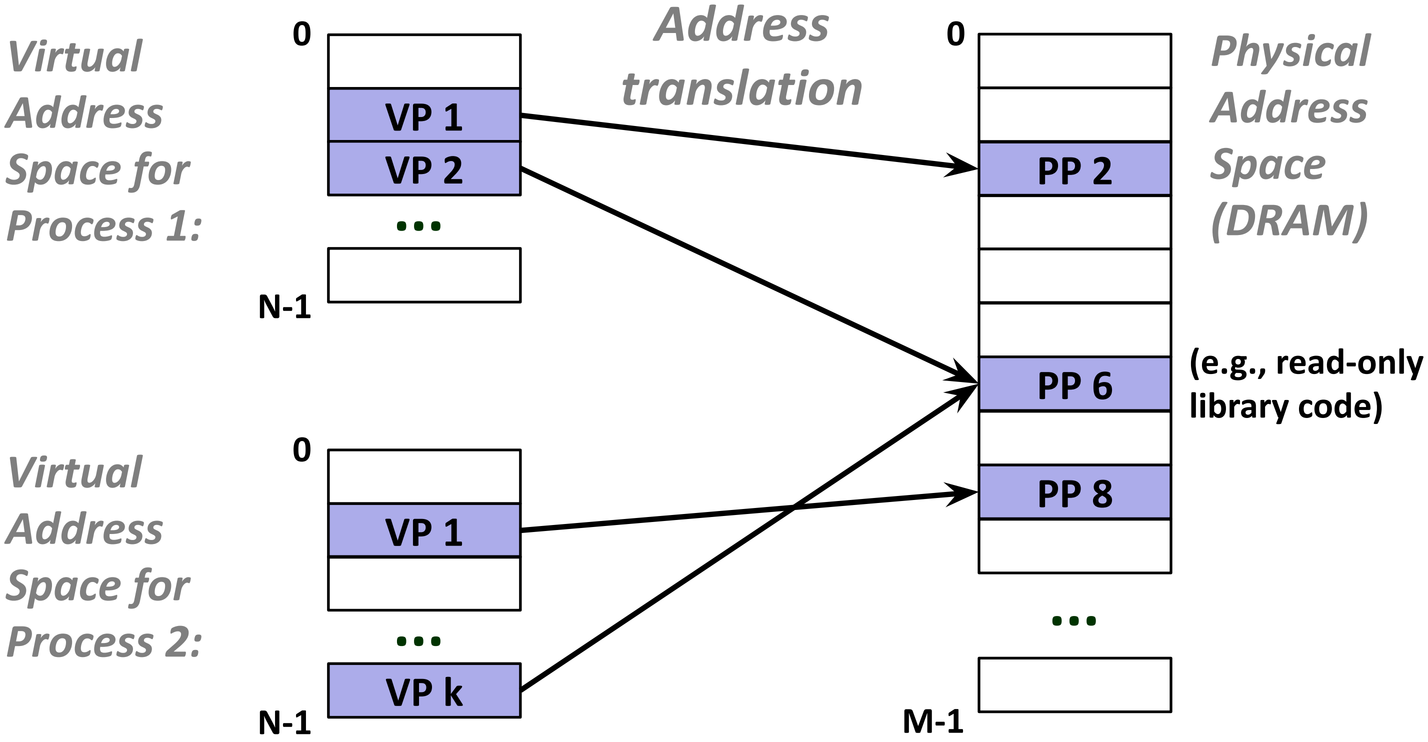
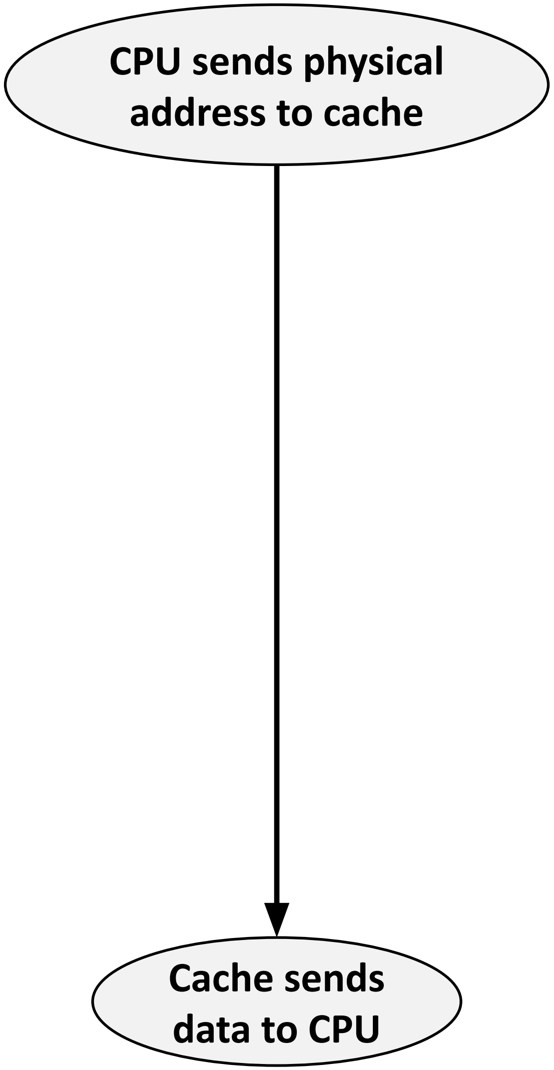
Virtual Memory

Review: Virtual Addressing

- Each process has its own *virtual address space*
- *Page tables* map virtual to physical addresses
- Physical memory can be shared among processes



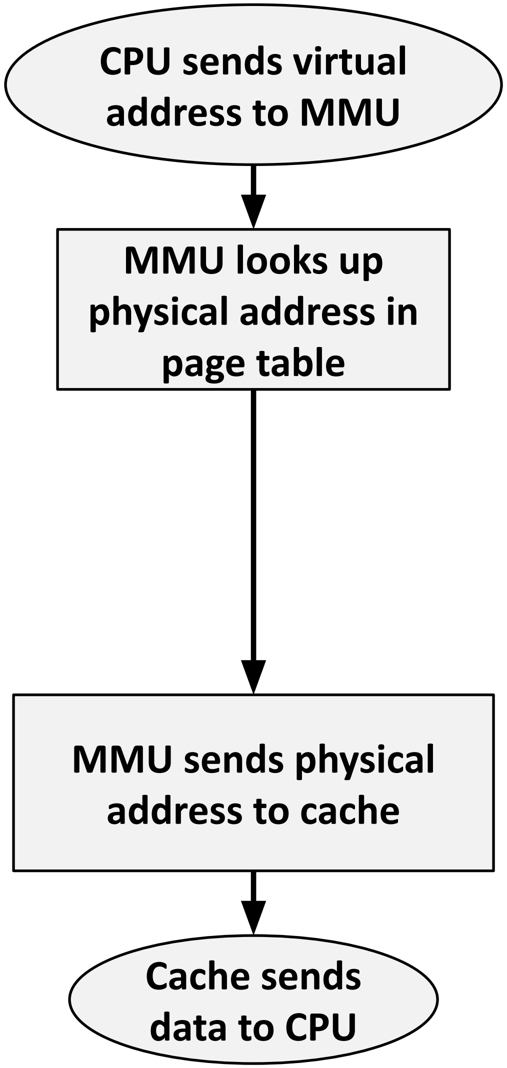
Review: Memory Accesses without VM



CPU sends physical address to cache

Cache sends data to CPU

Review: Memory Accesses with VM



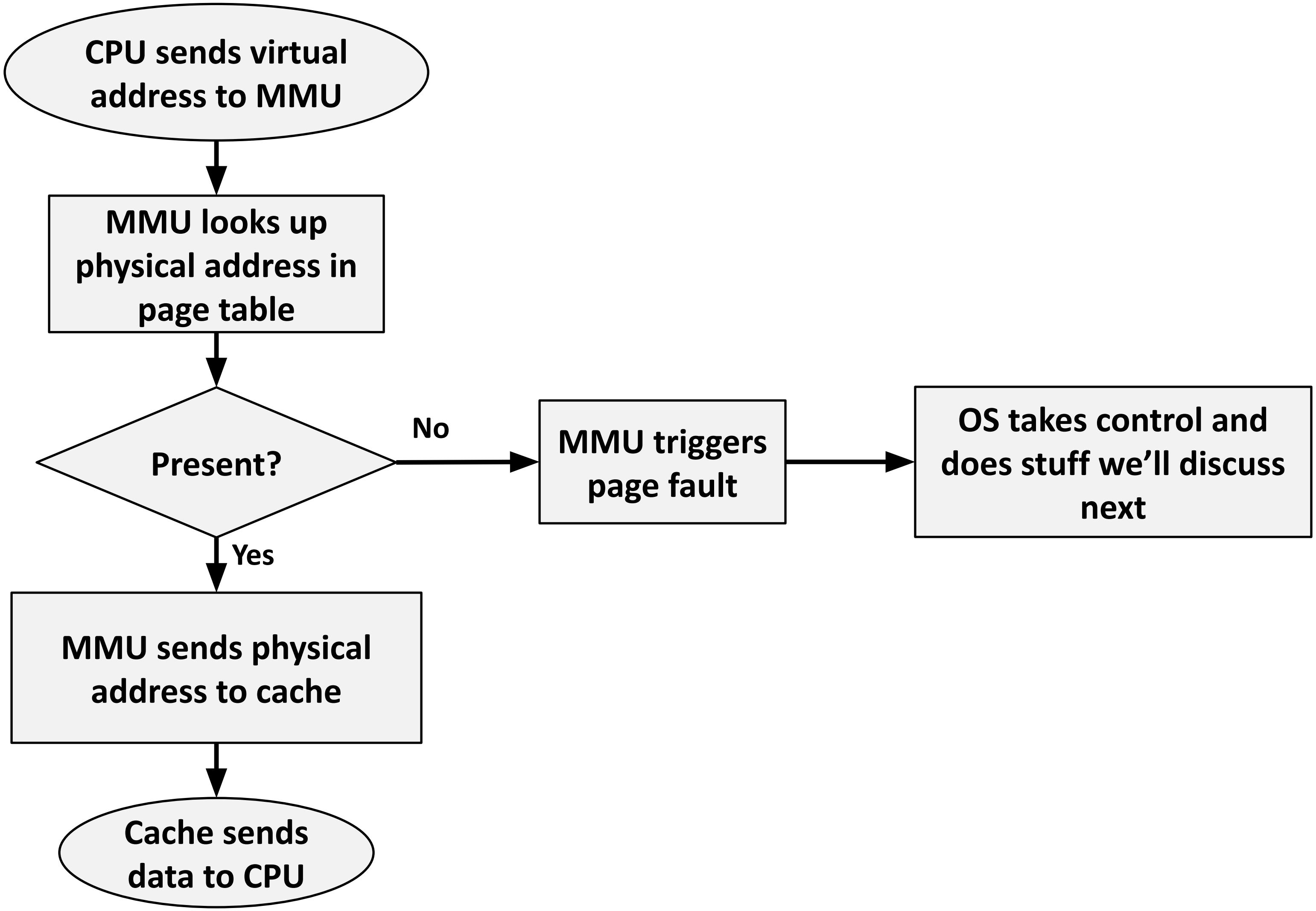
CPU sends virtual address to MMU

MMU looks up physical address in page table

MMU sends physical address to cache

Cache sends data to CPU

Review: Memory Accesses with VM



CPU sends virtual address to MMU

MMU looks up physical address in page table

Present?

No

MMU triggers page fault

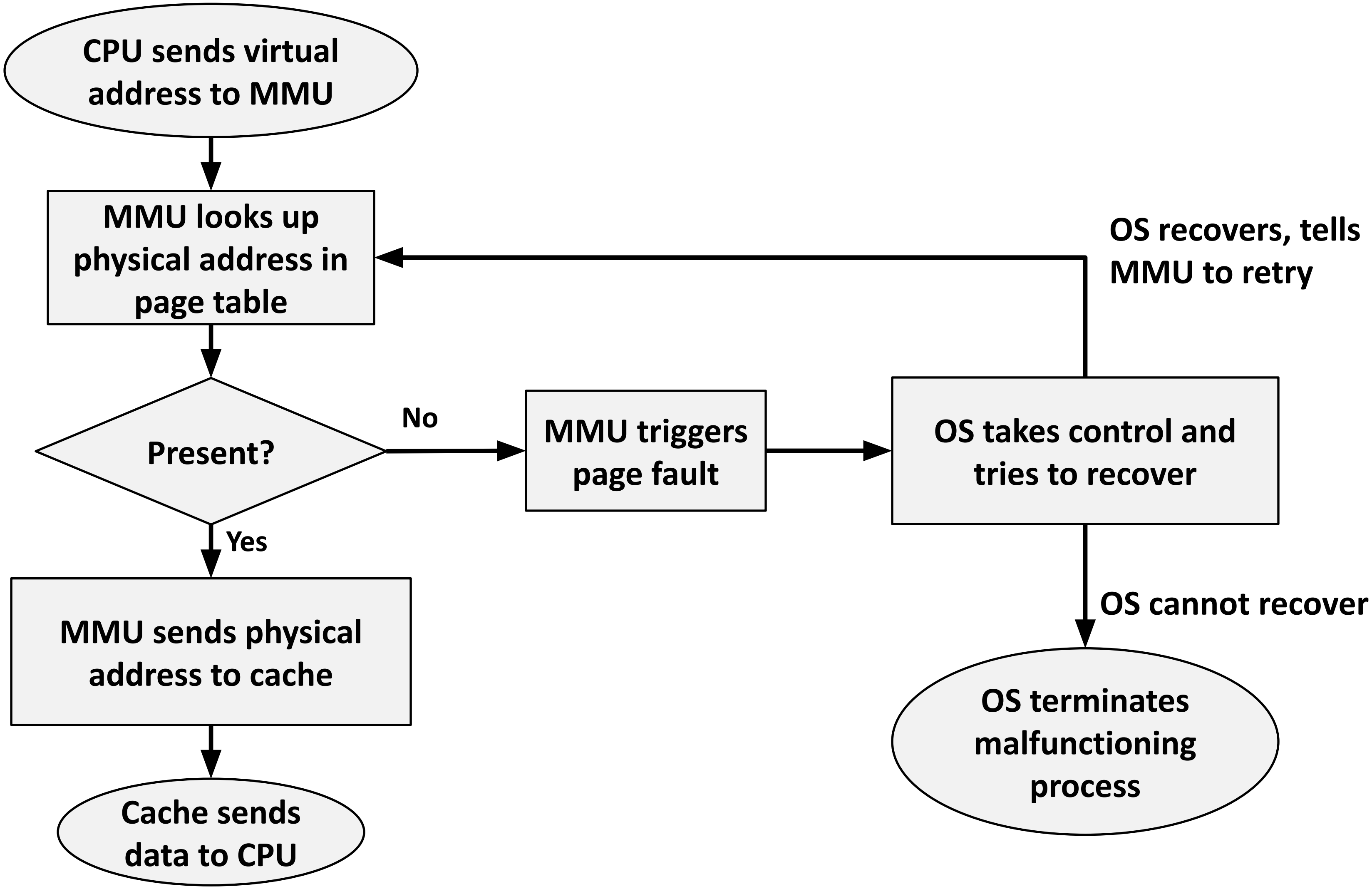
OS takes control and does stuff we'll discuss next

Yes

MMU sends physical address to cache

Cache sends data to CPU

Review: Memory Accesses with VM



CPU sends virtual address to MMU

MMU looks up physical address in page table

Present?

