Lecture 02: Processes and Threads

CS343 – Operating Systems

Branden Ghena – Spring 2024

Some slides borrowed from:
Stephen Tarzia (Northwestern), Jaswinder Pal Singh (Princeton), Harsha Madhyastha (Michigan), and UC Berkeley CS61C and CS162

Northwestern

Administrivia

- Getting Started Lab
  - Due on Thursday (don’t be late on this)

  - Purpose is to make sure that you’ve got everything set up right
    - SSH login for EECS servers (if this fails, IT turnaround is ~24 hours)
    - Github account and Git SSH access
    - Ability to build the Nautilus Kernel

  - Let us know if you’re having problems with this
    - Should not take long to complete
    - 94/120 of you have at least made your own repo (most are done)

Today's Goals

- Understand the operating system's view of a process.
- Explore the context switches and exceptional control flow.
- Understand the basics of system calls and signals.
- What are threads and why are they useful?

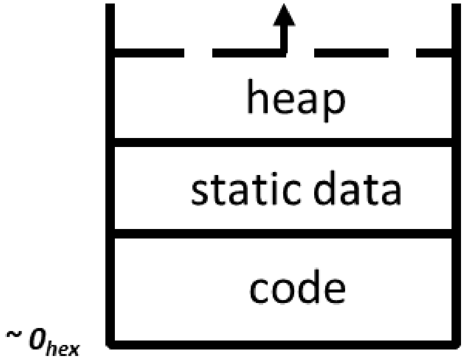
Outline

- **Processes**
- Context Switching
  - Running a Process
  - Exceptions
  - Running the Kernel
- System Calls
- Signals
- Threads

View of a process

- Process: program that is being executed
- Contains code, data, and a thread
  - Thread contains registers, instruction pointer, and stack

- Code and Data

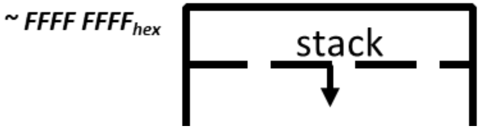


- Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Instruction Pointer
- Condition Codes

- Stack



POSIX processes also have file descriptors

- Integers specifying a file the process is interacting with
  - Process contains a table linking integers to files (and permissions)

- Default file descriptors
  - 0 - Standard input (stdin)
  - 1 - Standard output (stdout)
  - 2 - Standard error (stderr)

- Function calls to interact with files
  - `int      open  (const char *path, int oflag, ... );`
  - `ssize_t read  (int fildes, void *buf, size_t nbyte);`
  - `ssize_t write (int fildes, const void *buf, size_t nbyte);`

Example file descriptors

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID    USER   FD    TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd     DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd     DIR    8,5     4096       2 /
vim     6447 brghena  txt     REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem     REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem     REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem     REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem     REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem     REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem     REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem     REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem     REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem     REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem     REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem     REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem     REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem     REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem     REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem     REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem     REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem     REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem     REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem     REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem     REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem     REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem     REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena    0u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    1u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    2u    CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena    4u    REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Also the code files mapped to the address space

```
[brghena@ubuntu northwesternos.github.io] [master] $ lsof -p 6447
COMMAND  PID     USER   FD    TYPE DEVICE SIZE/OFF    NODE NAME
vim     6447 brghena  cwd     DIR    8,5     4096  524310 /home/brghena/Dropbox/class/cs343/northwesternos.github.io
vim     6447 brghena  rtd     DIR    8,5     4096       2 /
vim     6447 brghena  txt     REG    8,5  2906824 3418729 /usr/bin/vim.basic
vim     6447 brghena  mem     REG    8,5    51832 3415904 /usr/lib/x86_64-linux-gnu/libnss_files-2.31.so
vim     6447 brghena  mem     REG    8,5 14537584 3414469 /usr/lib/locale/locale-archive
vim     6447 brghena  mem     REG    8,5    47064 3415927 /usr/lib/x86_64-linux-gnu/libogg.so.0.8.4
vim     6447 brghena  mem     REG    8,5   182344 3416338 /usr/lib/x86_64-linux-gnu/libvorbis.so.0.4.8
vim     6447 brghena  mem     REG    8,5    14848 3416317 /usr/lib/x86_64-linux-gnu/libutil-2.31.so
vim     6447 brghena  mem     REG    8,5   108936 3416470 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
vim     6447 brghena  mem     REG    8,5   182560 3415356 /usr/lib/x86_64-linux-gnu/libexpat.so.1.6.11
vim     6447 brghena  mem     REG    8,5    39368 3415768 /usr/lib/x86_64-linux-gnu/libltdl.so.7.3.1
vim     6447 brghena  mem     REG    8,5   100520 3416225 /usr/lib/x86_64-linux-gnu/libtdb.so.1.4.2
vim     6447 brghena  mem     REG    8,5    38904 3416342 /usr/lib/x86_64-linux-gnu/libvorbisfile.so.3.3.7
vim     6447 brghena  mem     REG    8,5   584392 3415988 /usr/lib/x86_64-linux-gnu/libpcre2-8.so.0.9.0
vim     6447 brghena  mem     REG    8,5  2029224 3415140 /usr/lib/x86_64-linux-gnu/libc-2.31.so
vim     6447 brghena  mem     REG    8,5   157224 3416045 /usr/lib/x86_64-linux-gnu/libpthread-2.31.so
vim     6447 brghena  mem     REG    8,5  5416192 3416058 /usr/lib/x86_64-linux-gnu/libpython3.8.so.1.0
vim     6447 brghena  mem     REG    8,5    18816 3415275 /usr/lib/x86_64-linux-gnu/libdl-2.31.so
vim     6447 brghena  mem     REG    8,5    22456 3415526 /usr/lib/x86_64-linux-gnu/libgpm.so.2
vim     6447 brghena  mem     REG    8,5    39088 3415026 /usr/lib/x86_64-linux-gnu/libacl.so.1.1.2253
vim     6447 brghena  mem     REG    8,5    71680 3415157 /usr/lib/x86_64-linux-gnu/libcanberra.so.0.2.5
vim     6447 brghena  mem     REG    8,5   163200 3416142 /usr/lib/x86_64-linux-gnu/libselinux.so.1
vim     6447 brghena  mem     REG    8,5   192032 3416251 /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
vim     6447 brghena  mem     REG    8,5  1369352 3415780 /usr/lib/x86_64-linux-gnu/libm-2.31.so
vim     6447 brghena  mem     REG    8,5   191472 3414925 /usr/lib/x86_64-linux-gnu/ld-2.31.so
vim     6447 brghena   0u     CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   1u     CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   2u     CHR  136,3      0t0       6 /dev/pts/3
vim     6447 brghena   4u     REG    8,5    16384  524588 /home/brghena/Dropbox/class/cs343/northwesternos.github.io/.index.html.swp
```

Additional process contents

- Whatever else the OS thinks is useful
  - Process ID
  - Priority
  - Time Used
  - Process State

- Different OSes will attach different things to the "process abstraction"

