Cache Example, System Control Flow

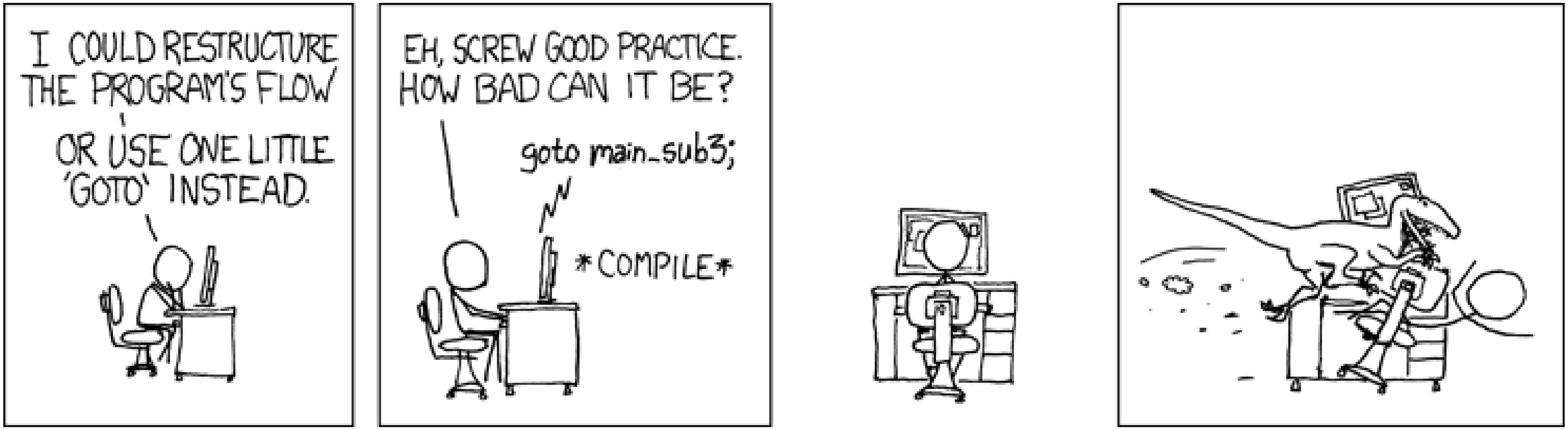
CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma, Hunter Zahn, John Kaltenbach, Kevin Bi
Sachin Mehta, Suraj Bhat, Thomas Neuman, Waylon Huang
Xi Liu, Yufang Sun



http://xkcd.com/292/

Administrivia

- Homework 3 due Friday
- Lab 4 released Wednesday
- Midterm Scores on Catalyst
  - +6 from Gradescope score; please double-check
- **Midterm Clobber Policy**
  - Final will be cumulative (half midterm, half post-midterm)
  - If you perform better relative to the rest of the class on the midterm portion of the final, you replace your midterm score
  - Replacement score = $(\mathrm{F_{MT}\ score} - \mathrm{F_{MT}\ avg}) \times \frac{\mathrm{MT\ stddev}}{\mathrm{F_{MT}\ stddev}} + \mathrm{MT\ mean}$
  - Course policies on website have been updated

Anatomy of a Cache Question

- Cache questions come in a few flavors:
  1) TIO Breakdown
  2) For fixed cache parameters, analyze the performance of the given code/sequence
  3) ~~For fixed cache parameters, find best/worst case scenarios~~
  4) For given code/sequence, how does changing your cache parameters affect performance?
  5) Average Memory Access Time (AMAT)

The Cache

- What are the important cache parameters?
  - Must figure these out from problem description
  - Address size, cache size, block size, associativity, replacement policy
  - Solve for TIO breakdown, # of sets, management bits
- What starts in the cache?
  - ~~Not always specified (best/worst case)~~

Code: Arrays

- Elements stored contiguously in memory
  - Ideal for spatial locality – if used properly
  - Different arrays not necessarily next to each other
- Remember to account for data size!
  - char is 1 B, int/float is 4 B, long/double is 8 B
- Pay attention to access pattern
  - Touch *all* elements (e.g. shift, sum)
  - Touch *some* elements (e.g. histogram, stride)
  - How many times do we touch each element?

