

Lecture 17

Processes

CS213 – Intro to Computer Systems
Branden Gena – Winter 2024

Slides adapted from:

St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Today's Goals

- Explore various mechanisms by which OS and processes interact
 - System calls and signals
- Discuss operations on files as example system calls
- Introduce the idea of "scheduling" processes

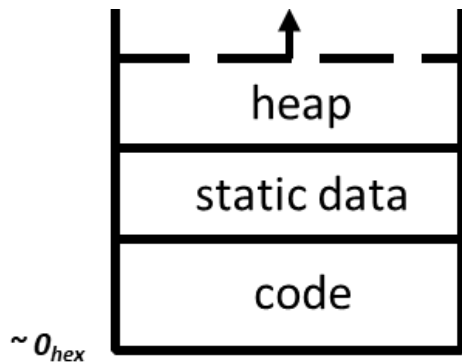
Outline

- **Process Control Flow**
- System Calls
- File I/O
 - Standard I/O
- Signals
- Scheduling Processes

Reminder: 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

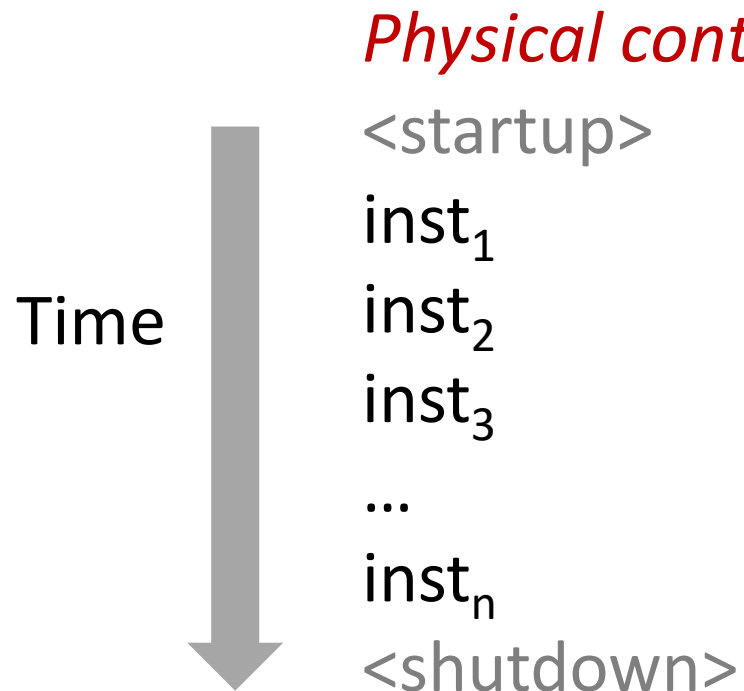


Questions remaining about processes

- Interaction mechanisms with OS
 - How do processes make requests of the OS?
 - How does the OS inform processes of various events?
- Both answered by the same basic mechanism:
exceptional control flow

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*)

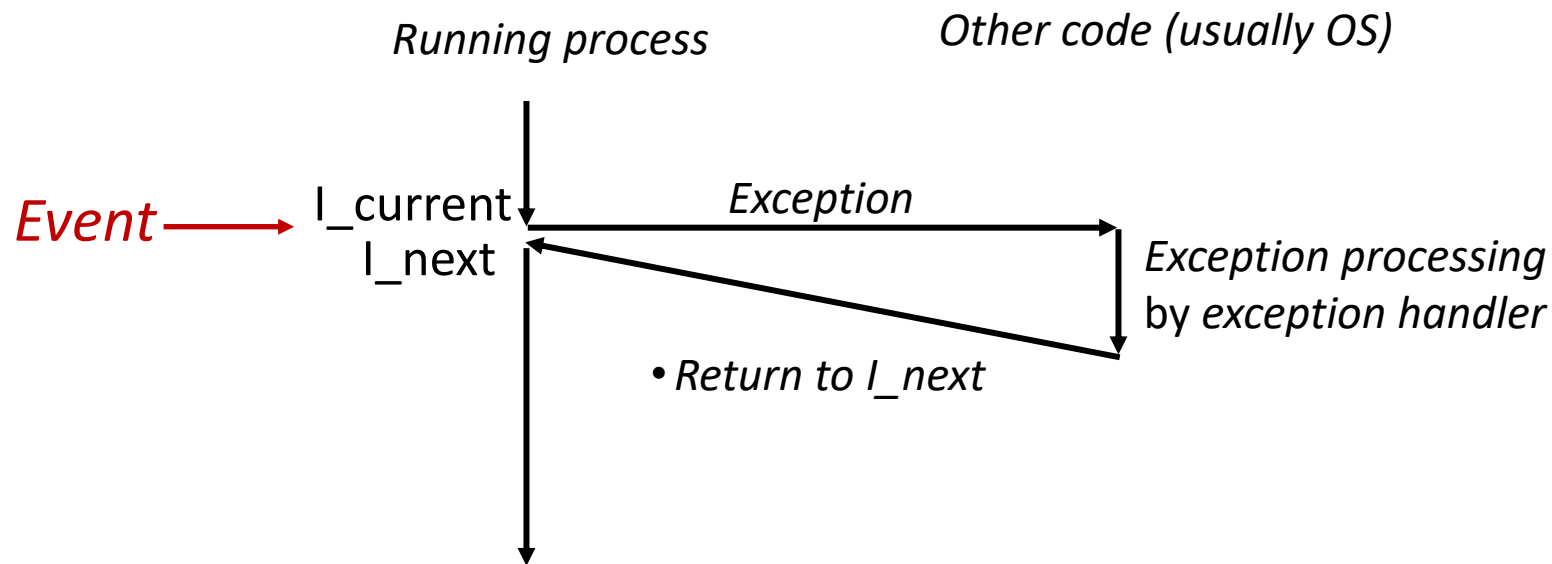


Altering control flow

- 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
 - 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

Outline

- Process Control Flow
- **System Calls**
- File I/O
 - Standard I/O
- Signals
- Scheduling Processes

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*

How does a process ask the OS to do something?

