Processes II

CSE 351 Winter 2021

Instructor:

Mark Wyse

Teaching Assistants:

Kyrie Dowling

Catherine Guevara

Ian Hsiao

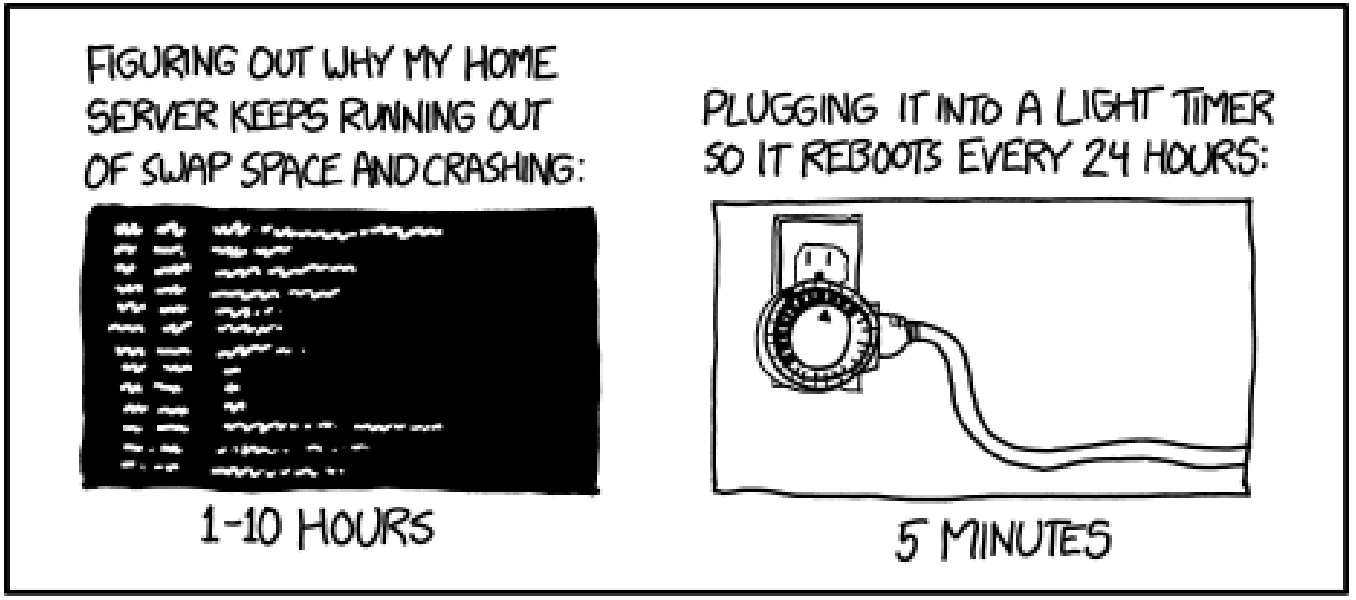
Jim Limprasert

Armin Magness

Allie Pfleger

Cosmo Wang

Ronald Widjaja



https://xkcd.com/1495/

Administrivia

- hw17 due tonight!
- hw18 due Friday, hw19 due Monday
- Study Guide 2 due Monday 3/1
- Lab 4 due Friday 3/5