No

MMU triggers page fault

OS takes control and tries to recover

OS recovers, tells MMU to retry

OS cannot recover

OS terminates malfunctioning process

Yes

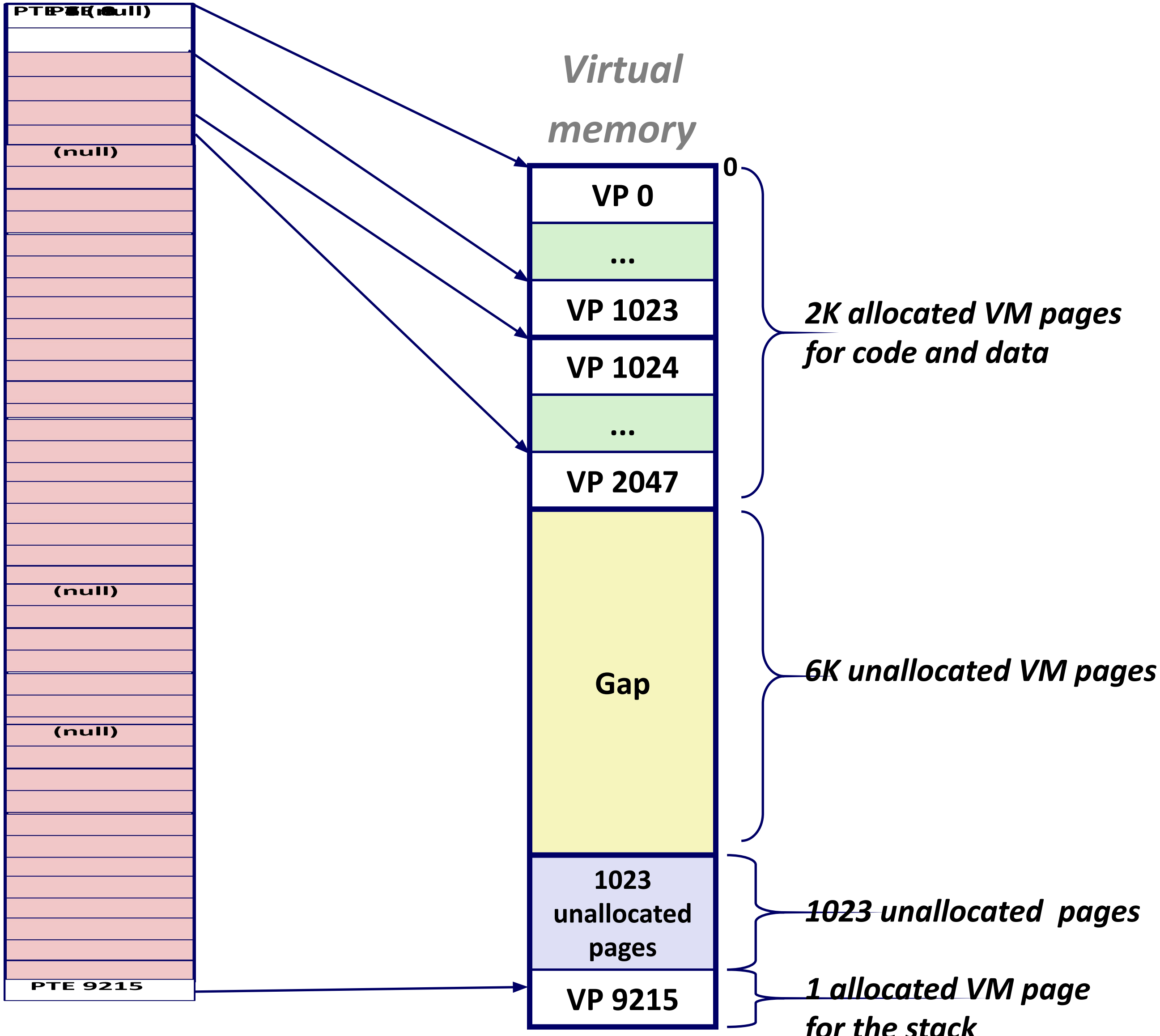
MMU sends physical address to cache

Cache sends data to CPU

We have a problem

2^{20} Entries of

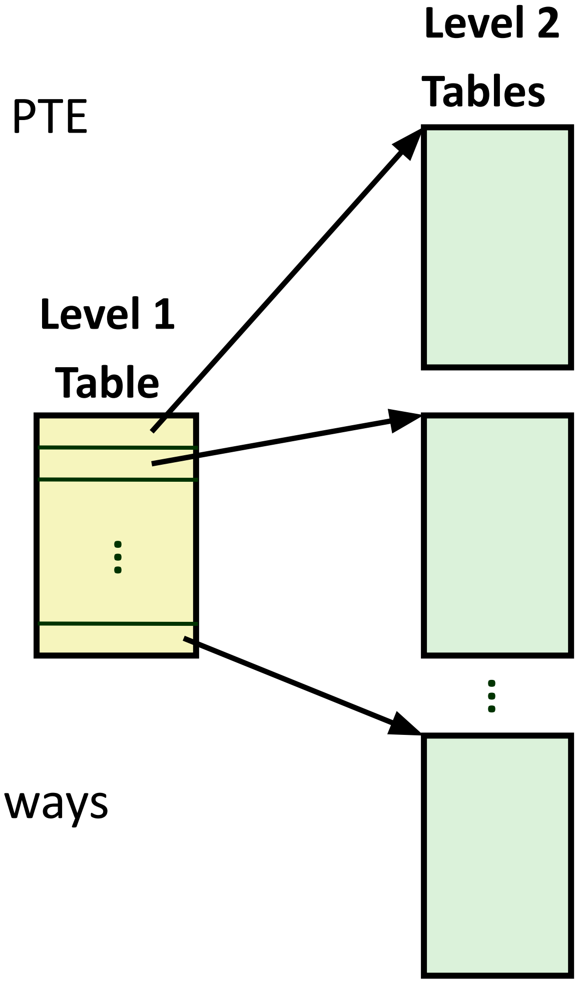
4 bytes each



32 bit addresses, 4KB pages, 4-byte PTEs

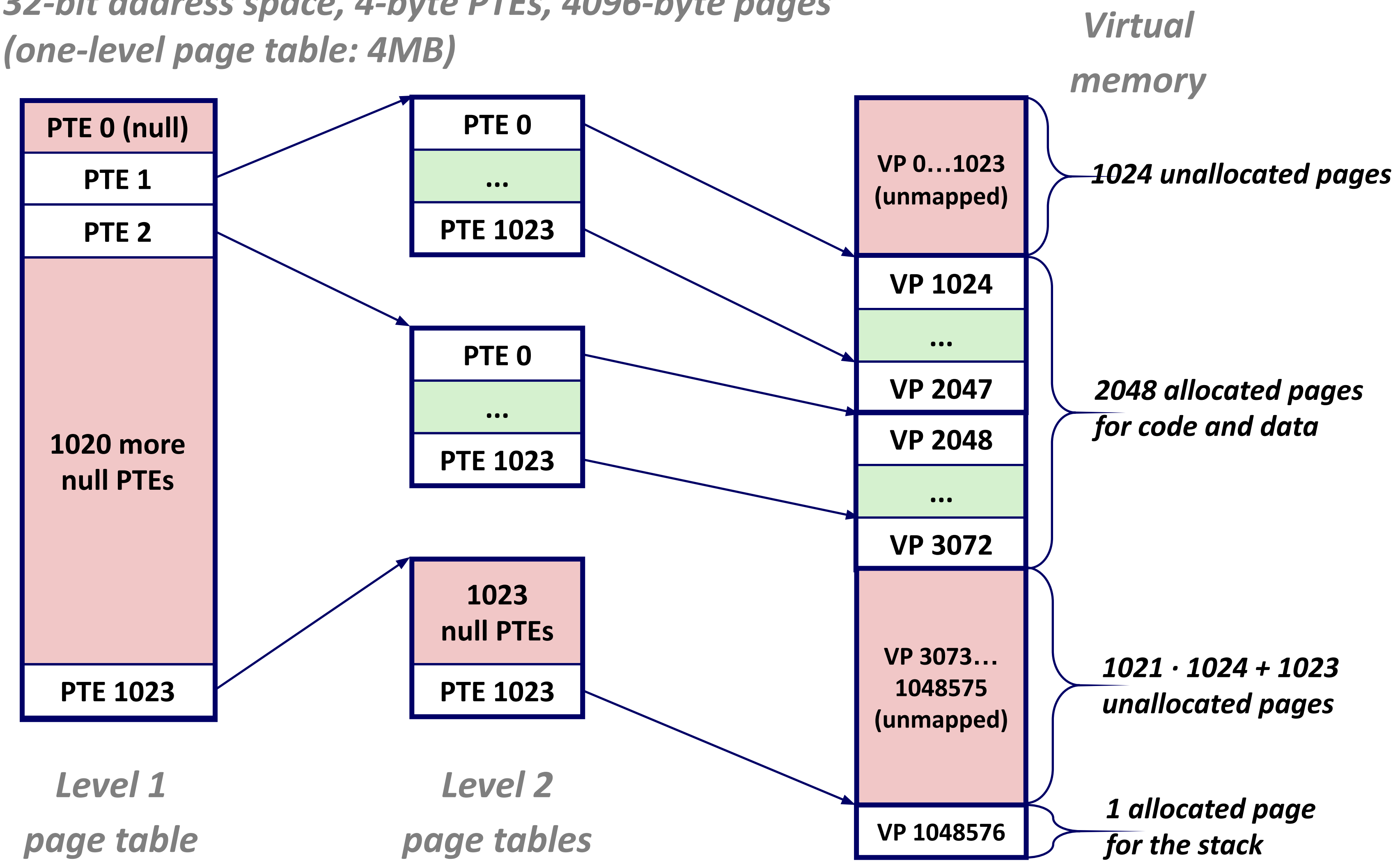
Multi-Level Page Tables

- **Suppose:**
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- **Problem:**
  - Would need a 512 GB page table!
    - $2^{48} * 2^{-12} * 2^{3} = 2^{39}$ bytes
- **Common solution: Multi-level page table**
- **Example: 2-level page table**
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)

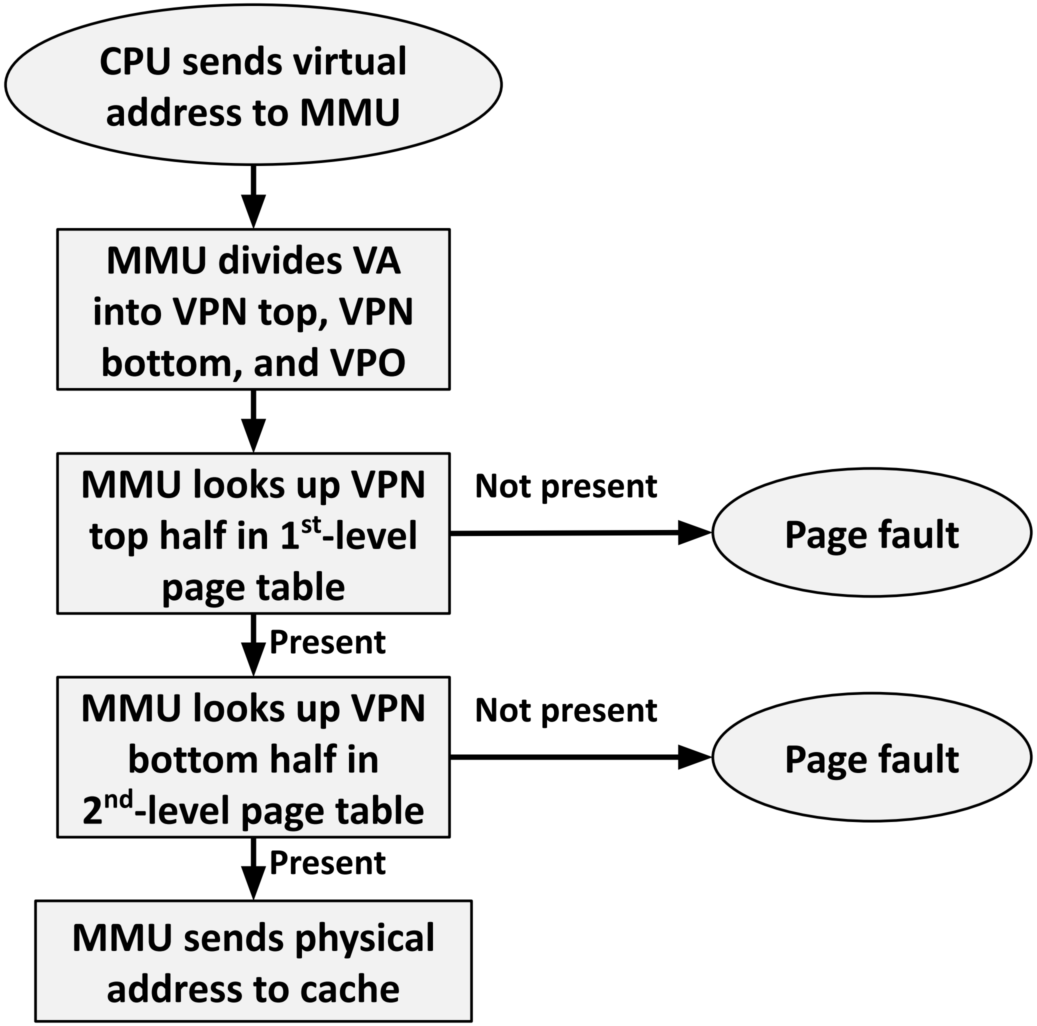


A Two-Level Page Table Hierarchy

32-bit address space, 4-byte PTEs, 4096-byte pages
(one-level page table: 4MB)