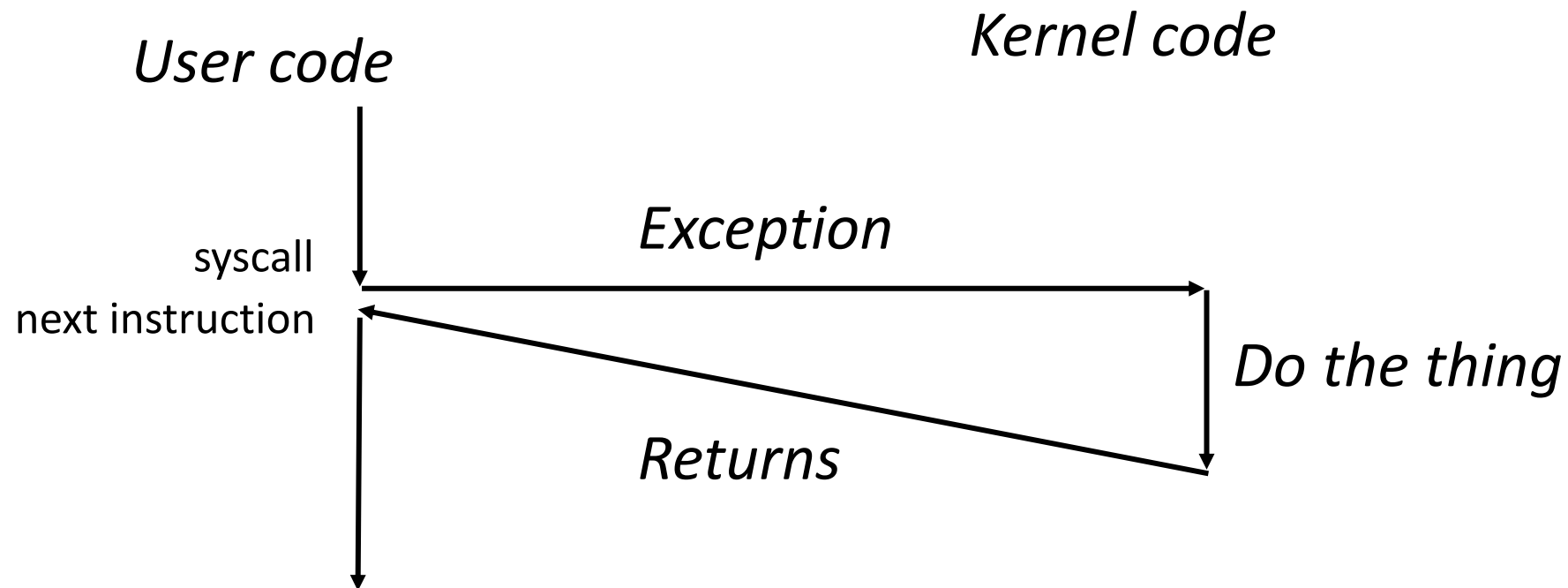
- Certain things can only be accessed from kernel mode
 - All of memory, I/O devices, etc.
 - Kernel: the portion of the OS that is running and in memory
- **Bad Idea** to allow processes to just enter kernel mode
 - We do NOT trust processes
 - So there shouldn't be any instruction that switches to 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



System call steps (simplification)

1. Process loads parameters into registers (just like a function call)
2. Process executes trap instruction (`int`, `syscall`, `svc`, etc.)
3. Hardware moves `%rip` to "handler" and switches to kernel mode
4. OS checks what the process wants to do from registers
5. OS decides *whether* the process is allowed to do so

Returning from a system call (simplification)

- After OS finishes whatever operation it was asked to do
 - And when the process is scheduled to run again
 - 1. OS places return result in a register (just like a function call)
 - 2. OS changes mode to user mode (and sets virtual memory stuff)
 - 3. OS sets `%rip` to instruction after the system call
-
- Process continues and can use results of system call

Linux system calls

- Example system calls

- <https://man7.org/linux/man-pages/man2/syscalls.2.html>

<i>Number</i>	<i>Name</i>	<i>Description</i>
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

Example using system calls

- Let's 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

The C function for syscalls just performs the correct assembly

- Example system call: fork

<i>Number</i>	<i>Name</i>	<i>Description</i>
57	fork	Create process

- C code:

```
fork();
```

- x86-64 assembly implementation:

```
fork:
```

```
    movq $57, %rdi
```

```
    syscall
```

Process management 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
- Replaces code and data, clears registers, sets `%rip` to start again

