Processes II & Virtual Memory I

Lecture Topics

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

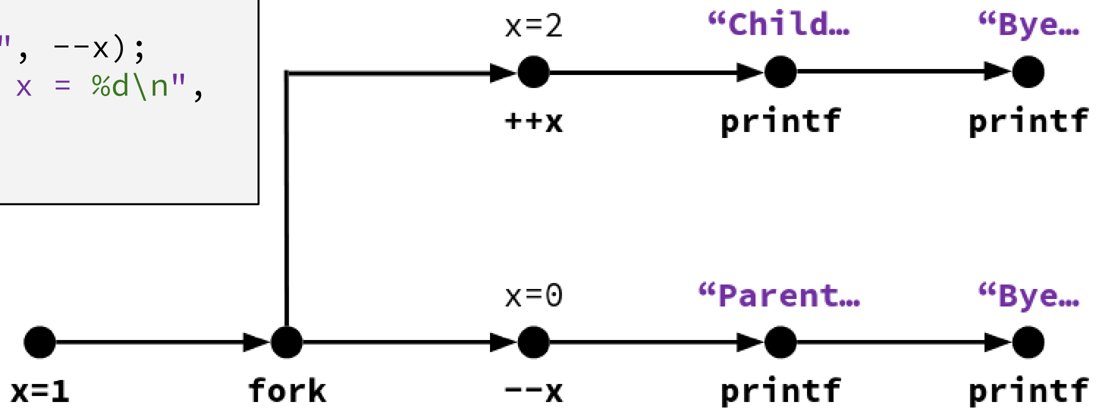
`fork` Example

```
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);
}
```

- Both parent and child start/continue execution after `fork`
- Child gets a *copy* of parent's data - both processes start with `x = 1`
  - Subsequent changes to `x` are **independent**
- Shared open files - `stdout` is the same for both
- **Can't predict execution order** of parent and child - up to the OS!

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);
}
```



Polling Question

Which of the two sequences of outputs are possible?

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

Seq1	Seq2
L0	L0
L1	Bye
Bye	L1
Bye	L2
Bye	Bye
L2	Bye

	Seq1	Seq2
A)	No	No
B)	No	Yes
C)	Yes	No
D)	Yes	Yes

Fork-Exec

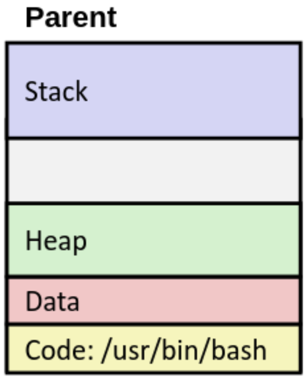
- `fork()` creates a copy of the current process
- `exec*()` replaces the current process' code and address space with the code for a different program
  - Whole family of `exec` calls – see `exec(3)` and `execve(2)`

```
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 Program

Very high-level diagram of what happens when you run the command “`ls`” in a Linux shell

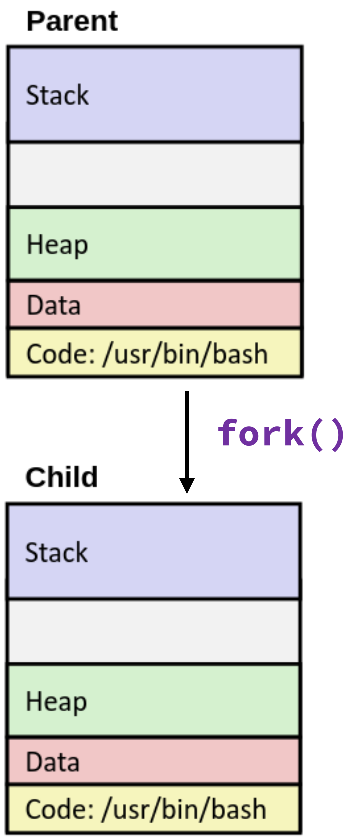
- This is the loading part of CALL!