Translating with a two-level page table



CPU sends virtual address to MMU

MMU divides VA into VPN top, VPN bottom, and VPO

MMU looks up VPN top half in 1st-level page table

Not present

Page fault

Present

MMU looks up VPN bottom half in 2nd-level page table

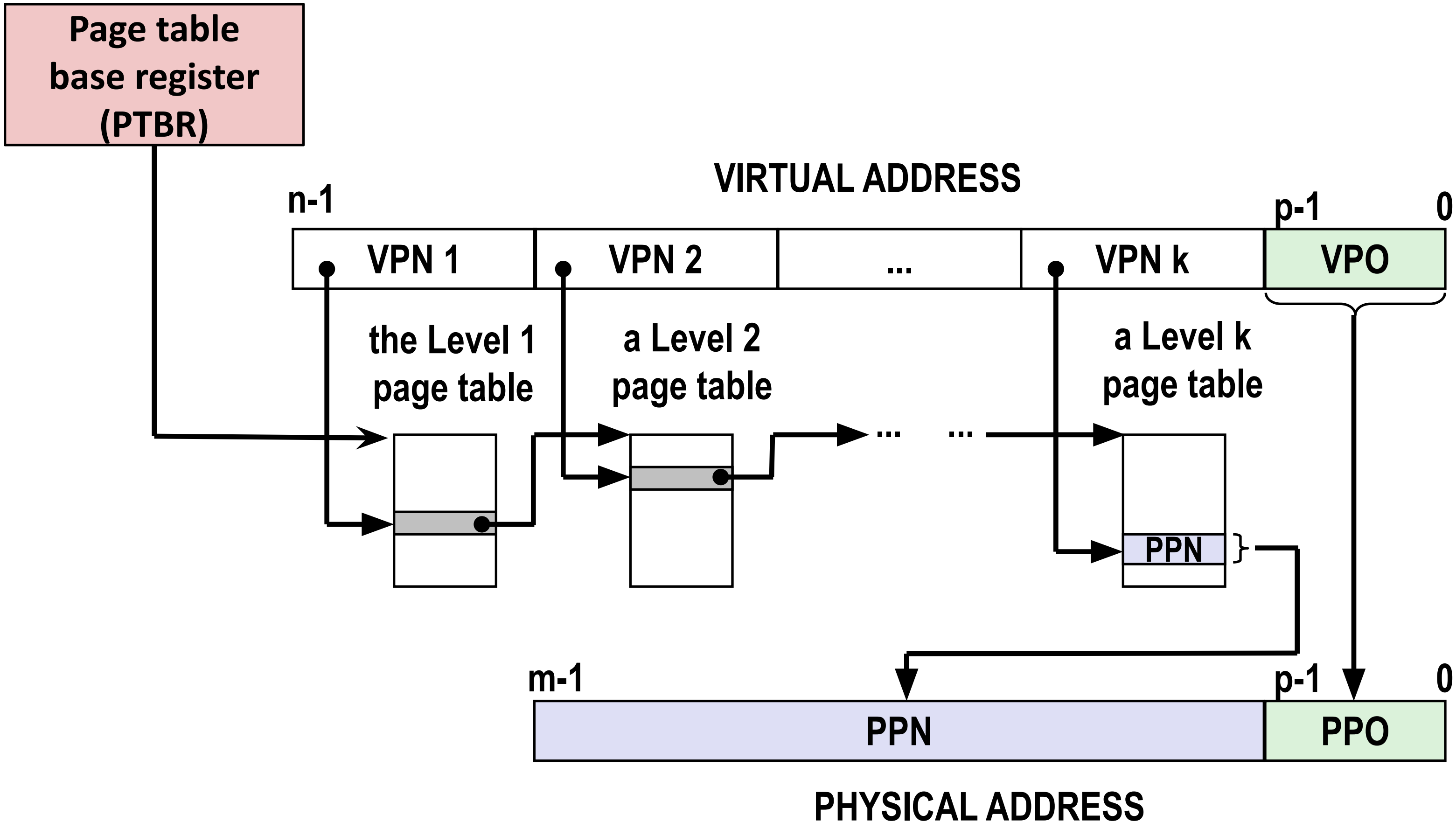
Not present

Page fault

Present

MMU sends physical address to cache

Translating with a k-level Page Table



Quick Question

Suppose we have a 64-bit virtual address space having a page size of 8KB and every page table entry is 4 bytes in size. How many levels of page table (paging) would be required such that every level of the page table fits in at most one page?

Quick Question

Suppose we have a system with 32-bit virtual address having the structure:

Page Directory	Virtual Page Number	Offset
10 bits	10 bits	12 bits

The page table entry is 4 bytes in size with 20 bits for the physical frame number.

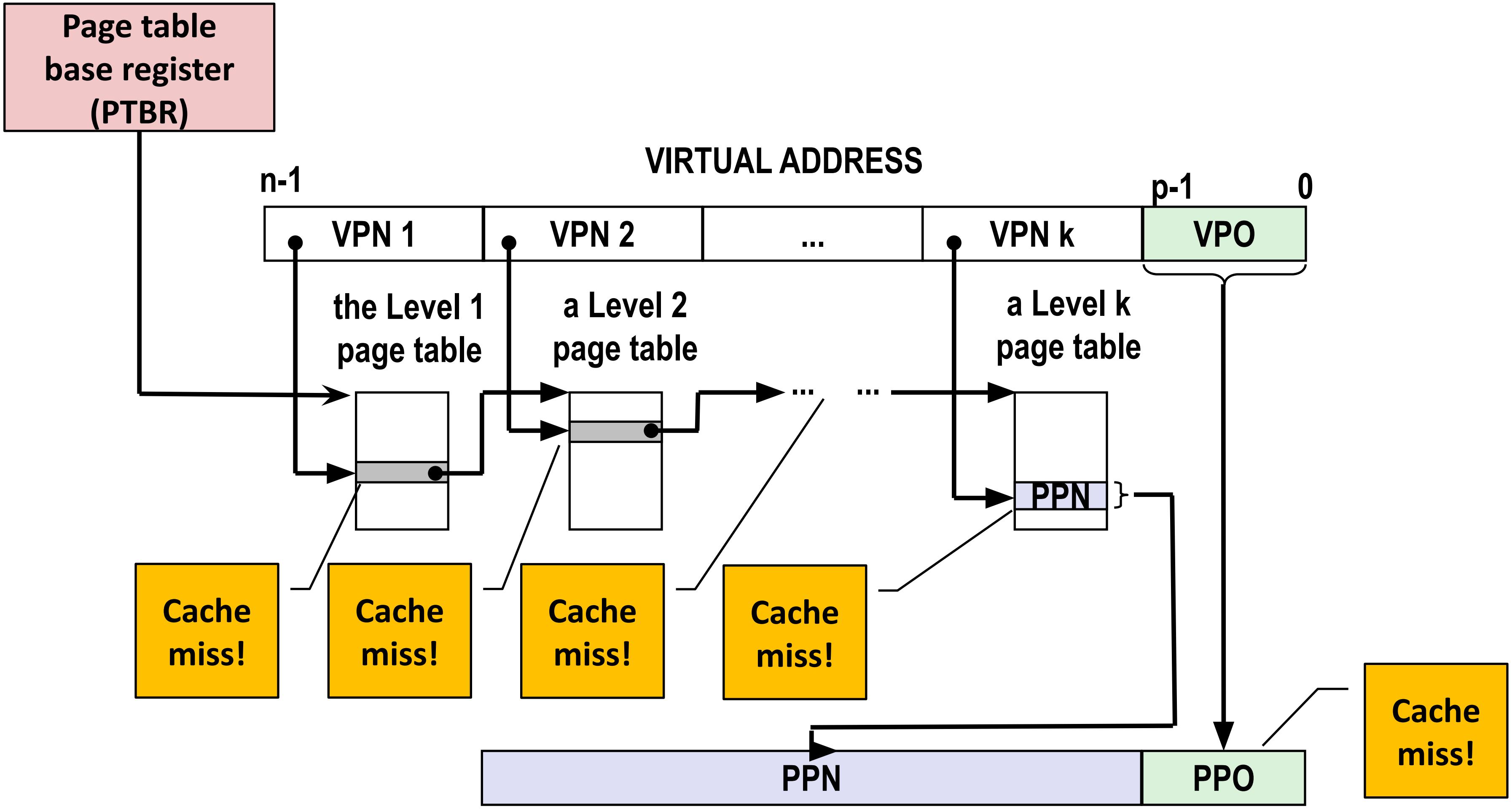
a. **What is the size of a page in this system?**

b. **What is the maximum amount of physical memory?**

c. **How many pages are required to represent the page table?**

d. **If we have an address space with two pages at the top of the address space and one page at the bottom of the address space, how much physical memory would be required for this mapping (in bytes)?**

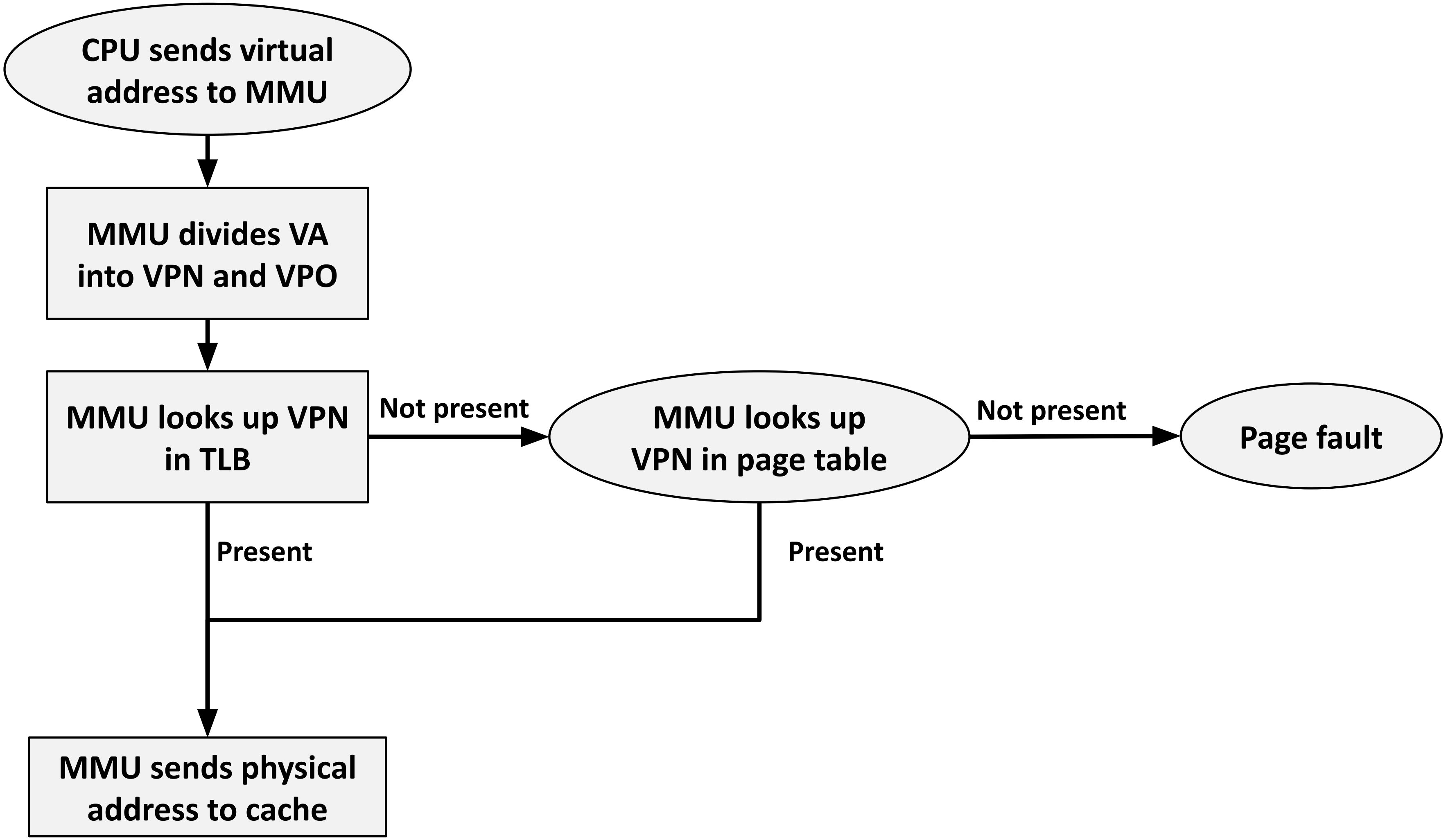
The problem (with k-level page tables)



Speeding up Translation with a TLB

- **Page table entries (PTEs) are cached like any other memory word**
  - PTEs may be evicted by other data references
  - PTE hit still costs cache delay
- **Solution: *Translation Lookaside Buffer* (TLB)**
  - Dedicated cache for page table entries
  - TLB hit = page table not consulted
  - Can be fairly small: one TLB entry covers 4k or more