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;
}
```

Creating a new process

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

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

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

Executing a new program

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

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

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

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

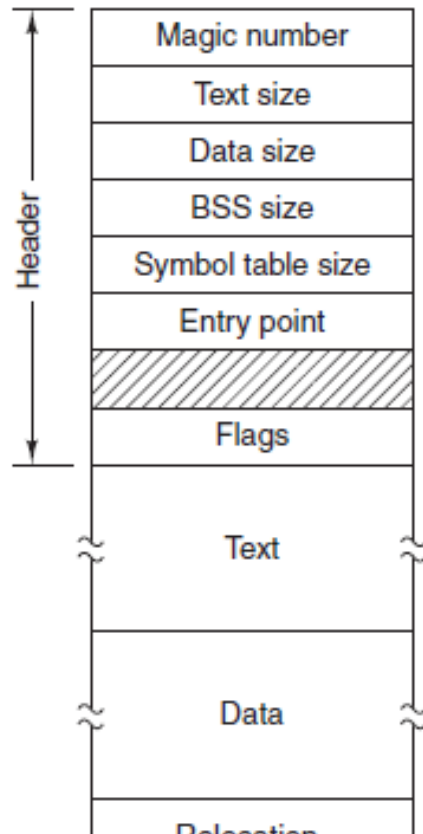
- Process Control Flow
- System Calls
- **File I/O**
 - Standard I/O
- Signals
- Scheduling Processes

Files

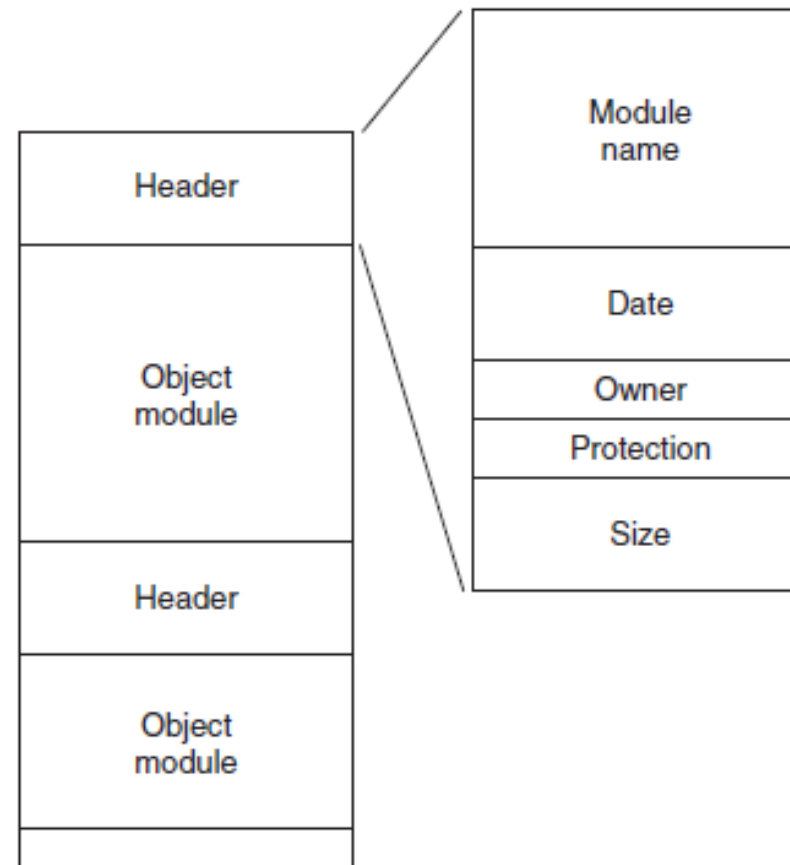
- Collections of data
 - Usually in permanent storage on your computer
- Types of files
 - Regular files
 - Arbitrary data
 - Think of as a big array of bytes
 - Directories
 - Collections of regular files
 - Special files
 - Links, pipes, devices (see CS343)

Sidebar: what about types of regular files?

- Text files versus Executables versus Tar files
 - All just differing patterns of bytes!
 - It really is just all data. The meaning is in how you interpret it.



Executable File



Archive (tar)

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena 4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena 4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena 235 Nov 18 21:42 arguments.c
```

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena 4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena 4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena 235 Nov 18 21:42 arguments.c
```

- Permissions for the owner and name of the owner
 - Read, Write, eXecute
 - Cannot execute `arguments.c`
 - For directories: Read contents, Write new contents, Traverse directory

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena 4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena 4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena 235 Nov 18 21:42 arguments.c
```

- Permissions for the group and name of the group
 - Example: I could make a CS213 group, add you all to it, and only give that group access to some folder or file

File permissions

- Files have owners and permissions associated with them

```
[brghena@ubuntu code] $ ls -la
total 32
drwxrwxr-x 2 brghena brghena 4096 Nov 18 22:07 .
drwxr-xr-x 4 brghena brghena 4096 Nov 18 18:38 ..
-rwxrwxr-x 1 brghena brghena 16704 Nov 18 21:42 arguments
-rw-rw-r-- 1 brghena brghena 235 Nov 18 21:42 arguments.c
```

- Permissions for everyone else on the computer
 - Not the owner and not in the group
 - For my personal machine, not particularly relevant
 - For Moore, probably don't want to let others read your files...

How does a process access files?

- This is an example use case for system calls
- Files are a shared and managed resource on the computer
 - Need to follow permissions settings
 - Handle if multiple processes try to edit a file simultaneously
- Also need to simplify what the interface looks like
 - Files are actually structures in filesystem likely on disk
 - But the process shouldn't need to care about the details of that

How do we interact with files?

- Analogy: think of a file as a book
 - Big array of characters (bytes)
1. Open the book, starting at the first page
 2. Read from the book
 3. Write to the book
 4. Change pages (without reading everything in between)
 5. Close the book when finished

System calls for interacting with files

1. Open the book, starting at the first page
 - `open()`
2. Read from the book
 - `read()`
3. Write to the book
 - `write()`
4. Change pages (without reading everything in between)
 - `lseek()`
5. Close the book when finished
 - `close()`

Higher-level methods of file interaction

- Here, we're talking about system calls to the OS
- C standard library also defines file interactions
 - `fopen`, `fread`, `fwrite`, `fseek`, `fclose`
 - All are wrappers on top of the actual syscalls
 - Buffers your interactions to make them more efficient
 - Reads/Writes large chunks of data at a time
 - Might collect multiple `fwrite`'s before doing a single real write
 - `fflush()` guarantees that the buffer is written *now*

Opening files

- `int open(const char *pathname, int flags);`
- `pathname` is the string path for the file
 - `"/home/brghena/class/cs213/s21/code/arguments.c"`
 - `"/arguments.c"`
 - `"arguments.c"`
- `flags` include access permission requests
 - Read only, Write only, Read and Write (`O_RDONLY`, `O_WRONLY`, `O_RDWR`)
 - Also can choose to append to a file (`O_APPEND`)
 - Or to create the file if it does not exist (`O_CREAT`)

Open returns a “file descriptor”

- `int open(const char *pathname, int flags);`
- OS keeps track of opened files for each process
 - File descriptor is **just a number** referring to the opened file
 - Non-negative number. Always the lowest unused, starting at zero
 - A “handle” to the file
- File descriptor is used in other calls to reference the file
 - That way the OS doesn't have to look up pathname every time
- Negative number instead specifies an error (for all of these calls)

Sidebar: how do you figure out how these calls work?

