CSCI 2330


when all 3 levels of the cache miss
so the cpu has to access the main memory
This little maneuver is gonna cost us 51 years
imgflip.com

Exam 2 graded

- High grade: 100
- Mean grade: 83
- Median grade: 84

TODO

- Assignment 4 due today

Overview

- Processes

Structure of course

- First third (Exam 1):
  - Data Representation
  - Memory
- Second third (Exam 2):
  - Machine Code
  - Memory exploits
- Final third (Final exam):
  - Memory hierarchy
  - Processes ← here

Processes

- Our systems model thus far has only looked at individual programs
- But modern computers have multiple programs running at any given time
- This is an important part of how computers work!

Operating system

- No agreed upon definition
- Roughly, collection of software for managing your computer
- Kernel – the core operating system program in charge of managing other programs and hardware



Operating system

- Kernel – the core operating system program in charge of managing other programs and hardware
- Has important responsibilities
- Including *protecting programs from each other*

Example

- Think of all of the horrible things you've done with pointers and memory
  - Overwriting random addresses
  - Failing to free all memory
- What happened?

Example

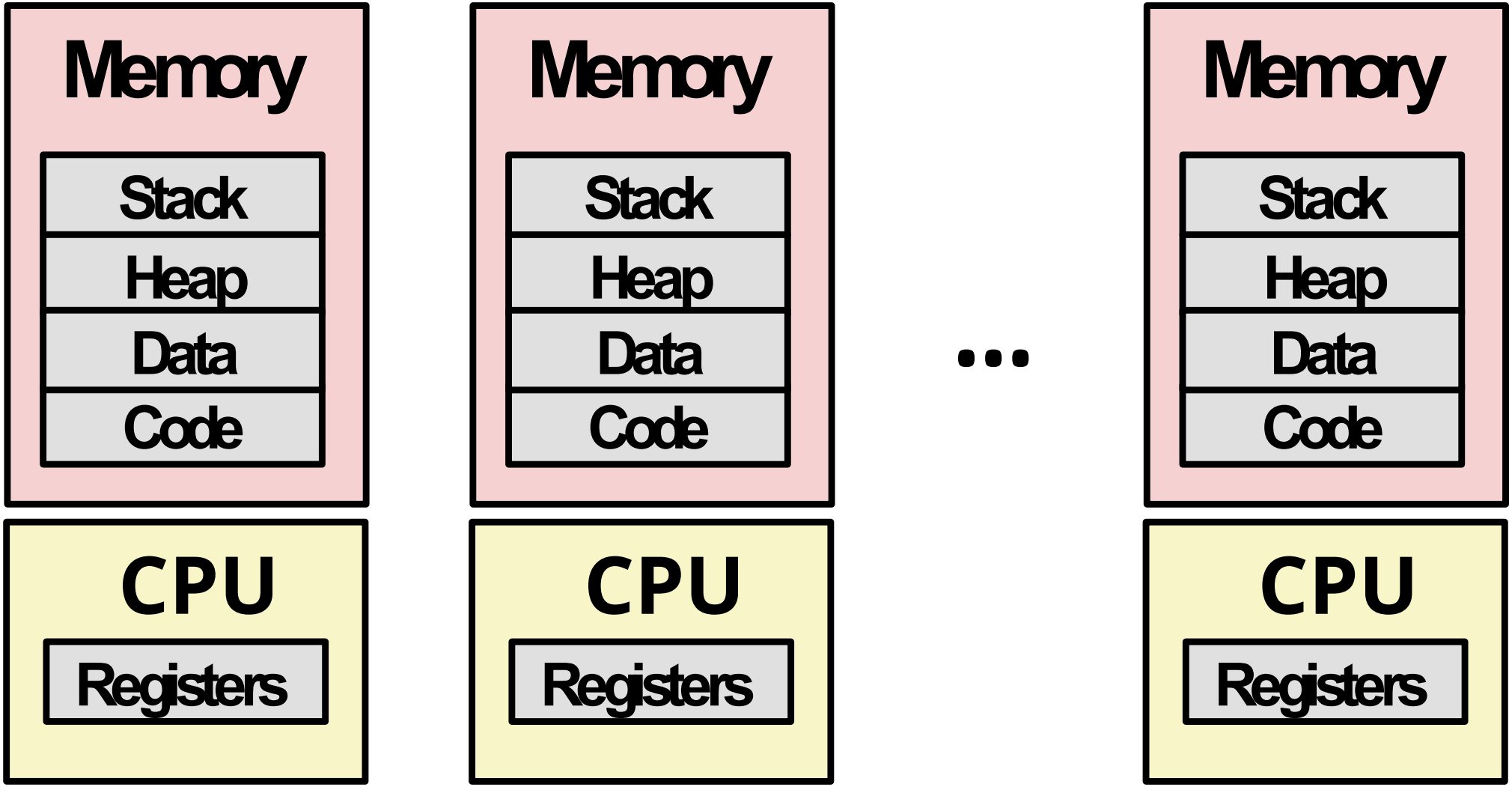
- Think of all of the horrible things you've done with pointers and memory
  - Overwriting random addresses
  - Failing to free all memory
- What happened?
  - The kernel protected the memory of other programs
  - The kernel freed the memory when your program exited