Translating with a TLB



CPU sends virtual address to MMU

MMU divides VA into VPN and VPO

MMU looks up VPN in TLB

Not present

MMU looks up VPN in page table

Not present

Page fault

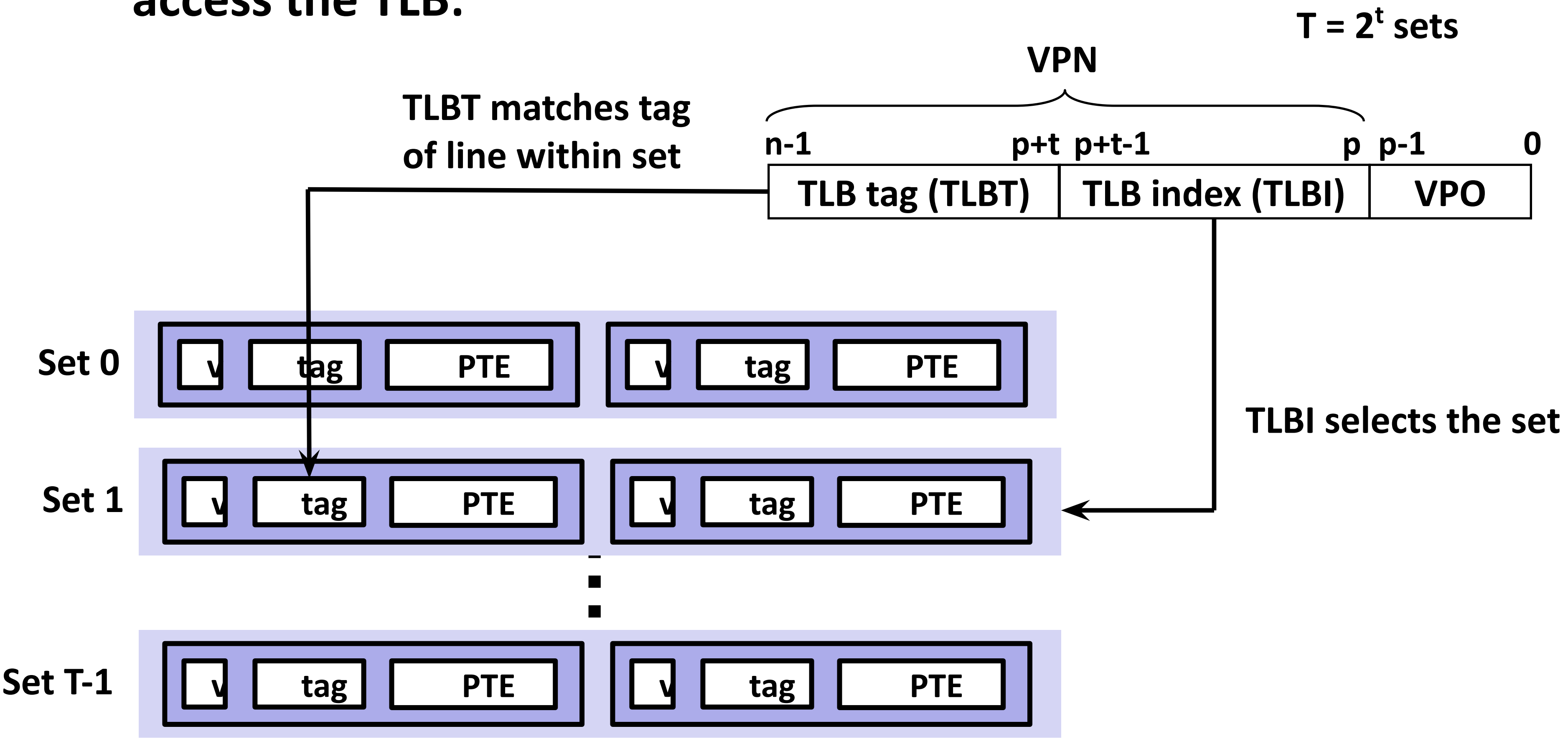
Present

Present

MMU sends physical address to cache

Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

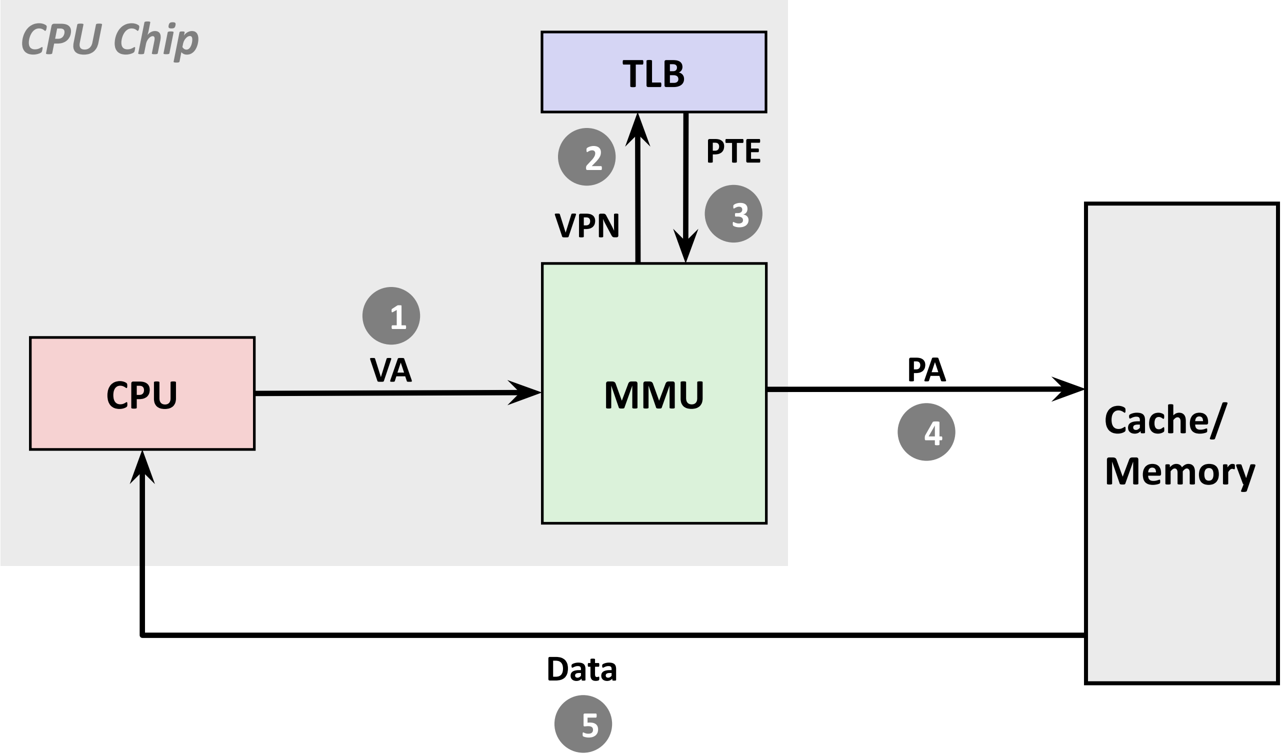


T = 2^t sets

TLBT matches tag of line within set

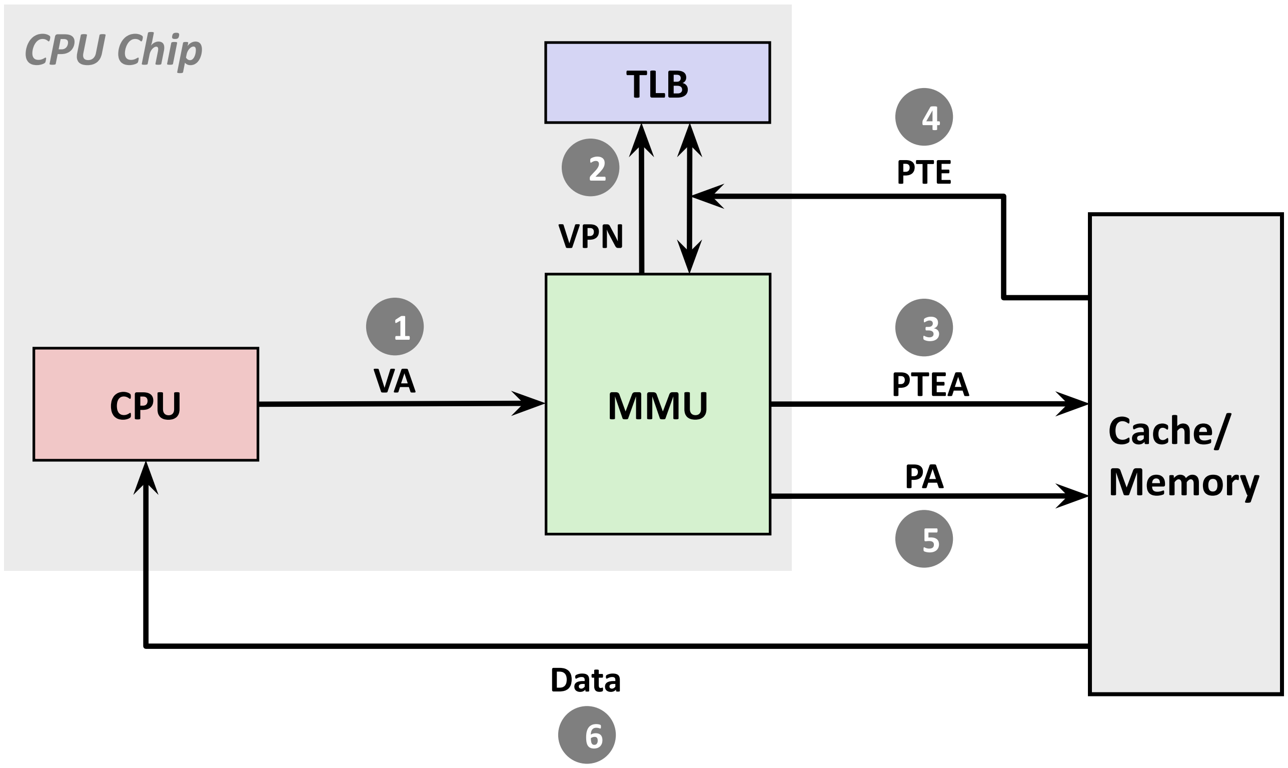
TLBI selects the set

TLB Hit



A TLB hit eliminates memory accesses to the page table

TLB Miss

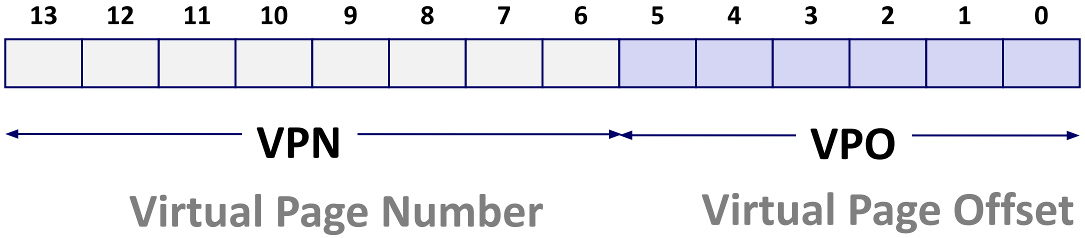


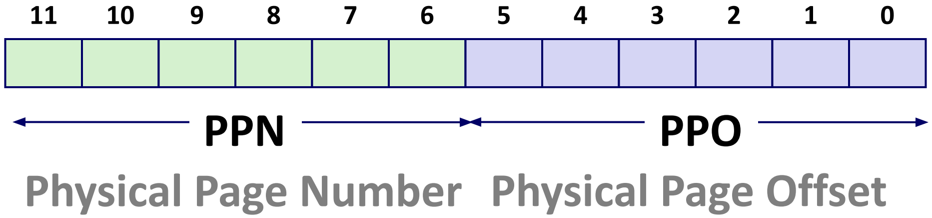
A TLB miss incurs additional memory accesses (PTE lookup)

Fortunately, TLB misses are rare. Why?

Simple Memory System Example

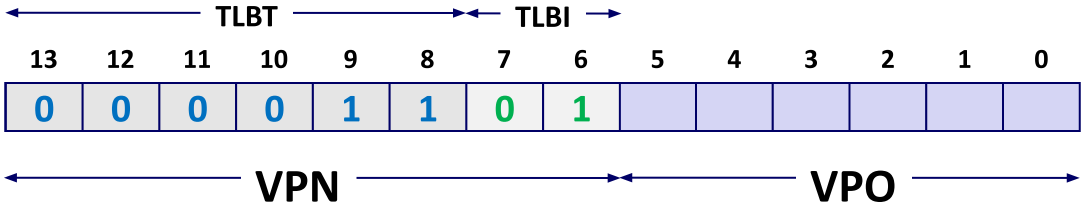
- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes





Simple Memory System TLB

- **16 entries**
- **4-way associative**



13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1						

VPN = 0b1101 = 0x0D

Translation Lookaside Buffer (TLB)

Set	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*	*Tag*	*PPN*	*Valid*
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

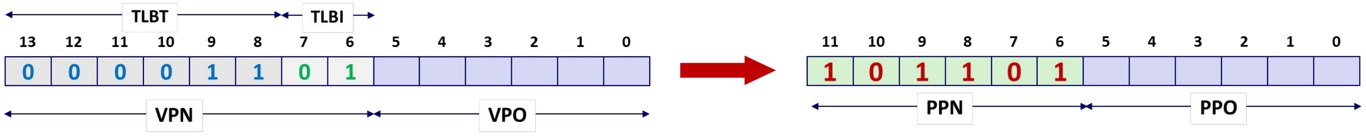
Simple Memory System Page Table

- **Only showing the first 16 entries (out of 256)**

VPN	*PPN*	*Valid*
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	*PPN*	*Valid*
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

0x0D → 0x2D

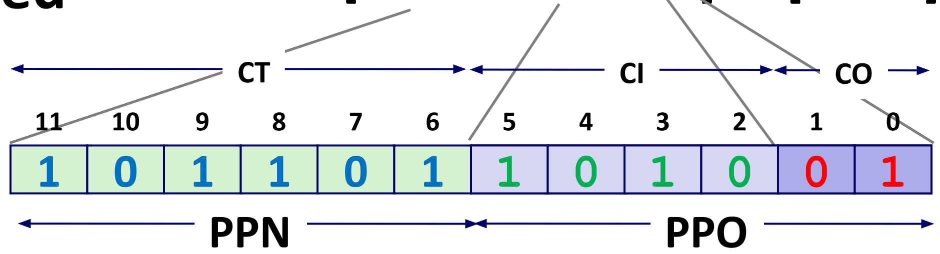


Simple Memory System Cache

- 16 lines, 4-byte cache line size
- Physically addressed
- Direct mapped

V[0b00001101101001] = V[0x369]

P[0b101101101001] = P[0xB69] = 0x15

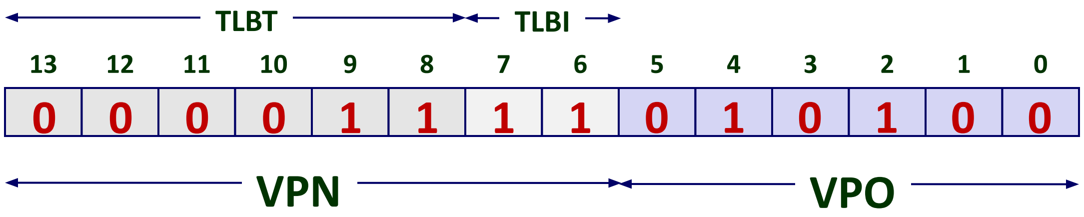


Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Address Translation Example

Virtual Address: `0x03D4`

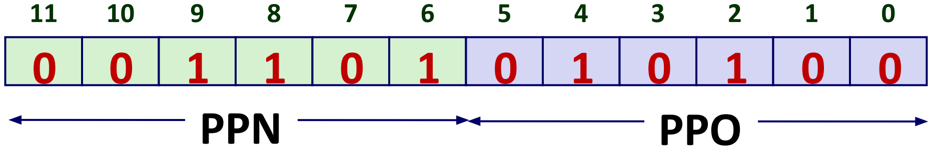


13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

TLB

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Physical Address



11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

Paging (aka Swapping)

- **Use (part of) disk as additional working memory**
- **Adds another layer to the memory hierarchy, but…**
  - "Main memory" is 10–1000x slower than the caches
  - Disk is **10,000x** slower than main memory
  - Enormous miss penalty drives design

- **Consequences**
  - Large page (block) size: 4KB and bigger
  - Always write-back and fully associative
  - Managed entirely in software
    - Plenty of time to execute complex replacement algorithms

Locality to the Rescue Again!

- **Paging is terribly inefficient**
- **Only works because of locality**
- **At any point in time, programs tend to access a set of active virtual pages called the *working set***
  - Programs with good temporal locality will have small working sets

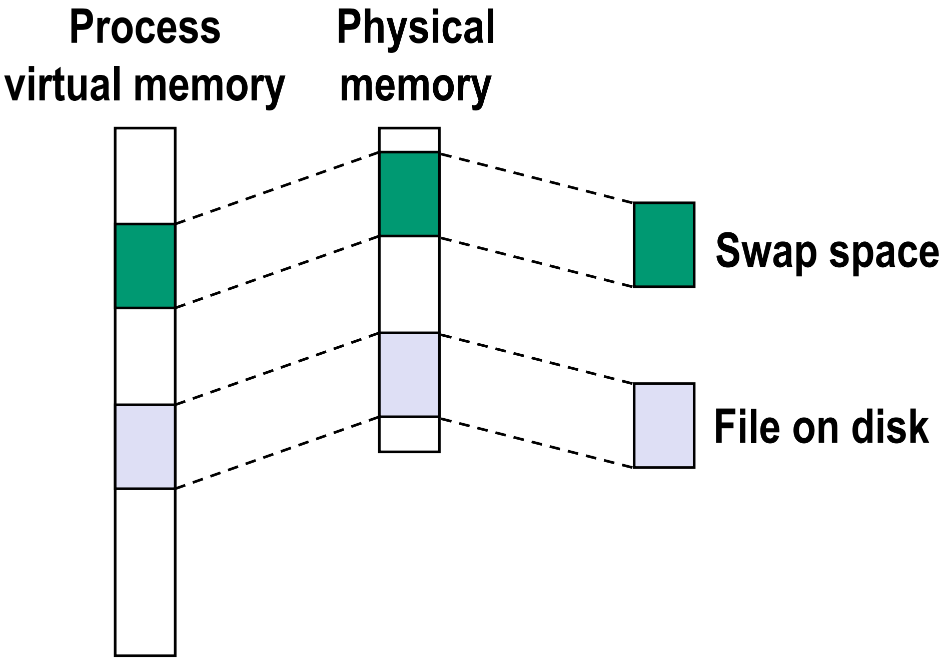
- **If working set size < main memory size**
  - Good performance after compulsory misses
- **If working set size > main memory size**
  - *Thrashing*: Performance meltdown, computer spends most of its time copying pages in and out of RAM
  - In the worst case, no forward progress at all (livelock)