Processes are an abstraction provided by the OS

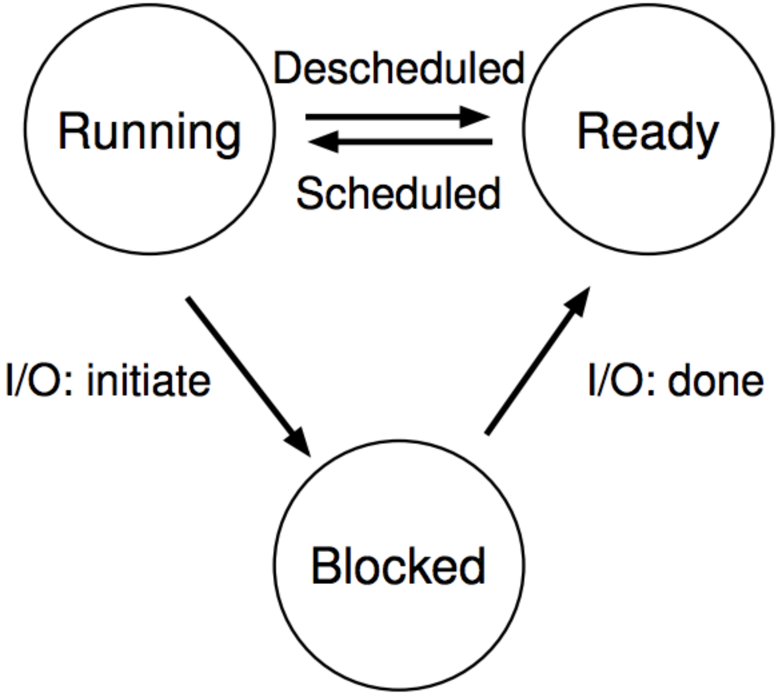
- The machine itself usually doesn’t support processes
  - Just has a processor and a set of registers
  - Memory is just arbitrary memory

- OS provides the abstraction
  - Multiple processes can run at the “same time”
  - Each has its own registers
  - Each has its own isolated memory

- Processes enable
  - Multiple functionalities on a computer
  - Multiprogramming of a system

Processes don't run all the time

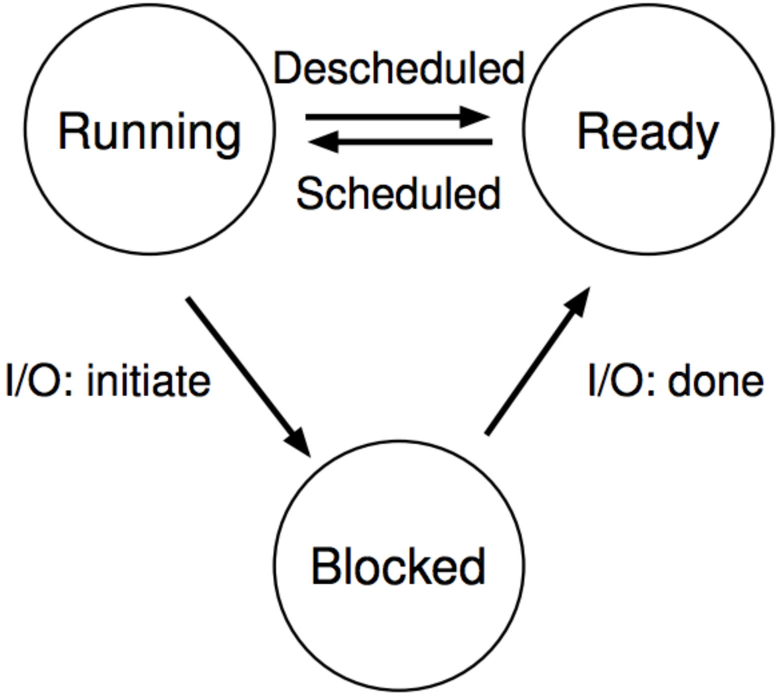
The three basic process states:



- OS ***schedules*** processes
  - Decides which of many competing processes to run.
- A ***blocked*** process is not ready to run.
- I/O means input/output – anything other than computing.
  - For example, reading/writing disk, sending network packet, waiting for keystroke, updating display.
  - While waiting for results, the process often cannot do anything, so it **blocks**, and the OS schedules a different process to run.

Multiprogramming processes

The three basic process states:



- When one process is Blocked, OS can schedule a different process that is Ready
- Even with a single processor, the OS can provide the illusion of many processes running simultaneously
- OS usually sets a maximum runtime before switching limit for processes (timeslice)

Key difference between kernel and processes: privilege

- Processes have limited access to the computer
  - Hardware supports different "modes" of execution (kernel and user)
  - Kernel mode has access to physical memory and special instructions

- They run when the OS lets them
- They have access to the memory the OS gives them

- They cannot access many things directly
  - Must ask the OS to do so for them

Break + Question

- Is it safe for two processes to have the same code section?

Break + Question

- Is it safe for two processes to have the same code section?

Usually yes!

- The OS can mark the code section as read-only

- Example: multiple instances of a shell share the same code

- Self-modifying code would be a problem…

Outline

- Processes
- **Context Switching**
  - **Running a Process**
  - Exceptions
  - Running the Kernel
- System Calls
- Signals
- Threads

Context: Tock Operating System

- Usually we'll use Nautilus as an example (last two labs are in it)
  - But Nautilus doesn't have a userspace!!
  - (also I honestly understand it less well than Tock)

- Tock OS
  - Embedded operating system
    - Targets resource-constrained embedded systems
  - Written in Rust
    - Reliability and Security are key goals
  - Multi-programming traditional OS environment
    - One core, with as many processes as you want

https://tockos.org/ - https://github.com/tock/tock

Switch to Process

- When a kernel decides to start running a process it does a context switch into the process
    - This includes starting a process for the first time
    - Or continuing running a process after it was stopped
        - Blocked for I/O or just timesliced off the processor

- High-level steps for switching into a process
    1. Scheduler decides which process should be running
    2. Save kernel register values to kernel stack
    3. Restore process register values (usually from a data structure)
    4. Switch to process mode instead of kernel mode
    5. Jump to next instruction in process

Tock ARM implementation: Switch to Process

- Tock ARM-v7m implementation
  1. Save kernel register values to kernel stack
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L237
  2. Restore process register values (usually from a data structure)
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L251
  3. Bonus: enter exception handler

  4. Switch to process mode instead of kernel mode
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L94
  5. Jump to next instruction in process
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L104

Tock RISC-V implementation: Switch to Process

- Tock rv32i implementation
  1. Save kernel register values to kernel stack
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/syscall.rs#L276
  2. Bonus: pause exceptions
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/syscall.rs#L300
  3. Restore process register values (usually from a data structure)
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/syscall.rs#L336
  4. Switch to process mode instead of kernel mode
     AND jump to next instruction in process
     AND enable exceptions again
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/syscall.rs#L370

Processes run until an exception occurs

- While the process is running, the OS is NOT running
  - In a single-core environment at least
- So, when does the OS kernel get to run again?
  - Whenever an exception occurs
- Hardware can be a source of this
  - Random device event occurring
  - Timer expiring (source of the process timeslice)
- Software can also cause this
  - System calls

Outline

- Processes
- **Context Switching**
  - Running a Process
  - **Exceptions**
  - Running the Kernel
- System Calls
- Signals
- Threads

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>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Altering control flow