Processes

- A *process* is the state of a running program
- What makes up the state of a running program?

Processes

- A *process* is the state of a running program
- What makes up the state of a running program?
  - Memory contents (including code)
  - Register values (including program counter)
  - Much more!


Memory
Stack
Heap
Data
Code
CPU
Registers
Memory
Stack
Heap
Data
Code
CPU
Registers
...
Memory
Stack
Heap
Data
Code
CPU
Registers

Process vs. program

- Program – sequence of instructions
- Compare:
- Process – the state of a running program


jeffrey@polaris: ~
jeffrey@polaris: ~
jeffrey@polaris: ~
jeffrey@polaris: ~
jeffrey@polaris:~$

Process vs. program

- A program can have 0 or more processes
- A process has exactly 1 program

Process vs. window

- Window – a (typically) rectangular graphical user interface (GUI) element
- A process can create zero or more windows

Running multiple processes

- How does this work?
- Suppose we have one processor
- Remember: process state consists of register values and memory state
  - So if each process has its own register values…
  - And we only have one set of registers…
  - How can we run more than one process simultaneously?

Running multiple processes

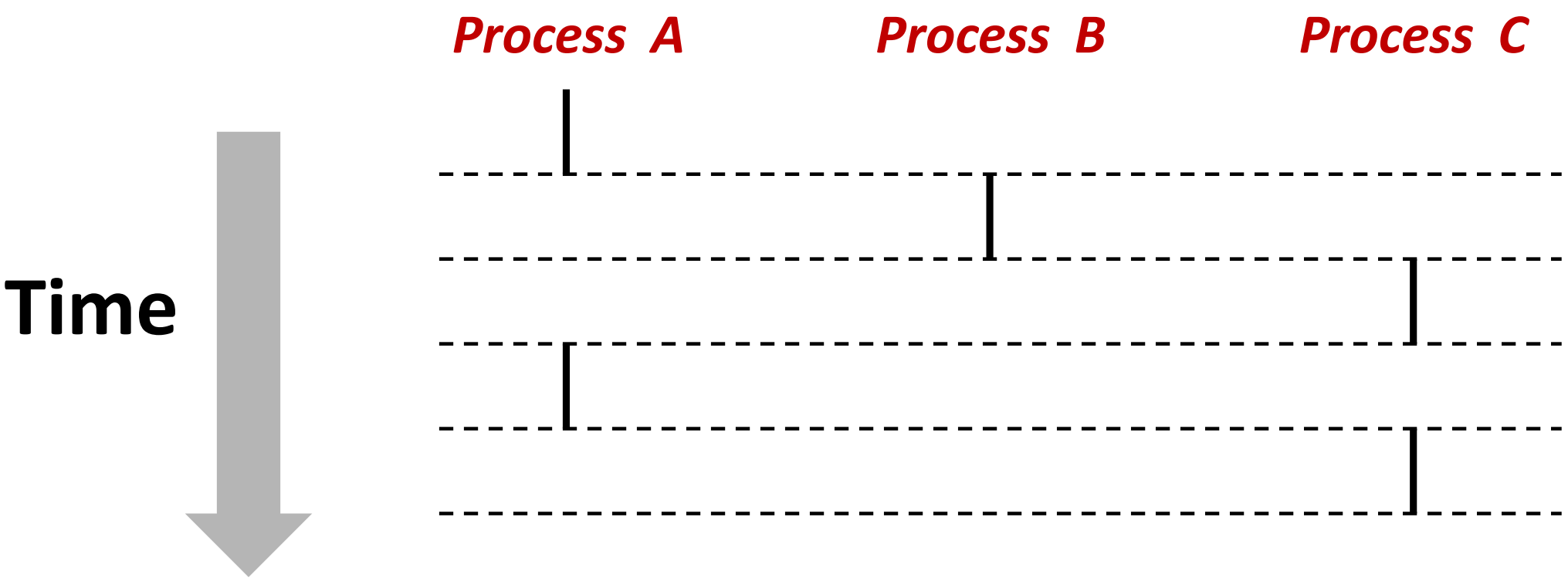
- But maybe I have four processors…
- Not much better!
- Can I only run four processes at once?
- How many processes am I running?