- Manual pages
- Online: <https://man7.org/linux/man-pages/man2/close.2.html>

close(2) — Linux manual page

[NAME](#) | [SYNOPSIS](#) | [DESCRIPTION](#) | [RETURN VALUE](#) | [ERRORS](#) | [CONFORMING TO](#) | [NOTES](#) | [SEE ALSO](#) | [COLOPHON](#)

CLOSE(2)

Linux Programmer's Manual

CLOSE(2)

NAME [top](#)

close - close a file descriptor

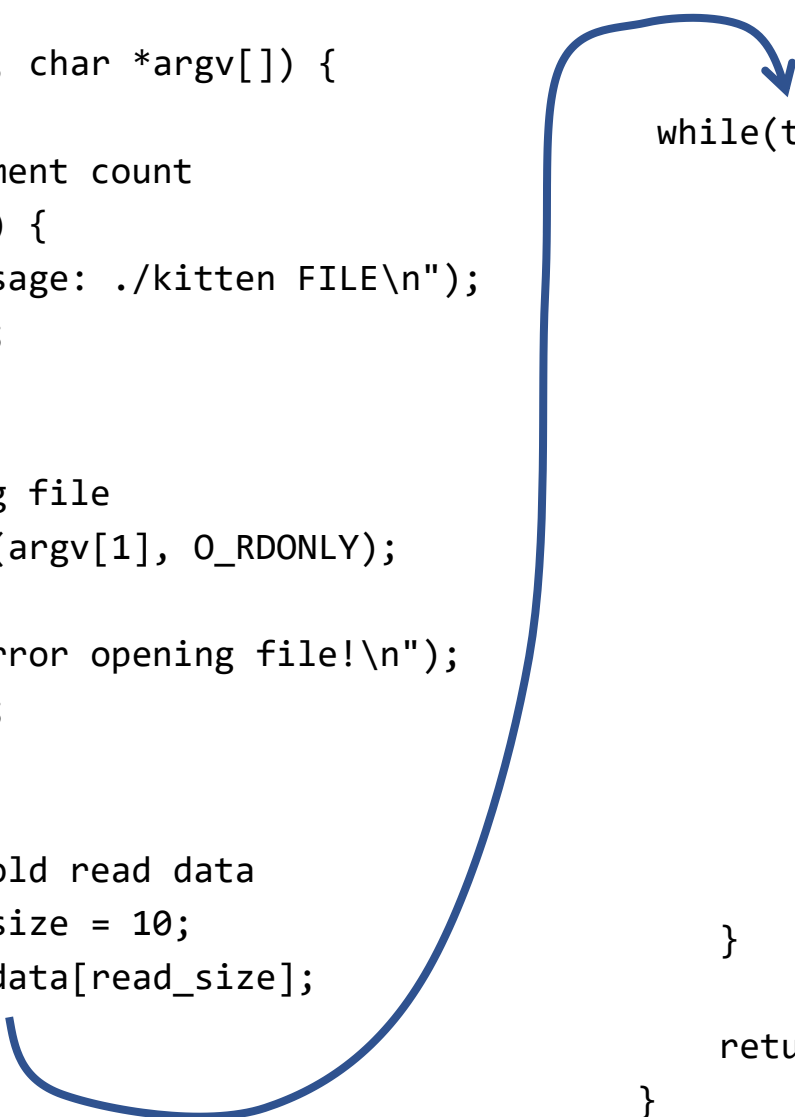
SYNOPSIS [top](#)

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

```
int close(int fd);
```


Example: read a file and print it to terminal

```
int main(int argc, char *argv[]) {  
  
    // check argument count  
    if (argc != 2) {  
        printf("Usage: ./kitten FILE\n");  
        return -1;  
    }  
  
    // try opening file  
    int fd = open(argv[1], O_RDONLY);  
    if (fd < 0) {  
        printf("Error opening file!\n");  
        return -1;  
    }  
  
    // array to hold read data  
    uint8_t read_size = 10;  
    uint8_t read_data[read_size];  
  
    while(true) {  
        // read from file  
        ssize_t read_length = read(fd, read_data, read_size);  
        if (read_length < 0) {  
            printf("Error reading file!\n");  
            return -1;  
        }  
        if (read_length == 0) {  
            break;  
        }  
  
        // print out data  
        for (int i=0; i<read_length; i++) {  
            printf("%c", read_data[i]);  
        }  
    }  
  
    return 0;  
}
```

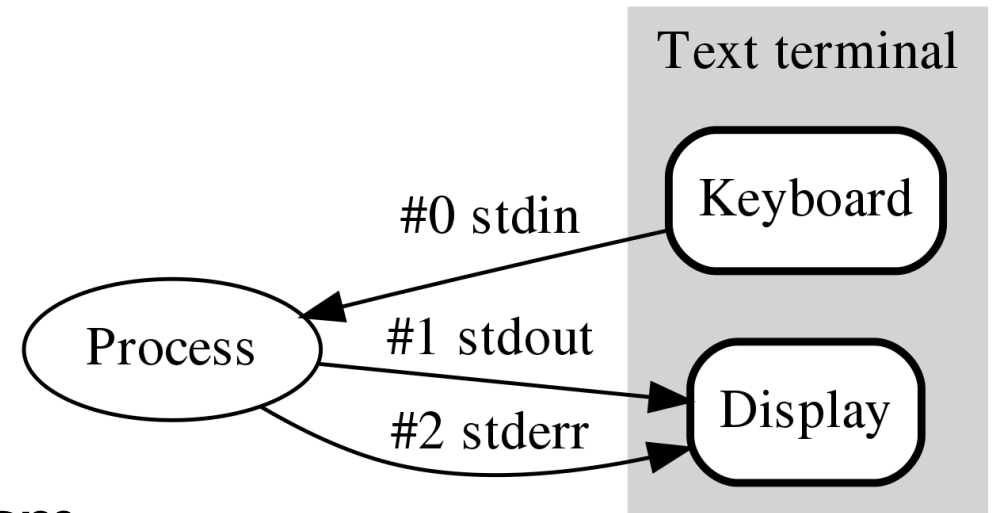


Outline

- Process Control Flow
- System Calls
- **File I/O**
 - **Standard I/O**
- Signals
- Scheduling Processes