exec-ing a Program (pt 2)

Very high-level diagram of what happens when you run the command “`ls`” in a Linux shell

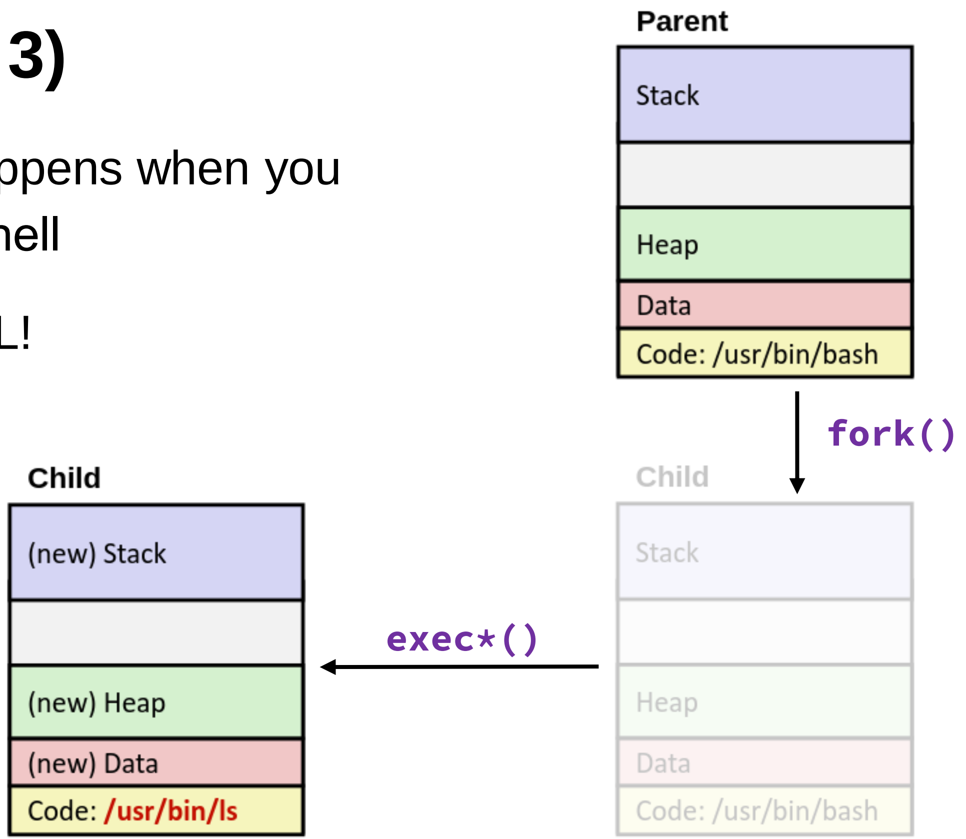
- This is the loading part of CALL!



exec-ing a Program (pt 3)

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 Topics

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

`exit`: Exiting a Process

- `void exit(int status)`
  - Explicitly exits a process
    - Status code: 0 = normal exit, nonzero = abnormal exit
- The `return` statement from `main()` also exits a process
  - The return value is the status code

- Terminated processes still take up system resources
  - Data structures maintained by the OS
  - A process can’t clean up all of its own resources when it exits, so whose responsibility is it?

Zombies

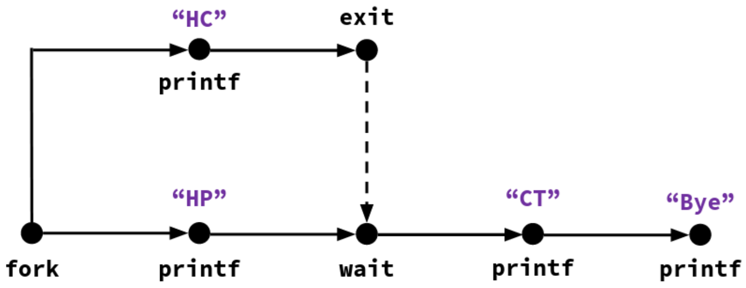
- A terminated process that is still consuming resources is called a **zombie**
- Parent needs to **reap** its zombie children (i.e. clean up its resources)
  - Parent is given exit status information, then transfers control to the OS to delete zombie process
- What if the parent exits before reaping the child?
  - Orphaned child is reaped by `init` process (process 1)
    - Note: on recent Linux systems, `init` has been renamed to `systemd`