Code: Linked Lists/Structs

- Nodes stored separately in memory
  - Addresses of nodes may be very different
  - Method of linking and ordering of nodes are important
- Remember to account for size/ordering of struct elements
- Pay attention to access pattern
  - Generally must start from "head"
  - How many struct elements are touched?

Access Patterns

- How many hits within a single block once it is loaded into cache?
- Will block still be in cache when you revisit its elements?
- Are there special/edge cases to consider?
  - Usually edge of block boundary or edge of cache size boundary

Cache Example Problem

a) 1 GiB address space, 100 cycles to go to memory. Fill in the following table:

	L1	L2
Cache Size	32 KiB	512 KiB
Block Size	8 B	32 B
Associativity	4-way	Direct-mapped
Hit Time	1 cycle	33 cycles
Miss Rate	10%	2%
Write Policy	Write-through	Write-through
Replacement Policy	LRU	n/a
Tag	17	11
Index	10	14
Offset	3	5
AMAT	AMAT L1 = 1 + 0.1 * 35 = 4.5	AMAT L2 = 33 + 0.02 * 100 = 35

Cache Example Problem

Using *only* L1$, `char A[]` is block aligned, and `SIZE=2^25`:

```
char *A = (char *) malloc (SIZE*sizeof(char));
/* number of STRETCHes */
for(i=0; i<(SIZE/STRETCH); i++) {
    /* go up to STRETCH */
    for(j=0;j<STRETCH;j++)     sum  += A[i*STRETCH+j];
    /* down from STRETCH */
    for(j=STRETCH-1;j>=0;j--) prod *= A[i*STRETCH+j];
}
```

- What does our access pattern of `A[]` look like?

Cache Example Problem

Using *only* L1$, `char A[]` is block aligned, and `SIZE=2^25`:

```
char *A = (char *) malloc (SIZE*sizeof(char));
/* number of STRETCHes */
for(i=0; i<(SIZE/STRETCH); i++) {
   /* go up to STRETCH */
   for(j=0;j<STRETCH;j++)     sum  += A[i*STRETCH+j];
   /* down from STRETCH */
   for(j=STRETCH-1;j>=0;j--) prod += A[i*STRETCH+j];
}
```

- What does our access pattern of `A[]` look like?
  - Mostly stride-by-1 with step size `sizeof(char)` = 1 B
  - 2nd inner `for` loop hits same indices as 1st inner `for` loop, but in reverse order
  - Always traverse full `SIZE`, regardless of `STRETCH`

Cache Example Problem

Using *only* L1$, `char A[]` is block aligned, and `SIZE=2^25`:

```
char *A = (char *) malloc (SIZE*sizeof(char));
for(i=0; i<(SIZE/STRETCH); i++) {
   for(j=0;j<STRETCH;j++)     sum  += A[i*STRETCH+j];
   for(j=STRETCH-1;j>=0;j--) prod += A[i*STRETCH+j];
}
```

b) As we double our STRETCH from 1 to 2 to 4 (…etc), we notice the number of cache misses doesn't change! What is the largest value of STRETCH *before* cache misses changes?

2^15, when working set size (`STRETCH*sizeof(char)`) exactly equals cache size **C**

Cache Example Problem

Using *only* L1$, `char A[]` is block aligned, and `SIZE=2^25`.

Cache size C = 32 KiB, block size K = 8 B, associativity N = 4.

```
char *A = (char *) malloc (SIZE*sizeof(char));
for(i=0; i<(SIZE/STRETCH); i++) {
   for(j=0;j<STRETCH;j++)      sum  += A[i*STRETCH+j];
   for(j=STRETCH-1;j>=0;j--) prod += A[i*STRETCH+j];
}
```

c) If we double our STRETCH from (b), what is the ratio of cache *hits* to *misses*?