Reading Review

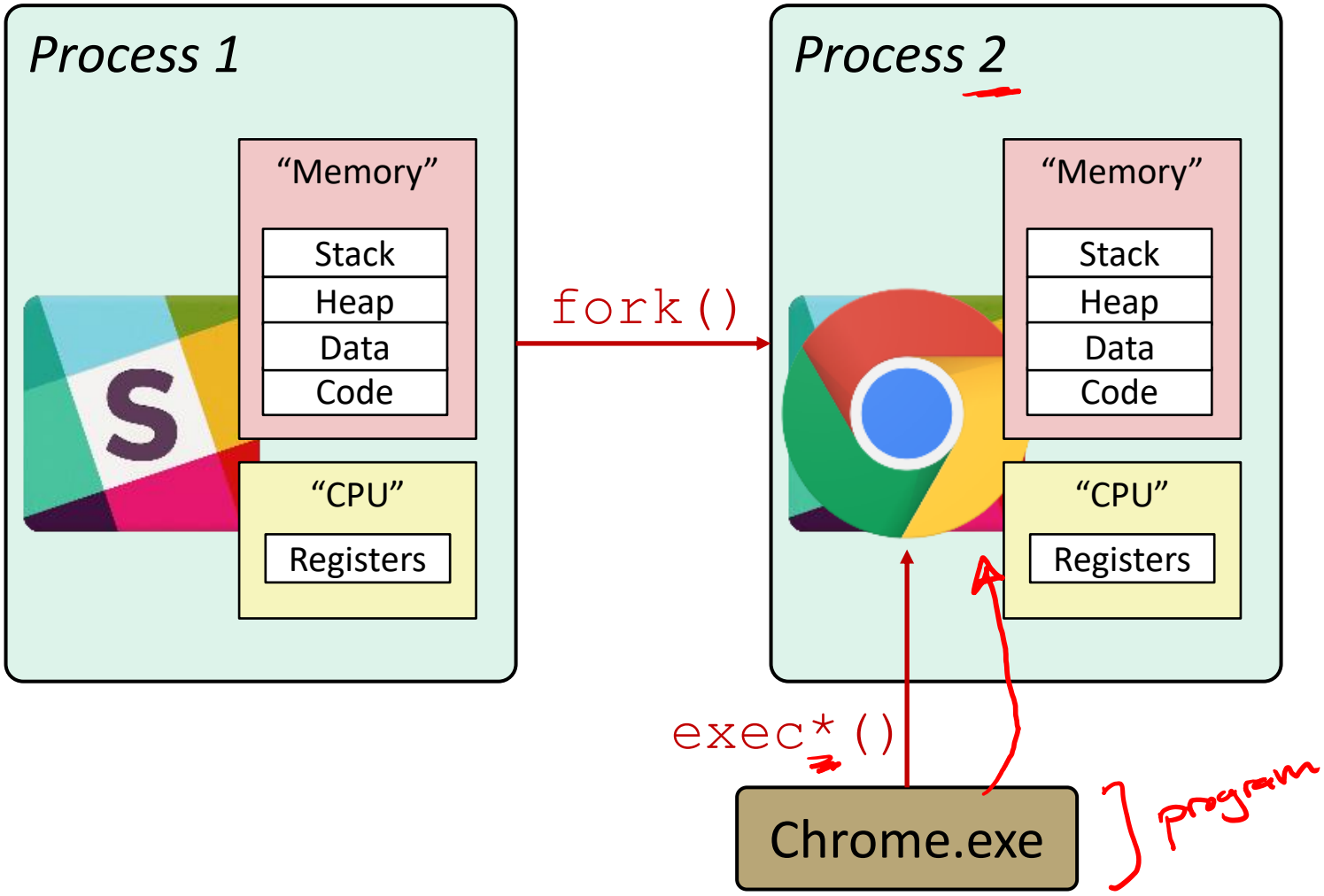
- Terminology:
  - `exec*()`, `exit()`, `wait()`, `waitpid()`
  - `init`/`systemd`, reaping, zombie processes
  - Virtual memory: virtual vs. physical addresses and address space, swap space

- Questions from the Reading?

Processes

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

Creating New Processes & Programs



Creating New Processes & Programs

- fork-exec model (Linux):
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*

- Other system calls for process management:
  - `getpid()`
  - `exit()`
  - `wait()`, `waitpid()`

`fork`: Creating New Processes

- **`pid_t`** `fork(`**`void`**`)`
  - Creates a new "child" process that is *identical* to the calling "parent" process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child's process ID (PID) to the parent process
- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent's virtual address space
  - Child has a different PID than the parent

```
pid_t pid = fork();
if (pid == 0) {
  printf("hello from child\n");
} else {
  printf("hello from parent\n");
}
```

- `fork` is unique (and often confusing) because it is called once but returns "twice"

Understanding `fork()`

Process X (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding `fork()`

Process X (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = Y

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Process Y (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork_ret = 0

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Understanding `fork()`

Process X (parent; PID X)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork ret = Y

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

hello from parent

Process Y (child; PID Y)

```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

fork ret = 0

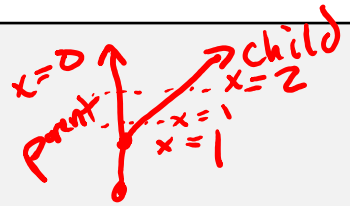
```
pid_t fork_ret = fork();
if (fork_ret == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