wait: Synchronizing with Children

- `int wait(int* child_status)`
  - Suspends the current process until one of its children terminates
    - Reaps that child, then returns its PID
  - 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` Example

```
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");
}
```



Feasible output:	Infeasible output:
HC	HP
HP	CT
CT	Bye
Bye	HC

wait Example 2: Zombies

```c
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 */
    }
}
```

```
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
```

Zombie child is still there

Need to kill parent for init to reap the child

`wait` Example 3: Non-terminating Child

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

Must explicitly kill the child, or it will run forever.

Lecture Topics

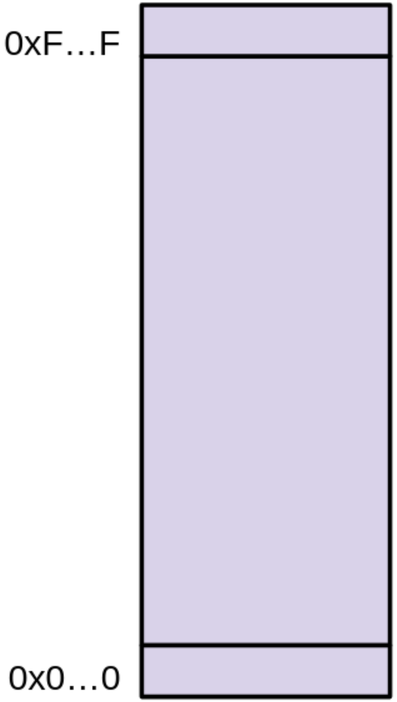
- Processes and context switching
  - Creating new processes
    - fork() and exec*()
  - Ending a process
    - exit(), wait(), waitpid()
    - Zombies
- **Virtual Memory (VM*)**
  - **Overview and motivation**
  - **VM as a tool for caching**
  - Address translation
  - VM as a tool for memory management
  - 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*

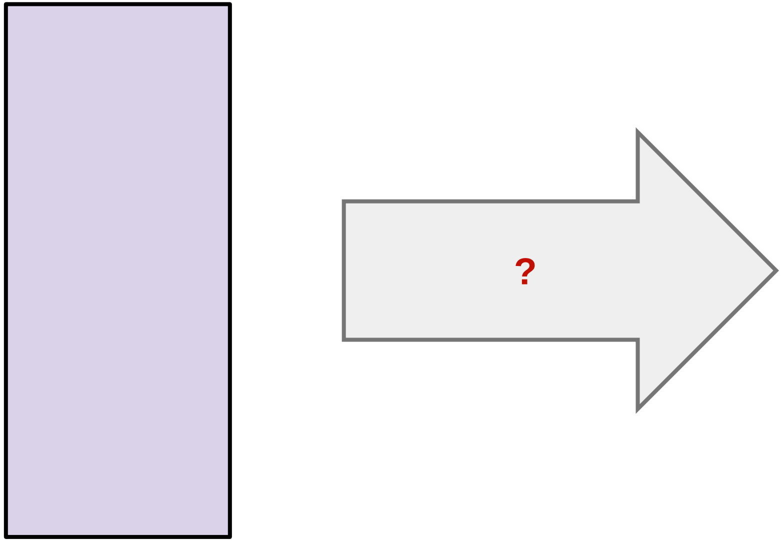
Memory as we know it so far… is *virtual*!