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
```

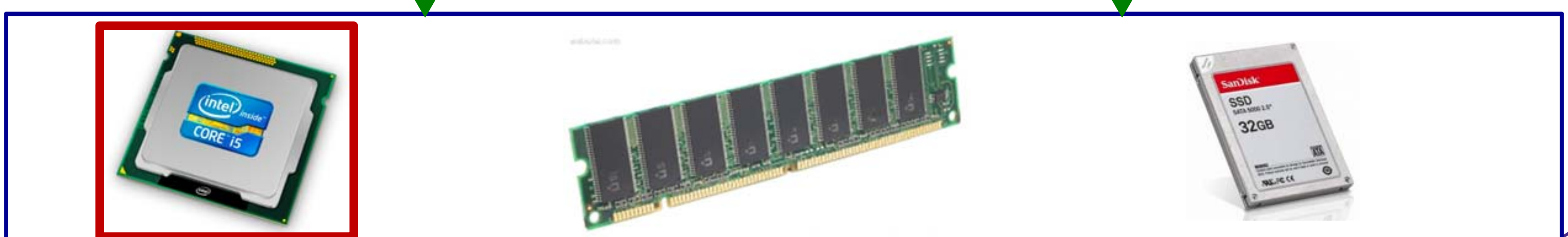
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:



- Data & addressing
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- Arrays & structs
- Memory & caches
- **Processes**
- Virtual memory
- Memory allocation
- Java vs. C

Leading Up to Processes

- System Control Flow
  - **Control flow**
  - **Exceptional control flow**
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)

Control Flow

- **So far:** we've seen how the flow of control changes as a *single program* executes
- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs
- *Exceptional* control flow is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU 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

time

<startup>
$instr_1$
$instr_2$
$instr_3$
…
$instr_n$
<shutdown>

Altering the Control Flow

- Up to now: two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in *program state*
- Processor also needs to react to changes in *system state*
  - Unix/Linux user hits "Ctrl-C" at the keyboard
  - User clicks on a different application's window on the screen