How do programs talk to users?

- We often gloss over this in CS211
 - `printf()`
 - `gets()`
- Work through the same file mechanism
 - Three special files created for each program
 - `stdin` – standard input (file descriptor 0)
 - `stdout` – standard output (file descriptor 1)
 - `stderr` – standard error (file descriptor 2)
- `printf(...)` -> `fprintf(1, ...)` -> handle arguments then `write(1, ...)`



Standard I/O is a process thing, not a C thing

- You can access them in Python, for instance
 - <https://docs.python.org/3/library/sys.html#sys.stdin>

```
sys.stdin  
sys.stdout  
sys.stderr
```

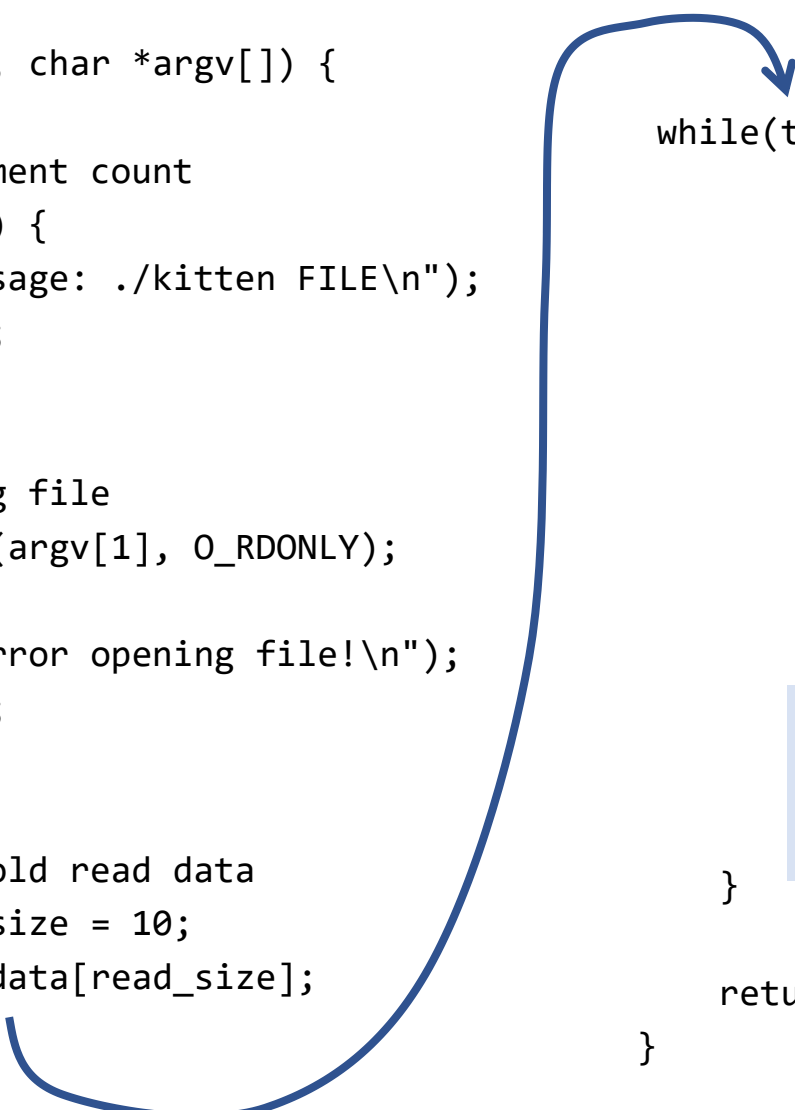
File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to `input()`);
- `stdout` is used for the output of `print()` and `expression` statements and for the prompts of `input()`;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular `text files` like those returned by the `open()` function. Their parameters are chosen as follows:

Example: printing to terminal with a write call

```
int main(int argc, char *argv[]) {  
  
    // check argument count  
    if (argc != 2) {  
        printf("Usage: ./kitten FILE\n");  
        return -1;  
    }  
  
    // try opening file  
    int fd = open(argv[1], O_RDONLY);  
    if (fd < 0) {  
        printf("Error opening file!\n");  
        return -1;  
    }  
  
    // array to hold read data  
    uint8_t read_size = 10;  
    uint8_t read_data[read_size];  
  
    while(true) {  
        // read from file  
        ssize_t read_length = read(fd, read_data, read_size);  
        if (read_length < 0) {  
            printf("Error reading file!\n");  
            return -1;  
        }  
        if (read_length == 0) {  
            break;  
        }  
  
        // print out data  
        ssize_t write_length = write(STDOUT_FILENO,  
                                    read_data, read_length);  
    }  
  
    return 0;  
}
```



Example: trace system calls for commands

- `strace -o syscalls.txt COMMAND`
 - Tracks every system call made by the command
 - Outputs to a file: `syscalls.txt`
- Try on
 - `cat`
 - `parallel-sum-ex`
 - `strace` itself

Break + Open Question

- How does `printf()` work?