- Programs refer to **virtual** memory addresses
  - System provides private addresses for 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 *definitely* don't have 2^w bytes of physical memory ***for every process***
  - Processes should not interfere with each other
    - Except for specific cases where they want to share code or data

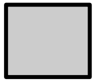


Problem 1: How does everything fit?

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



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



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

Problem 2: Memory Management

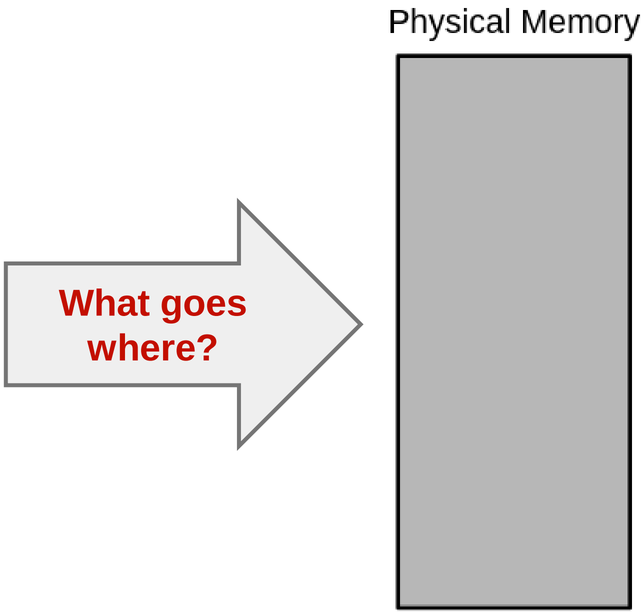
We have multiple processes:

Process 1
Process 2
Process 3
…
Process n

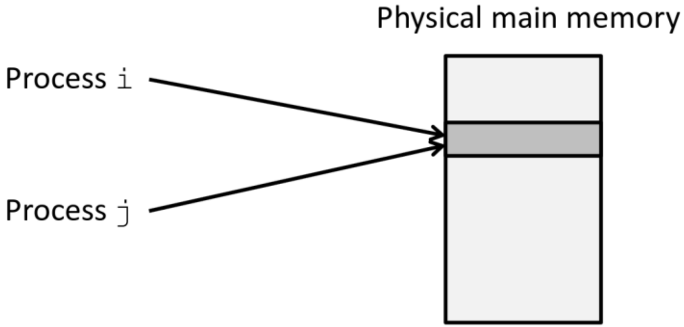
×

Each process has:

Stack
Heap
Static Data
Literals
Code
…



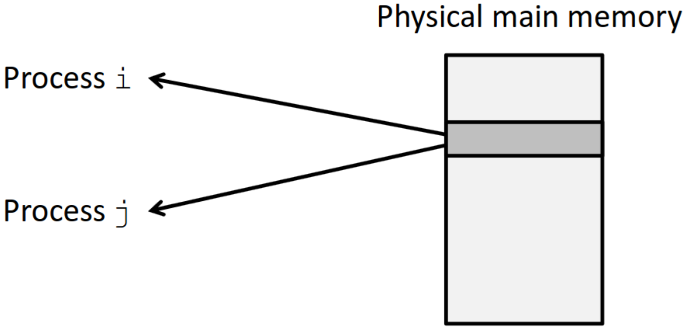
Problem 3: How to protect data?



What if two running programs both use the same address in their code?

We want to make sure processes don't access the same physical memory locations.

Problem 4: How to share data?



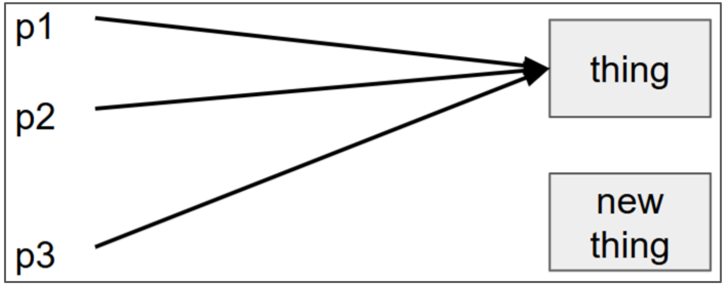
… Except sometimes we *do* want them to share memory!