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires
- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for "*exceptional*" control flow!

Java Digression #1

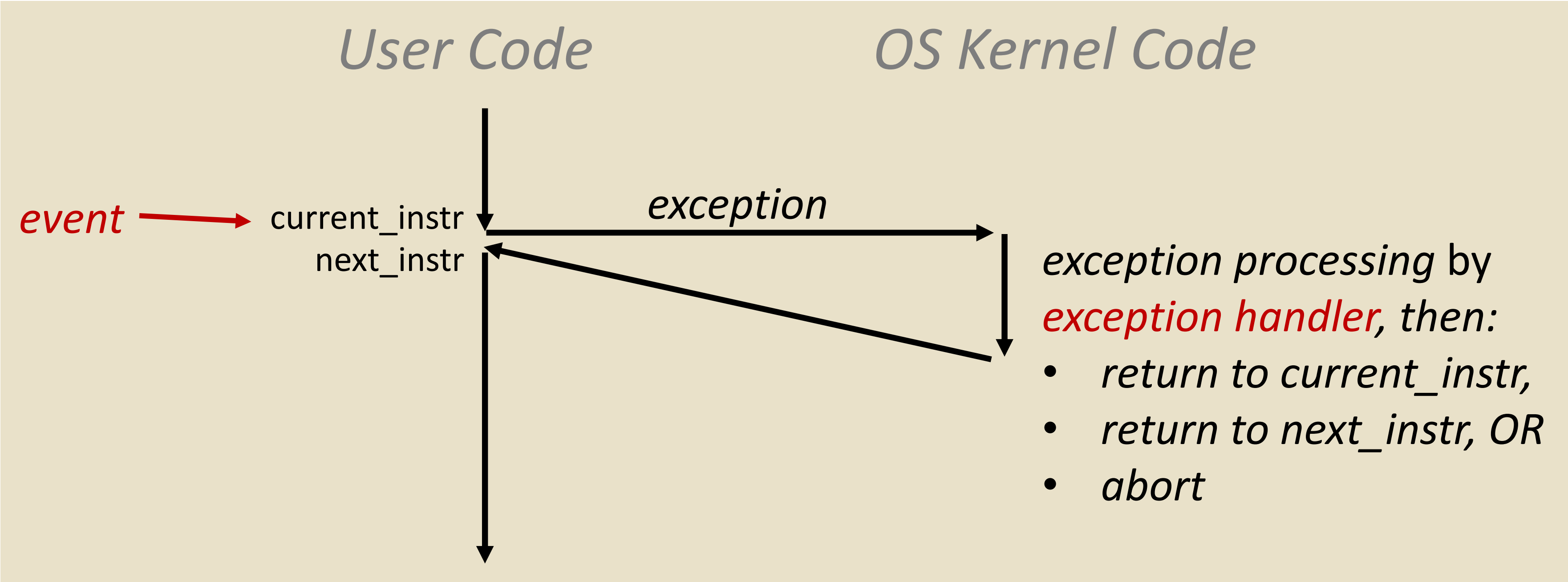
- Java has exceptions, but they're *something different*
  - Examples: NullPointerException, MyBadThingHappenedException, …
  - `throw` statements
  - `try/catch` statements ("throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none")
- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for "many call-stack returns at once"
  - Requires additions to the calling convention, but we already have the CPU features we need
- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently

Exceptional Control Flow

- Exists at all levels of a computer system
- Low level mechanisms
  - **Exceptions**
    - Change in processor's control flow in response to a system event (i.e., change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software
- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - We won't cover these – see CSE451 and CSE/EE474

Exceptions

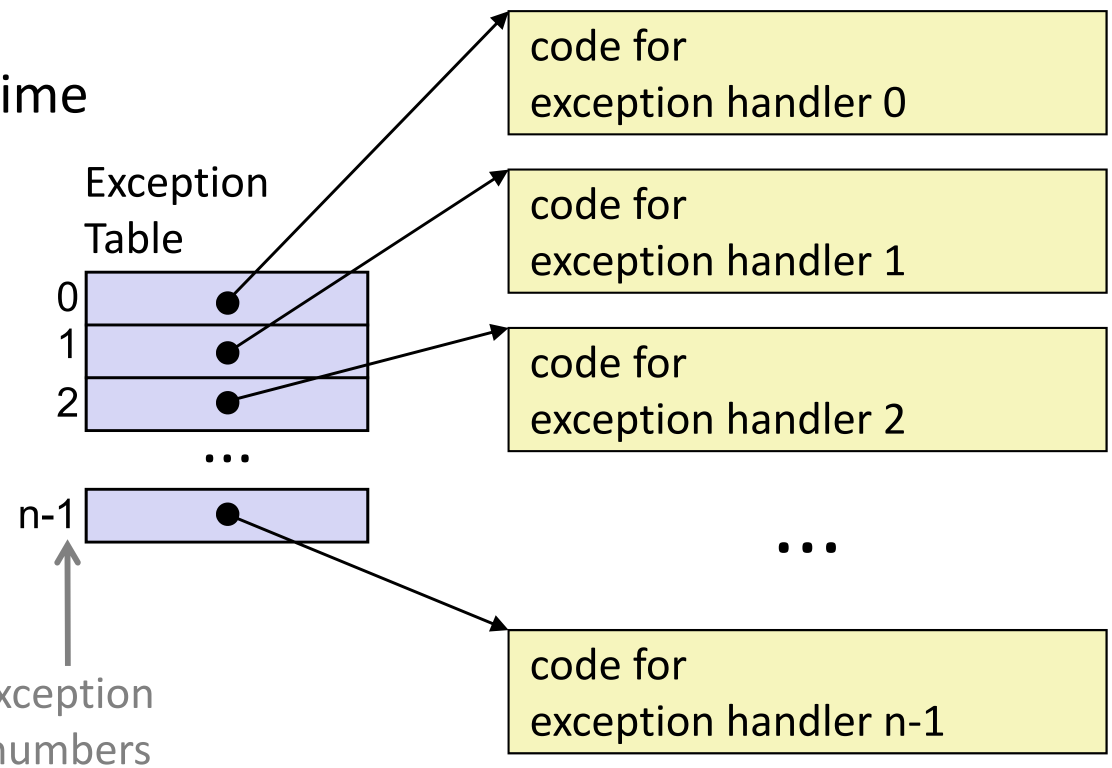
- An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C