hello from child

Which one appears first? Non-deterministic

Fork Example

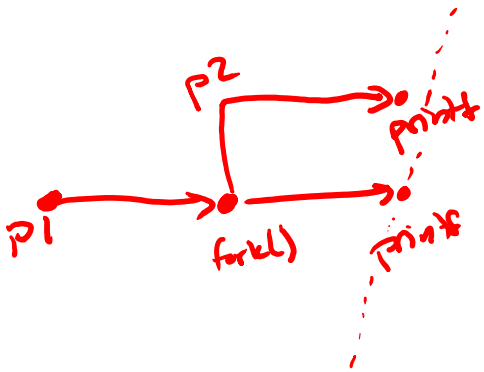
```
void fork1() {
    int x = 1;
    pid_t fork_ret = fork();
    if (fork_ret == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



- Both 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

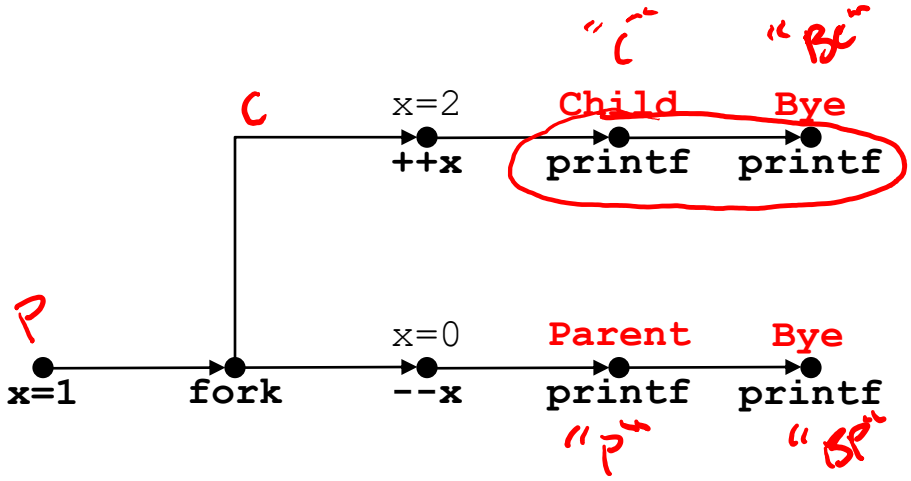
- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - `a` → `b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output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

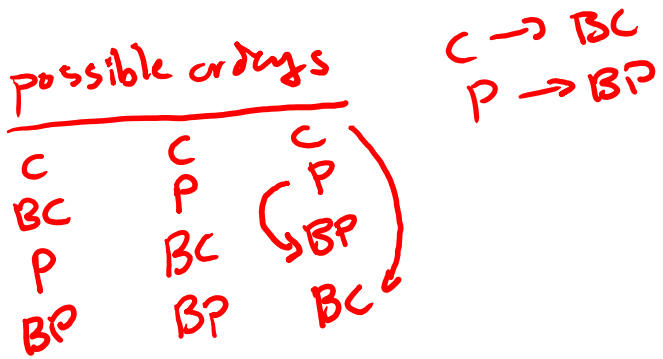


- Any *topological sort* of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right

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);
}
```