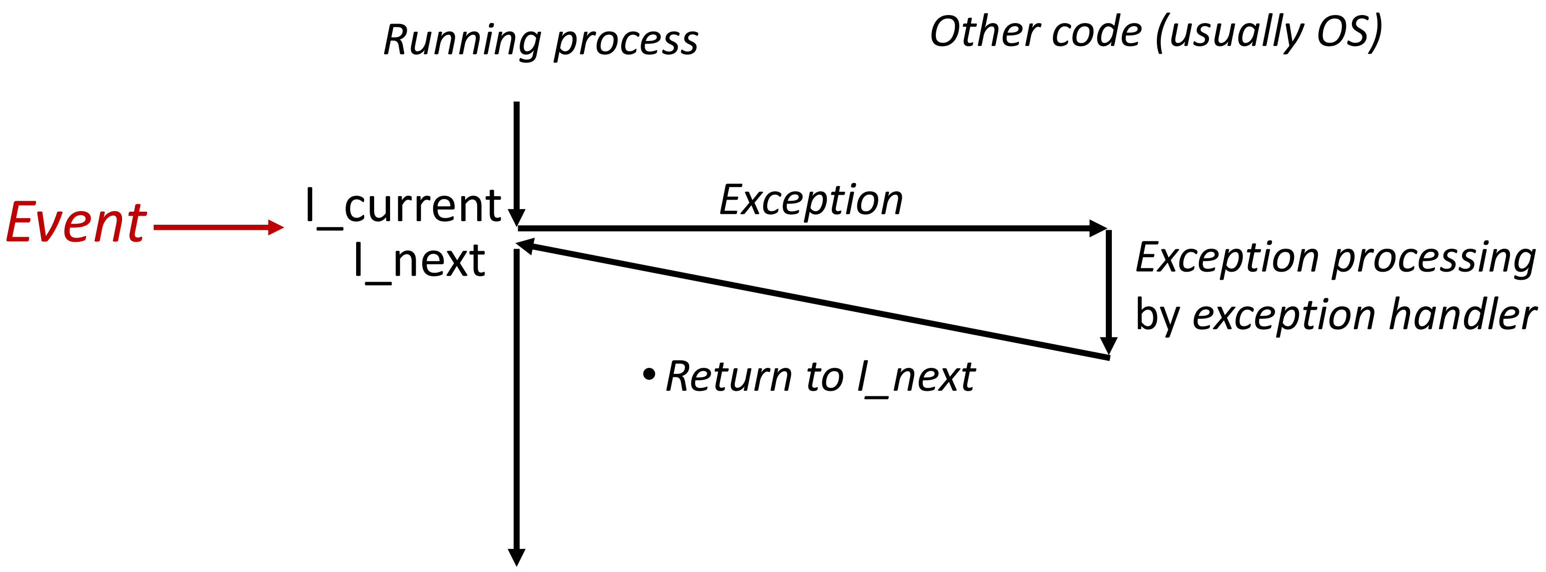
- Instructions that change control flow allow software to react changes in program state
  - Jumps/branches
  - Call/return

- Also need to react to changes in system state
  - Data arrives at network adapter
  - Instruction divides by zero
  - User hits Ctrl-C on the keyboard
  - System timer expires

- These mechanisms are known as “exceptional control flow”

Exceptional control flow

- Mechanisms that could cause exceptional control flow
  - Exceptions: events cause execution to jump to OS handler
  - Context switch: request or timeout causes execution to jump to OS
  - Signals: event plus OS causes execution to jump to process handler



Exceptions

- Hardware detects an event that OS software needs to resolve immediately

- Could be an error
  - Invalid memory access
  - Invalid instruction

- Could just be something the OS should handle (known as interrupts)
  - Page fault
  - USB device detected

- OS has a table of “exception handlers”, which are functions that handle each exception class (also known as interrupt handlers)
  - Hardware jumps execution to the proper handler

Tock exception vector

- Array of functions for each exception type
  - In Tock, most are “unhandled” which just crashes the system
  - Some are device interrupts (which don’t matter for today’s lecture

- Interrupt “vector”
  - https://github.com/tock/tock/blob/master/chips/nrf52/src/crt1.rs#L31

- SVC is a “service call” instruction
  - It’s used by processes to request an action from the OS
  - SVC_Handler is a function to handle those requests
    - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L76

Outline

- Processes
- **Context Switching**
  - Running a Process
  - Exceptions
  - **Running the Kernel**
- System Calls
- Signals
- Threads

How does a process ask the OS to do something?

- Certain things can only be accessed from kernel mode
  - All of memory, I/O devices, etc.
  - Kernel: the portion of the OS that is running and in memory

- **Bad Idea** to allow processes to just enter kernel mode
  - We do NOT trust processes
  - So there shouldn't be any instruction that switches to kernel *mode* unless that instruction also switches to kernel *code*

- Requirements
  1. Switch execution to the kernel
  2. Change into kernel mode
  3. Inform the kernel what you want it to do

Hardware can save us!

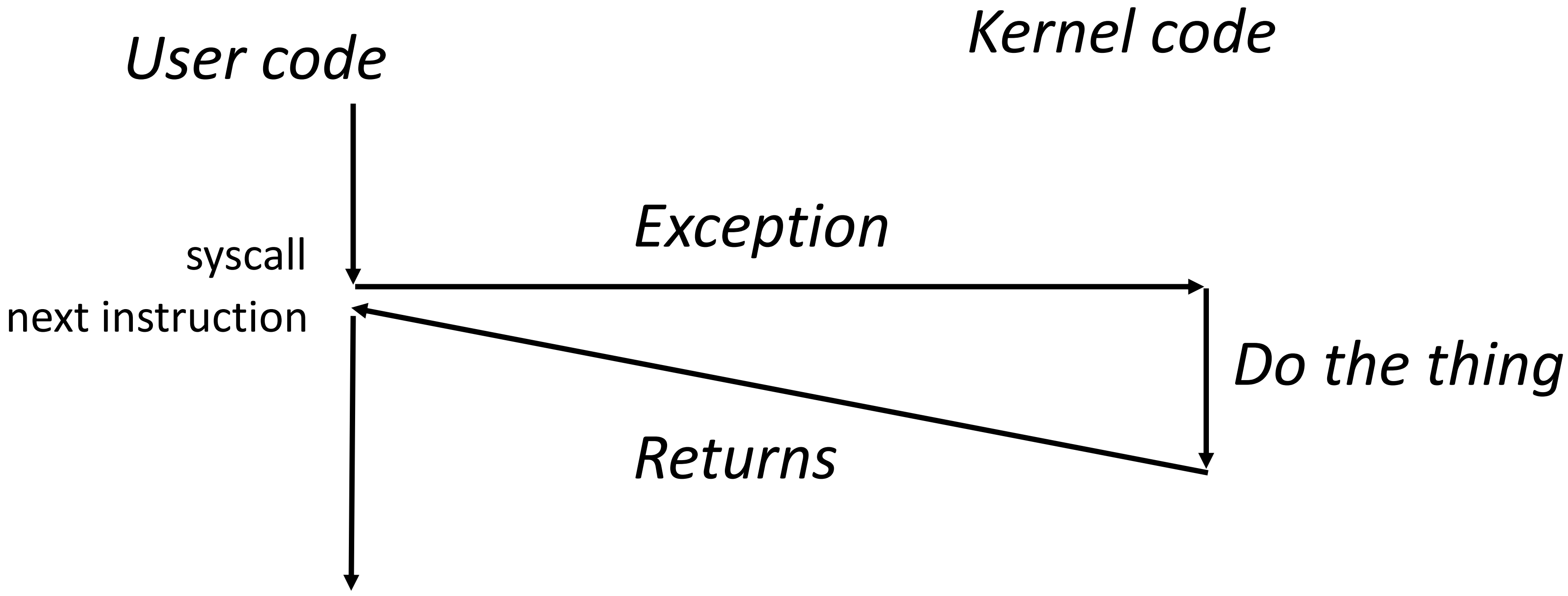
- Solution: trigger an exception to run an OS handler
  - Hardware instruction: trap

- When instruction runs:
  1. Mode is changed to kernel mode
     AND
  2. Instruction Pointer is moved to a known location in the kernel

- Same mechanism is used for other exceptions
  - Division by zero, invalid memory access
  - Also very similar to hardware interrupts

System call example

- System call: making a request of the OS from a process
  - Uses exceptional control flow to enter OS kernel
  - Returns back to process when complete
    - Instruction *after* the system call



Switch to Kernel

- Assume the process *wants* to switch to the kernel, what occurs?