All an illusion

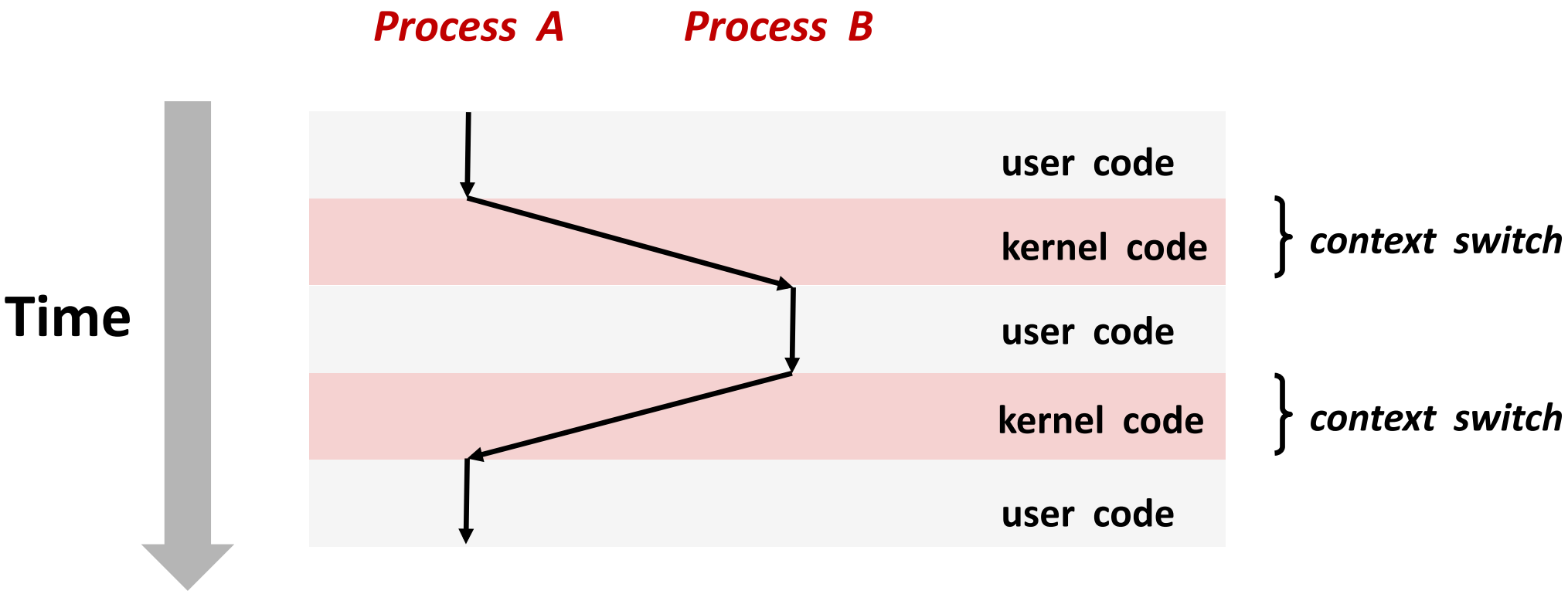
- The kernel creates the illusion that processes run simultaneously
- It does this by running each one for a very short time, quickly switching between each one
- This is called *time sharing*…

