in trouble!

FALSE!!

Summary

- Virtual memory provides:
  - Ability to use limited memory (RAM) across multiple processes
  - Illusion of contiguous virtual address space for each process
  - Protection and sharing amongst processes

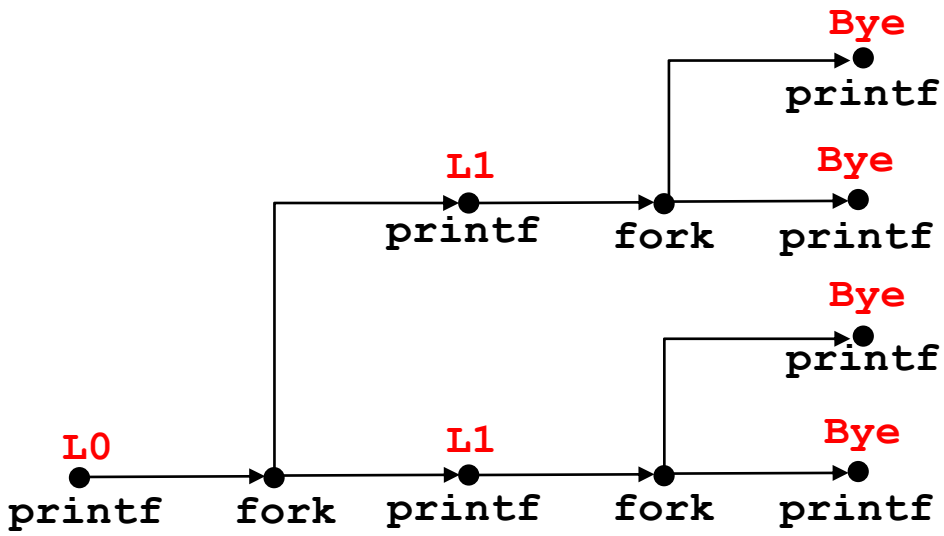
BONUS SLIDES

Detailed examples:

- Consecutive forks
- `wait()` example
- `waitpid()` example

Example: Two consecutive `forks`

```
void fork2() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



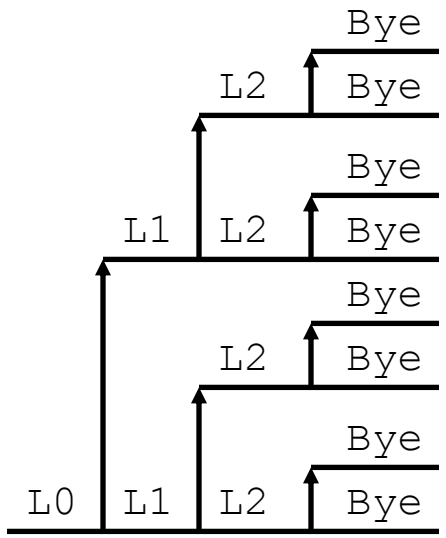
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

Example: Three consecutive `forks`

- Both parent and child can continue forking

```
void fork3() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

`waitpid()`: Waiting for a Specific Process

`pid_t waitpid(pid_t pid, int &status, int options)`

- suspends current process until specific process terminates
- various options (that we won't talk about)

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```