- High-level steps for switching to the kernel
  1. Process executes a system call
  2. Processor enters kernel mode and runs an exception handler
  3. Save process registers
  4. Restore kernel registers
  5. Figure out why a context switch occurred

Tock ARM Implementation: Switch to Kernel

- Tock ARM-v7m implementation
  1. Process executes a system call
     - https://github.com/tock/libtock-c/blob/master/libtock/tock.c#L257
  2. Processor enters kernel mode and runs an exception handler
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L76
  3. Bonus: return to switch_to_user implementation
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L135
  4. Save process registers
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L264
  5. Restore kernel registers
     - https://github.com/tock/tock/blob/master/arch/cortex-v7m/src/lib.rs#L272
  6. Figure out why a context switch occurred
     - https://github.com/tock/tock/blob/master/arch/cortex-m/src/syscall.rs#L262

Tock RISC-V Implementation: Switch to Kernel

- Tock rv32i implementation
  1. Process executes a system call
     - https://github.com/tock/libtock-c/blob/master/libtock/tock.c#L485
  2. Processor enters kernel mode and runs an exception handler
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/lib.rs#L335
  3. Save process registers
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/lib.rs#L358
  4. Bonus: return to switch_to_process implementation
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/lib.rs#L437
  5. Restore kernel registers
     - https://github.com/tock/tock/blob/master/arch/rv32i/src/syscall.rs#L382
  6. Figure out why a context switch occurred
     - https://github.com/tock/tock/blob/master/arch/cortex-m/src/syscall.rs#L262

Handling the system call

- Now the kernel is running again AND it knows why it was running
- If a fault occurred crash the process or something
- If a syscall occurred, read why from the process's registers
  - Which are saved in some data structure somewhere
  - Then figure out *what* to do about that request

Switching between process and kernel is a context switch

- Context switch: switching from process to kernel or kernel to process
  - Vague term. Sometimes refer to there-and-back as a context switch

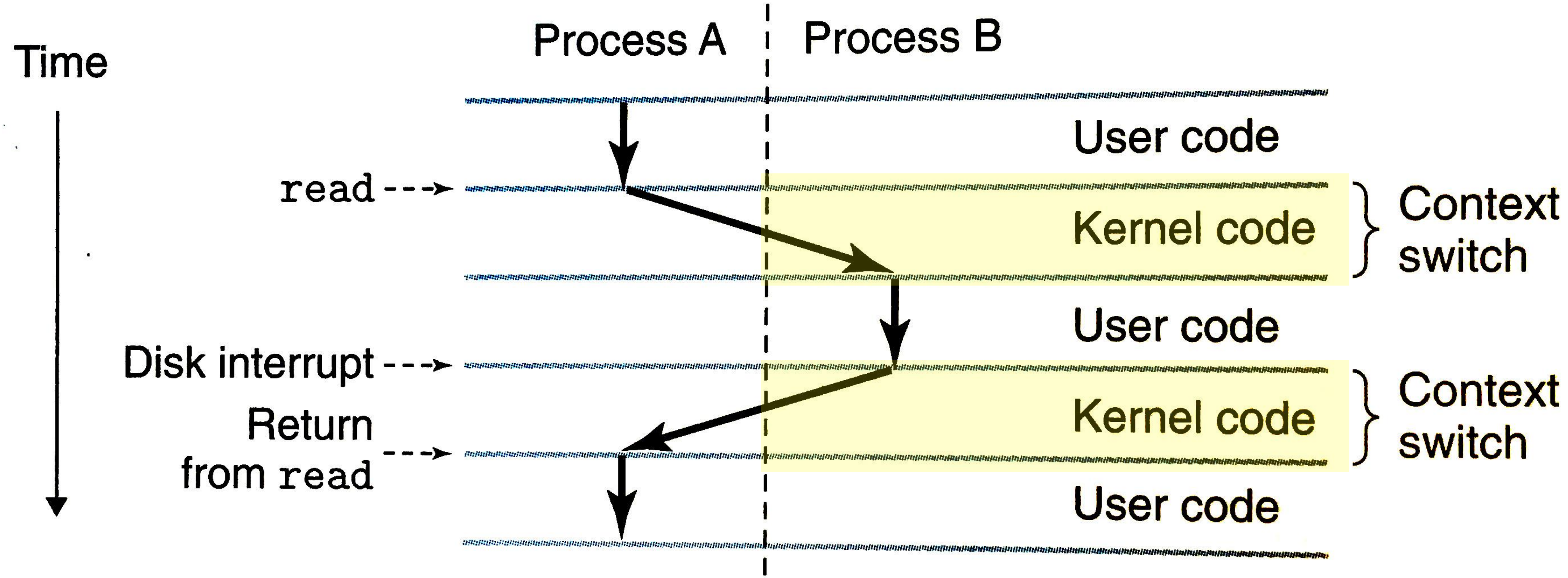


Diagram from Bryant & O'Hallaron book

Break + Question

- Context switches are expensive
  - Lots of context switches lead to poor performance
  - Why?

Break + Question

- Context switches are expensive
  - Lots of context switches lead to poor performance
  - Why?

  - Lots of memory manipulation
    - Saving and restoring registers
    - All cached data is almost certainly invalid

  - Triggering an exception isn't exactly quick
    - Processor needs to stop everything and jump somewhere

  - OS needs to figure out what's going on and respond to it
    - The figuring it out part can be a lot of code

Outline

- Processes
- Context Switching
  - Running a Process
  - Exceptions
  - Running the Kernel
- **System Calls**
- Signals
- Threads

Things a program cannot do itself

- Print “hello world”
  - *because the display is a shared resource.*
- Download a web page
  - *because the network card is a shared resource.*
- Save or read a file
  - *because the filesystem is a shared resource, and the OS wants to check file permissions first.*
- Launch another program
  - *because processes are managed by the OS*
- Send data to another program
  - *because each program runs in isolation, one at a time*

Linux system calls

- Example system calls
  - https://man7.org/linux/man-pages/man2/syscalls.2.html

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

Many other system calls

- POSIX contains many others, for example `time()`
  - And especially lots of old ones

- Windows or other operating systems will have entirely different system calls
  - Same basic idea for how they function though

Example system call usage

- Create new processes with system calls

- From process view:
  - Just look like regular C functions
  - Take arguments, return values

- Underneath:
  - Function uses special assembly instruction to trigger exception

Process system calls

`pid_t fork(void);`

- Create a new process that is a copy of the current one
- Returns either PID of child process (parent) or 0 (child)

`void _exit(int status);`

- Exit the current process (exit(), the library call cleans things up first)

`pid_t waitpid(pid_t pid, int *status, int options);`

- Suspends the current process until a child (*pid*) terminates

`int execve(const char *filename, char *const argv[], char *const envp[]);`

- Execute a new program, replacing the existing one

Creating a new process

```c
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

  printf("Both!\n");
  return 0;
}
```

Creating a new process

```
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    printf("Child!\n");
  } else {
    printf("Parent!\n");
  }

  printf("Both!\n");
  return 0;
}
```

← Existential crisis

Executing a new program

```c
#include <stdio.h>
#include <unistd.h>

int main(){
  if(fork() == 0) {
    execve("/bin/python3", ...);
  } else {
    printf("Parent!\n");
  }

  printf("Only parent!\n");
  return 0;
}
```

Creating your own shell

https://danishpraka.sh/2018/01/15/write-a-shell.html

```
void execute(char** args) {
  if (strcmp(args[0], "exit") == 0) {
    exit(); // exit the shell when requested
  }

  pid_t cpid = fork();
  if (cpid == 0) {
    if (execvp(args[0], args) < 0) { // child, execute new process
      printf("command not found: %s\n", args[0]);
    }
  } else {
    waitpid(cpid, & status, WUNTRACED); // parent, wait for process to be complete
}}

int main(){
  char** args;
  while(1){
    printf("> ");
    args = parse_incoming_text(); // complicated in C unfortunately
    execute(args);
}}
```

Creating your own shell

https://danishpraka.sh/2018/01/15/write-a-shell.html

```
void execute(char** args) {
  if (strcmp(args[0], "exit") == 0) {
    exit(); // exit the shell when requested
  }

  pid_t cpid = fork();
  if (cpid == 0) {
    if (execvp(args[0], args) < 0) { // child, execute new process
      printf("command not found: %s\n", args[0]);
    }
  } else {
    waitpid(cpid, & status, WUNTRACED); // parent, wait for process to be complete
}}

int main(){
  char** args;
  while(1){
    printf("> ");
    args = parse_incoming_text(); // complicated in C unfortunately
    execute(args);
}}
```

Break + Question

- What does the following code do?

```
#include <stdio.h>
#include <sys/types.h>

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

Break + Question

- What does the following code do?