Memory-Mapped Files

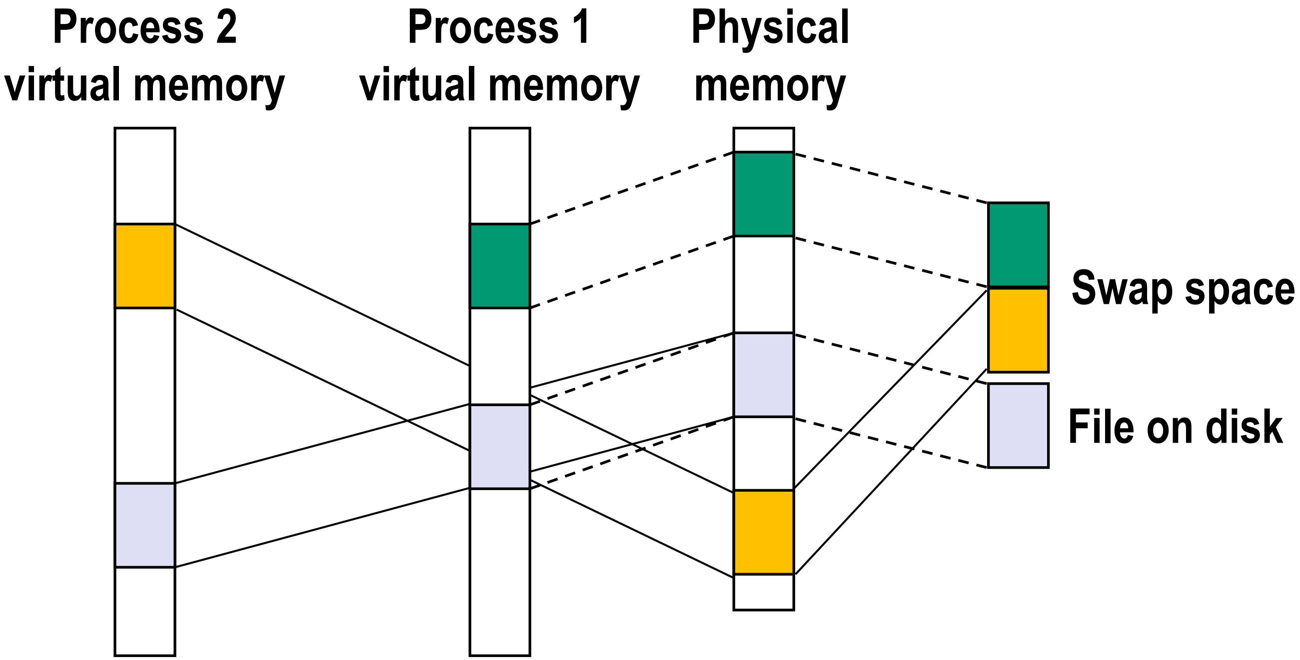
- **Paging = every page of a program's physical RAM is *backed* by some page of disk***
- **Normally, those pages belong to *swap space***
- **But what if some pages were backed by … files?**

* This is how it used to work 20 years ago.
Nowadays, not always true.

Memory-Mapped Files

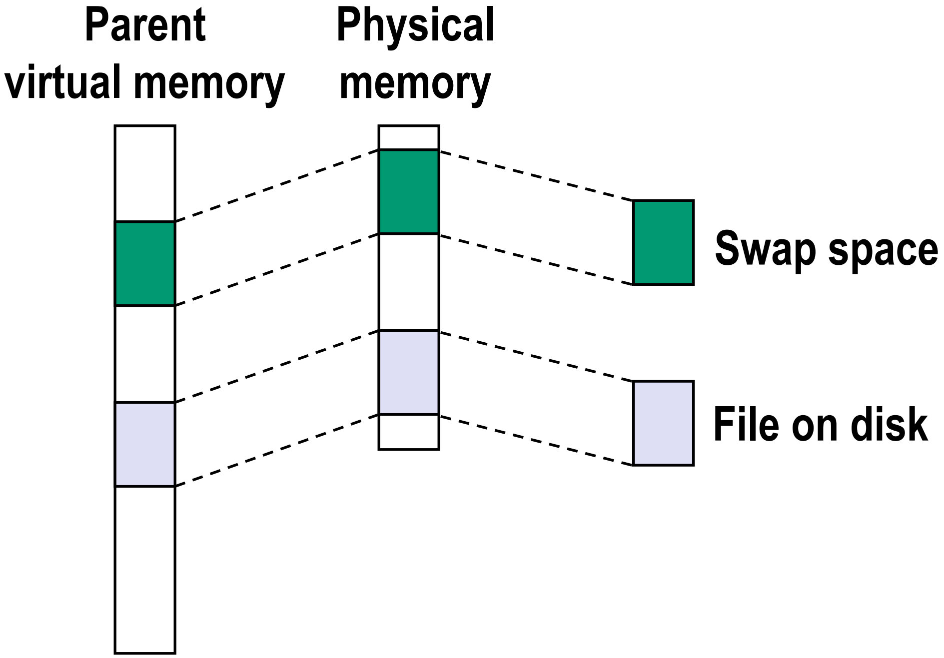


Memory-Mapped Files



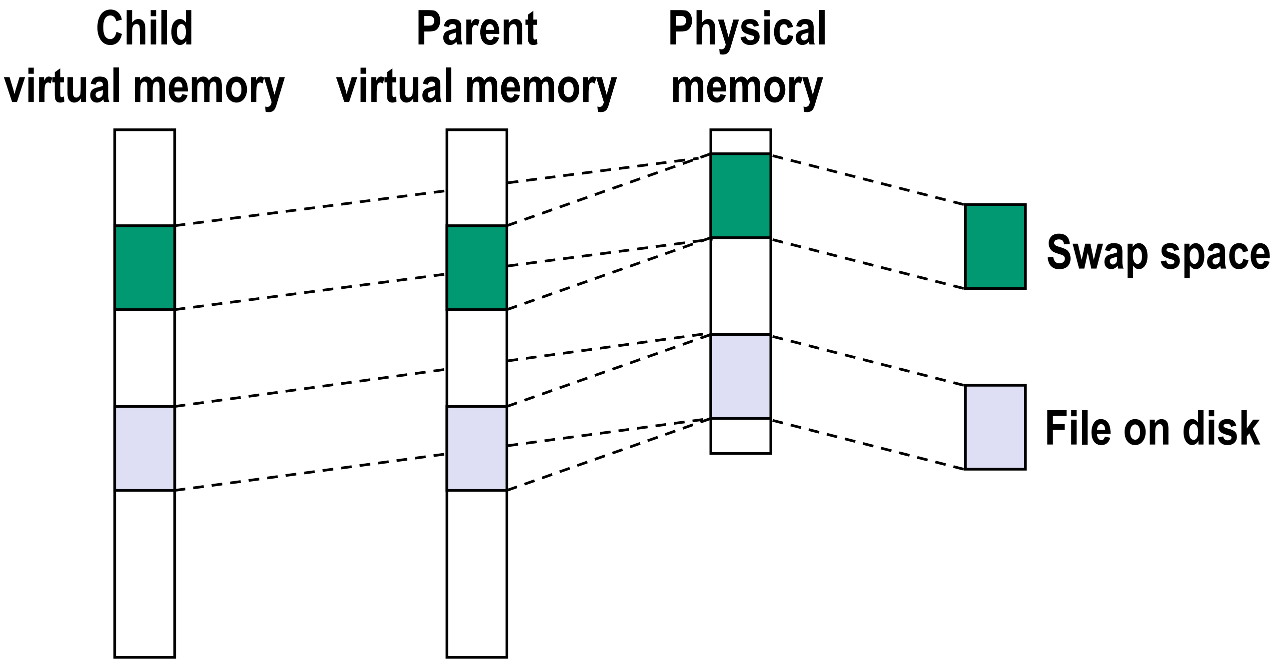
Copy-on-write sharing

- `fork` **creates a new process by copying the entire address space of the parent process**
  - That sounds slow
  - It *is* slow



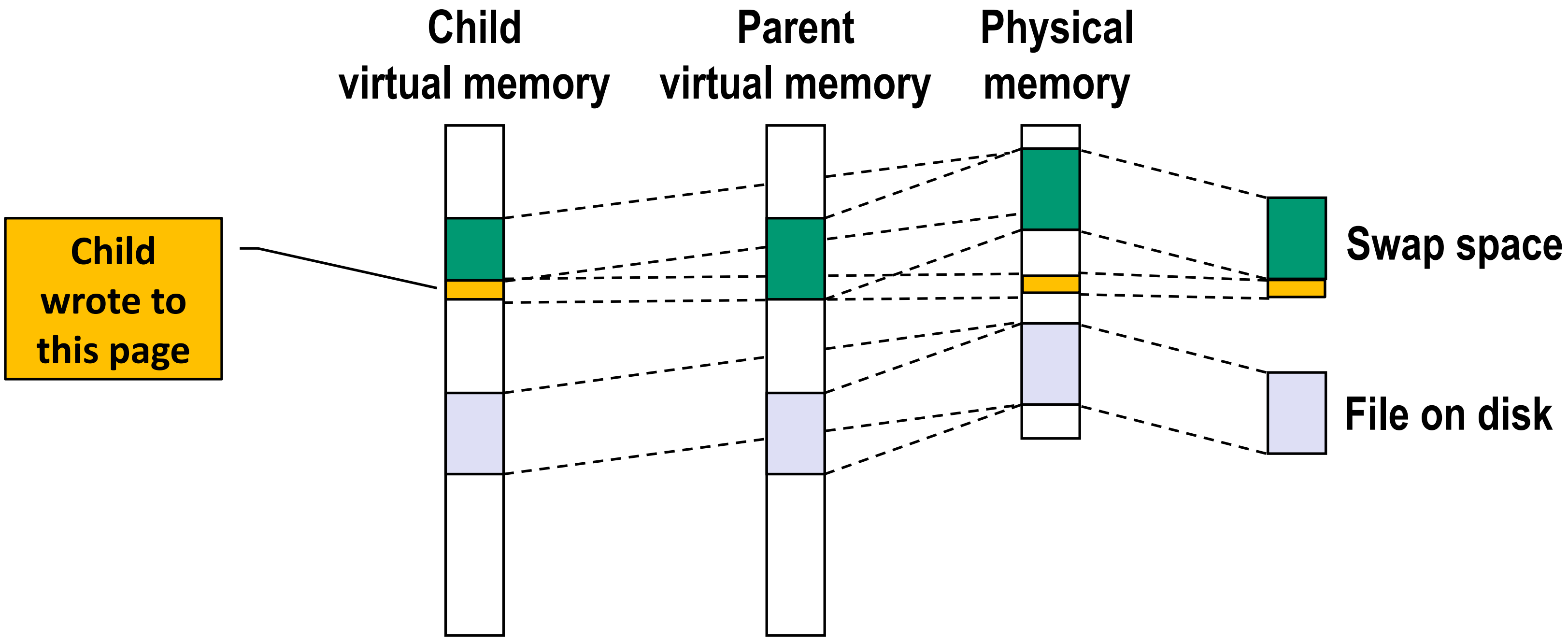
- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Copy-on-write sharing



- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Copy-on-write sharing



- **Clever trick:**
  - Just duplicate the page tables
  - Mark everything read only
  - Copy only on write faults

Summary

- **Multi-level page tables reduce total memory consumption of page tables**
- **Translation lookaside buffers reduce time cost of translation**
- **Real systems have 3 to 5 levels of page table**
- **Virtual memory makes nifty things possible**
    - Memory protection and process isolation
    - Paging/swapping (disk as extra RAM)
    - Memory-mapped files (RAM as cache for disk)
    - Copy-on-write sharing

Control Flow

- **Processors do only one thing:**
  - From startup to shutdown, each CPU core simply reads and executes (interprets) a sequence of instructions, one at a time *
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

<startup>

inst$_1$

inst$_2$

inst$_3$

…

inst$_n$

<shutdown>

* Externally, from an architectural viewpoint (internally, the CPU may use parallel out-of-order execution)

Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- **Insufficient for a useful system:**
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires

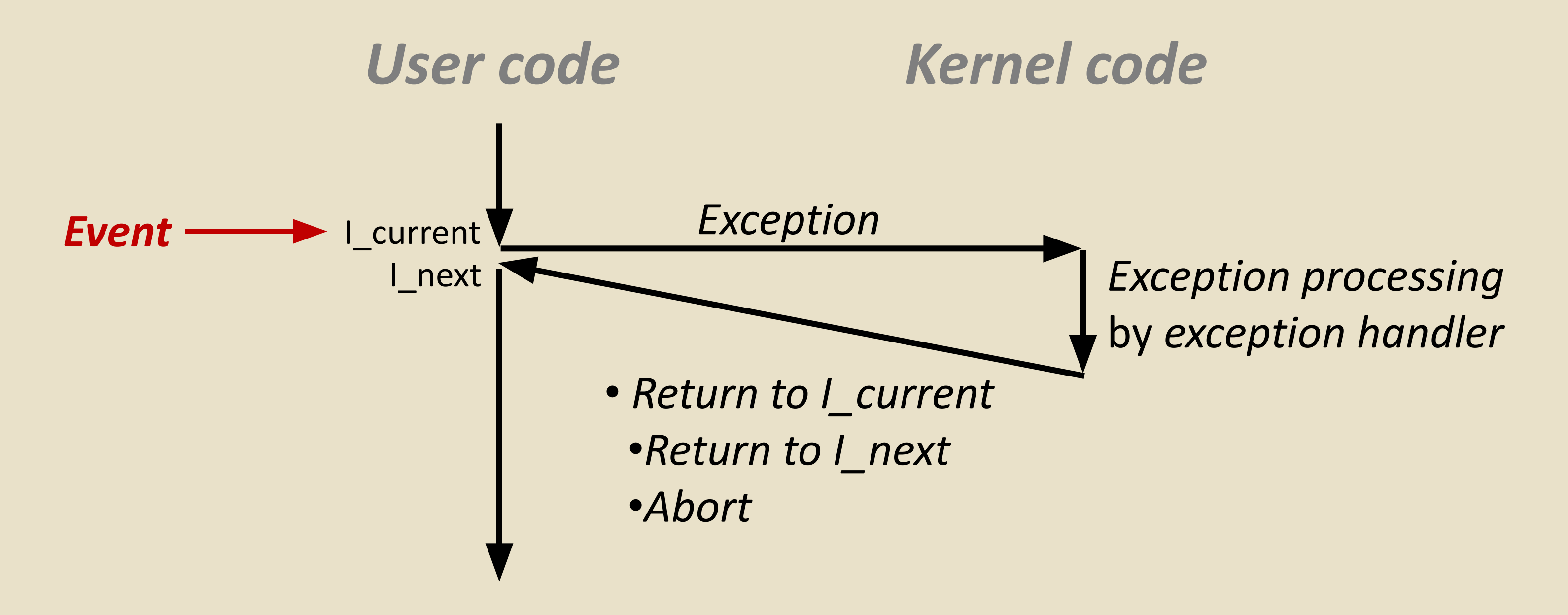
- **System needs mechanisms for "exceptional control flow"**