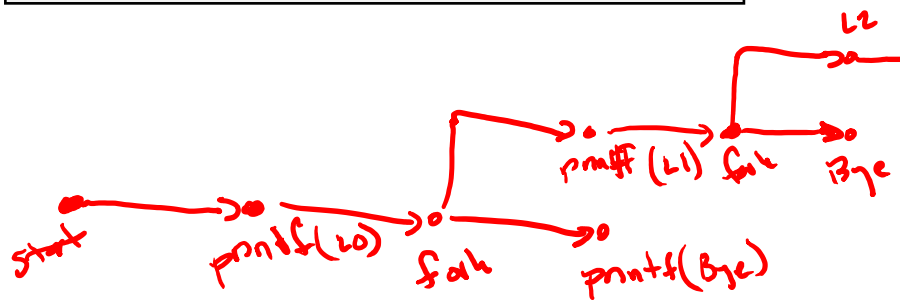


Polling Question

- Is the following sequence of outputs possible?
  - Vote in Ed Lessons

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

Seq 1:
L0
Bye
L1
L2
Bye
Bye



A. No
B. Yes
C. We're lost…

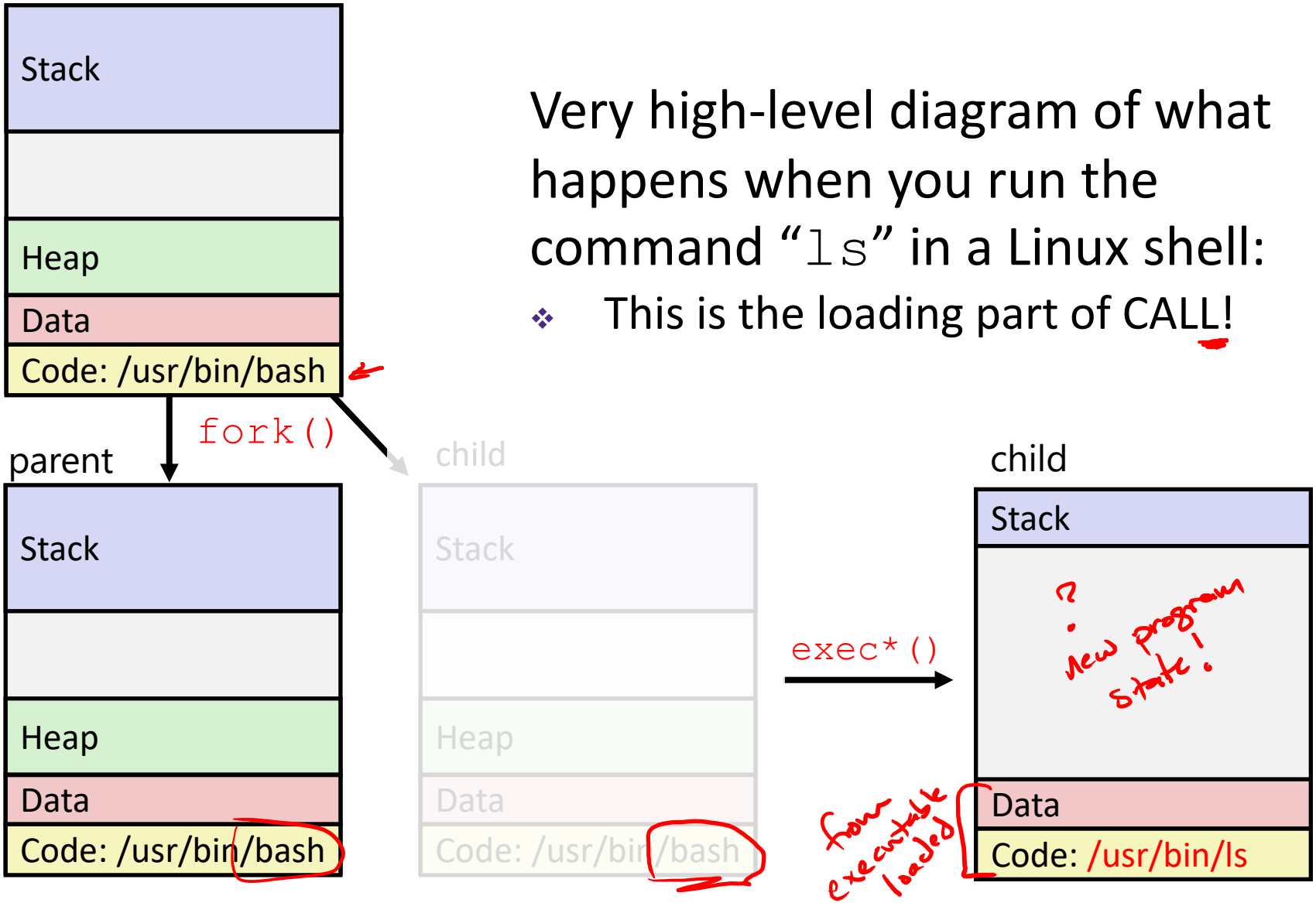
Fork-Exec

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!

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

```
int main()
return 0;      ← normal
return "non-zero";
```

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
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid of 1)
    - **Note:** on recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (*e.g.*, shells, servers) we need *explicit* reaping

`wait`: Synchronizing with Children

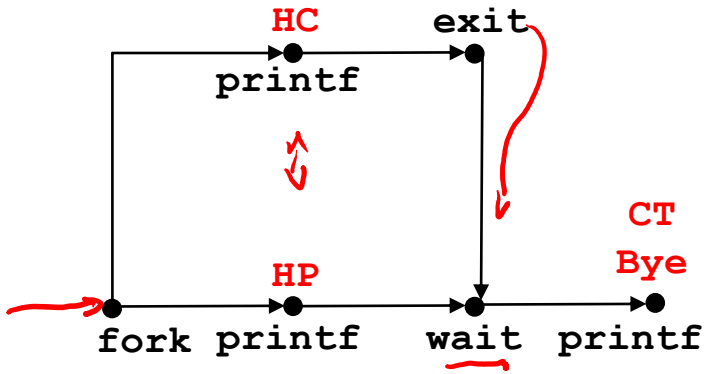
- **`int`** `wait(`**`int*`** `child_status)`
  - Suspends current process (*i.e.*, the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status`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

```
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");
}
```

forks.c



Feasible output:

HC
HP
CT
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 */
    }
}
```

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
```

don't exist → reaped!