Break + Open Question

- How does `printf()` work?
 1. Read in arguments and determine what it needs to format
 2. Create a new string buffer and write formatted values into it
 3. Call `write()` on `STDOUT` with the string

Outline

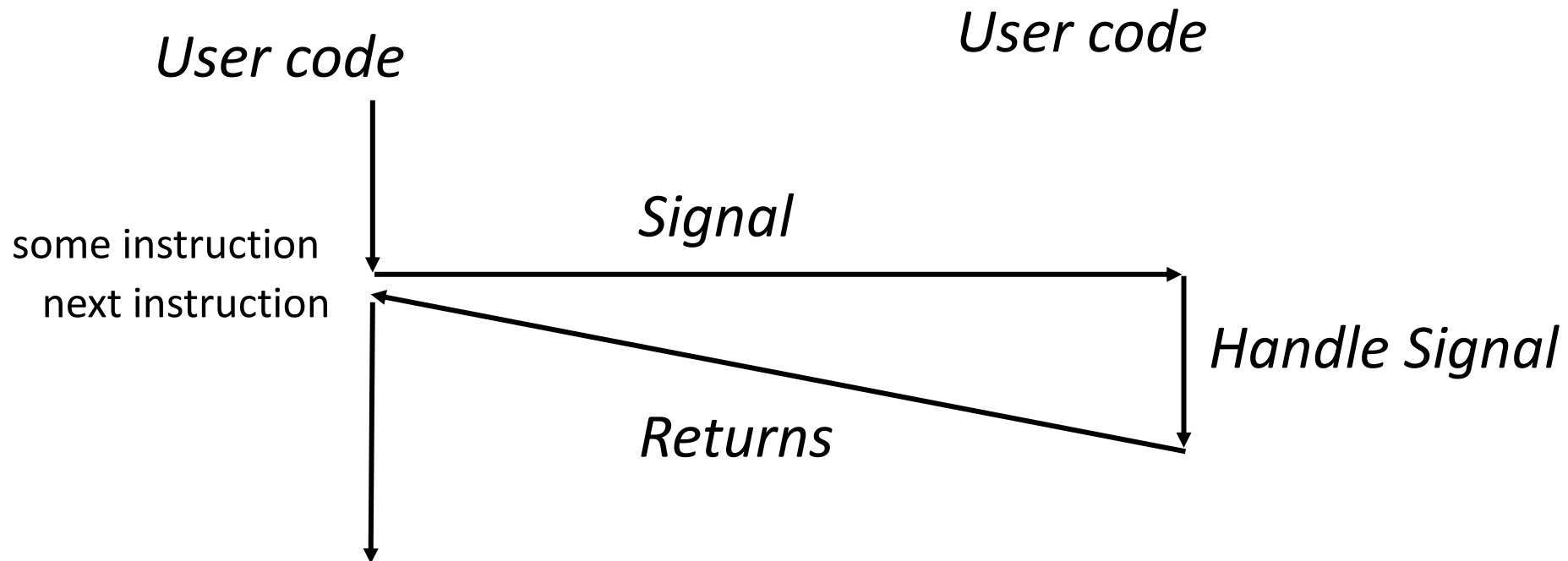
- Process Control Flow
- System Calls
- File I/O
 - Standard I/O
- **Signals**
- Scheduling Processes

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 a different version of exceptional control flow

- Signal is generated by the OS
- Interrupts user code and jumps to a signal handler
 - Then returns back to user code afterwards
 - Unless the signal handler ends the program (this is the default handler)



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) SIGTNT	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);`
- What are you 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)

Signals Examples

Example: catching a signal

```
void sighandler (int signum) {  
    printf("HA HA You can't kill me!\n");  
}  
  
int main (void) {  
    signal(SIGINT, sighandler);  
    printf("Starting\n");  
    while(true) {  
        printf("Going to sleep for a second...\n");  
        sleep(1);  
    }  
    return 0;  
}
```

```
#include <stdbool.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
#include <unistd.h>  
#include <signal.h>
```

Example: catching a segfault

```
int* pointer = 0x00000000;

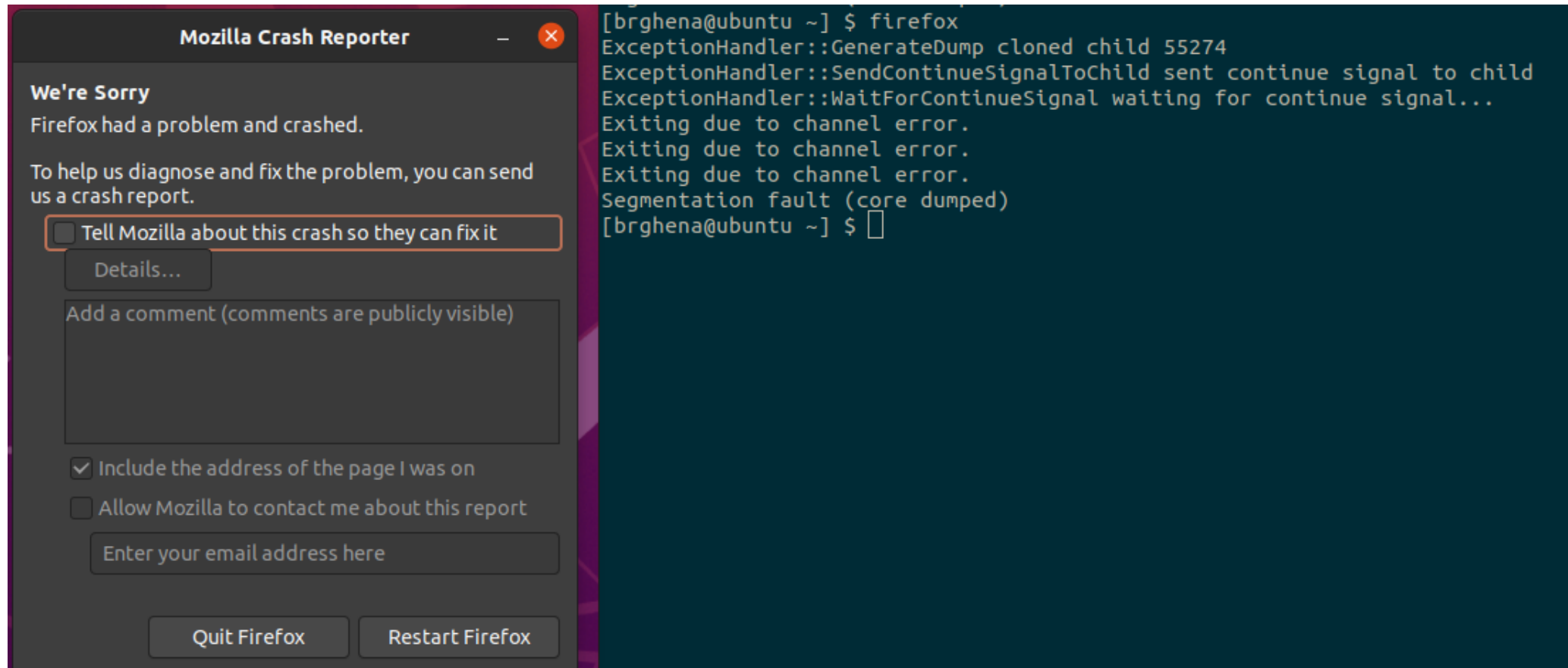
void sighandler (int signum) {
    printf("Oops, that pointer wasn't valid. Try again!\n");
    sleep(1);
}

int main (void) {
    signal(SIGSEGV, sighandler);
    printf("About to read from pointer 0x%08lX\n", (long)pointer);
    int test = *pointer;
    return(0);
}
```

