

LINUX CNO Programming



LINUX SYSTEMS PROGRAMMING

USER MODE DEVELOPMENT MODULE

LINUX CNO Programming



Series Overview



LINUX SYSTEM PROGRAMMING SERIES



- After the Linux System Programming Series, the student will understand the basic principles of Linux Programming which will provide the necessary pre-requisites for successfully continuing to the Linux Internals Series
- Assessments:
 - Daily Quizzes
 - 15 Labs
 - Final Assessment
 - Minimum score of 80%

Labs



- 15 labs
 - Do not count towards pass/failure of course
 - All solutions will be posted
- We are here to help
- Lab Tips:
 - Read the man-pages!
 - Always check return values
 - Free what you malloc
 - Use makefiles
 - Use gdb and valgrind



Quizzes and Final Assessment



- You are strongly encouraged to build, compile, and execute any code that might help answer questions
- Quizzes do not count towards passing/failure of the course
 - But are recorded
- All conversations will be taken outside during assessments





Learning Objectives

Given a workstation, device, and/or technical documentation, the student will be able to:

- Use common Linux tools such as gcc, GNU make, valgrind, and gdb
- Create a simple Linux application
- Use the C File API
- Create multiprocess applications that interact via signals
- Create multithreaded applications
- Implement various forms of thread synchronization
- Implement and link with static and dynamic libraries
- Implement networked clients and servers using the POSIX/BSD Socket API (IPv4 and IPv6)
- Use nonblocking I/O and I/O multiplexing techniques, including `select` and `epoll`
- Understand package managers such as apt and dnf/yum

Series Agenda



1. Build Environment
2. Linux API (glibc and POSIX)
3. File API
4. Libraries
5. Processes and Signals
6. Sockets
7. Threads
8. Synchronization
9. I/O Multiplexing
10. Services and Daemons
11. Packages



A Brief History of *nix

- Linux comes from a rich history of operating system development
- Unix (1973) – first portable operating system*
 - not programmed for a specific machine architecture
 - mostly written in C, aside from architecture-specific bits
 - Proprietary, closed-source
- Berkeley Software Distribution (BSD) (1977)
 - Based on Unix code
 - Ancestor of FreeBSD, OpenBSD
- Linux (1991) – developed initially by Linus Torvalds to be an open source operating system with a Unix-like API

**after initially being written in pure assembly*



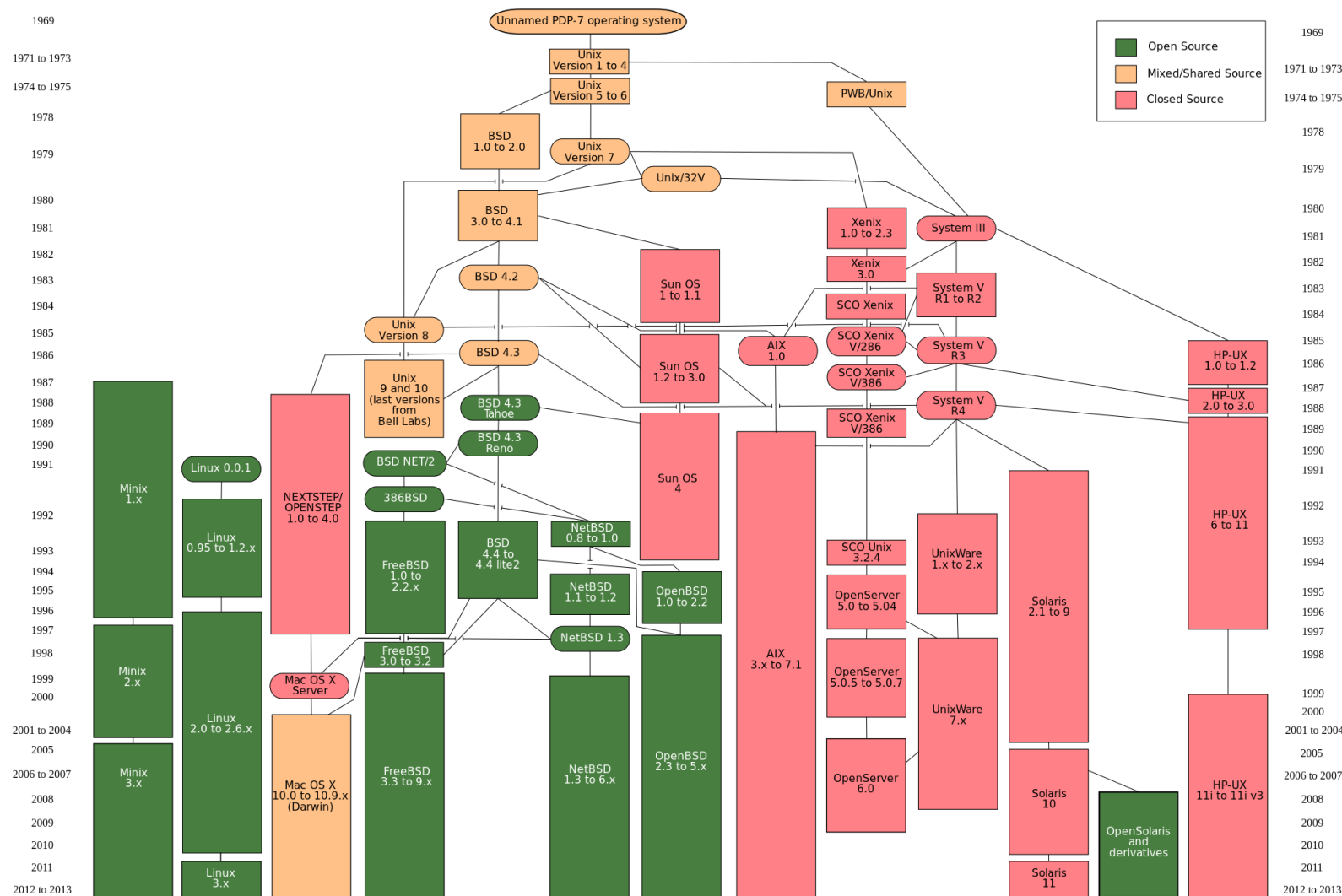
GNU



- **GNU's Not Unix** (GNU, recursive acronym)
 - Pronounced “Guh-new”, not like animal where G is silent
 - A free software effort announced in 1983 by Richard Stallman
 - Intended to create a free computing environment (kernel, system daemons, utilities, etc.) patterned after Unix
 - Free as in freedom (but also beer)
 - GNU Public License (GPL) came out of this effort
- In 1991 GNU had a large amount of system software, but no kernel
 - Linus Torvalds filled this hole with Linux
- Today, the GNU kernel, “Hurd”, is also available
(but not nearly as awesome)



Unix Family Tree



ManTech

LINUX CNO Programming



Build Environment



Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the compilation process for C Linux applications
- View and understand the intermediate outputs of gcc
- Use makefiles to automate compilation
- Use gdb and valgrind to debug programs in real time
- Use objdump to view assembly

Tools of the Trade

- Editor of your choice (vim, emacs, sublime, VS code, etc.)
- GCC
- GNU Make and Makefiles
- GDB
- Valgrind
- Objdump



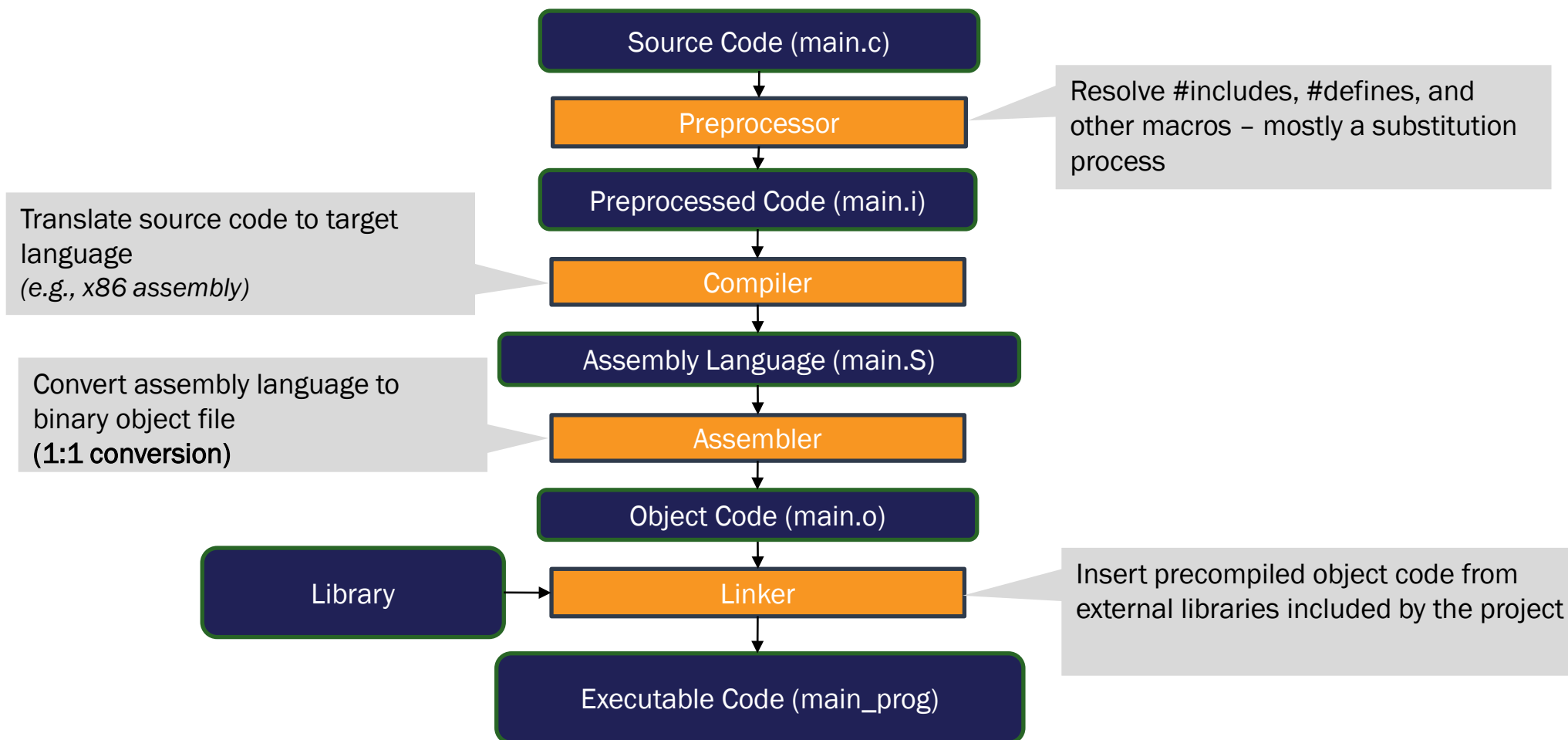
GCC



- The GNU Compiler Collection (GCC)
- Linux's compiler of choice
 - The kernel code is written specifically for this compiler suite, but some other compilers work too
- Can compile C, C++, Objective-C, Fortran, Ada, Golang, and more
- Standard toolchain for all compilation tasks
- Invoke it in your shell with `gcc`
- What happens when you have some source code in `main.c` and compile it with `gcc -o my_prog main.c`?
 - How does `main.h` get included?



Project Build Order



Preprocessor


- A logical unit of functionality which executes prior to the compiler
- Evaluates all preprocessor directives
- In a practical sense, this means that all statements starting with a '#' character are evaluated and replaced
 - `#define`
 - `#include`
 - `#ifdef`
- You can execute ONLY the preprocessor to see the preprocessed source code (useful for debugging macros):
 - `gcc -E main.c`



Compiler

- Translates source code to a target language
- Object code may be machine code (x86 Assembly for example) or an intermediate format (MSIL, Java bytecode)

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv)
4 {
5     printf("Hello World\n");
6     return 0;
7 }
```



```
1 .file "test.c"
2 .text
3 .section .rodata
4 .LC0:
5 .string "Hello World"
6 .text
7 .globl main
8 .type main, @function
9 main:
10 .LFB0:
11 .cfi_startproc
12 pushq %rbp
13 .cfi_def_cfa_offset 16
14 .cfi_offset 6, -16
15 movq %rsp, %rbp
16 .cfi_def_cfa_register 6
17 subq $16, %rsp
18 movl %edi, -4(%rbp)
19 movq %rsi, -16(%rbp)
20 movl $.LC0, %edi
21 call puts
22 movl $0, %eax
23 leave
```

- You can tell GCC to compile your source code without linking or assembling
 - `gcc -S main.c`
(creates main.s file)



Assembler

- Converts assembly language to machine language
- Assembly instructions map directly to the binary read by the target machine (and can be mapped directly back again)

```

1  .file   "test.c"
2  .text
3  .section .rodata
4  .LC0:
5  .string "Hello World"
6  .text
7  .globl  main
8  .type   main, @function
9  main:
10 .LFB0:
11  .cfi_startproc
12  pushq  %rbp
13  .cfi_def_cfa_offset 16
14  .cfi_offset 6, -16
15  movq   %rsp, %rbp
16  .cfi_def_cfa_register 6
17  subq   $16, %rsp
18  movl   %edi, -4(%rbp)
19  movq   %rsi, -16(%rbp)
20  movl   $.LC0, %edi
21  call   puts
22  movl   $0, %eax
23  leave

```