- `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);
    }
}
```

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

- 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 */* }

load new program

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

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

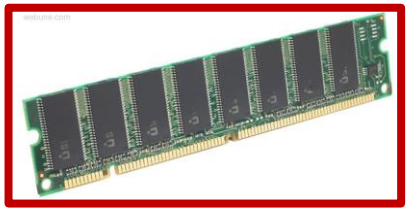
Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

Computer system:



OS:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Virtual Memory (VM*)

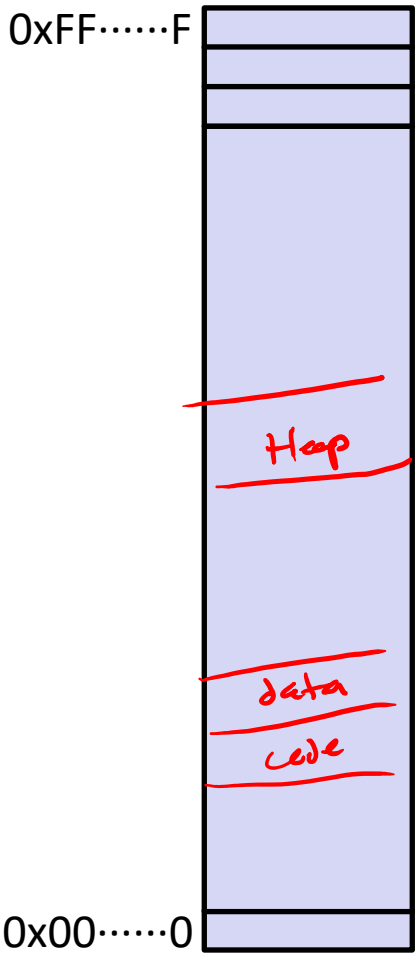
- **Overview and motivation**
- **VM as a tool for caching**
- Address translation
- VM as a tool for memory management
- VM as a tool for memory protection