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
```

Examples: sending a signal

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



The image shows a screenshot of a Mozilla Crash Reporter dialog box on the left and a terminal window on the right. The dialog box is titled "Mozilla Crash Reporter" and contains the following text: "We're Sorry", "Firefox had a problem and crashed.", "To help us diagnose and fix the problem, you can send us a crash report.", and a checkbox labeled "Tell Mozilla about this crash so they can fix it" which is currently unchecked. Below this checkbox is a "Details..." button and a text input field for "Add a comment (comments are publicly visible)". There are also checkboxes for "Include the address of the page I was on" (checked) and "Allow Mozilla to contact me about this report" (unchecked), followed by an input field for "Enter your email address here". At the bottom are "Quit Firefox" and "Restart Firefox" buttons. The terminal window on the right shows the command `firefox` being executed, followed by several lines of error output: "ExceptionHandler::GenerateDump cloned child 55274", "ExceptionHandler::SendContinueSignalToChild sent continue signal to child", "ExceptionHandler::WaitForContinueSignal waiting for continue signal...", "Exiting due to channel error.", "Exiting due to channel error.", "Exiting due to channel error.", and "Segmentation fault (core dumped)". The terminal prompt is `[brghena@ubuntu ~] $`.

Outline

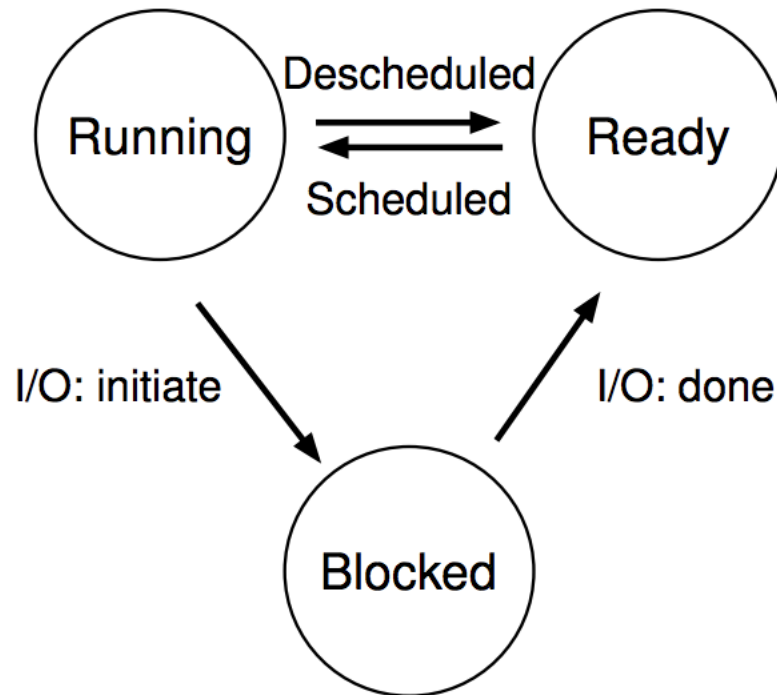
- Process Control Flow
- System Calls
- File I/O
 - Standard I/O
- Signals
- **Scheduling Processes**

Lies your operating system always told you

- “Every process on your computer gets to run at the same time!”
 - This is an *illusion*
- My desktop at home (running Windows)
 - Current load: 250 processes with 2987 threads
- So how does the magic work?

Processes don't run all the time

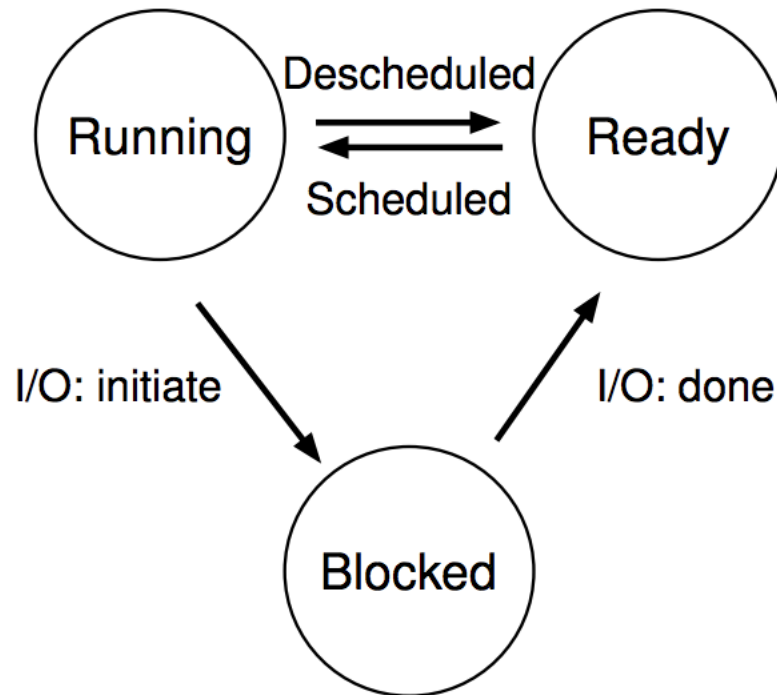
The three basic process states:



- OS *schedules* processes
 - Decides which of many competing processes to run.
- A *blocked* process is not ready to run and is waiting on I/O
- I/O means input/output – anything other than computing.