```

00001000: f30f 1efa 4883 ec08 488b 05e9 2f00 0048  ....H...H.../..H
00001010: 85c0 7402 ffd0 4883 c408 c300 0000 0000  ..t...H.....
00001020: ff35 e22f 0000 ff25 e42f 0000 0f1f 4000  .5./...%/.....@.
00001030: ff25 e22f 0000 6800 0000 00e9 e0ff ffff  .%/..h.....
00001040: f30f 1efa 31ed 4989 d15e 4889 e248 83e4  ....1.I..^H..H..
00001050: f050 5449 c7c0 c011 4000 48c7 c150 1140  .PTI....@.H..P.@
00001060: 0048 c7c7 2611 4000 ff15 822f 0000 f490  .H..&.@..../....
00001070: f30f 1efa c366 2e0f 1f84 0000 0000 0090  ....f.....
00001080: b828 4040 0048 3d28 4040 0074 13b8 0000  .(@@.H=(@@.t....
00001090: 0000 4885 c074 09bf 2840 4000 ffe0 6690  ..H..t..(@@...f.
000010a0: c366 662e 0f1f 8400 0000 0000 0f1f 4000  .ff.....@.
000010b0: be28 4040 0048 81ee 2840 4000 4889 f048  .(@@.H..(@@.H..H
000010c0: c1ee 3f48 c1f8 0348 01c6 48d1 fe74 11b8  ..?H...H..H..t..
000010d0: 0000 0000 4885 c074 07bf 2840 4000 ffe0  ....H..t..(@@...
000010e0: c366 662e 0f1f 8400 0000 0000 0f1f 4000  .ff.....@.
000010f0: f30f 1efa 803d 292f 0000 0075 1355 4889  ....=)/...u.UH.
00001100: e5e8 7aff ffff c605 172f 0000 015d c390  ..z...../...]}..
00001110: c366 662e 0f1f 8400 0000 0000 0f1f 4000  .ff.....@.
00001120: f30f 1efa eb8a 5548 89e5 4883 ec10 897d  ....UH..H....}
00001130: fc48 8975 f0bf 1020 4000 e8f1 feff ffb8  .H.u... @.....
00001140: 0000 0000 c9c3 662e 0f1f 8400 0000 0000  ....f.....
00001150: f30f 1efa 4157 4c8d 3db3 2c00 0041 5649  ....AWL.=.,..AVI

```

- You can tell GCC to compile and assemble your code (but not link it): `gcc -c main.c` (produces `main.o`)



Linker

- Combines one or more objects generated by the compiler into a final output program
- Resolves all cross-object references
 - Get addresses of symbols (e.g., functions) defined in other object files in your project
 - Insert symbols that your project imports from static libraries into the executable
 - Provide stubs for dynamic linking (with shared object files) at runtime (we'll get to this later)



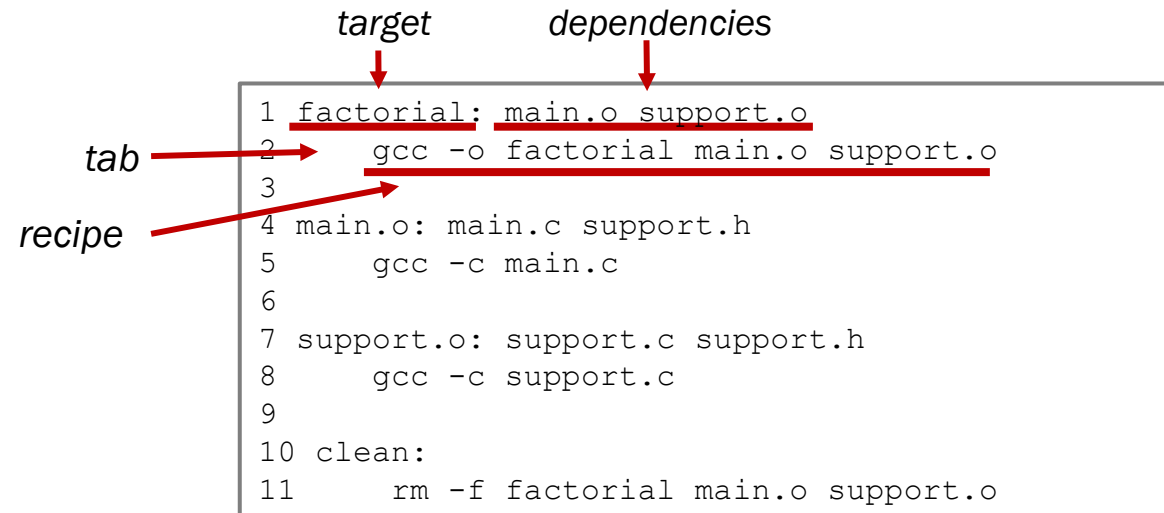
File Types

- **.h**
 - C header file
 - Used during compilation
- **.C**
 - C source code
 - Used during compilation
- **.a**
 - Static library
 - Used during compilation
- **.S**
 - Assembly language file
- **.i**
 - Preprocessed source
- **.ko**
 - kernel object file
 - module/driver
 - Kernel space
- **.SO**
 - Shared object file
 - Used during run time
 - User space
- **.out, no extension**
 - Executable file
 - Standard program
 - User space



Makefile Format

- A makefile consists of a list of rules
 - Rules consist of a target, a list of dependencies, and a recipe
 - Format:



```
1 factorial: main.o support.o
2 gcc -o factorial main.o support.o
3
4 main.o: main.c support.h
5     gcc -c main.c
6
7 support.o: support.c support.h
8     gcc -c support.c
9
10 clean:
11     rm -f factorial main.o support.o
```

The diagram illustrates the components of a Makefile rule. Red arrows point from labels to specific parts of the first rule (lines 1-3): 'target' points to 'factorial:', 'dependencies' points to 'main.o support.o', 'tab' points to the tab character before 'gcc', and 'recipe' points to the command 'gcc -o factorial main.o support.o'.

- Running `make factorial` or `make` will build the program
- Running `make clean` will remove all output files



Makefile Format *(continued)*

- Variables and special directives can shorten rules and make them easier to maintain

SRCS will be a list of all .c files in the directory

OBJS will be a list of .o files, derived from the names in SRCS

This rule will link all the object files in the current directory together to create the "factorial" executable

This rule will compile any .c files in the current directory into corresponding .o files, using the compiler and flags set before

```
CC= gcc
CFLAGS= -g -Wall

EXEC:= demo
SRCS:= $(wildcard *.c)
HDRS:= $(wildcard *.h)
OBJS:= $(SRCS:.c=.o)
LIBS:=

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) -o $@ $^ $(LIBS)

%.o: %.c $(HDRS)
        $(CC) -c $(CFLAGS) -o $@ $<

clean:
        rm -f $(EXEC) $(OBJS)
```

Variables



Make and Makefiles

- *make* automates the build process
 - Used to efficiently compile projects
 - Supports multiple targets (e.g., debug, arch-specific builds, multiple binaries, etc.)
 - Only recreates output files when dependencies have changed - saves time
 - Necessary for large projects, but still helpful for small ones
- Makefiles dictate how *make* should build your project
 - Makefiles can specify targets, their dependencies, and commands to run in order to build them
 - Running *make* from within a directory will use a file named Makefile (if found) as its input
 - Run `make <target>` to specify a target
- Rules for targets that are NOT the names of files to be created can be specified as well
 - Use `.PHONY` to list these targets in your makefile so that they do not conflict with names of files
 - Example: `.PHONY: clean all dothis thing`



Automatic Variables

- make supports “automatic variables” to make your life easier
- Some examples:

AUTOMATIC VARIABLE	MEANING
\$@	File name of the target of the rule
\$<	Name of first dependency in rule
\$^	Names of all the dependencies in rule
\$?	Names of all the dependencies in rule that are newer than the target

- See more at:
https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html





Lab 1

Hello World!

Tasks



- Using an editor of your choice, create main.c with the following

```
#include <stdio.h>
#define START_VAL      0

int main(int argc, char* argv[]) {
    char dummyStr[] = "Hello World!";
    int i;

    printf("argc %d\n", argc);
    for (i = START_VAL; i < argc; i++) {
        printf("%d - '%s'\n", i, argv[i]);
    }

    printf("%s\n", dummyStr);
    return 0;
}
```



Tasks

- Create a file named Makefile in the same directory that can compile main.c into a program

```
CC= gcc
CFLAGS= -g -Wall

EXEC:= demo
SRCS:= $(wildcard *.c)
HDRS:= $(wildcard *.h)
OBJS:= $(SRCS:.c=.o)
LIBS:=

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) -o $@ $^ $(LIBS)

%.o: %.c $(HDRS)
        $(CC) -c $(CFLAGS) -o $@ $<

clean:
        rm -f $(EXEC) $(OBJS)
```



Tasks

1. Run `gcc -E main.c > main.i`
 - Open `main.i` and see what happened to the `#include` and `#define` from the original coded
2. Run `gcc -masm=intel -Og -S main.i`
 - Open `main.s` and see the resulting assembly from the compilation process. Run `gcc -O3 -S main.i` and note the differences that optimizations make
3. Run `gcc -c main.s`
 - Open `main.o` with `xxd` or another hex editor to see the assembled code
4. Run `gcc -o lab_01 main.o`
 - Run your program with `./lab_01 A C T P`
5. Sit back and appreciate GCC and Make
 - run `make` to see the above process automated



Debugger – GDB

- In this course, we will use gdb
- gdb stands for GNU DeBug
 - Works for many programming languages
 - Runs on many Unix-like systems
- gdb features:
 - a command line interface
 - remote debugging
 - a disassembler
 - software / hardware breakpoints
 - dynamic expressions
 - a help system



GDB GEF

- In gdb, registers are referred to by `$<regname>`
 - `$rsp`
- gdb gets a lot of extra information from symbols and debugging information
 - `gcc -g` will add in debugging information
 - the `strip` command will remove extra symbols and debugging information
- gdb allows users to attach to running programs, or start the program itself
- There are configuration files that are available that can extend gdb and give it a nicer interface.
 - gdb **gef** is a simple one that makes it more usable and has some additional features
 - gef installation:
 - `git clone https://github.com/hugsy/gef.git`
 - `echo source `pwd`/gef/gef.py >> ~/.gdbinit`
- gdb commands can be abbreviated
 - instead of `'info registers'`, just type `'i r'`



GDB Details

- Due to Linux security features, by default processes can only debug their own child processes
- You can disable this setting, but it isn't recommended because it affects all processes on the system and represents a security risk
 - Temporarily disable: `echo "0" | sudo tee /proc/sys/kernel/yama/ptrace_scope`
 - Permanently disable: set `kernel.yama.ptrace_scope` to 0 in `/etc/sysctl.d/10-prtrace.conf`
- To attach to a running process
 1. Make sure you have the `CAP_SYS_PTRACE` capability, or the `PTRACE` scope set to 0
 2. Use `pidof` or another command to obtain PID of target process
 3. `gdb <--pid, -p> <pid of process>`
 - attached to the program
 - can also use the `attach` command within `gdb`



GDB Startup

- To launch and debug from gdb
 - `gdb <path to program>`
 - After launching gdb, you can supply command-line arguments with the `run` command (e.g., “`run arg1 arg2 arg3`”)
 - Can also run `gdb --args ./program Arg1 Arg2 ...` to combine steps 1-2
- Other options after launching gdb
 - `starti Arg1 Arg2 ...`
 - Starts the program, stopping at the first instruction



GDB Basics



`break (b) <addr/symbol>`

- places a breakpoint, when execution gets to here it will “pause” the program in gdb
- `b main`
- `b *0x414048`

`del <#>`

- deletes as breakpoint, taking breakpoint number as an argument

`watch`

- sets a “write” data breakpoint, or hardware breakpoint
- breaks when data changes here

`rwatch`

- sets a “read” data breakpoint, or hardware breakpoint
- break when data read here, also if data executed here



GDB Stepping

`continue (c)`

- lets the program continue running when stopped

`step (s)`

- steps a line of source code
- will step into function calls

`next (n)`

- steps a line of source code
- will step over function calls

`stepi (si)`

- step, but steps (into) an instruction instead of a source line

`nexti (ni)`

- next, but steps (over) an instruction instead of a source line



GDB in Reverse

`reverse-step (rs)`

- backsteps a line of source code
- will backstep into function calls

`reverse-stepi`

- reverse step, but backsteps (into) an instruction instead of a source line

`reverse-next (rn)`

- backsteps a line of source code
- will backstep over function calls

`reverse-nexti`

- reverse next, but backsteps (over) an instruction instead of a source line

`reverse-continue (rc)`

- Lets the program continue execution from where you are stopped - in reverse

`reverse-finish`

- Lets the program go back to the start of a function from where you are stopped – opposite of to the return

Note: type the command `target record-full` after `start` or `run` to start the built-in gdb record/replay required for reverse stepping



GDB Tracing/Info

`backtrace (bt)`

- reads from the stack to try to trace what functions called the current one

`print (p) <expression>`

- can take c expressions, with casting and dereferences
- can print out structures

`x/<#><format><unit>`

- some formats = `x`(hex), `s`(string), `i`(instruction)
- unit = `b`(byte), `h`(half), `w`(word), `g`(giant)

`disassemble (disas) [function/address]`

`info registers`

`info file`

- sections of the executable files that are loaded



GDB – More Commands

- `set <thing>=<value>`
 - `set $rax=0x50`
- `return <value>`
 - forces current function to return to its caller, with user defined return value
- `finish`
 - continues execution until the current function returns
- `info locals`
- `info variables`



Valgrind

- Valgrind is a tool for dynamic analysis
 - Pronunciation is “val” + “grinned” (like the Cheshire cat)
 - Named after the main entrance to Valhalla, in Nordic mythology. Only those judged worthy by the guardians of this entrance are allowed to pass through Valgrind
 - Is your code worthy of Valhalla?
- Useful for finding memory leaks, buffer overruns, invalid reads, use of uninitialized values, etc.
- invoke your program with valgrind
 - `valgrind ./my_program arg1 arg2`
- See valgrind help for additional tool options



Objdump

- Objdump can convert object files back to assembly language
 - Also useful for displaying symbol tables, specific object sections, side-by-side view with source, etc.