- *How does the system know where to jump to in the OS?*

Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - 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



Exception Table (Excerpt)

Exception Number	*Description*	*Exception Class*
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-255	OS-defined	Interrupt or trap

Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - **Asynchronous exceptions (interrupts)**
  - **Synchronous exceptions (traps & faults)**

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to "next" instruction
- Examples:
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - **Intentional**: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - **Unintentional** but possibly recoverable
    - Examples: *page faults*, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - **Unintentional** and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program

System Calls

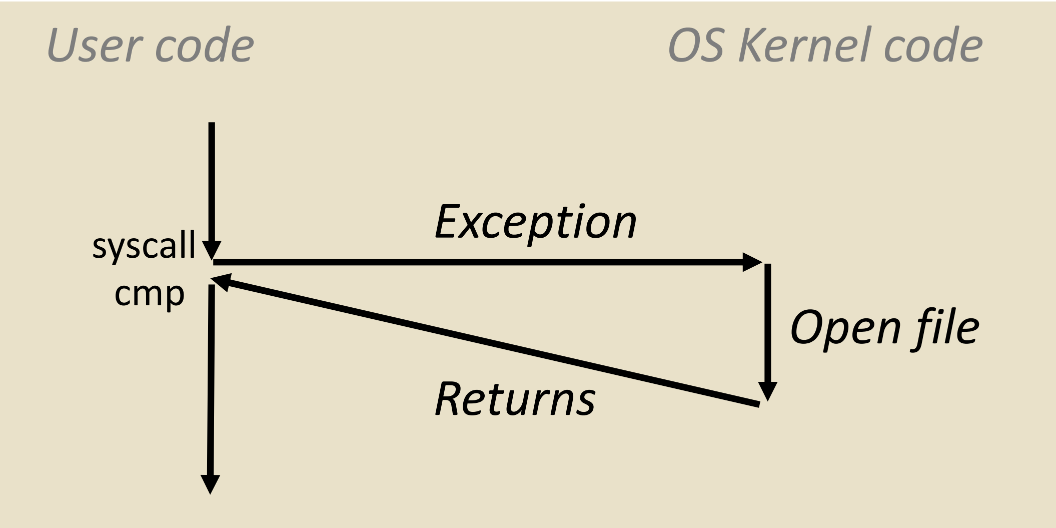
- Each system call has a unique ID number
- Examples for Linux on x86-64:

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

Traps Example: Opening File

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