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!*

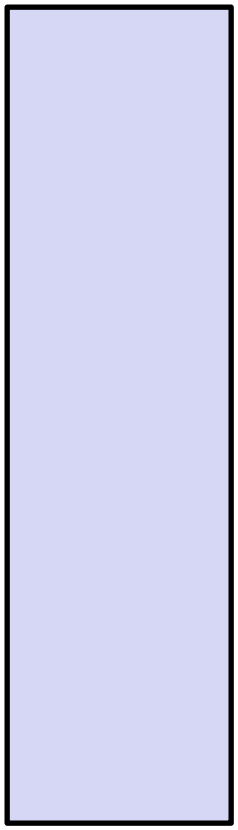
- Programs refer to virtual memory addresses
  - `movq (%rdi),%rax`
  - Conceptually memory is just a very large array of bytes
  - System provides private address space to each process
- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation 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)

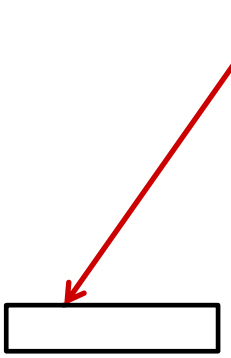
18 EiB



?

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.)

1 virtual address space per process, with many processes…

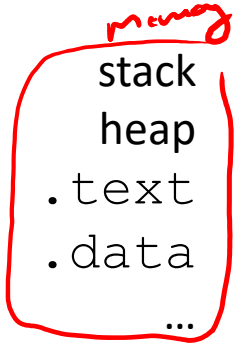
Problem 2: Memory Management

We have multiple processes:

- Process 1
- Process 2