- Inter-process communication
- Shared code
- etc.

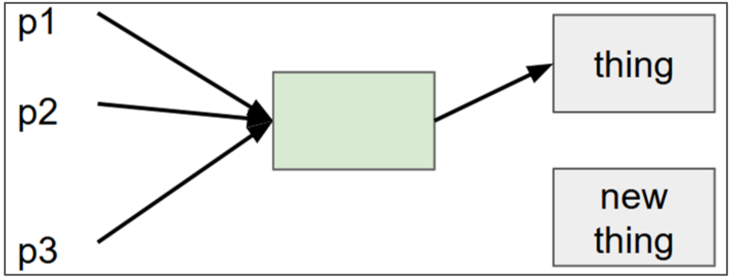
How can we solve these problems?

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

Without indirection:



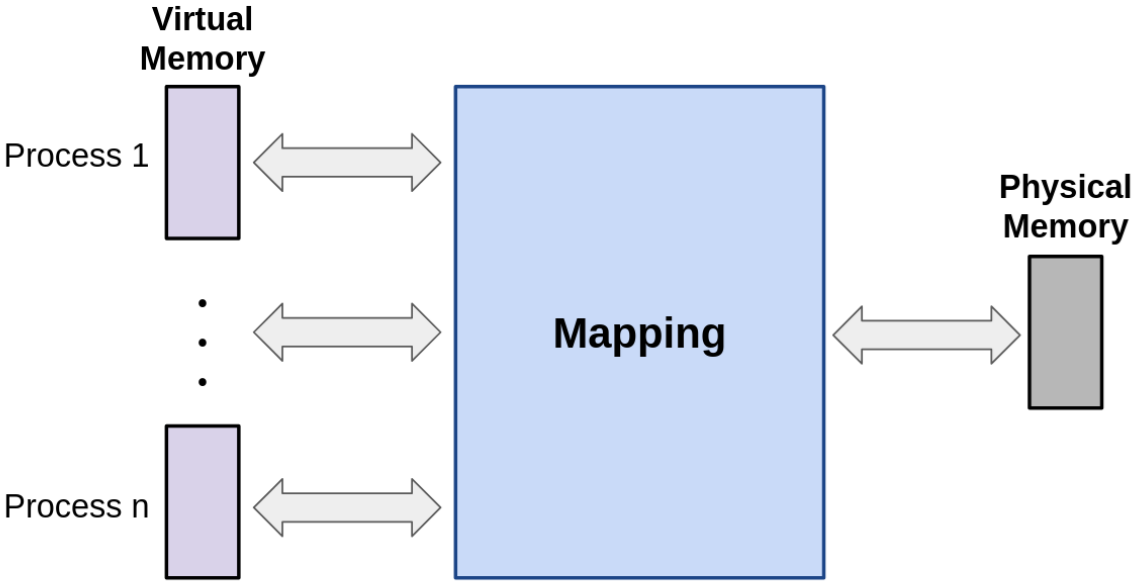
With indirection:



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 have to look up 2 things instead of 1)
    - But don’t have to track all uses of name/address (single source!)
- <u>Examples</u>:
    - Phone system: cell phone number portability
    - Domain Name Service (DNS): translation from name to IP address
    - Call centers: route calls to available operators, etc.