```
0000000000e5d70 <__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
```



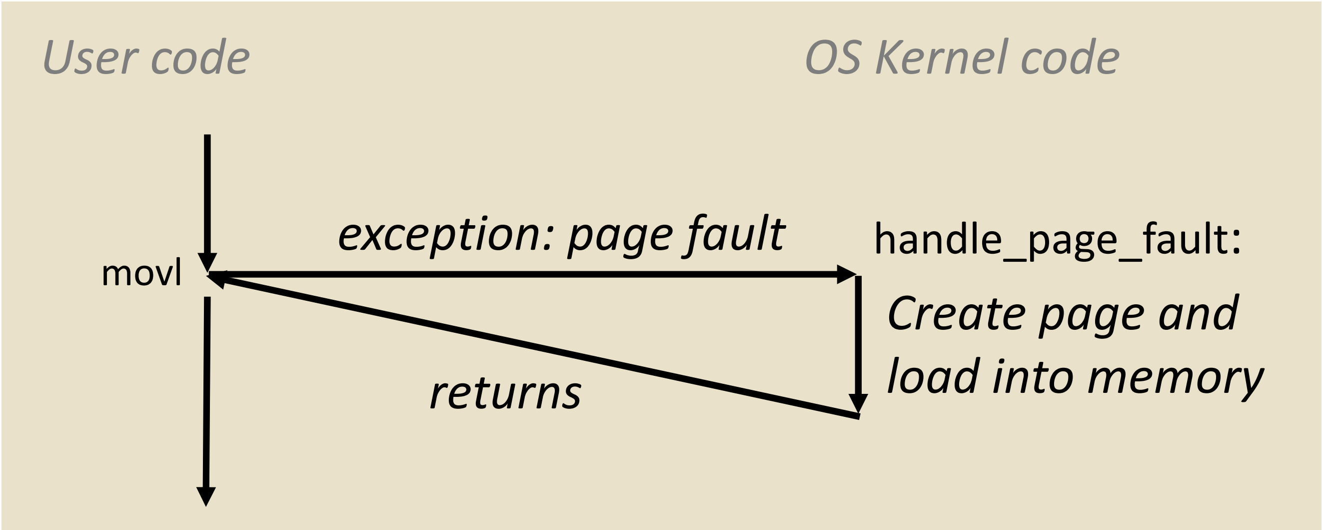
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

Fault Example: Page Fault

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

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

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

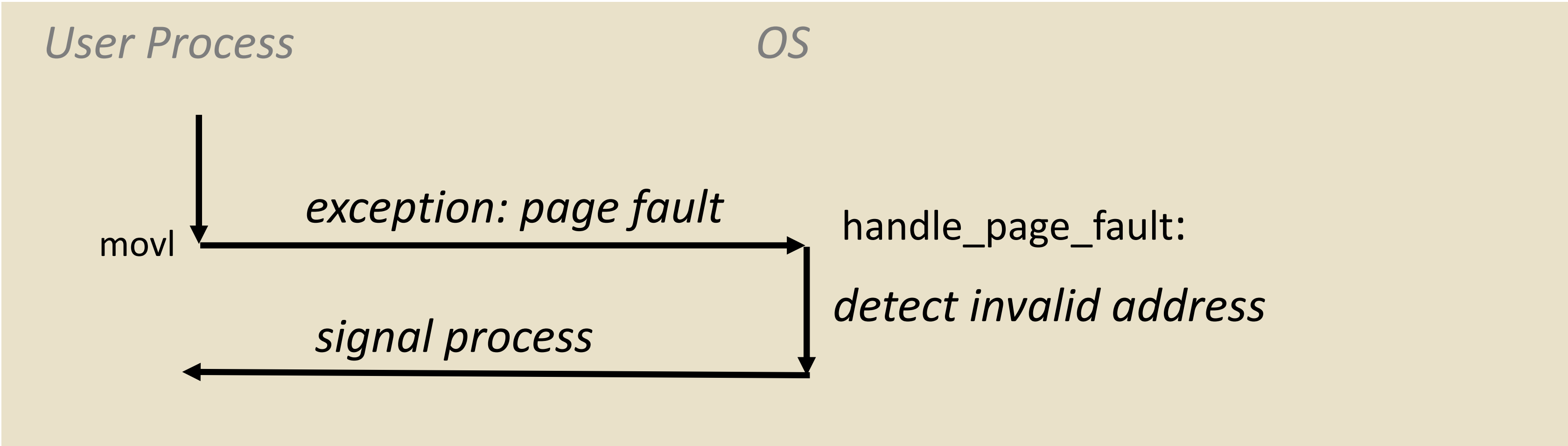


- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try

Fault Example: Invalid Memory Reference

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

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



- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

Summary

- Exceptions
  - Events that require non-standard control flow
  - Generated externally (interrupts) or internally (traps and faults)
  - After an exception is handled, one of three things may happen:
    - Re-execute the current instruction
    - Resume execution with the next instruction
    - Abort the process that caused the exception