- Process 3
- …
- Process n

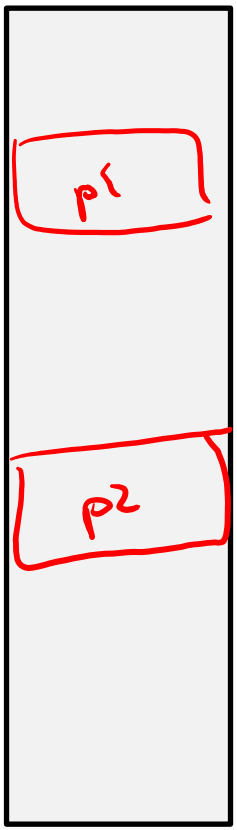
X

Each process has…

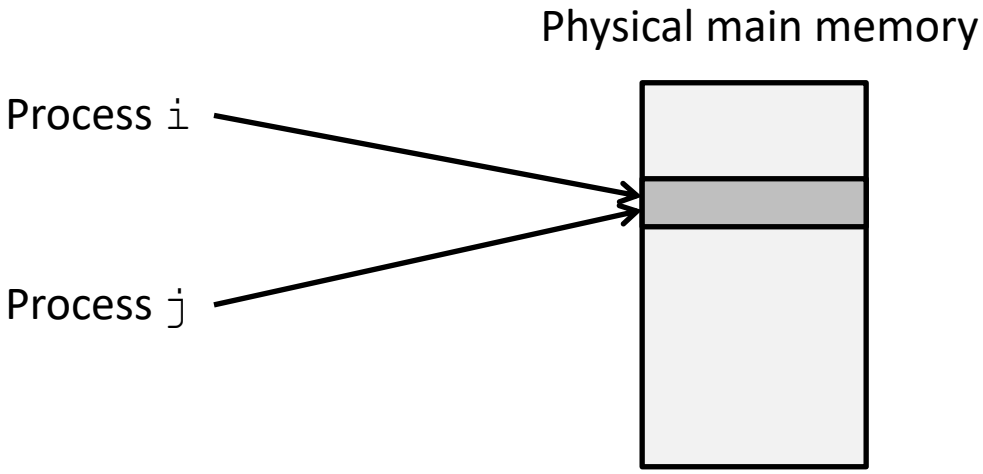


What goes where?

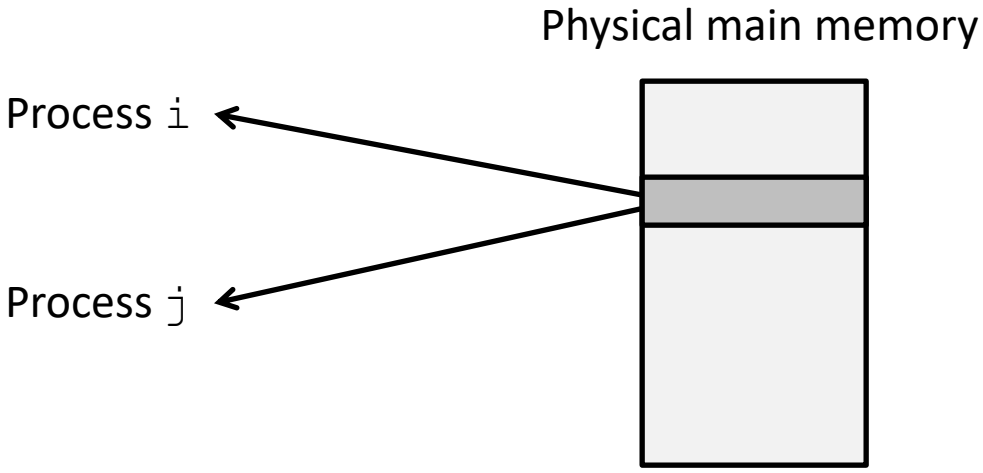
Physical main memory



Problem 3: How To Protect



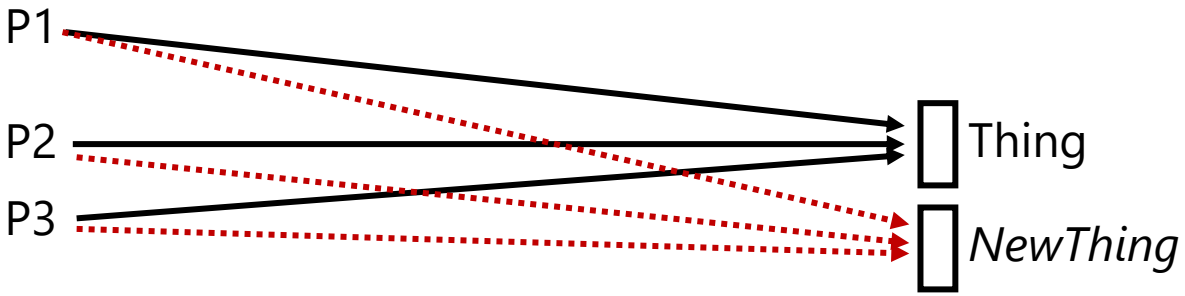
Problem 4: How To Share?



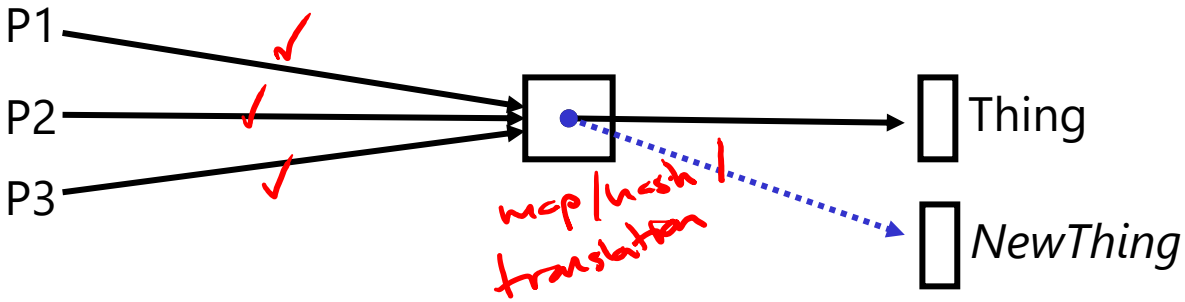
How can we solve these problems?

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

- Without Indirection



- With Indirection

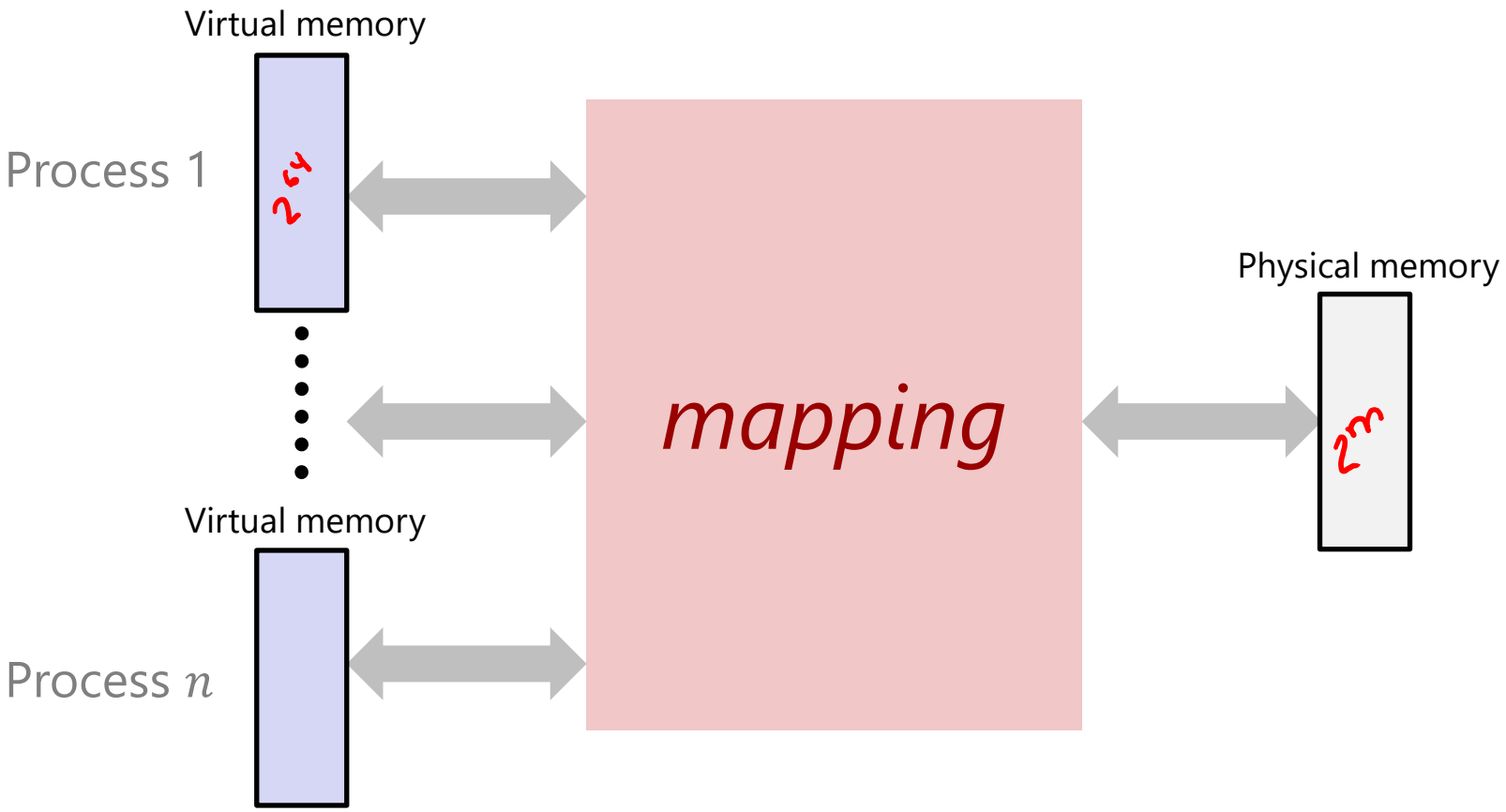


What if I want to move Thing?

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.
  - 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!)
- 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!