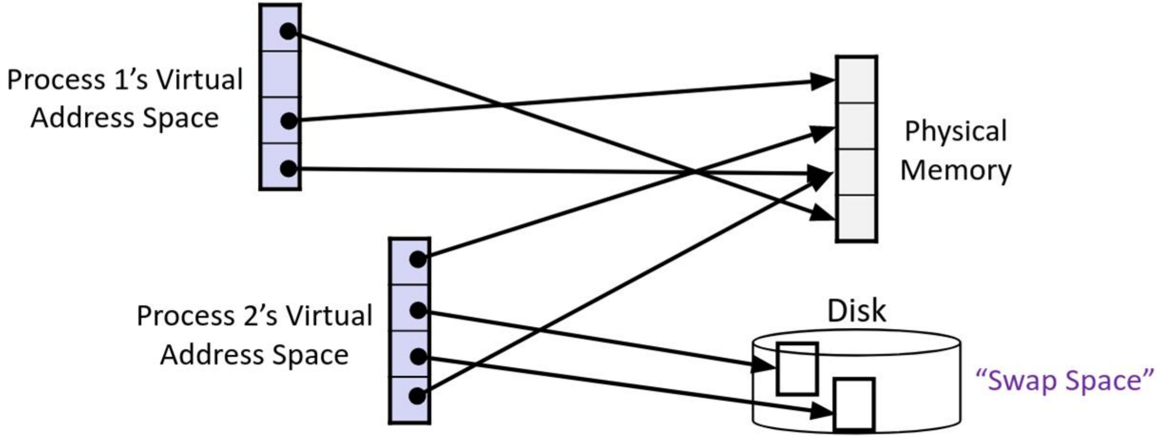
Indirection in Virtual Memory



- Each process gets its own private address space
  - Translates to some location in physical memory
- Solves previous problems!

Mapping

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



Address Spaces

- **Virtual Address Space:** Set of $N = 2^n$ virtual addresses
  - $\{0, 1, \ldots, N\text{-}1\}$
  - Corresponds to word size (so in x86-64, n = 64)
- **Physical Address Space:** Set of $M = 2^m$ physical addresses
  - $\{0, 1, \ldots, M\text{-}1\}$
  - Address length *m* depends on hardware
- Every byte in main memory has:
  - **One** physical address (PA)
  - Zero, one, or *more* virtual addresses (VAs)