Time-sharing!



Context switching

- How the kernel implements time sharing
- This is what each dashed line in the previous slide represents
- During each context switch, the processor switches execution to the kernel…



Note: “user” is the opposite of “kernel”, so user code is any non-kernel code

Execution timeline

- Process A
- Kernel code to context switch
- Process B
- Kernel code to context switch
- Process C
- ...

What does a context switch involve?

- Switching registers
- Memory state
- And more!
- Everything necessary so that your program thinks has been continuously running

Where does a program switch?

- We’ve disassembled lots of programs
- Anyone see any instructions for context switching?

Where does a program switch?

- We’ve disassembled lots of programs
- Anyone see any instructions for context switching?
- Any `jmp context_switch` instructions?

Where does a program switch?

- We’ve disassembled lots of programs
- Anyone see any instructions for context switching?
- Any `jmp context_switch` instructions?
- No!
- The kernel *interrupts* the running program…

Normal control flow

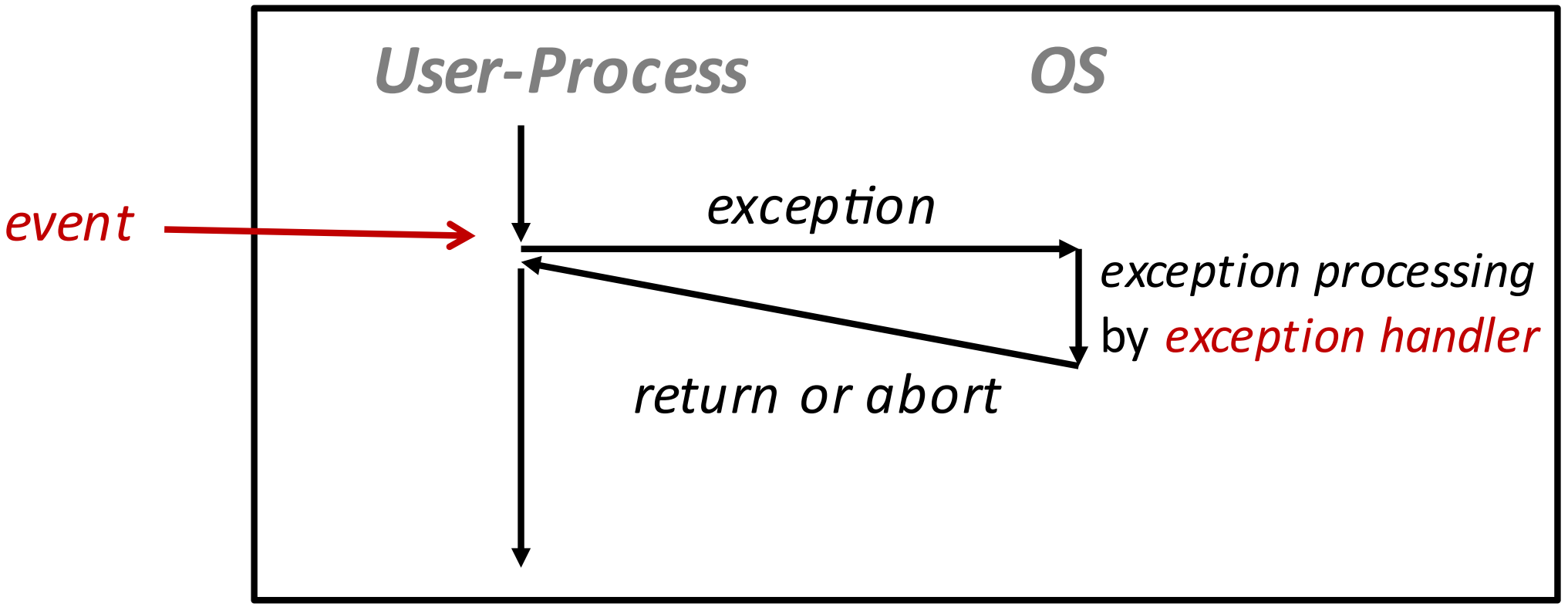
- Sequential control flow
  - … (`%rip` += instruction size)
- Jumps
  - `jmp foo` (`%rip = foo`)
- Function calls
  - `callq bar`, `ret` (`%rip = bar`, `%rip = RA`)
- Conditional jumps
  - `jz bletch` (`%rip = bletch` *if* zero flag set)