```
#include <stdio.h>
#include <sys/types.h>

int main() {
  while(1){
    fork();
  }
  return 0;
}
```

- Creates a new process
  - Then each process creates a new process
  - Then each of those creates a new process…

- Known as a Fork bomb!
  - Machine eventually runs out of memory and processing power and will stop working

- Defense: limit number of processes per user

Fork bombs in various languages

- Python fork bomb

```
import os
while 1:
   os.fork()
```

- Rust fork bomb

```
#[allow(unconditional_recursion)]
fn main() {
    std::thread::spawn(main);
    main();
}
```

- Bash fork bomb

```
:(){ :|:& };:
```

- Bash with spacing and a clearer function name

```
fork() {
    fork | fork &
}
fork
```

Outline

- Processes
- Context Switching
  - Running a Process
  - Exceptions
  - Running the Kernel
- System Calls
- **Signals**
- Threads

Alerting processes of events

- How do we let a process know there was an event?
  - Errors
  - Termination
  - User commands (like CTRL-C or CTRL-\)

- Events could happen whenever
  - Need to interrupt process control flow and run an event handler
  - Linux mechanism to do so is called “signals”

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called “Signals”

```
 1) SIGHUP     2) SIGINT   3) SIGQUIT   4) SIGILL  5) SIGTRAP
 6) SIGABRT    7) SIGBUS   8) SIGFPE    9) SIGKILL 10) SIGUSR1
11) SIGSEGV   12) SIGUSR2 13) SIGPIPE  14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT  19) SIGSTOP 20) SIGTSTP
21) SIGTTIN   22) SIGTTOU 23) SIGURG   24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO   30) SIGPWR
31) SIGSYS           ...
```

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called "Signals"

```
 1) SIGHUP      2) SIGINT    3) SIGQUIT    4) SIGILL    5) SIGTRAP
 6) SIGABRT     7) SIGBUS    8) SIGFPE     9) SIGKILL  10) SIGUSR1
11) SIGSEGV    12) SIGUSR2  13) SIGPIPE   14) SIGALRM  15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD  18) SIGCONT   19) SIGSTOP  20) SIGTSTP
21) SIGTTIN    22) SIGTTOU  23) SIGURG    24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO    30) SIGPWR
31) SIGSYS          ...
```

Process Errors

Signals are asynchronous messages to processes

- Sometimes the OS wants to send something like an interrupt to a process
  - Your child process completed
  - You tried to use an illegal instruction
  - You accessed invalid memory
  - You are terminating now

- In POSIX systems, this idea is called "Signals"