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}

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

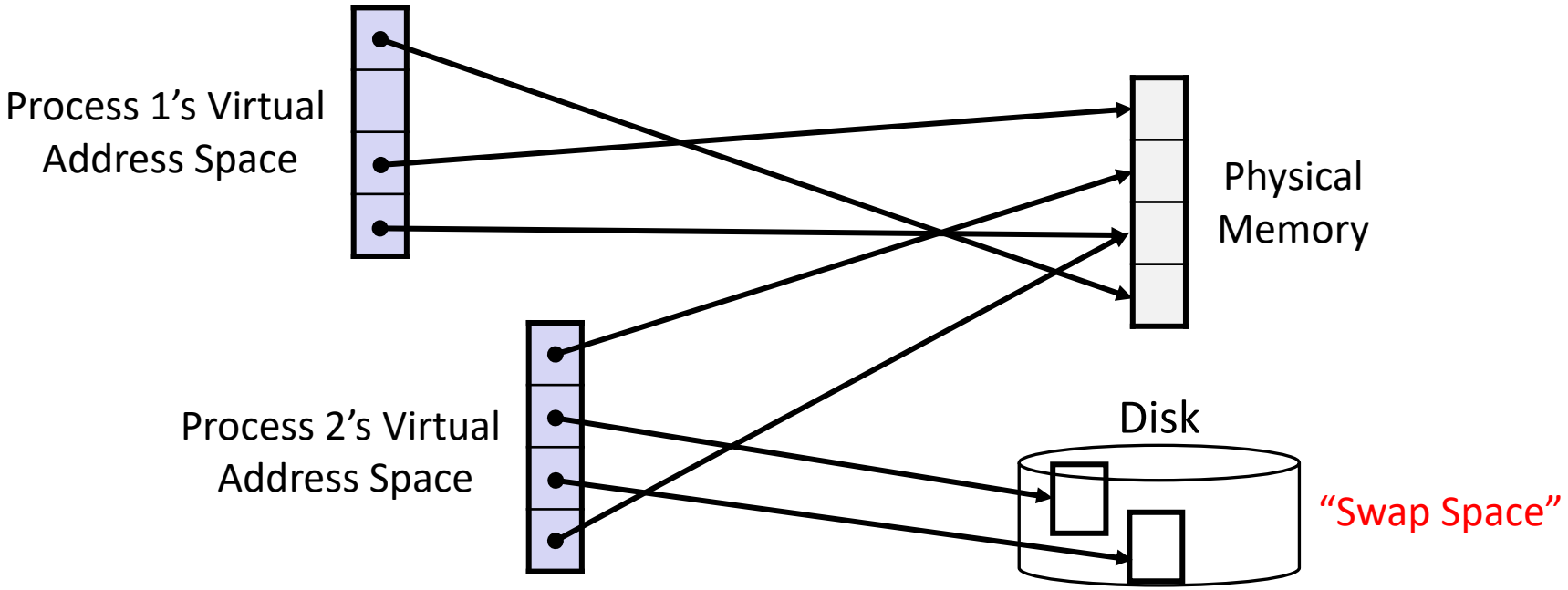
Polling Questions

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

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

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



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

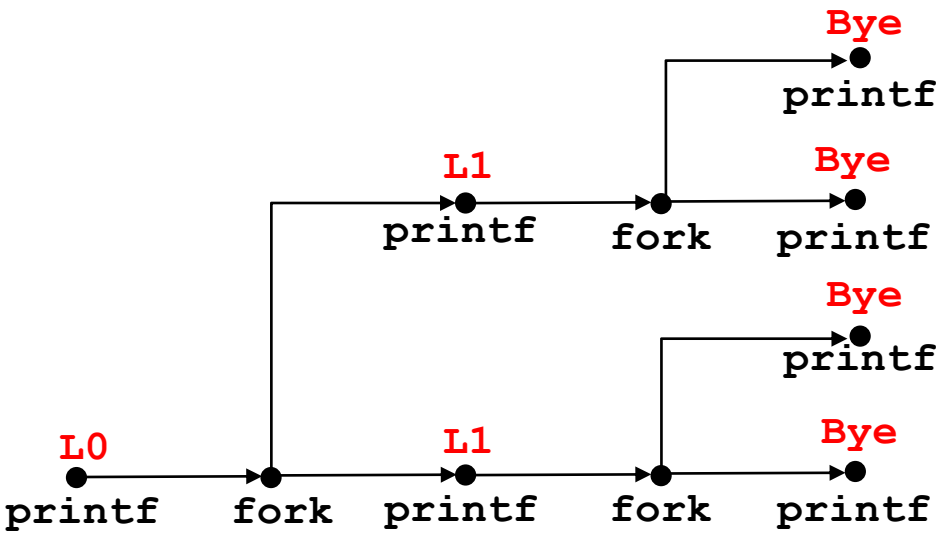
BONUS SLIDES

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");
}
```



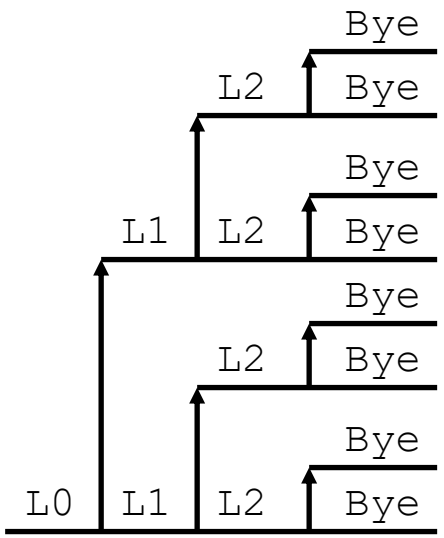
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

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);
    }
}
```