Exceptional control flow

- Machine control flow triggered by something other than the instruction semantics themselves
  - The trigger could be completely unrelated to the program!
- Context switching is one example of exceptional control flow

Exceptions

- The basic idea of exceptional control flow:
  - Some event occurs
  - That event triggers an *exception*
  - The kernel responds by interrupting the current process and jumping into kernel code
  - This kernel code is called an exception handler…
- There is an analogy to Python/Java exceptions here, but it’s best not to confuse them


User-Process
OS
event
exception
exception processing
by exception handler
return or abort

Exception handler

- What the exception handler does can vary depending on the exception and other variables
- In the previous slide, the running process was left running
- In a context switch, it would have been switched out with a different process

Exceptions

- There are two kinds of exceptions:
  - Asynchronous exceptions
  - Synchronous exceptions

Asynchronous exceptions

- A.k.a. hardware interrupts
- These can happen at any instant
- Caused by a hardware device
- Examples:
  - Typing a key
  - Moving a mouse
  - Timer hitting zero
  - Receiving a message over the network

Synchronous exceptions

- A.k.a. software interrupts
- Caused by an instruction
- Examples:
  - 5 / 0
  - *NULL

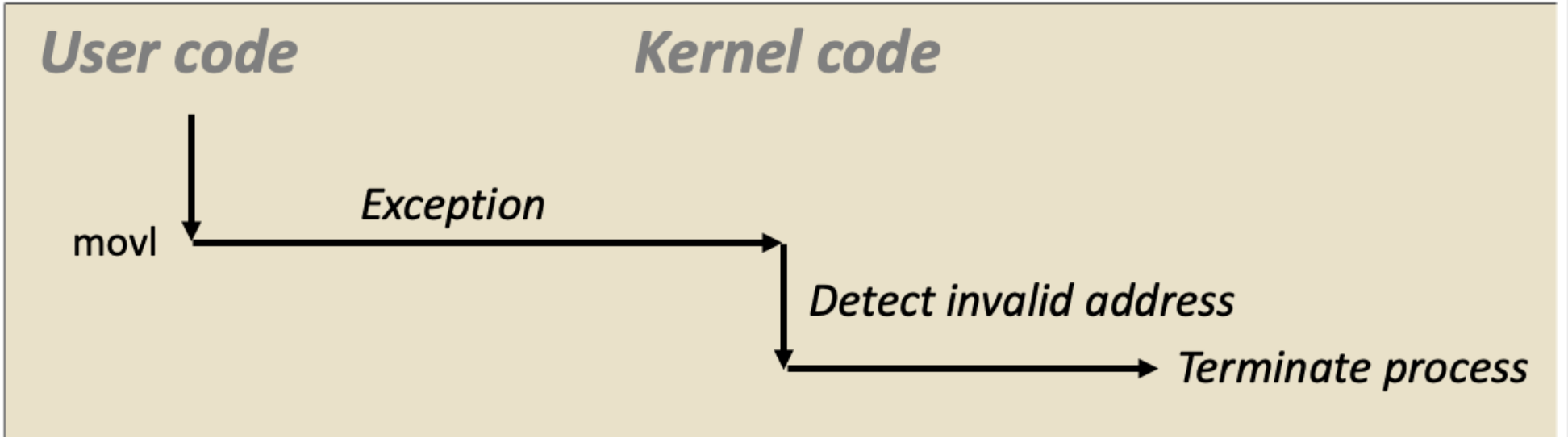