```

00001000: f30f 1efa 4883 ec08 488b 05e9 2f00 0048    ....H...H.../.H
00001010: 85c0 7402 ffd0 4883 c408 c300 0000 0000    .t...H.....
00001020: ff35 e22f 0000 ff25 e42f 0000 0f1f 4000    .5./...%./....@
00001030: ff25 e22f 0000 6800 0000 00e9 e0ff ffff    .%./...h.....
00001040: f30f 1efa 31ed 4989 d15e 4889 e248 83e4    ....l.I..^H..H..
00001050: f050 5449 c7c0 c011 4000 48c7 c150 1140    .PTI...@.H..P.@
00001060: 0048 c7c7 2611 4000 ff15 822f 0000 f490    .H..&.@.../....
00001070: f30f 1efa c366 2e0f 1f84 0000 0000 0090    ....f.....
00001080: b828 4040 0048 3d28 4040 0074 13b8 0000    .(@@.H=(@@.t....
00001090: 0000 4885 c074 09bf 2840 4000 ffe0 6690    ..H..t..(@@...f.
000010a0: c366 662e 0f1f 8400 0000 0000 0f1f 4000    .ff.....@.
000010b0: be28 4040 0048 81ee 2840 4000 4889 f048    .(@@.H..(@@.H..H
000010c0: c1ee 3f48 c1f8 0348 01c6 48d1 fe74 11b8    ..?H...H..H..t..
000010d0: 0000 0000 4885 c074 07bf 2840 4000 ffe0    ....H..t..(@@...
000010e0: c366 662e 0f1f 8400 0000 0000 0f1f 4000    .ff.....@.
000010f0: f30f 1efa 803d 292f 0000 0075 1355 4889    .(==)/...u.UH.
00001100: e5e8 7aff ffff c605 172f 0000 015d c390    ..z...../...].
00001110: c366 662e 0f1f 8400 0000 0000 0f1f 4000    .ff.....@.
00001120: f30f 1efa eb8a 5548 89e5 4883 ec10 897d    .....UH..H....}
00001130: fc48 8975 f0bf 1020 4000 e8f1 feff ffb8    .H.u... @.....
00001140: 0000 0000 c9c3 662e 0f1f 8400 0000 0000    .....f.....
00001150: f30f 1efa 4157 4c8d 3db3 2c00 0041 5649    ....AWL.=.,.AVI

```



```

Disassembly of section .init:

0000000000401000 <_init>:
401000:    f3 0f 1e fa                endbr64
401004:    48 83 ec 08                sub     $0x8,%rsp
401008:    48 8b 05 e9 2f 00 00        mov     0x2fe9(%rip),%rax
40100f:    48 85 c0                    test    %rax,%rax
401012:    74 02                       je      401016 <_init+0x16>
401014:    ff d0                       callq   *%rax
401016:    48 83 c4 08                add     $0x8,%rsp
40101a:    c3                          retq

Disassembly of section .plt:

0000000000401020 <.plt>:
401020:    ff 35 e2 2f 00 00        pushq   0x2fe2(%rip)
401026:    ff 25 e4 2f 00 00        jmpq    *0x2fe4(%rip)
40102c:    0f 1f 40 00                nopl    0x0(%rax)

0000000000401030 <puts@plt>:
401030:    ff 25 e2 2f 00 00        jmpq    *0x2fe2(%rip)
401036:    68 00 00 00 00 00        pushq   $0x0
40103b:    e9 e0 ff ff ff            jmpq    401020 <.plt>

```

- Example: `objdump -d main.o`
 Add `-M intel` for intel syntax



LINUX CNO Programming



Introduction to the Linux API



Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe the POSIX API and glibc
- Use man-pages to find and use Linux API functionality without using the Internet
- Properly check error codes and use errno
- Know basic coding conventions on Linux platforms

The POSIX API

- Linux supports much of the Portable Operating System Interface (POSIX), which defines standards for:
 - API endpoints for functions
 - `strcpy`, `mmap`, `fopen`, `read`, `socket`, etc.
 - Command-line tools
 - `df`, `du`, `awk`, `ed`, etc.
- The POSIX standard is periodically updated
 - Add new functions, base definitions, etc.
 - POSIX 2001 and 2008 are the most commonly supported
 - Most of the time you won't need anything beyond 2001



The GNU C Library (glibc)

- Implementation of the C standard library
 - Contains definitions of common functions
 - `strcpy`, `malloc`, `free`, `recv`, etc.
 - Used by C programs, but also many other higher-level languages (e.g., the Python interpreter)
- Provides heap implementation with `malloc`
- Wraps system calls
- Also supports non-portable GNU extensions
 - `asprintf` – allocating version of `sprintf`
 - Built-in linked list via `insque`, `remque`
 - Built-in hash table via `hsearch`
 - Use `#define _GNU_SOURCE` to enable



Man-pages

- Manual Pages (colloquially known as man-pages) provide information on functions and command-line tools
- Stored in `/usr/share/man`
- These are your best friend
- The manual pager, `man`, can be used to view man-pages from the shell
 - The displayed text will provide a summative descriptive
- Example: `man printf`, `man echo`, `man socket`
- Mind the Notes, Example Usage, and Bug subsections!



Man-page sections

- Manual entries can contain multiple sections
 - access a specific section with `man <section #> <function>`
- Some common section commands:
- Example:
`man 7 socket, man 2 read`
- Try it:
`man 1 printf` **vs** `man 3 printf`

man COMMAND	DESCRIPTION
man 1 <topic>	General commands
man 2 <topic>	System calls
man 3 <topic>	C standard library
man 4 <topic>	Drivers / Special Files
...	...



Searching Man-pages

- The utility `apropos` can search through man-pages for you
- You can also search for a string within the man-pages using the `-k` parameter
 - This only searches the name and description portions of a man-page
 - Example: `man -k socket`
- Search the full text of man-pages using an uppercase `-K` parameter
 - This option alone will also simply bring up the first man-page it finds with the given search term
 - You have to close the man-page it brings up to get to the next search result
- See a list of all the man-pages that contain a given search term using `-Kw`
 - Example: `man -Kw sockaddr_in6`
- Further refine your search to a given man-page section with `-s`
 - Example: `man -Kws 3 sockaddr_in6`



Linux API

- Combination of C standard library functions (usually provided by glibc) and system calls provided by the operating system
- Linux supports most of the POSIX standard
- POSIX and GNU standards specify a great deal of functionality, and Linux supports (most of) these, and some of its own
 - See “Conforming To” subsection of man-pages to see to what standard a given function belongs, if any
- Examples of non-portable Linux functionality: `sendfile`, `splice`, `pipe2`, `clone`



Error checking

- Always check return values for errors!
- Unix-like systems employ a global variable, `errno`, that stores a descriptive error code from the most recent system call and some library functions
 - It's actually a macro for a function call, to allow multithreading support
- The `perror` and `strerror` functions convert `errno` to a useful string for human readability
- To use `errno` and `perror`, `#include <errno.h>`
- To use `strerror`, `#include <string.h>`



Errno Usage

```
1 #include <unistd.h>
2 #include <stdio.h>
3
4 int main(int argc, char* argv) {
5     FILE* fp;
6     fp = fopen("doesnotexist", "r");
7     if (fp == NULL) {
8         perror("fopen");
9     }
10    return 0;
11 }
```

```
$ ./demo
$ fopen: No such file or directory
```



Errno Usage



```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <errno.h>
5
6 int main(int argc, char* argv[]) {
7     FILE* fp;
8     char filename[] = "doesnotexist";
9     fp = fopen(filename, "r");
10    if (fp == NULL) {
11        fprintf(stderr, "Error opening file %s: %s\n", filename, strerror(errno));
12    }
13    return 0;
14 }
```

```
$ ./demo
$ Error opening file doesnotexist: No such file or directory
```



Error Checking Gotchas

- Not all functions return the same value for failure cases
- Make sure to check the man-pages to see what a function returns

```
sock = socket(PF_INET, SOCK_STREAM, 0);
if (!sock) {
    perror("socket");
}
...
buffer = (char*)malloc(1024);
if (!buffer) {
    fprintf(stderr, "Failed in malloc\n");
}
...
if (!sprintf(buffer, "message number %d\n", i)) {
    fprintf(stderr, "Failed in sprintf\n");
}
```

- Which if statements are checking error values correctly?
 - Use `man` to find out



Coding Conventions

- GNU conventions
 - Structs:
 - Don't do typedefs and structure tags in the same declaration
 - Functions:
 - declarations all go in one place at the beginning of a file
 - Named with lower case words, underscore separated
 - Parameters, Variables:
 - lower case words, underscore separated
 - Macros, Constants:
 - Prefer `enum` instead of `#define` (GDB knows enums)
 - Upper-case, underscore separated
 - See <https://www.gnu.org/prep/standards/standards.html>



A Note About Includes

- When you `#include` header files with names enclosed by `<>`, gcc will search for them in the default include directory
 - This is `/usr/include` on most systems
- To find definitions for structures, variables, macros, etc. in these files, grep that directory recursively
 - Example: `grep -rn "term" /usr/include`
- Can you find the definition for the `stat` structure for your system?



More Macros

- There are many helpful predefined Macros:
 - `__FILE__` expands to the name of the current code file
 - `__LINE__` expands to the current line number in the code file
 - These can help with debugging code
 - `printf("Error at %s:%d\n", __FILE__, __LINE__);`
- Macros can have arguments as well.
 - Be careful! It is all just text replacement. Why does the example below fail?

```
#define MUL(X, Y)      (X * Y)
#define ERR(reason)    printf("Error (%s@%d) : %s\n", __FILE__, __LINE__, reason); exit(-1)

int main() {
    if (MUL(3+3, 6) == 21) {
        ERR("Bad Macro!");
    }
    return 0;
}
```

`if ((3+3 * 6) == 21) {`
 `3+18 = 21`



LINUX CNO Programming



File API



Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe the standard C File API
- Implement code to enumerate files
- Implement code to read and process files

File API

- The `open` function is a Unix system call that can open a file and return a file descriptor (integer)
 - Simple, unbuffered access to the file
 - Useful for more advanced use cases
- The `fopen` function is a C language standard that opens a file and returns a file *stream* (`FILE*`)
 - provides buffered I/O
 - Useful for traditional file reading and writing
 - `fprintf`, `fscanf`, etc.
- Obtain a stream `FILE*` from a file descriptor with `fdopen`



Standard Files

- By default, each Linux process has access to three file descriptors: standard input, standard output, and standard error
 - These have the integer values 0, 1, and 2, respectively
 - Best to access them using macros `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`, for portability
- Access their corresponding streams with `stdin`, `stdout`, `stderr`
- `fprintf` will print to your file of choice
 - `printf` is the same as `fprintf(stdout, ...)`



File Enumeration API

- `#include <sys/types.h>`
- `#include <dirent.h>`
- `opendir`
- `readdir`
- `scandir`
- `closedir`
- `rewinddir`



File I/O API



- `#include <stdio.h>`
- `fopen`
- `fread`
- `fgets`
- `fwrite`
- `fseek`
- `flock` (BSD, not C or POSIX – use `fcntl` if needed)
- `fflush`
- `feof`
- `fclose`



File Information API

- `#include <sys/types.h>`
- `#include <sys/stat.h>`
- `fstat`
- `stat`







Lab 2


`wc -l -w -m`

Tasks

- Build a program that mimics the output of the `wc` (wordcount) utility
- Only concern yourself with the output `wc -l -w -m`
 - Don't bother with user-specified options
 - Just take in a single command-line argument, a path to a file
- For reference, your output should appear as below when parsing the provided LICENSES file:
 - ```
391 2852 18941 LICENSES
```

  
*lines*

  
*words*

  
*characters*







## Lab 3