```
 1) SIGHUP       2) SIGINT    3) SIGQUIT    4) SIGILL    5) SIGTRAP
 6) SIGABRT      7) SIGBUS    8) SIGFPE     9) SIGKILL  10) SIGUSR1
11) SIGSEGV     12) SIGUSR2  13) SIGPIPE   14) SIGALRM  15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD  18) SIGCONT   19) SIGSTOP  20) SIGTSTP
21) SIGTTIN     22) SIGTTOU  23) SIGURG    24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF  28) SIGWINCH  29) SIGIO    30) SIGPWR
31) SIGSYS            ...
```

Process Termination

Sending signals

- OS sends signals when it needs to

- Processes can ask the OS send signals with a system call
  - `int kill(pid_t pid, int sig);`

- Users send signals through OS from command line or keyboard
  - Shell command: kill -9 *pid* (SIGKILL)
  - CTRL-C (SIGINT)

Handling signals

- Programs can register a function to handle individual signals
  - `signal(int sig, sighandler_t handler);`

- OS keeps track of signal handlers for each signal
  - Calls that function when a signal occurs

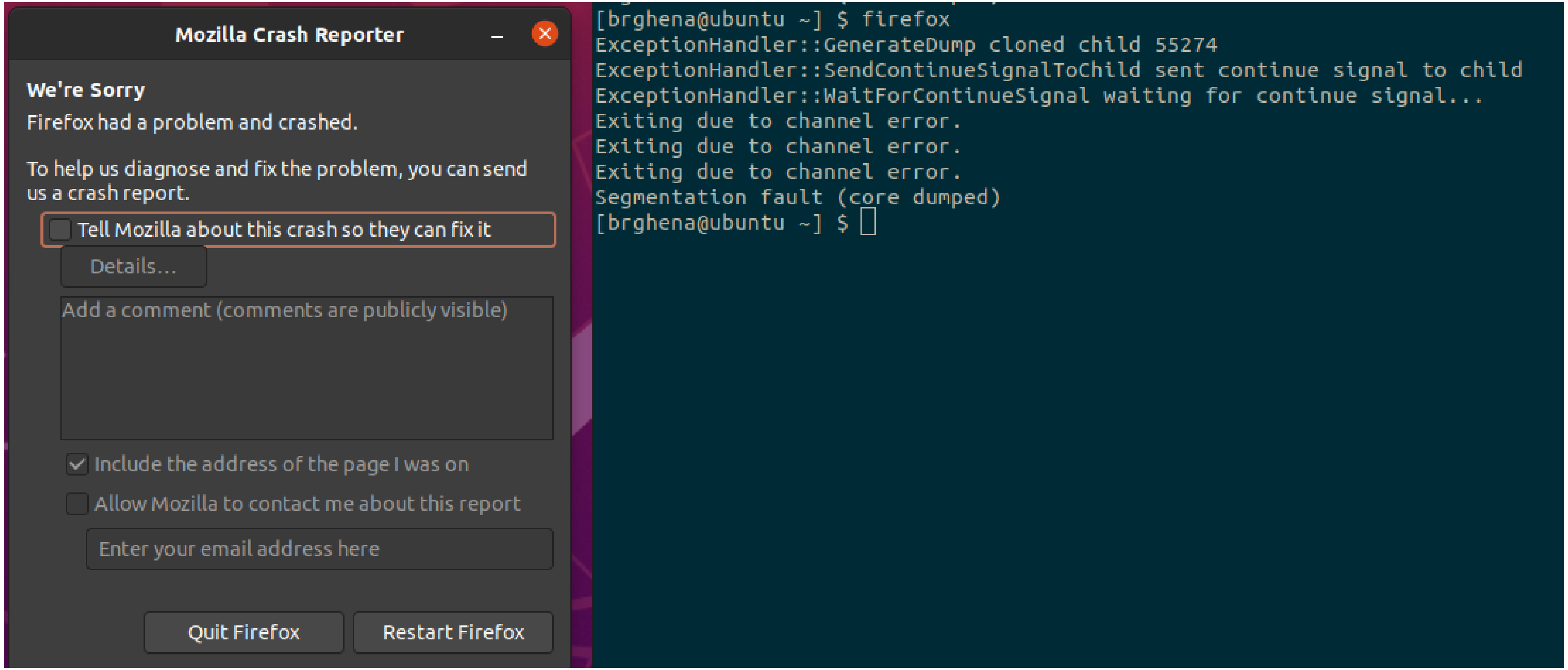
- What is the process supposed to do about it?
  - Do some *quick* processing to handle it
    - That needs to be "reentrant" safe

  - Reset the process and try again

  - Quit the process (default handler)

Examples: sending a signal

> kill -11 *pid* (11 is SIGSEGV – a.k.a segfault)



Outline

- Processes
- Context Switching
  - Running a Process
  - Exceptions
  - Running the Kernel
- System Calls
- Signals
- **Threads**

Software Tasks: Threads

Unit of execution *within* a process

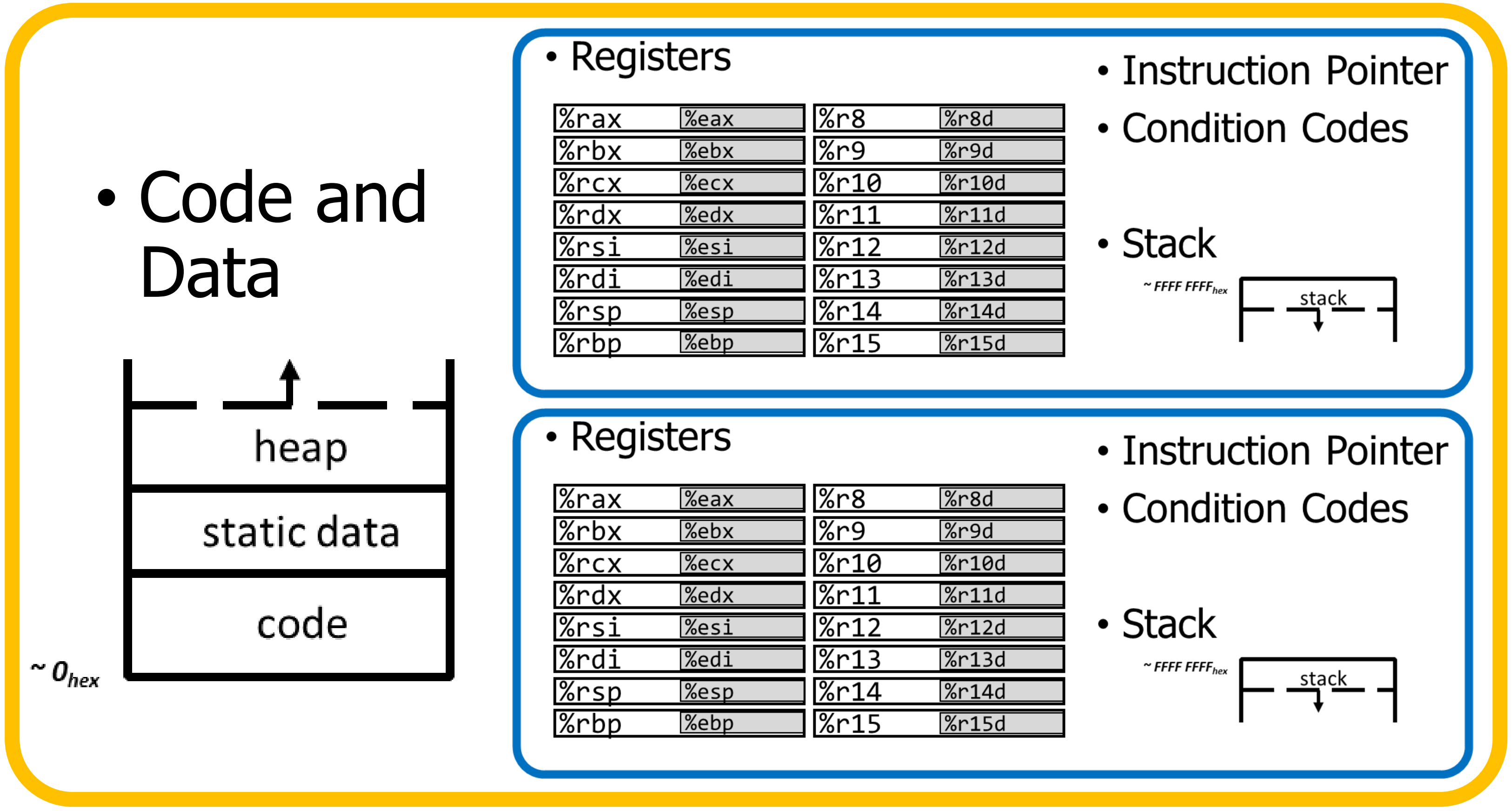
Processes discussed so far have a single thread

- They “have a single thread of execution”
- They “are single-threaded”

But a single process could have multiple threads

Alternate view of a process

- A process could have multiple threads
  - Each with its own registers and stack



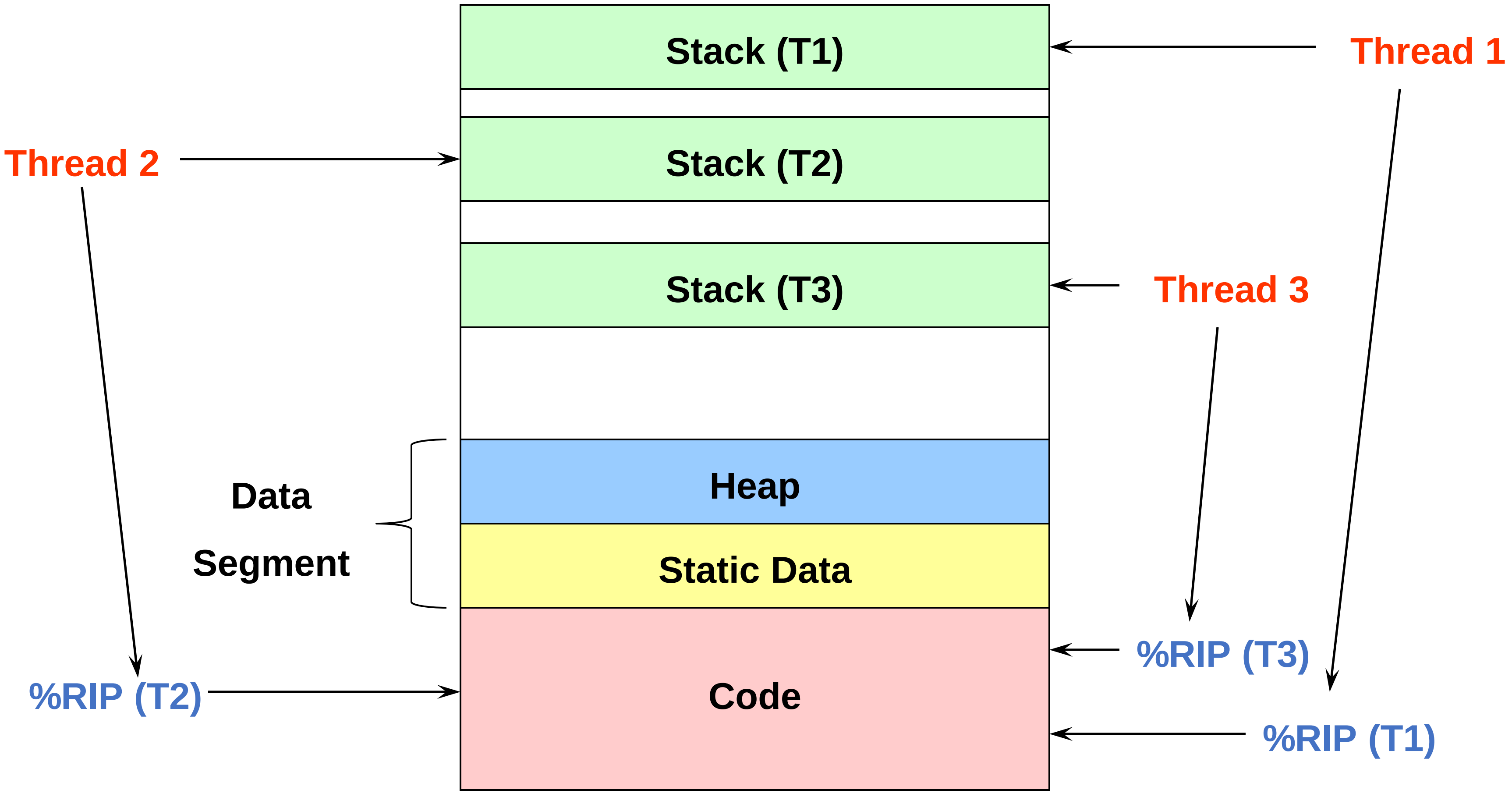
Threads have separate:

- Instruction Pointer
- Registers
- Stack Memory
- Condition Codes

Threads share:

- Code
- Global variables

Process address space with threads



Thread use case: web browser

Let's say you're implementing a web browser:

You want a tab for each web page you open:

- The same code loads each website (shared code section)
- The same global settings are shared by each tab (shared data section)
- Each tab does have separate state (separate stack and registers)

Disclaimer: Actually, modern browsers use separate processes for each tab for a variety of reasons including performance and security. But they used to use threads.

Thread use case: user interfaces

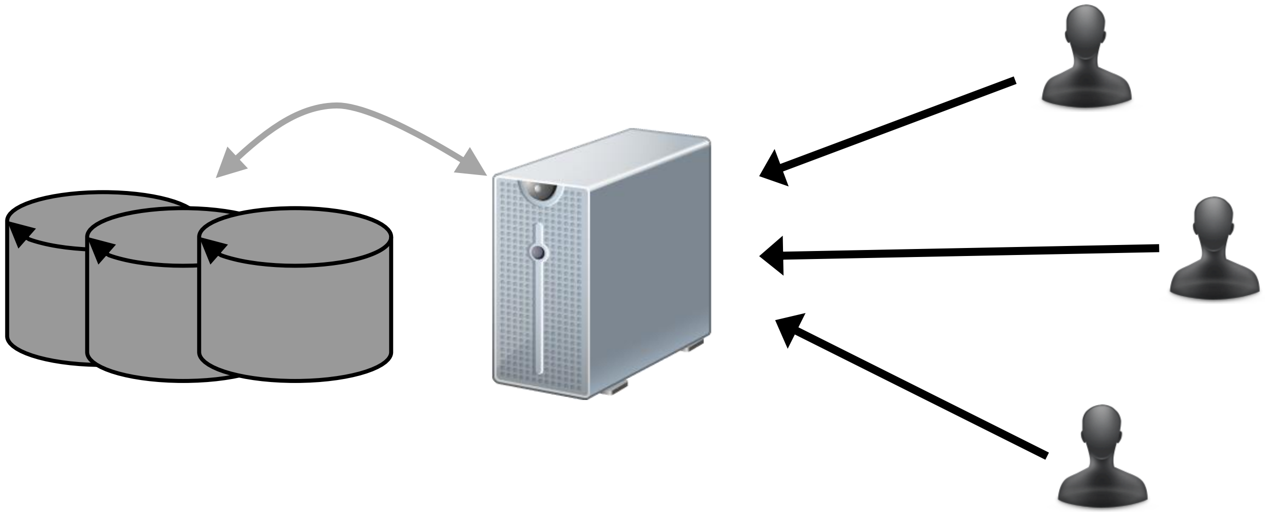
- Even if there is only a single processor core, threads are useful

- Single-threaded User Interface
  - While processing actions, the UI is frozen