Synchronous exception examples

- divzero.c
- segfault.c
- Which instruction is triggering the exception?

```
int a[1000];

int main() {
    a[5000] = 7;
    ...
}
```

```
80483b7:    c7 05 60 e3 04 08 07    movl   $0x7,0x804e360
```



Context switching

- Synchronous or asynchronous?
  - Is it triggered by an instruction in your program?

Context switching

- Synchronous or asynchronous?
  - Is it triggered by an instruction in your program?
- No, it is triggered by a hardware timer
  - Time reaches zero → exception generated
  - Then the kernel executes its timer handler

*Un*exceptional exceptions?

- So far synchronous exceptions have seemed… unintentional
- Do we ever want to intentionally create a software interrupt?
- Yes, when we want to switch to kernel code
- When we want the kernel to do something for us

System call

- Call into the kernel
- Common syscalls:
  - open/close/read/write
  - exit
  - sockets
- Full Linux syscall table

Let’s look at some examples!

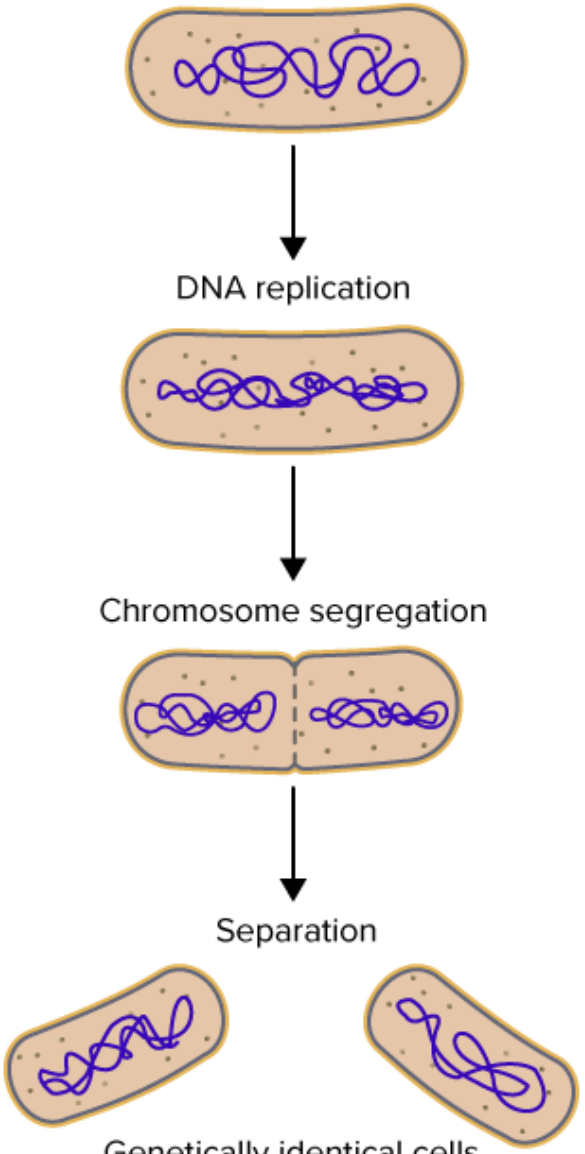
- hello.c
- hello2.c
- How do these print? How do they exit?

Exceptions not-so-exceptional

- They are part of the regular execution of a program
- Hundreds of exceptions occurring every second
- Allow us to write programs that think they are the only thing running on the entire machine
- Meanwhile the kernel is carefully handling the chaos of hundreds of exceptions a second

Where do processes come from?