```
ls -l
```

# Tasks

- Build a directory listing tool
  - Approximate the output of “`ls -l`”
  - Focus on the listing of directory contents and some attributes
  - Don’t worry about formatting things exactly like `ls`
- Start with enumerating the contents of a directory, then move on to displaying file attributes
- Hint: man-pages are your friend
- Useful functions:
  - `opendir`, `readdir`, `closedir`, `stat`, `getpwuid`, `getgrgid`, `localtime`, `strftime`



## Bonus Tasks

- Recursively enumerate directories (breadth-first or depth-first)
- Take in a command-line argument that specifies the path of the directory to enumerate, if present (otherwise default to the current directory)



# LINUX CNO Programming



## Libraries

SO what?



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Describe static and dynamic libraries and the differences between them
- Create static and dynamic libraries
- Create programs that link with static and dynamic libraries at compile-time
- Create programs that link with static and dynamic libraries at run-time

# Types of Libraries

- Static (archives)
  - .a files
  - Static linking is performed exclusively at link-time
  - Code for all the functions you use from the library is placed into the resulting binary
- Dynamic (shared objects)
  - .so files
  - Supports both linking during run-time as well as dynamic loading and unloading (to support things like plugins)
  - Code from library is not inserted into binary, but is assumed to be present on the target system for use during runtime





# Static Library (.a) Use

- **Upside:** The version of the library you test with will be the same as the one on the target machine
- **Upside:** Do not have to worry about whether the target machine has the library
- **Upside:** No overhead for calling library functions versus native program ones
- **Upside:** library symbols can be stripped out of the binary
- **Downside:** You are responsible for updating the library when bugs are found (cannot rely on users to update their libraries)
- **Downside:** Increase in binary size
- **Downside:** More cumbersome to include cross-platform functionality (glibc and other platform-specific libraries are dynamically-linked so programs use the correct one for the target platform)



# Shared Library (.so) Use

- **Upside:** reduces the size of program binaries
- **Upside:** reduces overall memory consumption of the machine as multiple processes can share a single instance of the shared object
- **Upside:** Skirts some licensing issues (LGPL allows inclusion of free software libraries in commercial products through dynamic linking)
- **Downside:** Dependency issues. What if the shared library isn't on the target computer? What if it's an older version than what you need?
- **Downside:** Run-time linking requires a level of indirection that adds a small amount of overhead





# Viewing Linked Shared Libraries

- You can view the libraries with which an executable links using `ldd`

```
[student@localhost ~]$ ldd /bin/ls
linux-vdso.so.1 (0x00007ffdf553e000)
libselinux.so.1 => /lib64/libselinux.so.1 (0x00007f2be091a000)
libcap.so.2 => /lib64/libcap.so.2 (0x00007f2be0913000)
libc.so.6 => /lib64/libc.so.6 (0x00007f2be074d000)
libpcre2-8.so.0 => /lib64/libpcre2-8.so.0 (0x00007f2be06bf000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f2be06b9000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2be098f000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f2be0698000)
```

↑  
*Shared  
library*

↑  
*Location on disk*

↑  
*Address where library gets loaded  
into the executable (will change  
due to address space  
randomization, which we'll cover in  
another class)*



# Creating a Library

- Create a collection of functions, globals, etc. that you would like to export as a library
  - Adopt a naming convention that prevents name conflicts with symbols from the programs that link with your library
  - Do not make a function named main 😊
  - Create a header file that contains the declarations of the functions you want to export

## Header file

```
1 #ifndef ACTP_FACTORIAL_H
2 #define ACTP_FACTORIAL_H
3
4 unsigned int actp_factorial(unsigned int num);
5
6 #endif
7
```

*Exported functions  
prefixed with actp\_ to  
prevent name conflicts*

## Source file

```
1 #include <stdio.h>
2 #include "actp_factorial.h"
3
4 unsigned int actp_factorial(unsigned int num) {
5 unsigned int counter
6 unsigned int result = 1;
7 for (counter = 1; counter <= num; counter++) {
8 result *= counter;
9 }
10 return result;
11 }
```



# Library Constructors and Destructors

- You can make constructors and destructors that will be invoked when your library is loaded and unloaded
- Use gcc attributes
  - `__attribute__((constructor))`
  - `__attribute__((destructor))`
- Example:
  - Define two functions, `on_load` and `on_unload`, which take no parameters and return `void`
  - Declare them like this
    - `void on_load(void) __attribute__((constructor));`
    - `void on_unload(void) __attribute__((destructor));`



# Creating a Static Library

- Static libraries are created from object files using the ar utility (see `man 1 ar`)
- To turn a set of object files into a .a file usable for static linking:
  - `ar rcs <lib file> <obj 1> <obj 2> ...`
  - Example: `ar rcs libactp.a actp_factorial.o`
- Provide your public header files with your .a file to users of your library



# Linking with a Static Library

- Include the header files from the library in your code
  - Example: `#include "actp_factorial.h"`
- Use whatever functionality wanted from the library
- When compiling, tell GCC to link with the library
  - `gcc -L. -o demo main.c -l:libactp.a`
    - `-L` = the path to search for extra libs
    - `-l:` = the full name of the library file
  - The binary “demo” now contains all the functions main.c used from libactp.a



# Creating a Dynamic Library

- Dynamic libraries need to be loadable at run-time, which means they must be position independent
- Position independent code (PIC) can be generated by GCC by passing the `-fPIC` option
  - Example: `gcc -c -fPIC -o actp_factorial_pic.o actp_factorial.c`
- After compiling all of your library source for PIC, combine them into a shared library
  - Example: `gcc -shared -o libactp.so actp_factorial_pic.o`
  - A lot of other shared library options can be supplied here
- Provide your public header files with your `.so` file to users of your library



# Linking with a Dynamic Library

- Include the header files from the library in your code
  - Example: `#include "actp_factorial.h"`
- Use whatever functionality wanted from the library
- When compiling, tell GCC to link with the library
  - `gcc -L. -Wl,-rpath=. -o demo main.c -l:libactp.so`
    - `-Wl,-rpath=` = the path to find the library at runtime (unnecessary if the library is in a standard location such as `/var/lib`)
    - `-L` = the path to search for extra libs
    - `-l:` = the full name of the library file
      - Can also use shorthand "`-l<name>`" for linking with `lib<name>.so`
  - The binary "demo" will now find `libactp.so` at runtime and load it from disk to use its functionality



# Explicit Library Loading

- Dynamic libraries can also be loaded explicitly by the programmer during run-time
- This does not require the compiler's participation at link-time
  - Aside from linking with the dynamic linking loader (`-ldl`)
- Need to `#include <dlfcn.h>`
- The `dlopen` function loads a shared object
- The `dlsym` function resolves a given symbol for use by the program
- The `dlclose` function unloads the shared object (when its reference count is zero)







## Lab 4

Linking

# Tasks

1. Create a static library and a Makefile to build and link it with a main application that uses it
  - The library should have at least 3 functions
    - A function that gets invoked when the library loads
    - A function that gets invoked when the library unloads
    - Some function that can be called directly by the app
2. Modify the makefile so that it creates a dynamic library instead
3. Modify the code in the main application so that it uses the dynamic library explicitly, using `dlopen`, `dlsym` and `dlclose`



# LINUX CNO Programming



## Processes



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the nature of processes in Linux
- Programmatically create new processes for multiprocess programs
- Programmatically start a new process with another image
- Properly reap terminated child processes and obtain information from them

# Processes

- A process is an instance of a program
- The kernel maintains process state and determines when to service and run the process
- In Linux, a process represents a single unit of execution and has
  - Memory (stack, heap, kernel stack, etc.)
  - State (running, ready, waiting, etc.)
  - Set of open file descriptors and other resources
  - etc.
- This is in contrast to Windows, wherein processes themselves are not schedulable entities (but threads within them are)





# Process Hierarchy

- When one process creates another, the first is called the parent
- The new process is called the child
- Every process has a parent
- Children can have their own children
- Children processes of the same parent are called siblings
- The ancestor of all userspace processes is init (PID 1)
  - This is started by part of the kernel scheduler (swapper) as part of the startup sequence



# Creating a Process

- Programmatically creating a new process is typically done using the `fork` system call
- `fork` essentially creates a clone of the process that calls it
  - After a successful fork call, two processes are running the same code and have equivalent copies of memory\*
- How do you tell difference between the two processes?
  - `fork` returns twice, once for the calling process, and once for the newly created one
  - the child is returned the value zero
  - the parent is returned the PID of the child

*\*An optimization called “copy on write” makes this more efficient. We will cover this in Linux Internals*



# Fork Example



```
6 int main() {
7 pid_t pid;
8
9 pid = fork();
10 if (pid == -1) {
11 perror("fork");
12 return EXIT_FAILURE;
13 }
14 else if (pid == 0) {
15 printf("I'm the child\n");
16 }
17 else {
18 printf("I'm the parent, and my child's PID is %d\n", pid);
19 }
20 printf("I'm quitting\n");
21 return EXIT_SUCCESS;
22 }
```





# Fork Example

- Output from previous code:

```
student@student-virtual-machine:~/linux/demos$./process_demo
I'm the parent, and my child's PID is 26142
I'm quitting
student@student-virtual-machine:~/linux/demos$ I'm the child
I'm quitting
```

- Why did a shell prompt appear before the child process print outs finished?
- Will this always happen?



# The Exec Family

- Often you will not want to clone your existing process, but launch a completely different one
- The `exec` family of functions enable this for you
  - On success, `exec` functions never return to the calling process
  - Instead, they load a new program (specified by the caller) into the process
- Pairing a `fork` with an `exec` function is the recipe for programmatically launching any program of your choosing
- Separating process creation from the program it loads allows programmers to perform any setup they wish between these steps



# The Exec Family

- There are many exec functions, each with slightly different functionality:
  - `execl`, `execvp`, `execle`, `execv`, `execvp`, `execvpe` (see `man 3 exec`)
- `l` means the function takes program arguments in a variable arguments `list` format (like `printf`)
- `v` means the function takes program arguments in the same form as `argv` in a `main` function
- `p` means the function will search for the specified program as the shell does (`PATH`, current dir, etc.)
- `e` means the function can be supplied a custom environment in which the specified program should run
- All of these are front-ends for the `execve` function (see `man 2 execve`)



# Exec Example



```
6 int main() {
7 char* ls_args[3];
8 char path[] = "/bin/ls";
9 ls_args[0] = path;
10 ls_args[1] = "-l";
11 ls_args[2] = NULL;
12
13 printf("I'm gonna turn into ls -l\n");
14
15 if (execv(ls_args[0], ls_args) == -1) {
16 perror("execv");
17 return EXIT_FAILURE;
18 }
19 return EXIT_SUCCESS;
20 }
```

```
student@student-virtual-machine:~/linux/demos$./exec
I'm gonna turn into ls -l
total 20
-rwxrwxr-x 1 student student 8673 Nov 28 14:46 exec
-rwxrw-r-- 1 student student 369 Nov 28 14:47 exec.c
```





## Lab 5

`fork`, `execv` and Firefox

# Tasks

- Create a program that uses `fork` and `execv` to launch Firefox
- Using command-line arguments to Firefox, make it launch a private browsing session and navigate to canvas
  - `firefox --private-window https://canvas.class.net`
- Use `wait` or `waitpid` to wait for and detect when the user closes firefox
- Launch Firefox again whenever the user closes it
- Make sure that you have no other instances of firefox running before you attempt to run your program or you may encounter odd behavior



# Reaping Zombies



- When a child process terminates, it becomes a “zombie” and needs to be “reaped” by its parent to clean up resources
- The zombie process state allows parents to obtain data from their terminated children
- Parents must “reap” terminated children to clean up system resources
- The wait and waitpid system calls are used to reap children and obtain information from them
  - Also allows the caller to suspend execution until a child process changes state



ManTech



# Reparenting

- What happens if a parent terminates before all of its children terminate?
- The children become orphans (obviously)
- If another thread in the same thread group as the parent exists, the children get adopted by that thread
- If not, they get adopted by init
  - The init daemon process periodically reaps its zombie children to prevent system resource leakage
- As of Linux 3.4, a process can also register itself as a “subreaper”, which means it becomes the parent of any orphaned descendents
  - See `prctl` with `PR_SET_CHILD_SUBREAPER`





# LINUX CNO Programming



## Signals



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Create programs that send and receive signals
- Modify the behavior of signals, including writing signal handlers
- Use signals to control other processes programmatically

# Signals



- Processes can receive and send signals to one another to indicate certain directives or conditions
- Each signal has an associated *disposition* which dictates how a process responds to the signal upon reception
  - For example, upon receiving a SIGINT (possibly sent via Ctrl+C), the default disposition is for the receiving process to terminate
  - Other dispositions are possible: ignore, stop, continue, terminate and core dump
- There are numerous, but finite signals
  - Some common ones: SIGHUP (hangup), SIGINT (interrupt), SIGSTOP, SIGCONT, SIGSEGV (seg fault), SIGCHLD (child stopped/terminated)



# Changing Signal Disposition

- Processes can change the disposition of a given signal using `signal` and `sigaction`

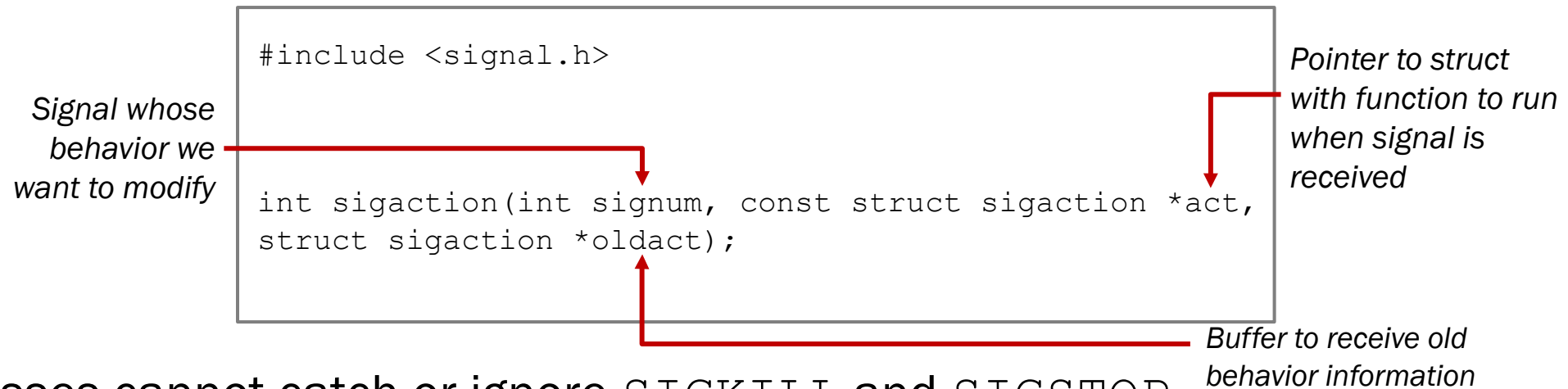
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <signal.h>
4
5 int main() {
6 signal(SIGINT, SIG_IGN);
7 while (1) { printf("can't Ctrl+C me!\n"); }
8 return EXIT_SUCCESS;
9 }
```