Exceptional Control Flow

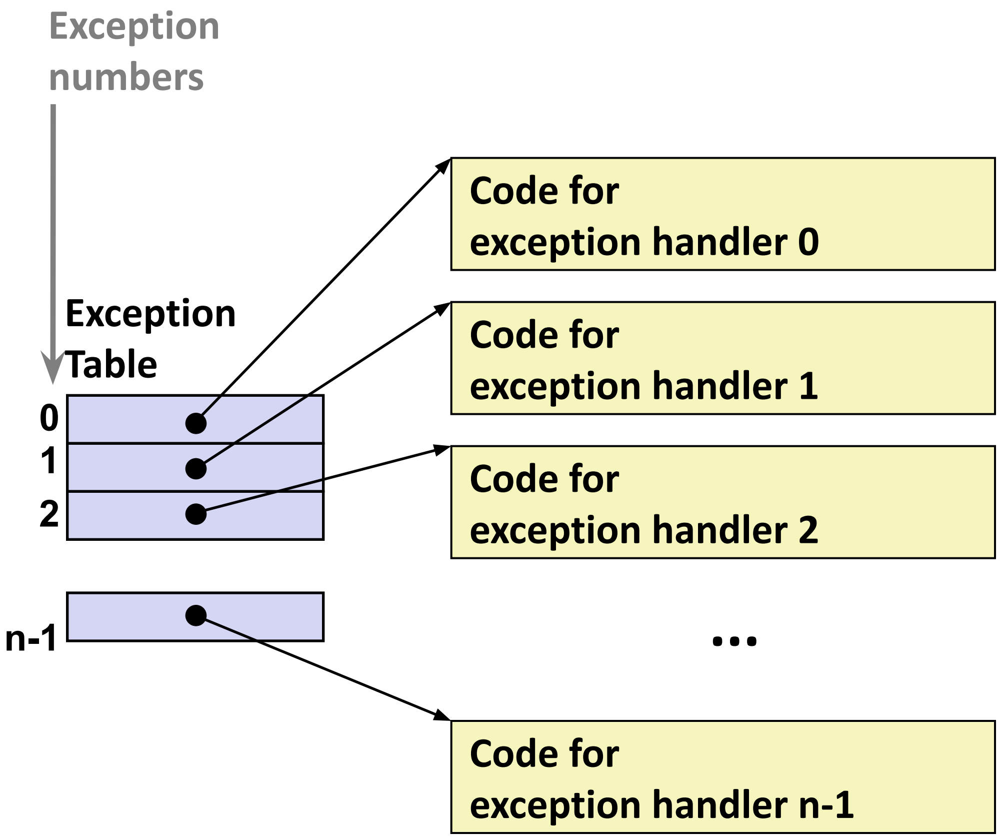
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 4. **Nonlocal jumps**: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

Exceptions

- **An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)**
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

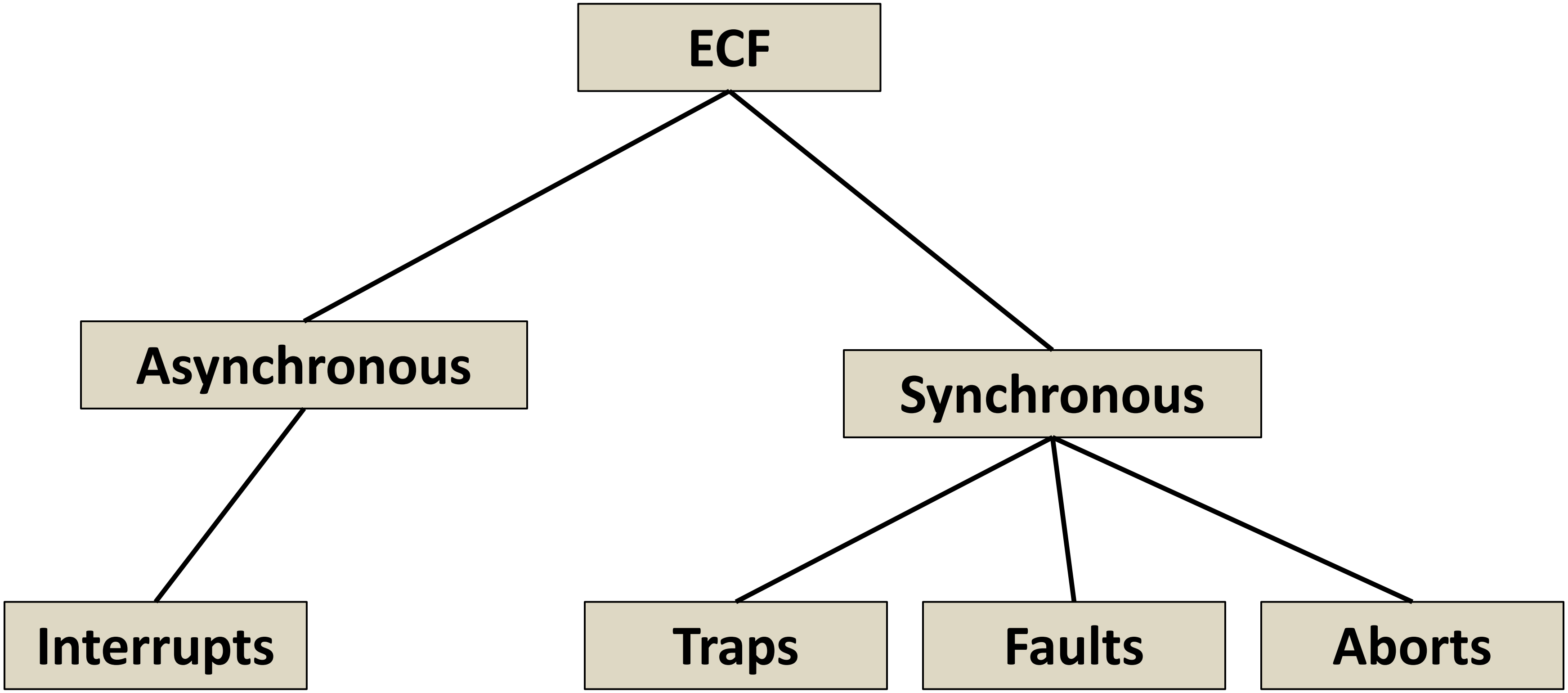


Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

(partial) Taxonomy



ECF

Asynchronous

Synchronous

Interrupts

Traps

Faults

Aborts

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional, set program up to "trip the trap" and do something
    - Examples: ***system calls***, gdb breakpoints
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

System Calls

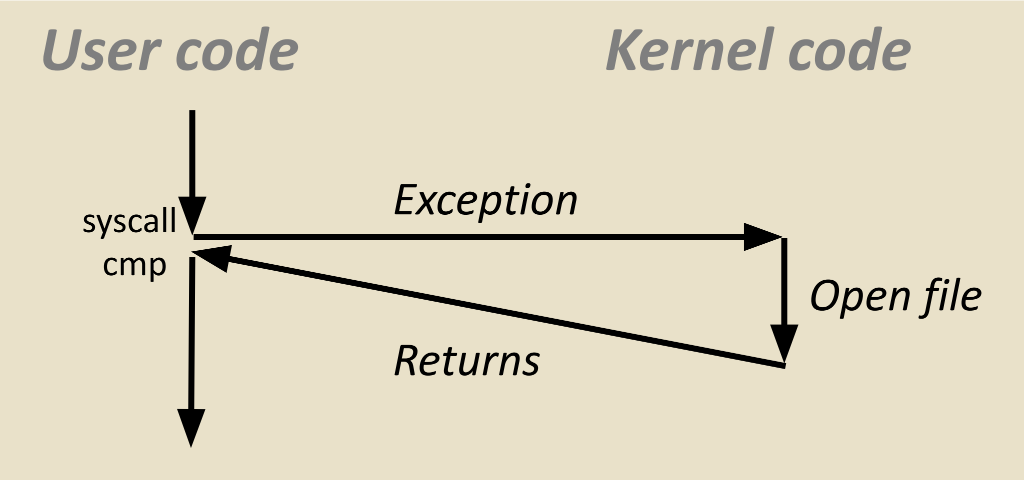
- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	***Name***	***Description***
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

System Call Example: Opening File

- User calls: **`open(filename, options)`**
- Calls **`__open`** function, which invokes system call instruction **`syscall`**

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax    # open is syscall #2
e5d7e:   0f 05                   syscall               # Return value in %rax
e5d80:   48 3d 01 f0 ff ff    cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

System Call

- User calls: **`open (f`**
- Calls __**`open`** funct

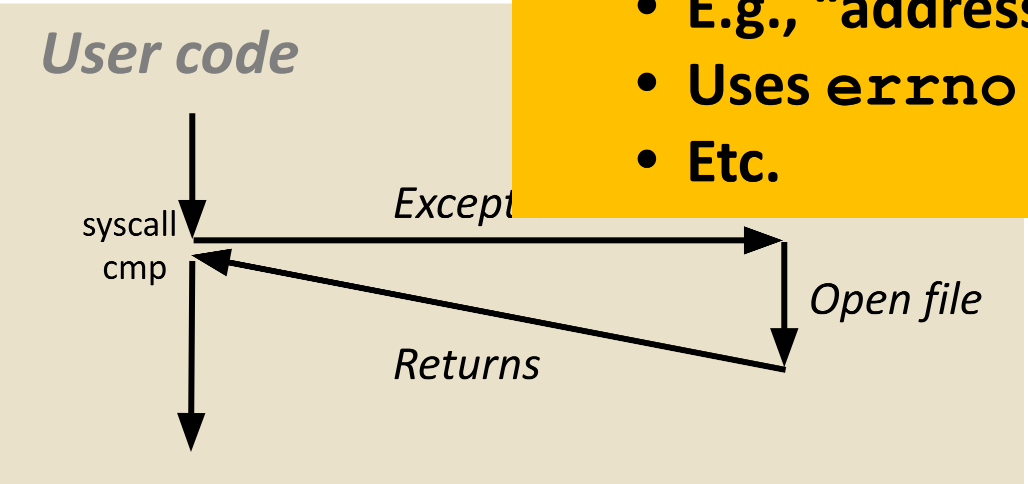
```
00000000000e5d70 <_
...
e5d79:   b8 02 00 00 00
e5d7e:   0f 05              s
e5d80:   48 3d 01 f0 ff ff
...
e5dfa:   c3                re
```

Almost like a function call

- **Transfer of control**
- **On return, executes next instruction**
- **Passes arguments using calling convention**
- **Gets result in `%rax`**

One Important exception!

- **Executed by Kernel**
- **Different set of privileges**
- **And other differences:**
  - **E.g., "address" of "function" is in `%rax`**
  - **Uses `errno`**
  - **Etc.**



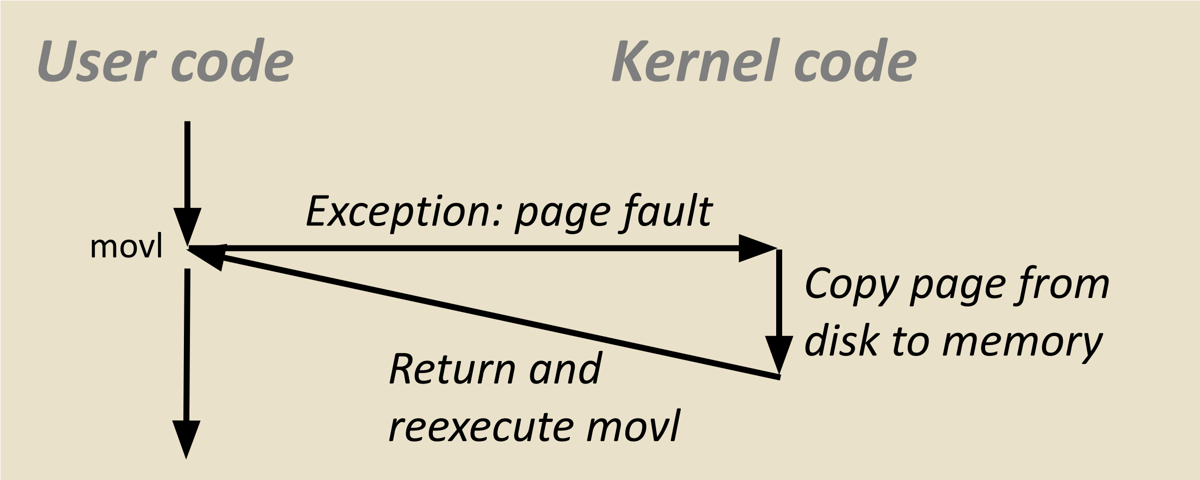
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:   c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