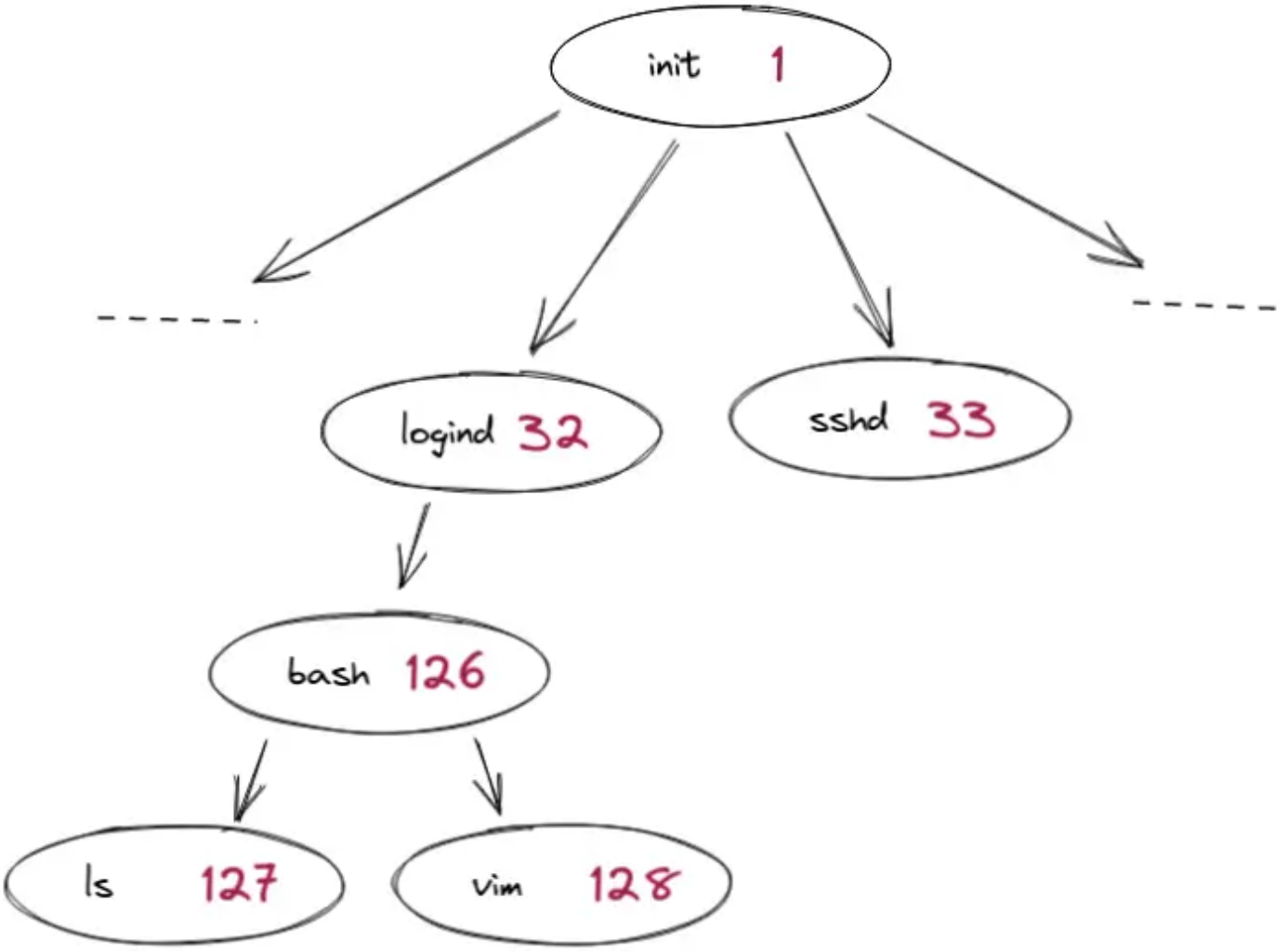

DNA replication
Chromosome segregation
Separation
Genetically identical cells

Binary fission

- Processes reproduce similarly
- Split into two processes with the same memory and code
- Before we go there, how do we conventionally create processes at a high level?