 - For example, reading/writing disk, sending network packet, waiting for keystroke
 - While waiting for results, the OS **blocks** the process, waiting to do more computation until the result is ready

Multiprogramming processes

The three basic process states:



- Even with a single processor, the OS can provide the illusion of many processes running simultaneously
 - And also use this opportunity to get more useful work done
- When one process is Blocked, OS can schedule a different process that is Ready
- OS can also swap between various Ready processes so they all make progress

Scheduling

- We know that multiple processes will be sharing the CPU
 - Possibly multiple threads in each process
 - Possibly multiple cores in the CPU

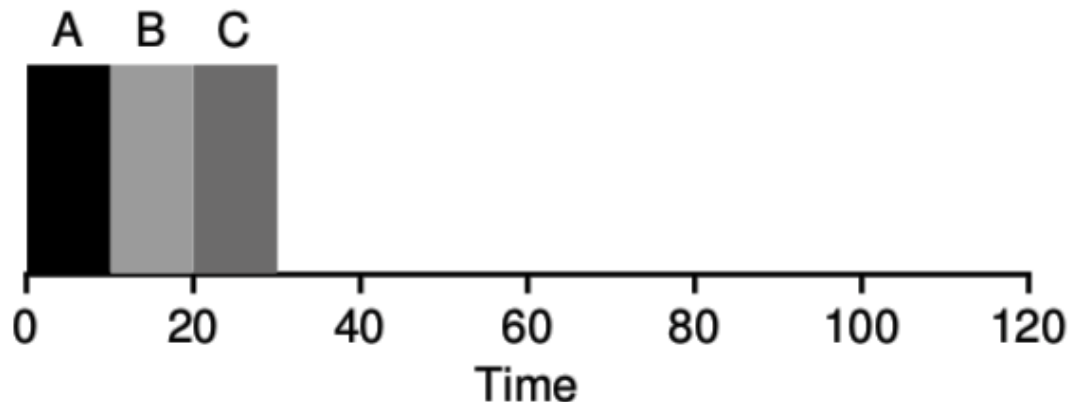
- Scheduling is creating a *policy* for sharing the CPU
 - Which process/thread is chosen to run, and when?
 - When (if ever) does the OS change which process is running?

When can the OS make scheduling decisions?

- Whenever the OS is actually running
 - i.e. after a context switch
- Possible triggers
 - System calls
 - Process/Thread creation/termination
 - I/O requests
 - Hardware events (interrupts)
 - I/O complete
 - Timer triggers

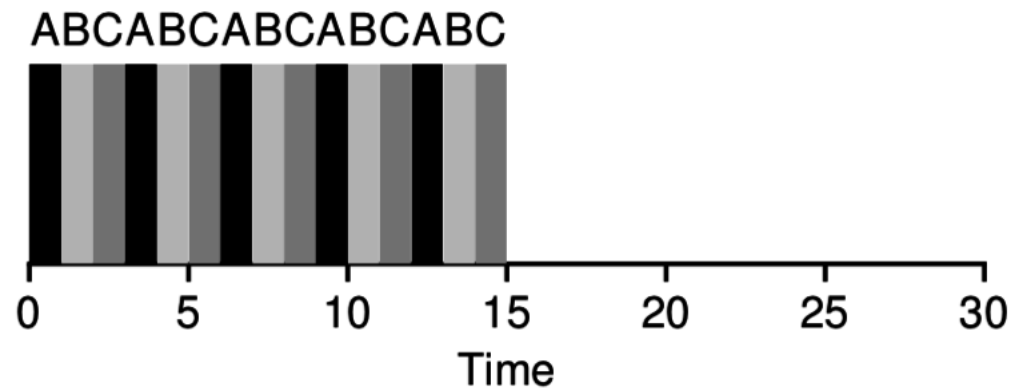
Example scheduler: FIFO Scheduling

- First In, First Out (FIFO)
 - also known as First Come First Served (FCFS)
- Policy
 - First job to arrive gets scheduled first
 - Let a job continue until it is complete
 - Then schedule next remaining job with earliest arrival



Example scheduler: Round Robin Scheduling

- Round Robin scheduling runs a job for a small *timeslice* (quanta), then schedules the next job



- Over time each job makes growing amounts of progress
- If we switch fast enough, it seems to the user that all jobs are running in parallel

CS343 – Operating Systems

- 213 topics in more depth: Concurrency, Processes, Virtual Memory
 - Deal with data races directly
 - Implement virtual memory system
 - Also, totally new topics: File Systems and Devices
- Focus: “how does the Operating System make the computer work”?

Outline

- Process Control Flow
- System Calls
- File I/O
 - Standard I/O
- Signals
- Scheduling Processes