- `SIG_IGN` ignores the signal, `SIG_DFL` sets the default disposition



# Signal Handlers

- `sigaction` can also be used to register a signal handler, which is a function that gets invoked whenever the calling process receives a given signal
  - `signal` can also be used for this, but is not portable (so don't)



- Processes cannot catch or ignore `SIGKILL` and `SIGSTOP`
  - So *signum* cannot be one of these



# Signal Handlers

- The `sigaction` structure

```
struct sigaction {
 void (*sa_handler) (int);
 void (*sa_sigaction) (int, siginfo_t *, void *);
 sigset_t sa_mask;
 int sa_flags;
 void (*sa_restorer) (void);
};
```

- Zero it out before setting any fields
- `sa_handler` will be a pointer to your handler function OR `SIG_IGN` / `SIG_DFL`
- `sa_flags` control finer points of signal behavior, usually set to `SA_RESTART`
  - See `man 2 sigaction` for more information



# Signal Handler Example

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4 #include <stdlib.h>
5
6 void signal_handler(int signum) {
7 write(STDOUT_FILENO, "Got SIGINT\n", 11);
8 _exit(EXIT_SUCCESS);
9 }
10
11 int main() {
12 struct sigaction sa = { 0 };
13 struct sigaction old_sa;
14
15 sa.sa_handler = signal_handler;
16 sigemptyset(&sa.sa_mask);
17 sa.sa_flags = SA_RESTART;
18
19 if (sigaction(SIGINT, &sa, &old_sa) == -1) {
20 perror("sigaction");
21 return EXIT_FAILURE;
22 }
23
24 while (1) { sleep(1); }
25 return EXIT_SUCCESS;
26 }
```

Signal handler  
which calls  
write and  
\_exit\*

Register handler for  
SIGINT

\*Not all functions are safe  
to call inside a signal  
handler, see man 7 signal  
for a list of safe functions



# Sending Signals

- Programmatically, the `kill` system call sends a given signal to a given process or group of processes
  - Despite its name, not all signals will “kill” a process
- A command-line tool, also named `kill`, is available
  - Example: `kill -SIGINT `pidof my_program``
- The `raise` function sends a signal to the calling process
- See `man 7 signal` for more information





# Signal Handling and System Calls

- Numerous system calls can be interrupted by signal handling
- You can handle this three different ways:
  - Set `SA_RESTART` in the `sigaction` call
    - This automatically restarts a blocking system call when the signal handler finishes running
  - Manually restart it by checking the `errno` value of the system call for `EINTR` and calling it again
  - Use `sigprocmask` to temporarily block reception of specified signals before issuing the blocking call. Reset the signal mask afterward to restore signal reception



# Handling Repeated Signals

- Receiving multiple signals of the same type is not differentiated from receiving a single signal of that type
  - Signal handlers should interpret receiving a signal X as receiving one or more signals of type X
- Example: a parent process receives a SIGCHLD signal when a child process terminates
  - If the parent registers a signal handler for reaping zombie children, it should call `waitpid` repeatedly

```
void child_reaper(int signum) {
 int saved_errno = errno;
 while (waitpid(-1, NULL, WNOHANG) > 0) {}
 errno = saved_errno;
 return;
}
```

*We use a while loop here so that we can handle the case wherein multiple children have terminated*



## Other Useful Signal Functions

- You can synchronously wait for a given signal you expect to receive using `sigwait`
- You can view pending signals for the current thread using `sigpending`
  - these are signals that have been blocked by your process, possibly with the use of `sigprocmask`
- You get a signal file descriptor to `read` and react to signals instead of using handlers using `signalfd`
  - This allows event-driven programs using `select`, `epoll`, etc. to handle I/O and signal handling with a single event loop





## Lab 6

Sending signals

# Tasks



- Create a program that launches the toofast binary and remotely controls it with signals
- Your program should `fork` and then `execv` toofast
- When a user presses the given key, send signals to your child process to make it perform the appropriate action
  - `u` = speed up playback
  - `d` = slow down playback
  - `p` = pause playback
  - `s` = start/resume playback
  - `q` = stop playback
- View the source of toofast (in `child_main.c`) and determine which signals to send for each action



# LINUX CNO Programming



## Sockets





# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Create network applications using the POSIX socket API
- Create sockets of different protocols including UDP, TCP, and Unix domain sockets
- Use IPv4 and IPv6 addresses
- Use getaddrinfo to create, bind and connect sockets
- Create clients and servers for connectionless protocols
- Create clients and servers for connection-based protocols

# The Socket API

- The POSIX socket API is taken directly from the BSD sockets
  - “BSD sockets” and “POSIX sockets” are synonymous
- From the perspective of userspace programs, sockets are merely integers, which represent a file descriptor
- Since sockets are file descriptors, normal file operations such as `write` and `read` can be performed on them
  - These correspond to sending and receiving network data, respectively
- Sockets also have special I/O functions that allow more control over network-specific behaviors:
  - `send`, `sendmsg`, `sendmmsg`
  - `recv`, `recvmsg`, `rcvmmmsg`





## The Socket API *(continued)*

- The API is used by every major operating system
  - Windows modified it slightly (e.g., `closesocket` instead of `close`), and added some needless extensions
- Integrated so deeply within Linux that the glibc functions are *extremely* light-weight wrappers over system calls of the same name
- Languages outside of C/C++ often wrap the API directly for their networking functionality (e.g., Python), or wrap its network system calls (e.g., Golang)
- Supports asynchronous IO



# Creating a Socket

- `#include <sys/socket.h>`
- Use the `socket` system call
- The `socket` function takes three parameters
  - The protocol family (BlueTooth, NFC, Netlink, Internet, etc.)
  - The socket type (Stream, Datagram, Sequential Packet, Raw etc.)
    - Since Linux 2.6.27 this argument can also be a bitwise OR flag containing additional options (such as `SOCK_NONBLOCK`)
  - The protocol to use (TCP, UDP, ICMP, etc.)
    - Typically, there is no more than 1 protocol supporting the family and type specified, so you can omit this parameter by making it 0
    - If multiple protocols exist for the given family and type, specify it explicitly
      - Otherwise, the OS will choose the best-suited support protocol given the other two parameters



## Creating a Socket *(continued)*

- When a socket is created, all subsequent operations performed on that socket use the specified protocol
- This means you can use the same functions for all network operations, regardless of the protocol used (`write`, `send`, `read`, `recv`, `close` etc.)
- Example TCP Socket setup
  - `int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);`
  - `PF_INET` means **Internet Protocol Family**
  - `SOCK_STREAM` means give us a reliable, two-way, connection-based socket
  - `IPPROTO_TCP` to request TCP
  - `AF_INET` can also be used here instead of `PF_INET`, as they resolve to the same value. However, `AF_*` technically stands for **address family** and should be used for the `sockaddr` structure



# Socket Addresses



- The sockaddr structure is a generic structure that can represent any address structure for all supported protocols
  - IPv4, IPv6, Unix domain, etc.
- Socket functions use it to determine the actual address type
- Think of it as an abstract class or interface in other languages
- This allows network functions to have the same signature for all address types

```
struct sockaddr {
 unsigned short sa_family;
 char sa_data[14];
};
```

Socket functions use *sa\_family* to determine which address type they are about to operate on, and then cast it to the appropriate type

Actual address data stored here,  
but opaque without a cast to  
appropriate type



# IPv4 Socket Addresses

- The `sockaddr_in` structure is the address type specific to IPv4 addresses, which contain a 4-byte IP address and 2-byte port

```
struct sockaddr_in {
 short sin_family;
 unsigned short sin_port;
 struct in_addr sin_addr;
 sin_zero[8];
};

struct in_addr {
 unsigned long s_addr;
};
```

Set this to `AF_INET` so socket functions know what kind of address you are sending

Port, in network byte order (big endian). Use `htons` to convert this from your machine's endianness to network byte order (`htons` stands for *host to network short*, which is a no-op on big endian architectures)

Set this using:

- constants, such as `INADDR_ANY` (0.0.0.0) or `INADDR_LOOPBACK` (localhost)
- `inet_pton`, which converts human readable IPv4 and IPv6 address strings to binary format



# IPv6 Socket Addresses

- The `sockaddr_in6` structure is the address type specific to IPv6 addresses, which contain a 16-byte IP address and 2-byte port (among other things)

```
struct sockaddr_in6 {
 sa_family_t sin6_family;
 in_port_t sin6_port;
 uint32_t sin6_flowinfo;
 struct in6_addr sin6_addr;
 uint32_t sin6_scope_id;
};