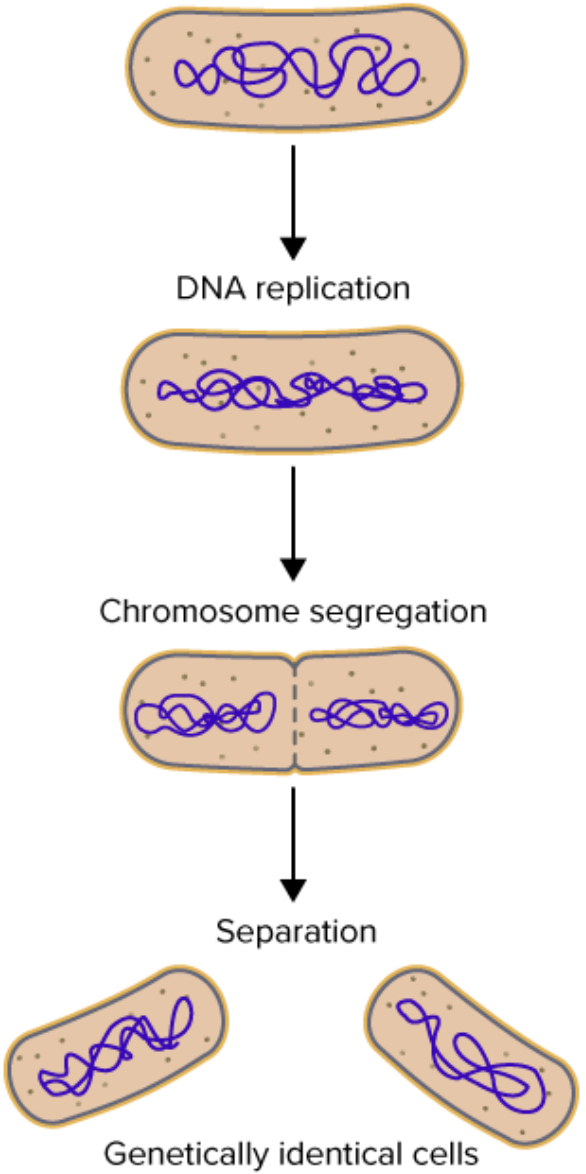
Creating processes

- Clicking on program in dock
  - Dock creates a process for clicked program
  - The dock is the parent, the new process the child
- Typing in a program in a terminal shell
  - Shell creates a process for typed command
  - The shell is the parent, the new process the child
- We can represent processes as a tree…


init 1
logind 32
sshd 33
bash 126
ls 127
vim 128

But wait…

- If processes reproduce by division, then why is one the parent?
- Well, we don't want the processes to be *exactly* the same…
- Otherwise, all we could create is more docks



fork system call

- This is one trippy system call!



fork system call

- Creates an exact copy of the current process
  - Same memory
  - Same register contents
- With one exception: `%rax`, the fork return value
  - Parent: `fork()` returns PID of newly created process
  - Child: `fork()` returns 0

forktest.c

- Let’s walk through this line-by-line…

Concurrent processes

- Remember these are running concurrently
- The order in which the parent vs. child processes are scheduled is not guaranteed
- Concurrent – execution order is not deterministic
- Simultaneous – execution at the same time (e.g., possible if you have multiple processors)

order.c

Why might fork fail?

```
if (f < 0) { // error, fork failed
    printf("fork failed\n");
}
```

- Ran out of memory
- Ran out of process ids (process ids on many systems are 16-bit ints!)

!?!?

```
while (1) {
    fork();
}
```

- What does this code do?
- What does the process tree look like?

Fork bomb!

```
while (1) {
    fork();
}
```

- What does this code do?
- What does the process tree look like?

TODO

- Assignment 4 due today