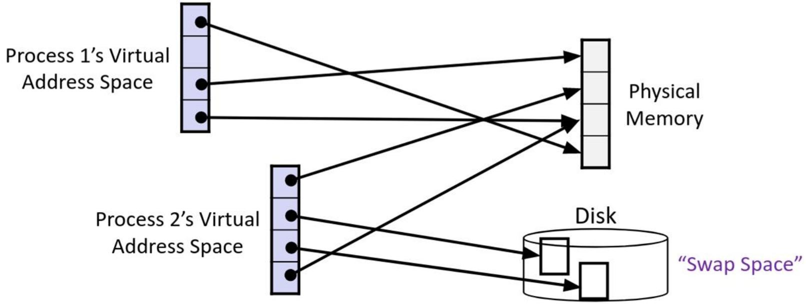
Review Questions

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

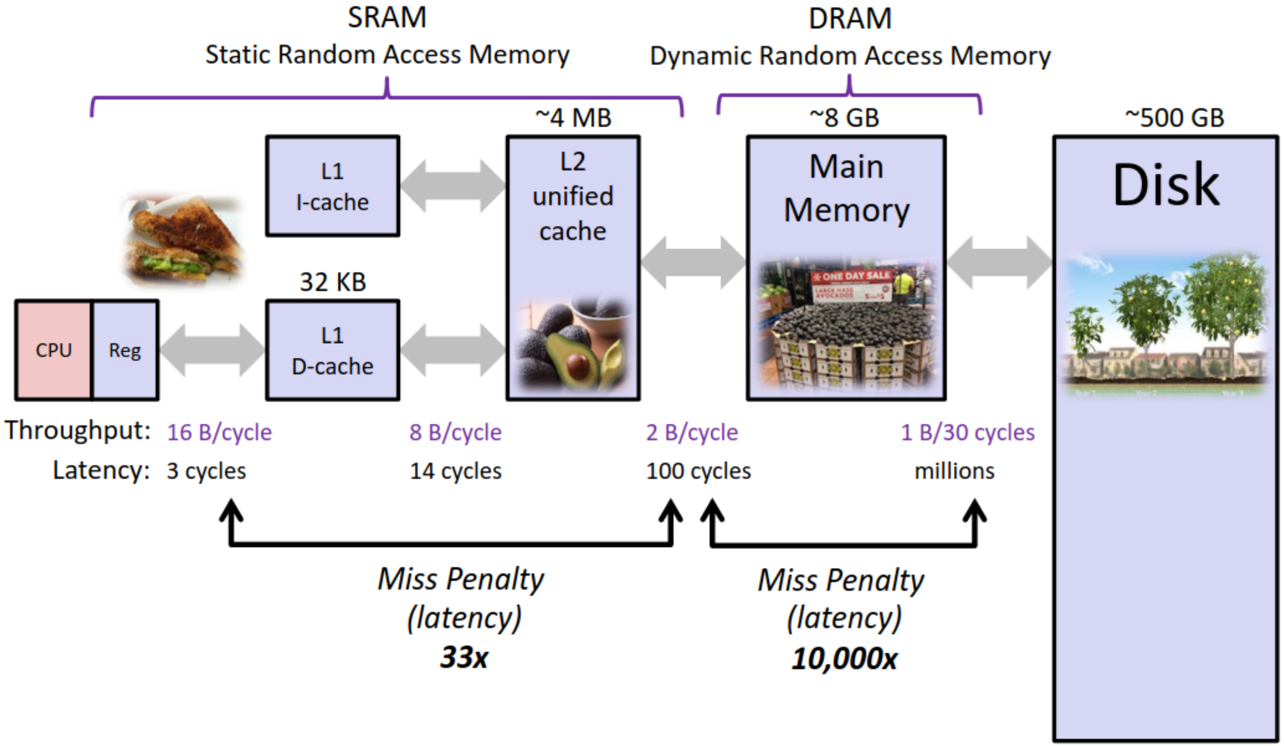
2. True or False: A 32-bit machine with 8 GiB of RAM installed would never use all of it (in theory).

VM and the Memory Hierarchy

- Think of memory (virtual or physical) as an array of bytes, now split into **pages**
  - Pages aligned (size is P = 2^p bytes), similar to cache blocks
  - Each virtual page can be stored in any physical page (no fragmentation!)
- Pages of virtual memory are usually stored in physical memory, but spill to disk when we run out of space
  - *Kind of like a cache!*



Memory Hierarchy: Core 2 Duo



Virtual Memory Design Consequences

- **Large page size:** typically 4-8 KiB or 2-4 MiB
    - Can be up to 1 GiB (for “Big Data” apps on big computers)
    - Much larger than cache blocks
- **Fully associative**
    - Any virtual page can be placed in any physical page
- Highly sophisticated, expensive **replacement algorithms** in OS
    - Too complicated and open-ended to be implemented in hardware
- **Write-back** rather than write-through
    - *Really* don’t want to write to disk every time we modify memory
    - Some things may never end up on disk (e.g., stack for short-lived process)

Why does VM work on RAM/disk?

- Avoids disk accesses because of *locality*
  - Same reason that L1 / L2 / L3 caches work
- The set of virtual pages that a program is “actively” accessing at any point in time is called its **working set**
  - If (working set of one process ≤ physical memory):
    - Good performance for one process (after compulsory misses)
  - If (working sets of all processes > physical memory):
    - **Thrashing**: Performance meltdown where pages are swapped between memory and disk continuously
- *This is why your computer can feel faster when you add RAM*

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)
- `exit` or `return` from `main` to end a process
- `wait` or `waitpid` used to synchronize parent/child execution and to **reap** child

- **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