User code

Kernel code

movl

Exception: page fault

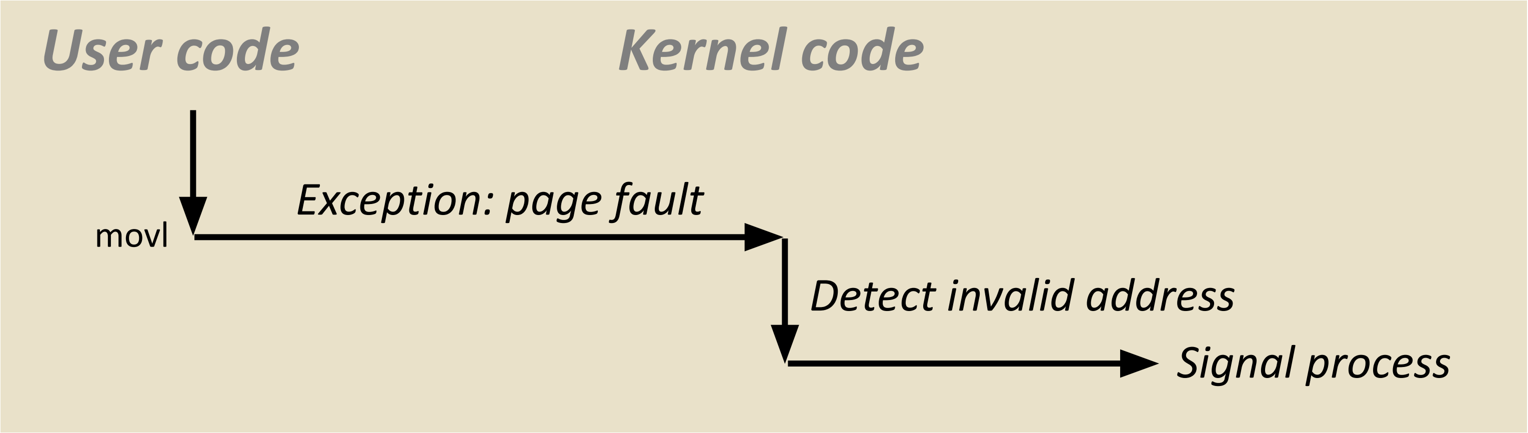
Copy page from disk to memory

Return and reexecute movl

Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

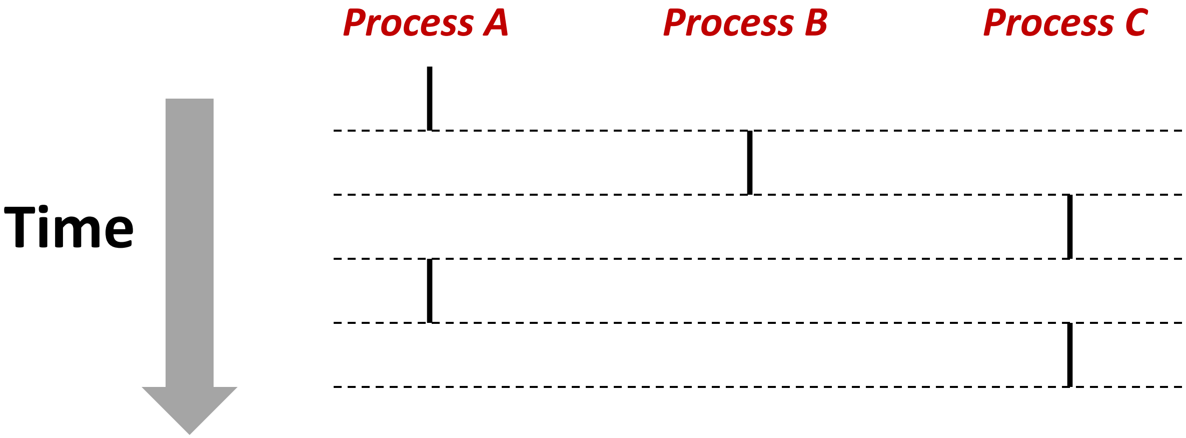
```
80483b7:   c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Sends **`SIGSEGV`** signal to user process
- User process exits with “segmentation fault”

Concurrent Processes

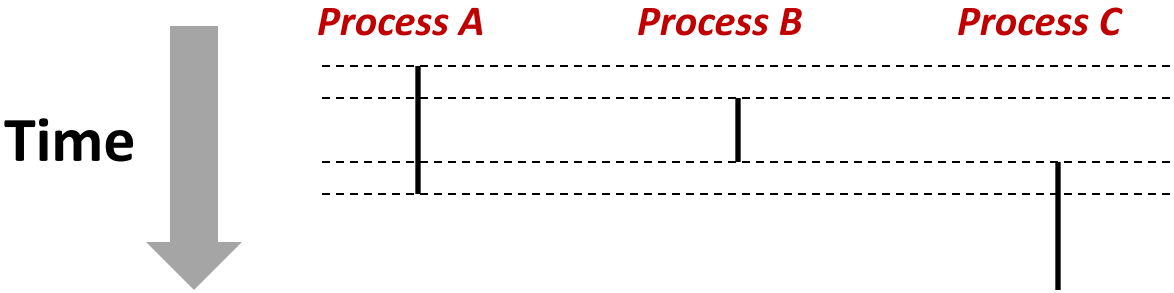
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

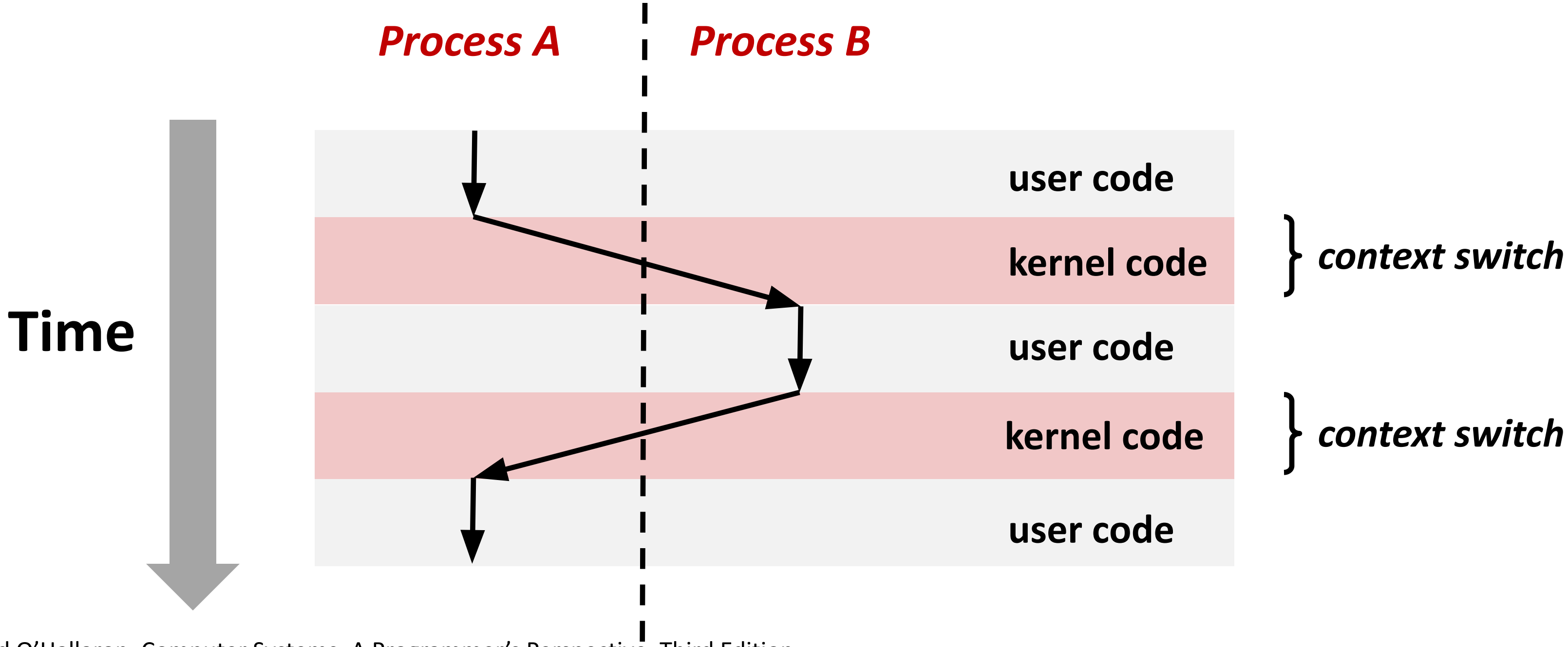
- **Control flows for concurrent processes are physically disjoint in time**

- **However, we can think of concurrent processes as running in parallel with each other**



Context Switching

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **Control flow passes from one process to another via a *context switch***



System Call Error Handling

- **On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.**
- **Hard and fast rule:**
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`
- **Example:**

```
    if ((pid = fork()) < 0) {
        fprintf(stderr, "fork error: %s\n", strerror(errno));
        exit(-1);
    }
```

Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(-1);
}
```

```
 if ((pid = fork()) < 0)
    unix_error("fork error");
```

Note: csapp.c exits with 0.

- But, must think about application. Not alway appropriate to exit when something goes wrong.

Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens[1]-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```
  pid = Fork();
```

- NOT what you generally want to do in a real application

[1] e.g., in “UNIX Network Programming: The sockets networking API“ W. Richard Stevens

Obtaining Process IDs

- **`pid_t getpid(void)`**
    - Returns PID of current process

- **`pid_t getppid(void)`**
    - Returns PID of parent process

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- **`void exit(int status)`**
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- **`exit` is called once but never returns.**

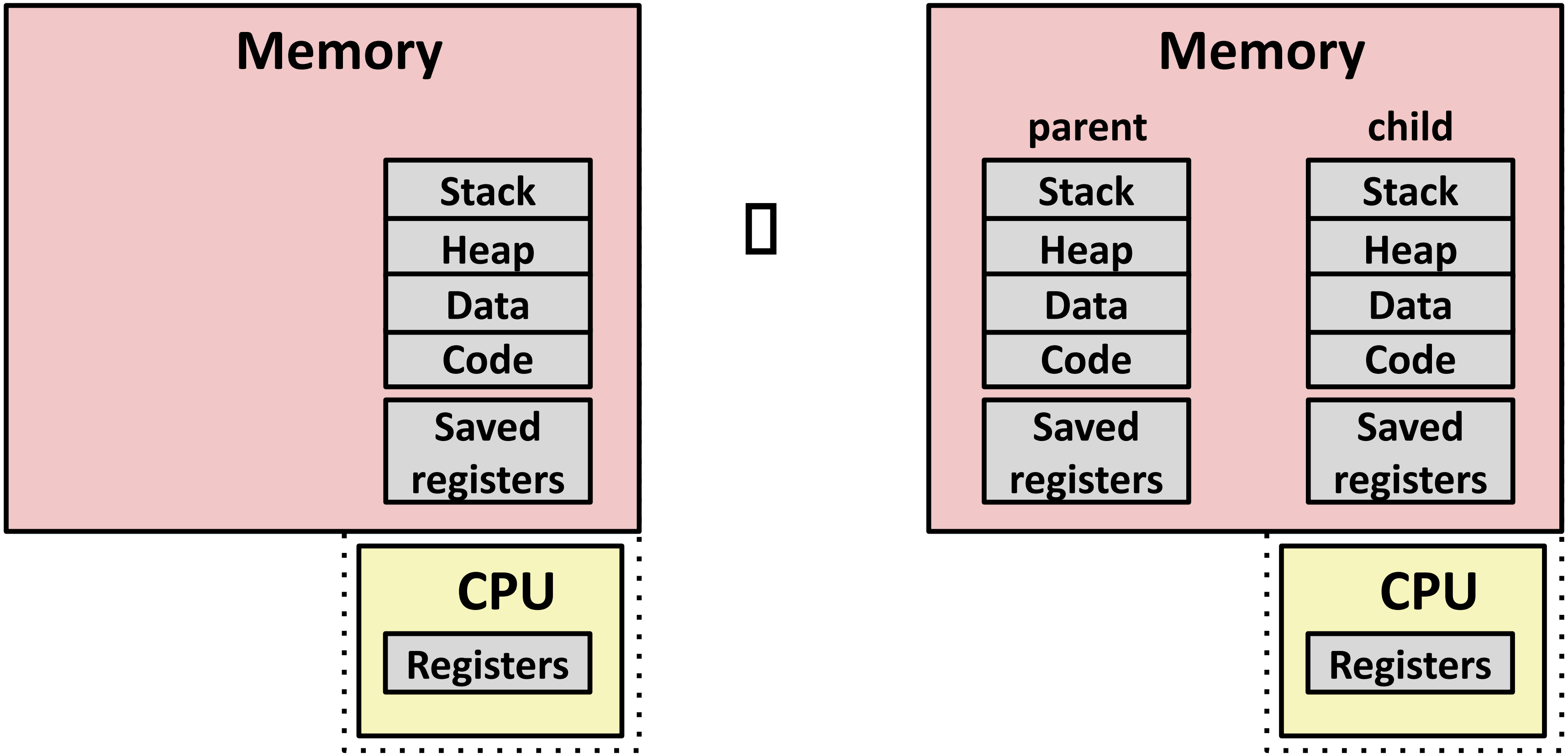
Creating Processes

- ***Parent process* creates a new running *child process* by calling `fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

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

Conceptual View of `fork`



- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of parent or child

The `fork` Function Revisited

- **VM and memory mapping explain how `fork` provides private address space for each process.**

- **To create virtual address for new process:**
  - Create exact copies of current **`mm_struct`**, **`vm_area_struct`**, and page tables.
  - Flag each page in both processes as read-only
  - Flag each **`vm_area_struct`** in both processes as private COW

- **On return, each process has exact copy of virtual memory.**

- **Subsequent writes create new pages using COW mechanism.**

`fork` Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {   /* Child */
        printf("child : x=%d\n", ++x);
    return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}                                  fork.c
```

- **Call once, return twice**
- **Concurrent execution**
  - **Can't predict execution order of parent and child**

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

`fork` Example

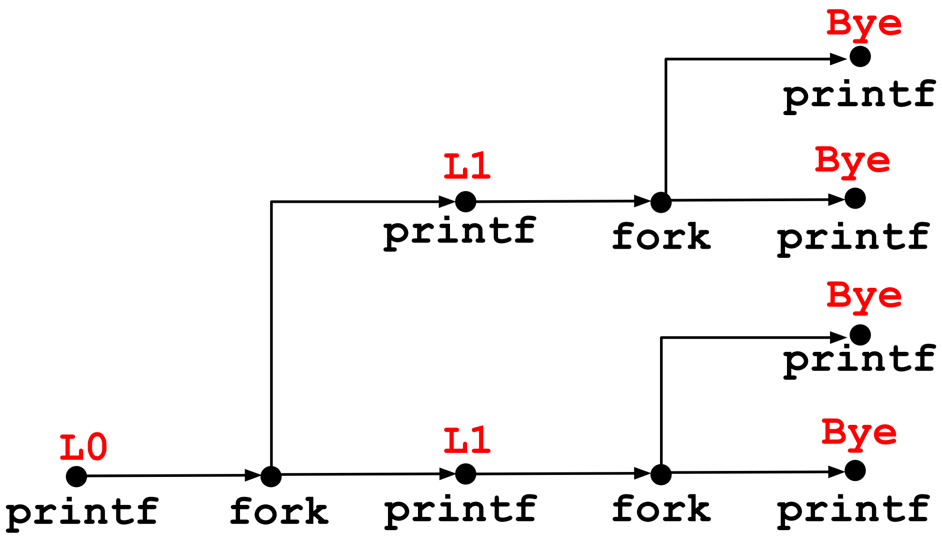
```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

`fork` Example: Two consecutive `fork`s

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



Feasible output:

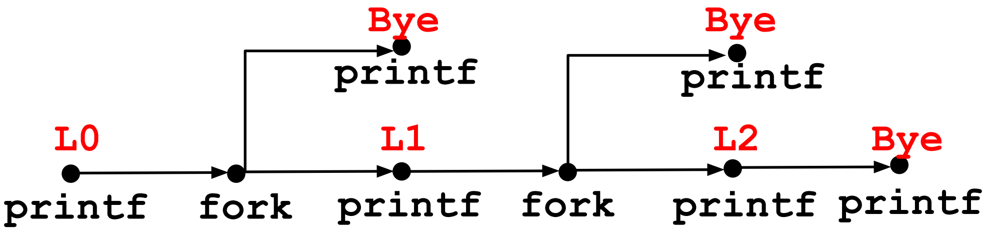
```
L0
L1
Bye
Bye
L1
Bye
Bye
```

Infeasible output:

```
L0
Bye
L1
Bye
L1
Bye
Bye
```

`fork` Example: Nested `fork`s in parent

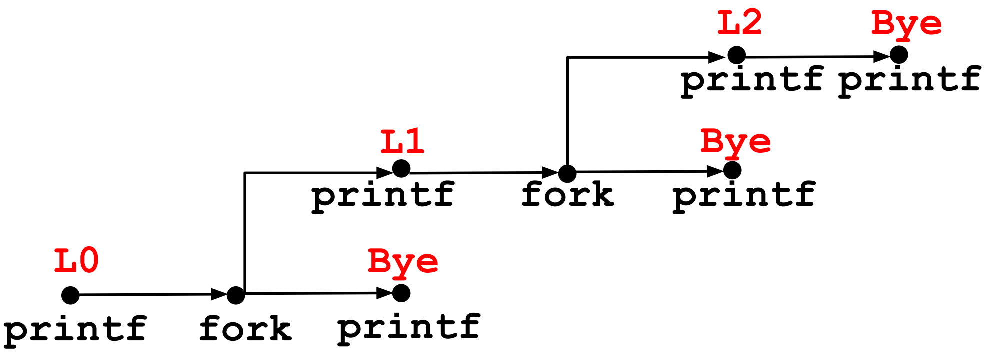
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
     }
    printf("Bye\n");
}                                    forks.c
```