```
main() {
   while(true) {
         check_for_UI_interactions();
         process_UI_actions(); // UI freezes while processing
   }
}
```

Thread use case: web server

- Example: Web server
  - Receives multiple simultaneous requests
  - Reads web pages from disk to satisfy each request



Web server option 1: handle one request at a time

Request 1 arrives
Server reads in request 1
Server starts disk I/O for request 1
Request 2 arrives
Disk I/O for request 1 finishes
Server responds to request 1
Server reads in request 2

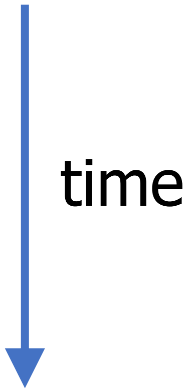
time

- Easy to program, but slow
  - Can’t overlap disk requests with computation
  - Can’t overlap either with network sends and receives

Web server option 1: event-driven model

- Issue I/Os, but don't wait for them to complete

  Request 1 arrives
  Server reads in request 1
  Server starts disk I/O for request 1
  Request 2 arrives
  Server reads in request 2
  Server starts disk I/O for request 2
  Disk I/O for request 1 completes
  Server responds to request 1



- Fast, but hard to program
  - Must remember which requests are in flight and which I/O goes where
  - Lots of extra state

Web server option 3: multi-threaded web server

- One thread per request. Thread handles only that request.

Main Thread	**Thread 1**	**Thread 2**
Request 1 arrives		
Create thread →		
	Read in request 1	
	Start disk I/O	
Request 2 arrives		
Create thread →		
		Read in request 2
		Start disk I/O
	Disk I/O finishes	
	Respond to request 1	
	Exit	

time ↓

- Easy to program (maybe), and fast!
  - State is stored in the stacks of each thread and the thread scheduler
  - Simple to program if they are independent…

More Practical Motivation

Back to Jeff Dean's "Numbers Everyone Should Know"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

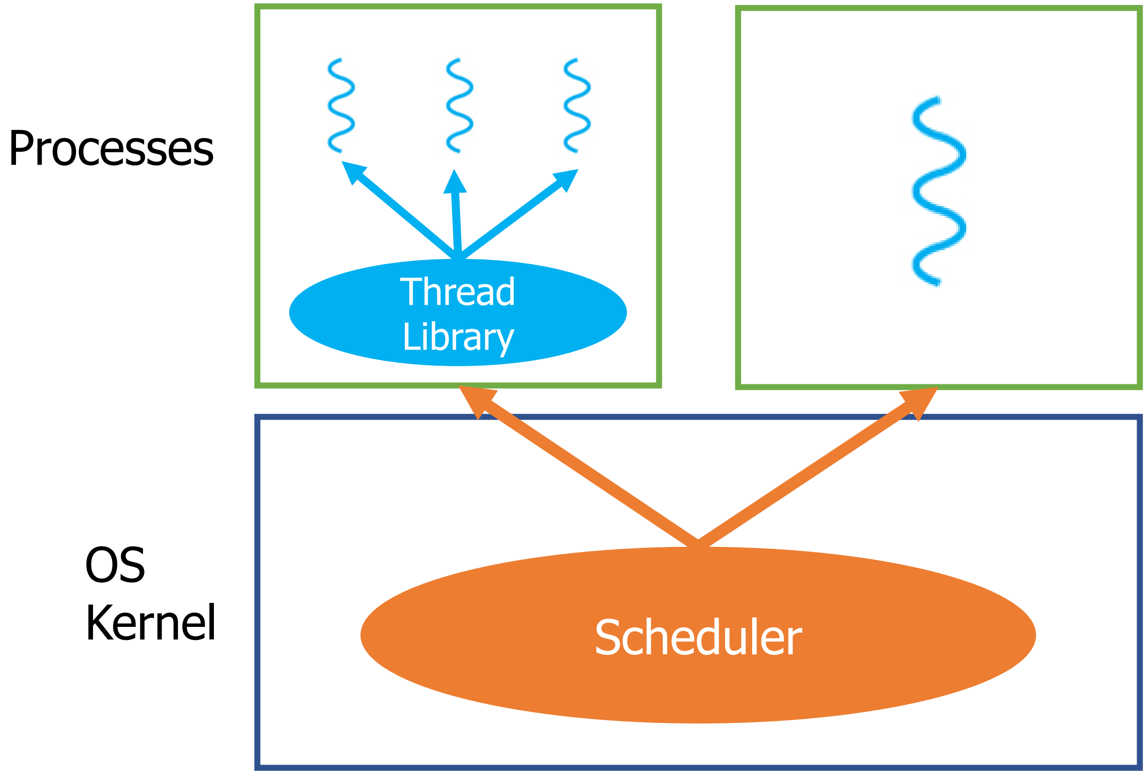
Handle I/O in separate thread, avoid blocking other progress

Models for thread libraries: **User Threads**

- Thread scheduling is implemented within the process
  - OS only knows about the process, not the threads

- Upsides
  - Works on any hardware or OS
  - Performance is better when creating and switching

- Downsides
  - A system call in any thread **blocks all threads**



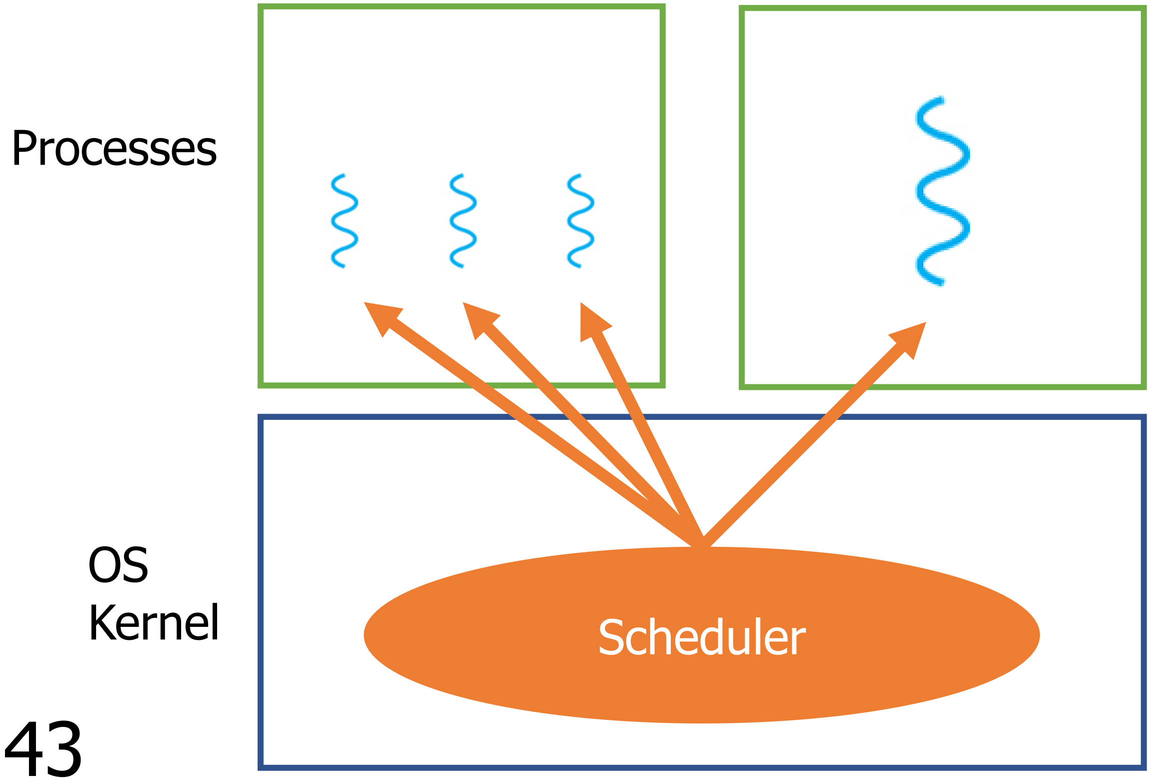
Models for thread libraries: **Kernel Threads**

- Thread scheduling is implemented by the operating system
  - OS manages the threads within each process

- Upsides
  - Other threads can continue while one blocks on I/O
  - No additional scheduler

- Downsides
  - Higher overhead

- This is what we'll focus on in CS343



Threads versus Processes

Threads

- **`pthread_create()`**
  - Creates a thread
  - ***Shares*** all memory with all threads of the process.
  - Scheduled independently of parent
- **`pthread_join()`**
  - Waits for a particular thread to finish
- Can communicate by reading/writing (shared) global variables.

Processes

- **`fork()`**
  - Creates a single-threaded process
  - ***Copies*** all memory from parent
    - Can be quick using copy-on-write
  - Scheduled independently of parent
- **`waitpid()`**
  - Waits for a particular child process to finish
- Can communicate by setting up shared memory, pipes, reading/writing files, or using sockets (network).

POSIX Threads Library: pthreads

- https://man7.org/linux/man-pages/man7/pthreads.7.html

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void*), void *arg);
```