struct in6_addr {
 unsigned char s6_addr[16];
};
```

Set this to `AF_INET6` so socket functions know what kind of address you are sending

*htons again*

*inet\_pton again*



# Clients and Servers

- Sockets can be used to create both clients and servers
  - In many protocols, clients are defined to be the entity that initiates the communications
- The API behavior of clients and servers varies based on whether the protocol uses connection-based sockets (`SOCK_STREAM` or `SOCK_SEQPACKET`)
- Connectionless (datagram) sockets:
  - Clients and servers both send/receive data to/from a specified address, possibly communicating with multiple peers using the same socket
- Connection-based sockets:
  - Clients **connect** their socket to an address and then send/receive data to that specific server only
  - Servers **accept** client connections, which creates a new socket for communication with only a single client
    - Servers often accept more than one client



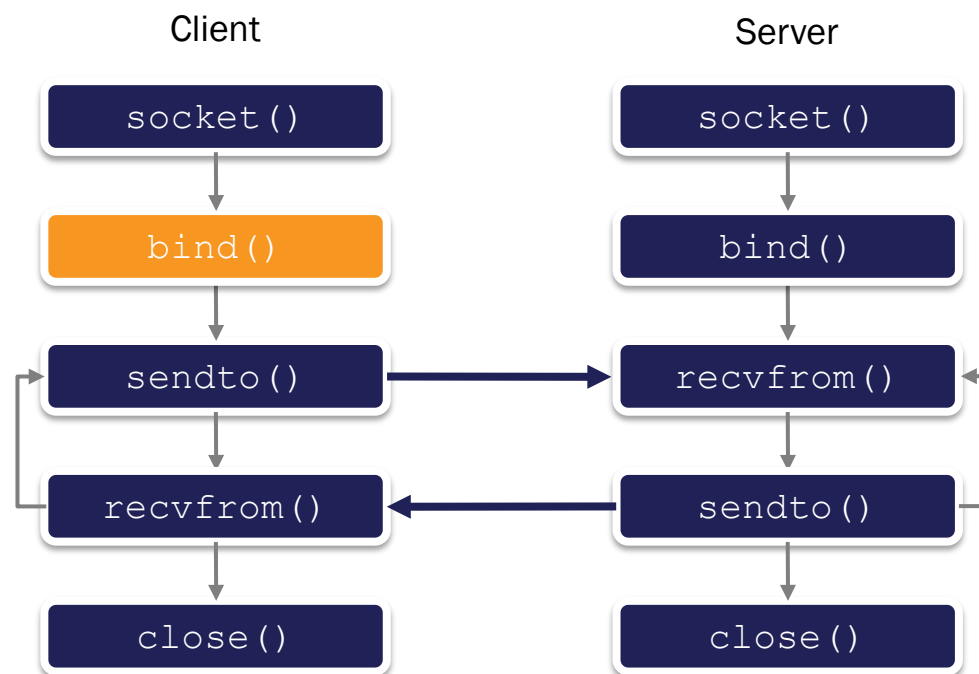
# Connectionless Interactions

- Servers need to call `bind`, which sets a socket's source address (which includes a port) to the one provided
  - This allows clients to send data to the server using its address
- Clients can also call `bind`, if they wish to set their own source address, but this is typically not necessary
- Sending data to a remote host is done with `sendto`
  - Takes a destination address as a parameter
- Receiving data is done with `recvfrom`
  - Receives a source address via an output parameter
- Close a socket using `close`





# Connectionless Interactions Illustrated



■ = optional

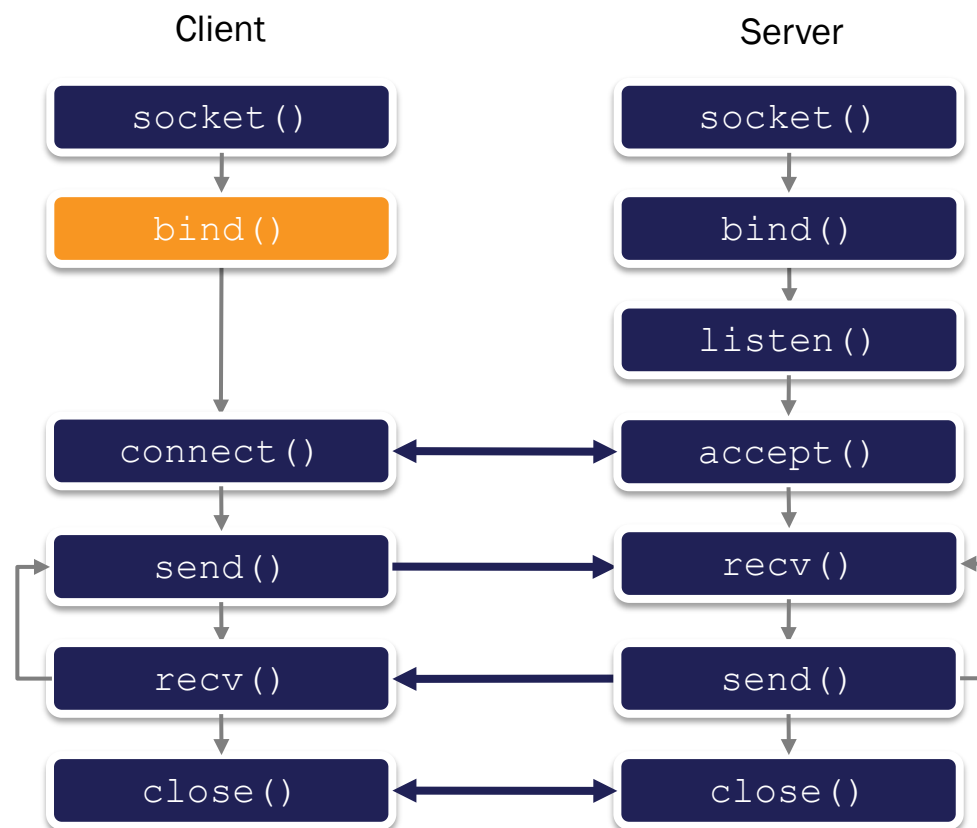


# Connection-based Interactions

- As with datagram sockets, `bind` needs to be called by servers to set a source address
- Unlike datagram sockets, servers also call `listen`, which turns the socket into a passive socket that contains a queue of incoming client connections
- Servers then call `accept`, which retrieves a client connection from the queue and returns a **new** socket to communicate with the client
- Clients call `connect` to specify the server the socket will be paired with for all subsequent operations
- When `close` is called, it also terminates the connection in addition to tearing down the socket (see also: `shutdown`)



# Connection Illustrated



## More on Connect

- Most connection-based protocol sockets can only call `connect` once
  - Under TCP, create a new socket if you want to connect repeatedly to a server or connect to a different one
- Connectionless clients can also call `connect` – this does not actually result in network activity, but will associate future send operations with the specified address
  - Call `connect` again to associate with another address, or with an `AF_UNSPEC` address to disassociate



## More on Accept / RecvFrom

- Using the POSIX socket API, you can use the same code to handle both IPv4 and IPv6
  - A socket is a socket is a socket
  - All network functions remain the same
- But functions like `accept` and `recvfrom` require a pointer to an address to fill out
- Do you allocate a `sockaddr_in` or a `sockaddr_in6`?
- Neither: use `sockaddr_storage`, which is big enough to fit IPv4 and IPv6 addresses (and others, depending on the implementation)



# Resolving Names

- `gethostbyname` was historically used to perform DNS lookups to obtain an address for use with `connect` and is now deprecated
- `getaddrinfo` now has that role, performing host lookup, reverse host lookup, and more
- `getaddrinfo` returns a linked list of `addrinfo` structures, which contain fields useful for `socket` and `bind` (source addresses) or `socket` and `connect` (destination addresses)
- This is essentially the Swiss army knife of name and address resolution



# getaddrinfo Example



```
1 struct addrinfo hints = { 0 };
2 struct addrinfo* ai_result;
3 struct addrinfo* ai;
4 int ret;
5 int sock;
6
7 hints.ai_family = AF_INET; /* IPv4 */
8 hints.ai_socktype = SOCK_STREAM;
9
10 ret = getaddrinfo(host, port, &hints, &ai_result);
11 if (ret != 0) {
12 fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
13 return -1;
14 }
15
16 for (ai = ai_result; ai != NULL; ai = ai->next) {
17 sock = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
18 if (sock == -1) {
19 perror("socket");
20 continue; /* try next result */
21 }
22 if (connect(sock, ai->ai_addr, ai->ai_addrlen) == -1) {
23 perror("connect");
24 close(sock);
25 continue; /* try next result */
26 }
27 break; /* connected! break out */
28 }
29 freeaddrinfo(ai_result);
30 if (ai == NULL) {
31 fprintf(stderr, "Could not find a suitable address\n");
32 return -1;
33 }
```

*"hints" is an input `addrinfo` structure that provides a set of filters for `getaddrinfo`. In this case we are asking for IPv4 addresses suitable for streaming sockets.*

*"host" is the hostname or IP address string we look up, "port" is a port number or service name, in string format (e.g., "80" or "HTTP")*

*We loop through the list of addresses returned by `getaddrinfo`, attempting to create a socket and connect to the returned address until success*

*Free up allocated memory with `freeaddrinfo`*



# Unix Domain Sockets

- Linux supports an inter-process communication method via the socket interface called Unix Domain Sockets
  - Specify `PF_UNIX` in `socket` call
- This allows different processes on the same machine to speak to one another through sockets
  - Very efficient and light-weight
- Processes can also pass ownership of file descriptors to one another through Unix Domain Sockets
  - Useful for privilege separation and more advanced multi-process architectures







## Lab 7

Basic TCP Client and Server (Optional)

# Tasks



- Use `ncat` to host a simple server to which your client will connect
  - `ncat -l -k 8080`
  - `-l` for listening socket, `-k` for keep alive, `8080` for port
- Use the socket API to connect to the server and send a newline-delimited greeting message
  - Use `getaddrinfo` to perform host lookup (for this lab you will be using “localhost” as the hostname)
- Manually send a message back from the server using `stdin`
- Receive a newline-delimited response from the server in your client program and print it out
- Make your client be able to repeat steps 2 – 5 automatically



# Tasks

1. Use the socket API to create a TCP server
2. Use `getaddrinfo` to find a suitable address for binding
3. Listen for clients on port 8080
4. Receive a newline-delimited message from a client and respond with your own message
5. Repeat step 4 until the client disconnects
6. Handle multiple simultaneous clients using `fork`
7. Create a signal handler for `SIGCHLD` to detect when a child process finishes and reap it
  - You can use `ps -ef | grep <progname>` to see if your handler is working (see how many instances of your server exist)





## Lab 8

UDP (IPv6 | IPv4) Client & Server

# Tasks

1. Use the socket API to create a client and server that use UDP over IPv6
  2. The server can simply echo back everything that the client sends it
  3. Use the manpages to figure out how to make a single UDP IPv6 socket accept messages from both IPv6 and IPv4 clients
- ncat can be made to use UDP if you pass it the -u flag





## More Advanced API Usage

- The POSIX API also supports a more advanced set of sending and receiving functions: `sendmsg` and `recvmsg`
- These functions require more setup, but a great deal of flexibility and power
- Rather than providing a simple, contiguous buffer for data to be send or received, these functions allow us to send and receive data from multiple buffers at different memory addresses
  - This is useful for sending protocol data packets directly from your in-memory representations of their packet structure (e.g., sending HTTP headers over TCP)
  - This is also useful for receiving packet data directly into your in-memory representation for that packet structure (e.g, reception of DNS records over UDP)
  - Additional controls for buffering and other behavior can be set with `sendmsg` and `recvmsg` as well. See the man pages for details



# Msghdrs and Iovecs

- sendmsg and recvmsg both operate on msghdr structures

*If receiving, msg\_name is a pointer to the address of the sender. If sending, this is a pointer to the address of the recipient*

*Note: msg\_iovlen is the number of buffers being sent from/received into – not their total lengths*

*msg\_control can be used to send additional data (like file descriptor ownership). See man cmsg for more details.*

```

struct iovec {
 void *iov_base;
 size_t iov_len;
};

struct msghdr {
 void *msg_name;
 socklen_t msg_namelen;
 struct iovec *msg_iov;
 size_t msg_iovlen;
 void *msg_control;
 size_t msg_controllen;
 int msg_flags;
};

```

*/\* Scatter/gather array items \*/*  
*/\* Starting address \*/*  
*/\* Number of bytes to transfer \*/*  
*/\* optional address \*/*  
*/\* size of address \*/*  
*/\* scatter/gather array \*/*  
*/\* # elements in msg\_iov \*/*  
*/\* ancillary data, see below \*/*  
*/\* ancillary data buffer len \*/*  
*/\* flags on received message \*/*

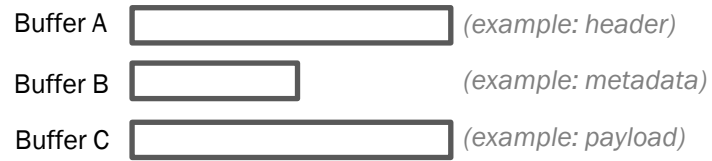
*iov\_base points to the beginning of a single buffer to be sent/received*

*iov\_len is the number of bytes to send/recv from that specific buffer*



# Example

- Before recvmsg (Scatter Input)



Incoming network bytes



```
struct iovec myiovs[3];
myiovs[0].iov_base = buffer_a;
myiovs[0].iov_len = 32;
myiovs[1].iov_base = buffer_b;
myiovs[1].iov_len = 16;
myiovs[2].iov_base = buffer_c;
myiovs[2].iov_len = 32;
```

- After recvmsg (Scatter Input)



Recvmsg places a set of incoming network bytes into multiple buffers specified by the user.

Sendmsg can do the same thing in reverse. It will take many source buffers and “gather” them into a single contiguous payload that will be sent over the network.







## Lab 9

Unix Sockets and File Descriptor  
Passing

# Tasks



1. Create client program that uses `sendmsg` to send a message to another process using Unix domain sockets (`PF_UNIX`)
  - `sendmsg` has a more advanced (but also more powerful) interface for sending messages, so reference the manpages carefully see how to use it
2. You may use `SOCK_DGRAM` or `SOCK_STREAM`
3. Create a server program that uses `recvmsg` to receive a message from the other process
  - `recvmsg`, like `sendmsg`, may require some manpage reading to use correctly
4. Use `man cmsg` to see how to use your existing client and server to attach a file descriptor to your messages as ancillary data
5. Have the client pass its standard output to the server, and have the server write to that descriptor



# LINUX CNO Programming



## Threads



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Understand the POSIX threading library and use the Linux Native Threading Library
- Pass arbitrary arguments to threads to limit shared memory
- Programmatically create threads
- Programmatically send signals to threads
- Programmatically stop threads and obtain information from them



# POSIX Threads

- POSIX Threads (called pthreads) is a standard threading API supported by many operating systems (including Linux)
  - Note that pthreads is an API, not an implementation
- The Linux threading implementation is known as the Native POSIX Thread Library (NPTL)
  - Successor to LinuxThreads, a non-POSIX threading implementation
  - Supports kernel-aware threading (threads are schedulable by the kernel)
- Extremely light-weight and efficient
- `#include <pthread.h>`
- Link using `-lpthread`



# Shared Memory

- Threads share memory, but have their own execution stack
- This means from one thread you can access
  - Global variables
  - Heap variables
  - Stack locations in another thread
    - Typically avoid this
- When two or more threads attempt to access the same memory problems can arise
- We will learn how to protect programs from these issues later



# Thread Scheduling

- Where and how threads are scheduled to run is implementation-defined
- With pthreads under NPTL, threads are scheduled by the kernel and can be split across cores/cpus by default
- Under pthreads, the programmer does not define when threads should be run or how they should be interleaved
  - This is nice, because it allows the system to make these decisions based on run-time knowledge
- Threads can be suspended by the system when they
  - Perform a system call
  - Voluntarily give up the processor by sleeping
  - Are just executing normally



# POSIX Threading API



- `pthread_create`
- `pthread_join`
- `pthread_detach`
- `pthread_exit`
- `pthread_self`
- `pthread_setschedparam` = priority
- `pthread_kill`
- See `man 7 pthreads` for more





# Creating a Thread

- ```
int pthread_create(  
    pthread_t* thread,  
    const pthread_attr_t* attr,  
    void * (*start_routine) (void*),  
    void* arg);
```
- Example start routine declaration:
 - ```
void* thread_func(void* arg);
```
- The calling thread receives the ID of the new thread through the `thread` output parameter
- The new thread begins execution at `start_routine`, which is passed `arg` as its argument
- The `start_routine` can return a pointer on return (or `pthread_exit`) that the parent can obtain with `pthread_join`



# Passing Arguments to Threads

- What if you want to pass multiple arguments?
- Standard practice is to create a structure that contains all your arguments, and then use a pointer to that structure as the parameter
- Inside the thread function, cast `arg` to the type of your structure

```
struct thread_args {
 int id;
 char name[32];
 char
color[32];
}
```



# Passing Arguments to Threads *(continued)*