Feasible or Infeasible?	Feasible or Infeasible?
L0	L0
Bye	L1
L1	Bye
Bye	Bye
Bye	L2
L2	Bye
Infeasible	Feasible

`fork` Example: Nested `fork`s in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}                           forks.c
```



Feasible or Infeasible?	Feasible or Infeasible?
L0	L0
Bye	Bye
L1	L1
Bye	L2
Bye	Bye
L2	Bye
Infeasible	Feasible

Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then the orphaned child should be reaped by **`init`** process (pid == 1)
    - Unless ppid == 1! Then need to reboot…
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
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 */
    }
}
```

```
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
```

- **`ps`** shows child process as “defunct” (i.e., a zombie)
- Killing parent allows child to be reaped by **`init`**

Non-terminating Child Example

```
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);
    }
}
```

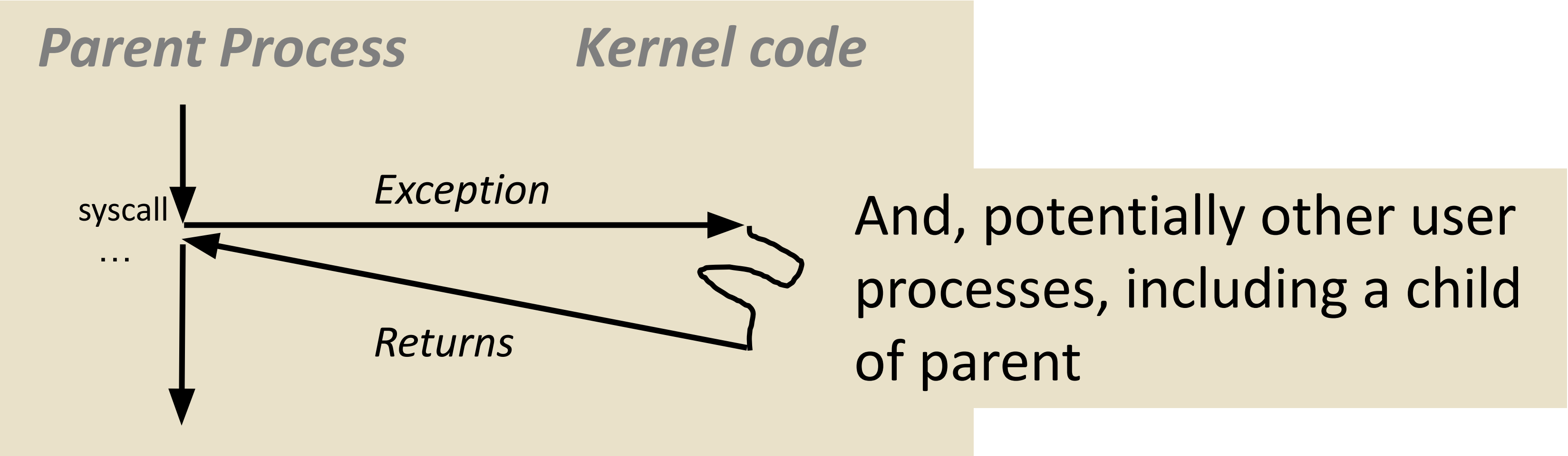
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9   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

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

`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Implemented as syscall



And, potentially other user processes, including a child of parent

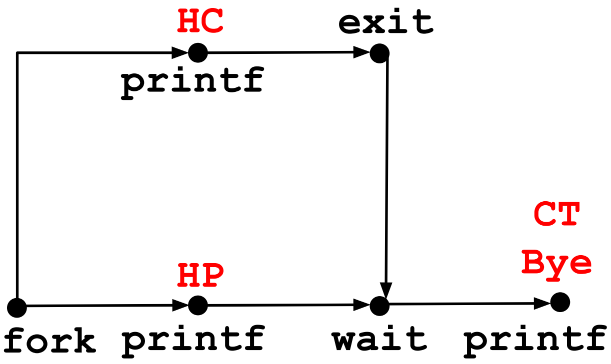
`wait`: Synchronizing with Children

- **Parent reaps a child by calling the `wait` function**

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the `pid` of the child process that terminated
  - If `child_status != NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

`wait`: Synchronizing with Children

```
void fork9() {
    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(s):

```
HC   HP
HP   HC
CT   CT
Bye  Bye
```

Infeasible output:

```
HP
CT
Bye
HC
```

Another 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, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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 terminate abnormally\n", wpid);
    }
}
```

forks.c

`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 (see textbook)

```
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 = N-1; i >= 0; 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 terminate abnormally\n", wpid);
    }
}
```

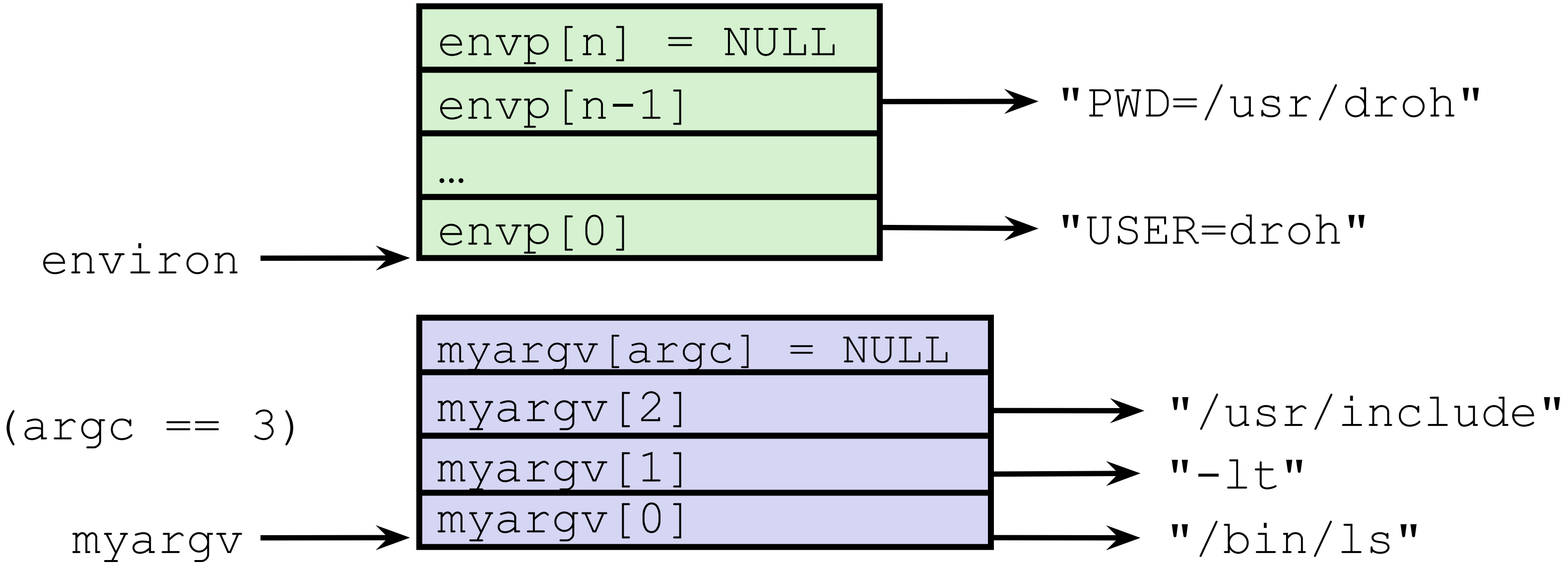
forks.c

`execve`: Loading and Running Programs

- **`int execve(char *filename, char *argv[], char *envp[])`**
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

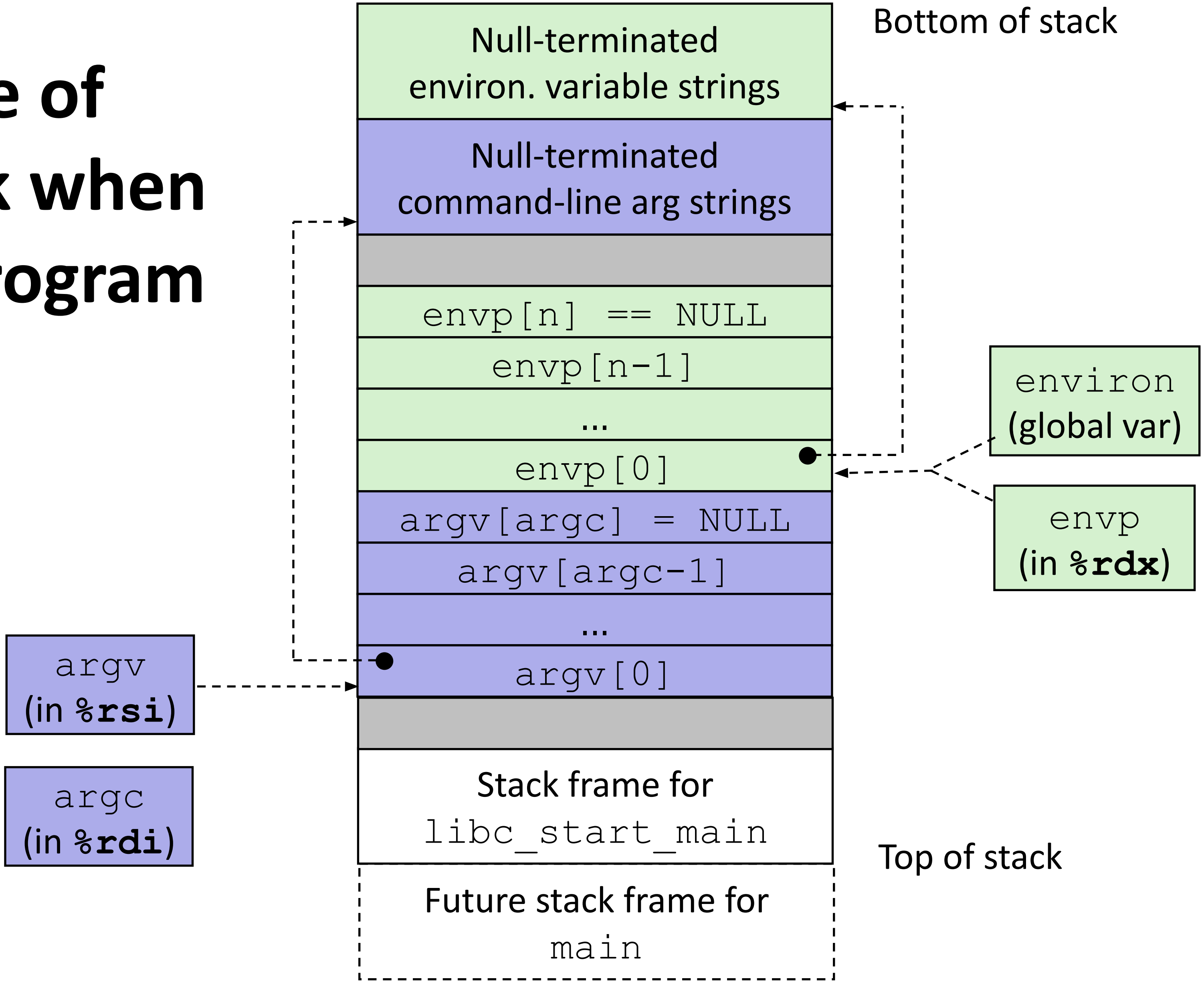
`execve` Example

- **Execute** `"/bin/ls -lt /usr/include"` **in child process using current environment:**

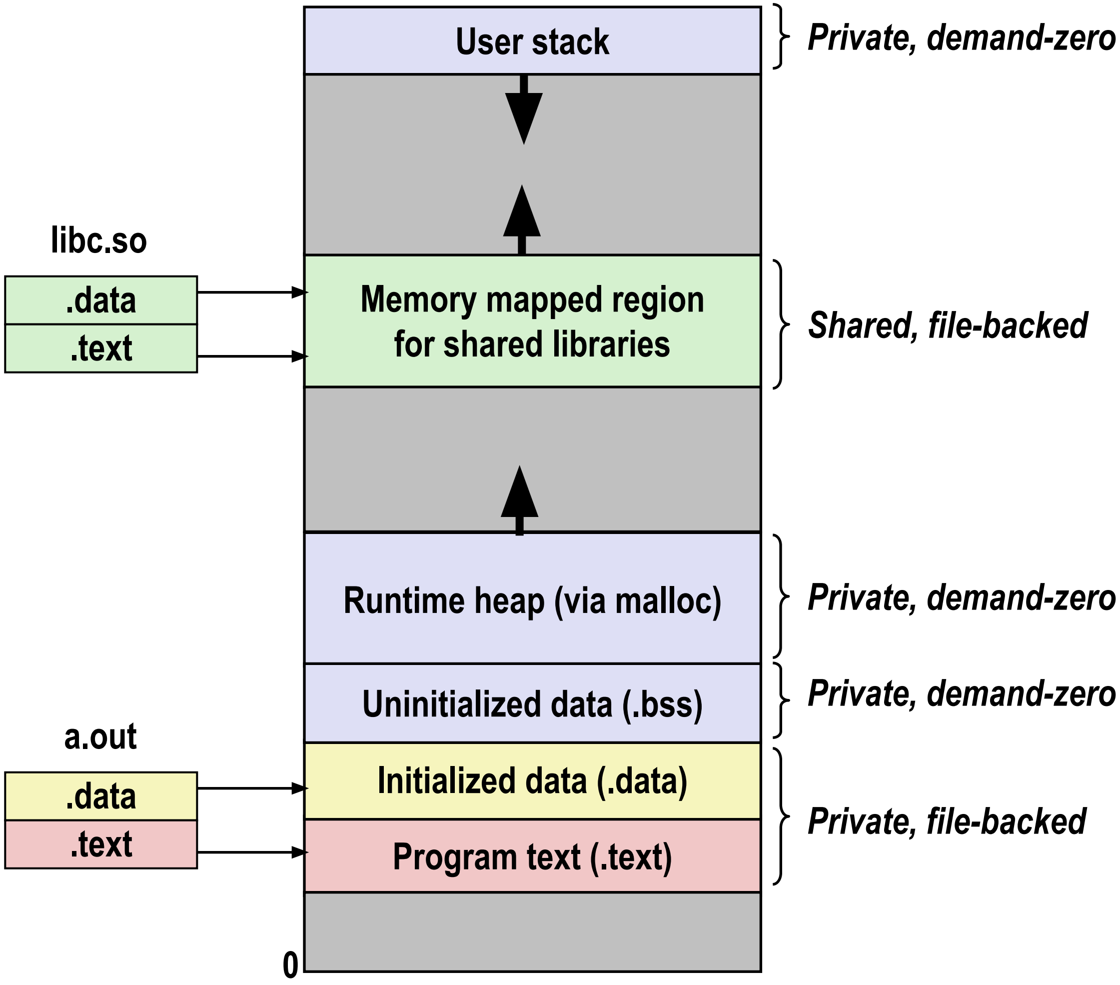


```
if ((pid = Fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Structure of the stack when a new program starts



The `execve` Function Revisited



- **To load and run a new program `a.out` in the current process using `execve`:**
- **Free `vm_area_struct`’s and page tables for old areas**
- **Create `vm_area_struct`’s and page tables for new areas**
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.
- **Set PC to entry point in `.text`**
  - Linux will fault in code and data pages as needed.