  - thread is created executing *start_routine* with *arg* as its sole argument.
  - return is implicit call to pthread_exit

```
void pthread_exit(void *value_ptr);
```

  - terminates the thread and makes *value_ptr* available to any successful join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

  - suspends execution of the calling thread until the target *thread* terminates.
  - On return with a non-NULL *value_ptr* the value passed to *pthread_exit()* by the terminating thread is made available in the location referenced by *value_ptr*.

Pthread system call example

- What happens when pthread_create() is called in a process?

```
Library:
   int pthread_create(…) {
      Do some work like a normal function
      Put syscall number into register
      Put args into registers
      Special trap instruction
```

← `clone` (56) syscall on Linux

```
Kernel:
   Get args from regs
   Do the work to spawn the new thread
   Store return value in %eax
```

```
      Get return values from regs
      Do some more work like a normal function
   };
```

Outline

- Processes
- Context Switching
  - Running a Process
  - Exceptions
  - Running the Kernel
- System Calls
- Signals
- Threads

- Bonus: Thread example

Threads Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);          /* last thing in the main thread  */
}
```

Threads Example

- Reads N from process arguments
- Creates N threads
- Each one prints a number, then increments it, then exits
- Main process waits for all of the threads to finish

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);            /* last thing in the main thread  */
}
```

Threads Example

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Check your understanding

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program?
2. Does the main thread join with the threads in the same order that they were created?
3. Do the threads exit in the same order they were created?
4. If we run the program again, would the result change?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread  */
}
```

Check your understanding

```
(base) CullerMac19:code04 culler$ ./pthread 4
Main stack: 7ffee2c6b6b8, common: 10cf95048 (162)
Thread #1 stack: 70000d83bef8 common: 10cf95048 (162)
Thread #3 stack: 70000d941ef8 common: 10cf95048 (164)
Thread #2 stack: 70000d8beef8 common: 10cf95048 (165)
Thread #0 stack: 70000d7b8ef8 common: 10cf95048 (163)
```

1. How many threads are in this program? **Five**
2. Does the main thread join with the threads in the same order that they were created? **Yes**
3. Do the threads exit in the same order they were created? **Maybe??**
4. If we run the program again, would the result change? **Possibly!**

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
  long tid = (long)threadid;
  printf("Thread #%lx stack: %lx common: %lx (%d)\n", tid,
         (unsigned long) &tid, (unsigned long) &common, common++);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  long t;
  int nthreads = 2;
  if (argc > 1) {
    nthreads = atoi(argv[1]);
  }
  pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
  printf("Main stack: %lx, common: %lx (%d)\n",
         (unsigned long) &t,(unsigned long) &common, common);
  for(t=0; t<nthreads; t++){
    int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t=0; t<nthreads; t++){
    pthread_join(threads[t], NULL);
  }
  pthread_exit(NULL);           /* last thing in the main thread */
}
```