```
15 int main() {
16 int ret;
17 pthread_t tid;
18 struct thread_args targs = { .id = 0 };
19 strcpy(targs.name, "sammy");
20 strcpy(targs.color, "blue");
21 ret = pthread_create(&tid, NULL, thread_func, &targs);
22 if (ret != 0) {
23 fprintf(stderr, "pthread_create: %s\n", strerror(ret));
24 exit(EXIT_FAILURE);
25 }
26 pthread_join(tid, NULL);
27 return EXIT_SUCCESS;
28 }
29
30 void* thread_func(void* arg) {
31 struct thread_args* targs = (struct thread_args*)arg;
32 printf("Thread ID: %d\n", targs->id);
33 printf("\tHi! My name is %s\n", targs->name);
34 printf("\tMy favorite color is %s\n", targs->color);
35 return NULL;
36 }
```

Set up parameters

Create thread

Wait for the thread to finish

Cast arg to use  
parameters



# Passing Arguments to Threads *(continued)*

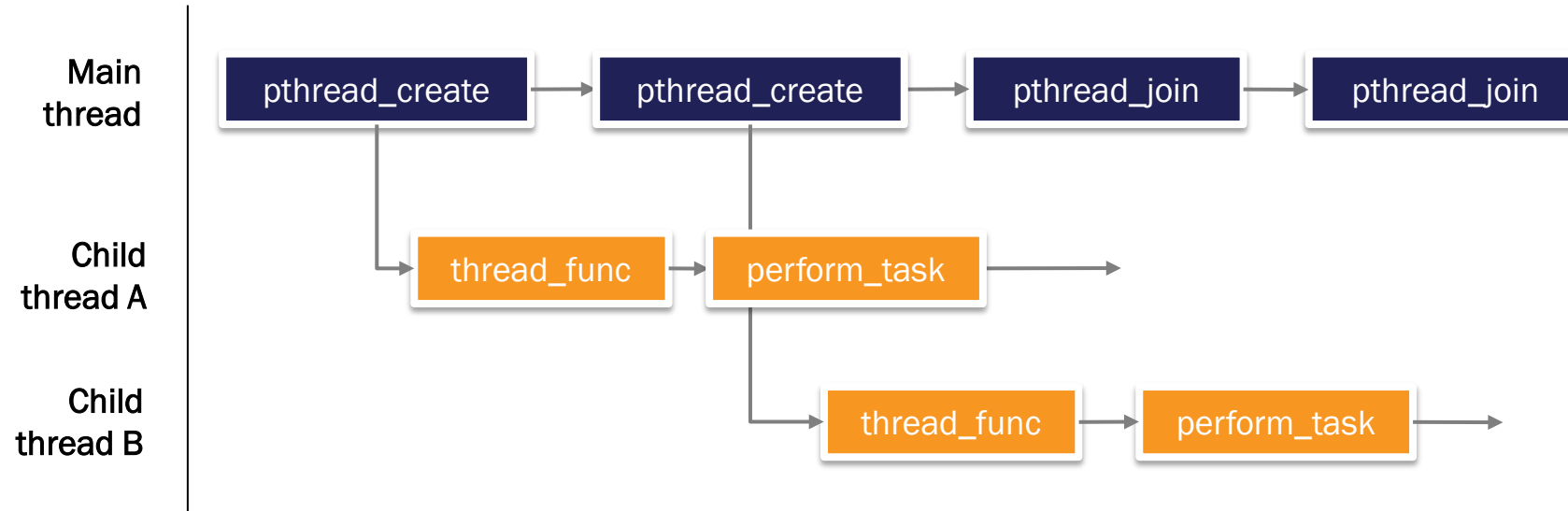
- Output from previous slide

```
student@student-virtual-machine:~/linux/lab_06$./demo
Thread ID 0:
 Hi! My name is sammy
 My favorite color is blue
```

- In the previous slide, is it safe to store the thread parameters on the main thread's stack? Why or why not?



# Thread Creation and Destruction



- Which child thread will finish first?
- If both child threads finish before `pthread_join` is called by the main thread, what happens?
  - See man pages



# Thread Internals

- Each thread in a process shares
  - The process ID
  - Parent process
  - open file descriptors
  - Current directory and root directory
- Each thread in a process has its own
  - `errno` value
  - Signal mask (use `pthread_sigmask`)
  - Thread ID (from `pthread_t` set during `pthread_create`)
  - CPU affinity
  - Capabilities (we will talk about this in Linux internals)



# Threads and Signals

- Signals sent to a process are received by a single thread
  - If the signal is a result of a hardware fault or other thread-caused event, the signal is sent to the thread that caused it
  - Otherwise, the signal is sent to an arbitrary thread that has the signal unblocked
    - Under Linux this tends to be the main/parent thread
- When using signal handlers while multithreading you have some options
  - Block the signal in all but one thread to guarantee where the signal will be handled
  - Mirror the signal to other threads in the program using `pthread_kill`





## Lab 10

Threads



# Tasks



- Create a multithreaded program that creates a number of threads specified with a command line argument
- Pass multiple arguments to threads from the main thread using pointers to structures holding 3+ arguments
  - One of these should be a sequentially-assigned thread ID
  - Feel free to use the `struct` defined in the earlier slides
- The thread function should
  - Print out when it starts and its thread ID
  - Increment a global counter by one, and loop to do this 100 times
  - Print the new counter value
  - Exit



# Tasks



- The parent thread should
  - Create parameters for all its children threads
  - Create child threads
  - Wait for all of its child threads to finish
  - Print the final value of the global counter
    - Is the final counter value always the expected value? Why or why not?



# LINUX CNO Programming



## Synchronization



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

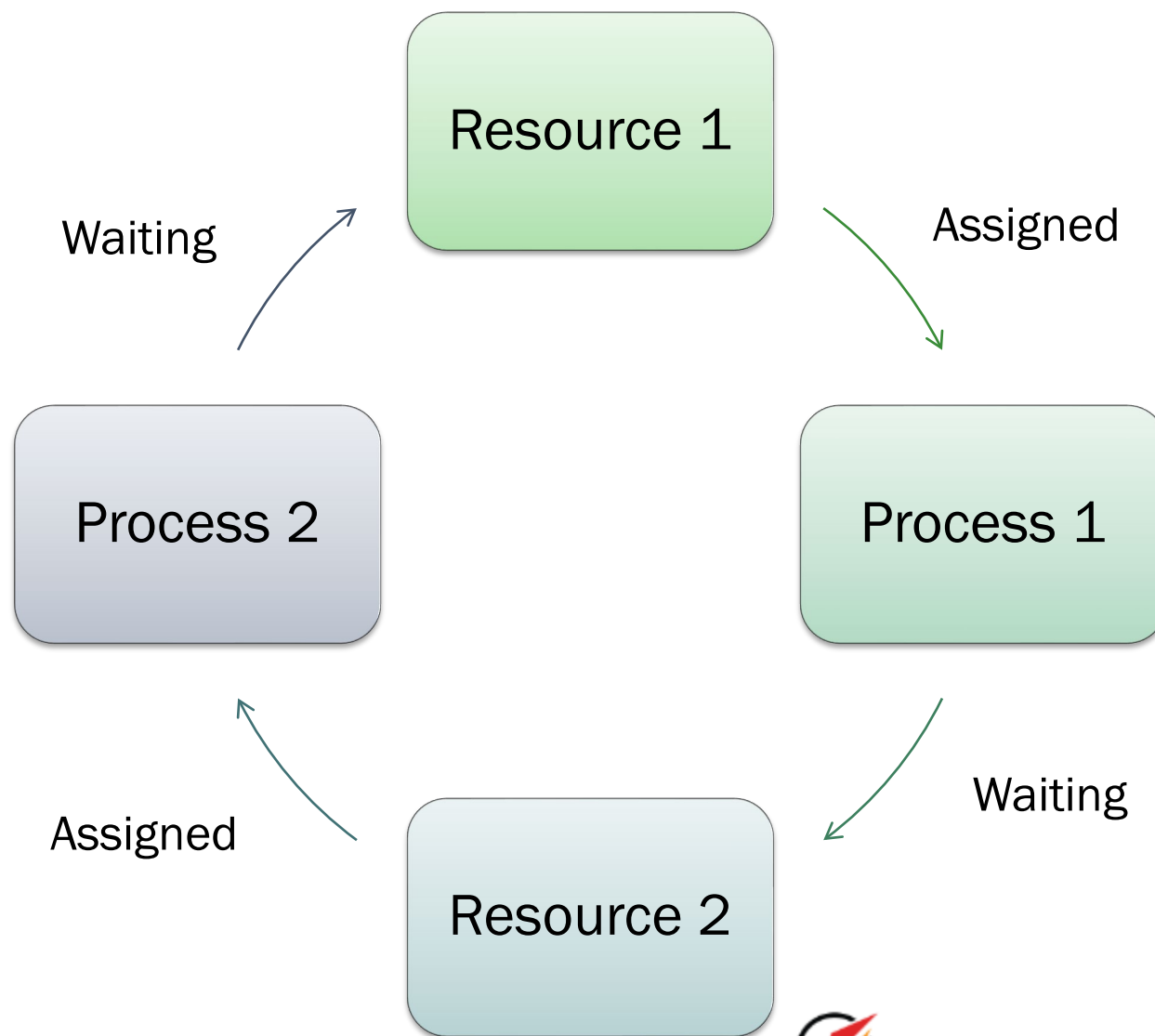
- Identify critical sections in code and note race conditions
- Use various synchronization primitives to protect critical sections and enforce proper thread ordering and consumption of resources:
  - Mutexes
  - Semaphores
  - Condition variables
  - Barriers
  - Spinlocks
  - Compiler atomics
- Solve coordination problems like the producer consumer problem

# No Resources?

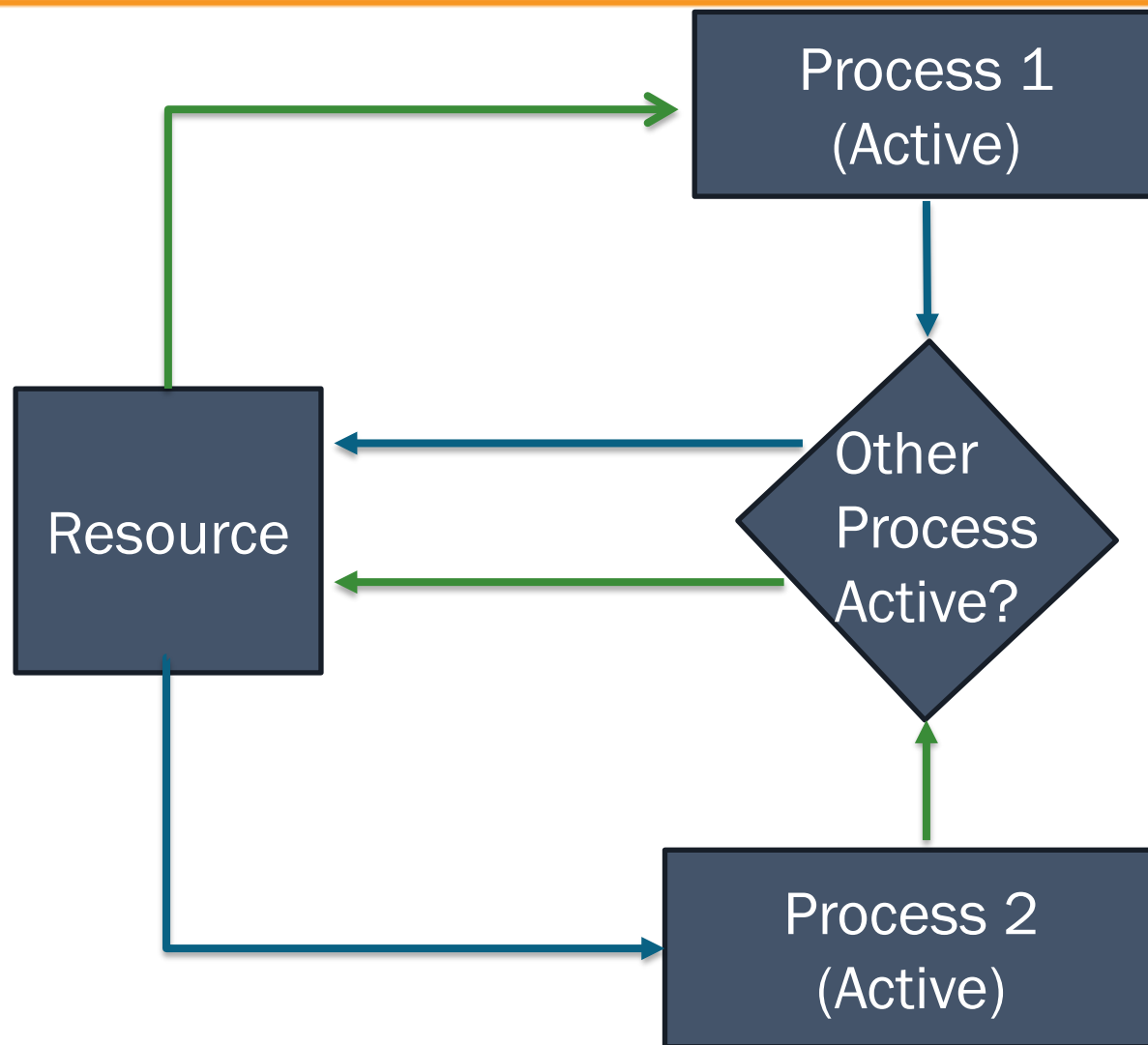
- 3 Scenarios:
  - Deadlock
    - processes block each other due to resource acquisition
    - none of the processes make progress waiting on the resource being held
  - Livelock
    - the states of the processes constantly change
    - the processes still depend on each other and can never finish their tasks
  - Starvation
    - process is unable to gain regular access to the shared resources it requires
    - unable to make any progress



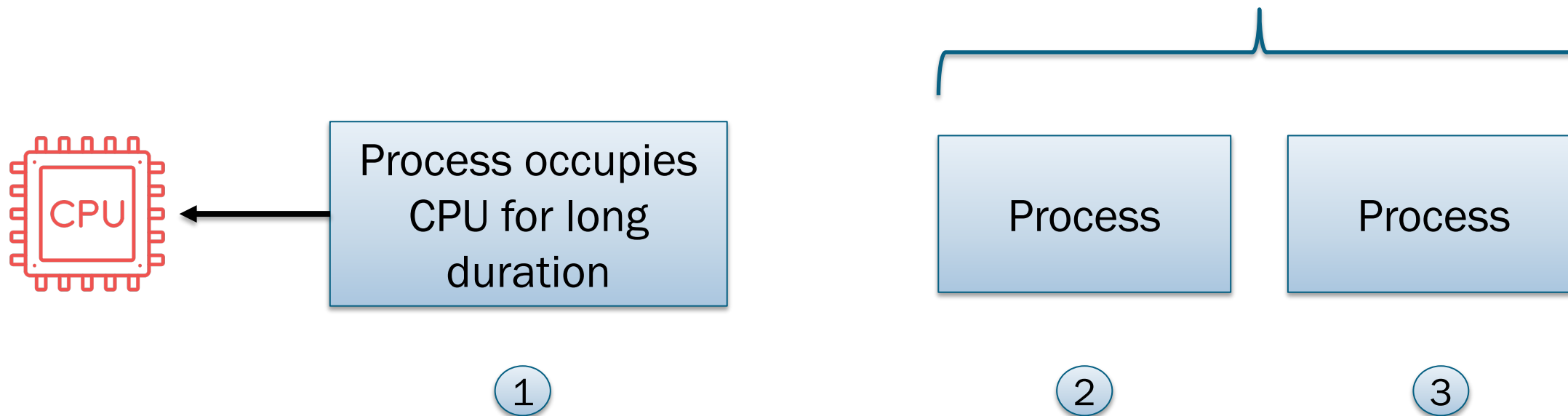
# Deadlock



# Livelock



# Starvation





# Synchronization Overview

- Synchronization refers to methods that manage process/thread scheduling, to protect program integrity
- Examples of this include
  - Limiting the number of threads rendering video at a time
  - Threads waiting for a queue to be nonempty before dequeuing
  - Only allowing one thread to access a global variable at a time
- Operating systems and language runtimes provide synchronization primitives, which allow programmers to implement synchronization correctly and efficiently



# List of Basic Synchronization Primitives

- Mutex – `pthread_mutex_t`
- Spinlock – `pthread_spinlock_t`
- Semaphore – `sem_t`
- Condition Variable – `pthread_cond_t`
- Barrier – `pthread_barrier_t`
- GCC Atomic Built-ins



# The Volatile Keyword

- variables shared between threads should be declared: **volatile**. This tells the compiler that:
  - the variable can change value at any time
  - it should not cache the value in a register for optimization
  - it should directly read the variable's value from memory every time it is accessed
  - it should store updates to the variable's value in memory after modification
- Example:

```
volatile int g_var = 0;
```



# Terminology

- **Race Condition:** In a threading context, a condition where the output of a program is dependent on thread scheduling or timing
  - Example: Multiple threads attempting to modify global value, as in the previous lab
- **Critical Section:** A region of code that is protected with synchronization primitives so that only one thread can be executing it at a time

*Note: Windows has an object called `CRITICAL_SECTION` that is used to protect critical sections. Don't confuse the hammer with the nail, like Microsoft did*



# Mutexes and Mutices

- A **mutex** (amalgam of mutual exclusion) allows programmers to define and protect critical sections
- A thread can request a “lock” on a mutex
  - If the mutex lock is already being held by another thread, the calling thread will be put to sleep
  - If the mutex lock is not being held by another thread, the calling thread obtains the lock and is allowed to continue executing
- A thread put to sleep waiting on a mutex lock will be woken up and given the lock when it becomes available
  - If multiple threads are waiting for the lock, it is up to the system which thread obtains the lock when it becomes available
- Mutexes support the idea of lock ownership – the thread that holds a lock must be the thread that unlocks it



## Mutexes *(continued)*

- `#include <pthread.h>`
- `pthread_mutex_init` or `PTHREAD_MUTEX_INITIALIZER`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_mutex_destroy`
- Link with `-pthread`





## Lab 11

Mutexes

# Tasks



- Use a mutex to protect the shared counter variable from the previous lab
- Run your new protected version a few times and compare it to the output of your previous one





# Spinlocks

- Spinlocks are like a mutex, but they do not put the calling thread to sleep when the requested lock is being held by another thread
  - Useful for when you **know** the lock will be unlocked soon
  - Rule of thumb: use this only when the cost of blocking is greater than the cost of running a tight loop (`while (1) {}`) for a while. Use a mutex otherwise.
- `pthread_spin_init`
- `pthread_spin_lock`
- `pthread_spin_trylock`
- `pthread_spin_unlock`
- `pthread_spin_destroy`



# Semaphores



- Semaphores are used to control and coordinate access to shared resources
- Semaphores internally contain a counter and a queue of waiting threads
  - Upon creation, the counter is initialized to a given number
  - Each time a thread wants to perform an action that accesses a protected resource, it pends on the semaphore
- The semaphore checks its counter value
- If the counter value is  $> 0$ , it is decremented and the thread continues execution
- If the counter value is 0, the thread is put to sleep and added to the internal queue



# Semaphores *(continued)*

- When a thread is done using a resource, it can post to the semaphore
  - If there are threads waiting on the semaphore queue, the first thread in line is woken up and allowed to continue execution
  - If there are no threads waiting on the semaphore, its internal counter is incremented by one
- Semaphores are called binary semaphores if their max counter value is 1, otherwise they are counting semaphores
- A binary semaphore and mutex are very similar
  - They differ only in the concept of lock ownership
  - A mutex must be unlocked by the thread that locked it, but a semaphore can be posted to by any thread



# Semaphores *(continued)*

- Semaphore example situations:
  - You have 100 threads that each need to open a file during their lifetime, but you can only have 10 files open at a time
  - 20 worker threads continually fetch a client socket from a queue that is filled by a main thread. The worker threads should only attempt to dequeue when the queue is not empty
- `#include <semaphore.h>`
- `sem_init` and `sem_open`
- `sem_wait`
- `sem_post`
- `sem_destroy` and `sem_unlink`
- Link with `-pthread`





## Lab 12

Semaphores

# Tasks



- Modify the previous lab so that threads wait on a semaphore immediately after creation
- When the user presses a key on the keyboard, a single thread should be allowed to finish execution
- If the user presses 'q', the program should signal all child threads to terminate, wait on them, and then quit
- Experiment with different initial value of the semaphore



# Condition Variables

- Condition variables allow threads to safely check for (and signal the occurrence of) a programmer-defined event
- Condition variables are used in conjunction with a mutex
- If a thread wants to wait for a particular condition to be met before continuing, it can
  - Lock the mutex
  - Check for the condition (e.g., `client_is_connected == 1`)
  - If the condition is not met, wait on the condition variable
- If some other thread wants to set the condition it can
  - Lock the mutex
  - Update the condition (`client_is_connected = 1`)
  - Set the condition variable to wake up waiting threads





## Condition Variables *(continued)*

- There are two ways of signaling that a condition has been met
  - Signal – unblock *at least* one waiting thread\*
  - Broadcast – unblock all waiting threads
- Since threads may be woken up before a condition has been met, proper practice is to wait on the condition in a loop
  - Example

*while, instead of  
if, in case we get  
woken up early*

```
pthread_mutex_lock(&m);
while (g_red_start == 0) {
 pthread_cond_wait(&red_cond, &m);
}
pthread_mutex_unlock(&m);
```

*Developer-defined condition*

*pthread\_cond\_wait  
will unlock mutex m if we  
have to sleep*

*\* While a signal is intended to wake up only one waiting thread, this is sometimes impossible on multiprocessor systems. This is called a spurious wakeup. See "Multiple Awakenings by Condition Signal" in man 3 pthread\_cond\_broadcast*





## Condition Variables *(continued)*

- `pthread_cond_init` or `PTHREAD_COND_INITIALIZER`
- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- `pthread_cond_destroy`





## Lab 13

Condition Variables

# Tasks

- Extend the previous lab, using a condition variable and mutex, so that all **red** threads execute completely when the user presses the 'r' key
- All other functionality should be the same



# Barriers



- Barriers allow a group of threads to all reach a certain point of execution before continuing
- When initializing a barrier, the programmer sets the number of threads that need to reach the barrier before any of the threads is allowed to proceed past it
- Each thread that needs to wait at the barrier calls `pthread_barrier_wait`
  - The thread sleeps if the number of threads waiting at the barrier is less than the threshold set during initialization
  - When the specified number of threads reach the barrier, all waiting threads are woken up and allowed to proceed
- `pthread_barrier_init`
- `pthread_barrier_wait`
- `pthread_barrier_destroy`



# GCC Atomic Built-ins

- GCC supports some built-in functions that translate to atomic operations
- These do not require special constructs from the OS, unlike a mutex, which makes them more light-weight
- `__atomic_fetch_add(&global,1, __ATOMIC_SEQ_CST);`
- When compiled, the above becomes

```
lock addl $0x1,0x2f0e(%rip)
```

*The lock instruction prefix provides mutual exclusion at the CPU-level for the suffixed instruction. See intel docs for details.*

- Numerous other atomic functions are available
  - `__atomic_fetch_sub`
  - `__atomic_fetch_xor`
  - `__atomic_add_fetch`
  - `__atomic_store_n`
  - `__atomic_load_n`
  - See GCC manual handout for a complete list
- `__atomic_fetch_add` is similar to `InterlockedIncrement` on Windows



# Thread Safety

- Thread safety refers to the behavior of a function or object when executed concurrently by multiple threads
  - Thread safe elements can be used by multiple threads without fear of race conditions
  - Thread unsafe elements need to be protected before use in multi-thread environments
- Some things are thread safe by default
  - For example, if no globals or other shared memory are used
  - Is errno thread safe?
- Others must be protected
  - Example: wrapping a queue's enqueue and dequeue functions in a mutex lock and unlock can make it thread safe



# Thread Safety *(continued)*

- If you are unsure of a built-in function's thread safety, check the manpages
  - Many manpages contain information about thread safety in the attributes section
  - `man 3 strtok`

## ATTRIBUTES

For an explanation of the terms used in this section, see `attributes(7)`.

| Interface               | Attribute     | Value                 |
|-------------------------|---------------|-----------------------|
| <code>strtok()</code>   | Thread safety | MT-Unsafe race:strtok |
| <code>strtok_r()</code> | Thread safety | MT-Safe               |

Thread Unsafe

Thread Safe







## Lab 14

Producer Consumer Problem



# Tasks



- You will be modifying your TCP server lab to handle multiple simultaneous clients via threading
- The main thread will `accept` new client connections and place the new socket descriptors in a global queue (provided)
- Worker threads should dequeue a client socket descriptor from the queue, send messages back and forth with that client until the client disconnects, and then repeat
- This scenario is an example of the producer consumer problem
  - The main thread “produces” client connections
  - Worker threads “consume” client connections



## Tasks *(continued)*



- Apply good programming practice to your lab:
  - Thread pool – To bound resource usage, only create a given number of worker threads (specified at the command line). Do this at program startup, before any clients connect
  - No spinning – No thread should ever busy wait. If no work can be done, then block



## Tasks *(continued)*



- Use synchronization primitives to coordinate the efforts of the main thread and worker threads
- Protect the shared connection queue
- Do not allow the queue to have more than 10 elements at a time
- Do not allow worker threads to dequeue from an empty queue



# Hints

---

- Use a mutex to protect accesses to the queue
- Use a condition variable **or** a semaphore to limit the amount of elements in the queue
- Use a condition variable **or** a semaphore to prevent workers from attempting to dequeue from an empty queue



# LINUX CNO Programming



## I/O Multiplexing

Poll Dancing



# Learning Objective

Given a workstation, device, and/or technical documentation, the student will be able to:

- Perform I/O operations on multiple resources concurrently with a single thread
- Use various I/O multiplexing constructs such as select, poll, and epoll
- Create a scalable network server capable of handling thousands of concurrent with a single thread

# Scalability Problems

- Forking and threading allow a program to do multiple things at the same time
- But blocking limits efficiency (waiting on user, file, sockets)
  - What happens if you have eight threads for a webserver, one for each core, and eight clients connect to it?
  - What happens when every thread is blocked, waiting for a request from a client
  - What happens if a ninth client connects and sends a request?
- You could fork or add a thread for each new client, but this isn't very scalable
  - Increased context switching
  - Difficult to bound resource usage



## Solution: I/O Multiplexing

- I/O multiplexing allows a **single** process or thread to handle I/O operations on **multiple** resources concurrently
- Rather than giving up the CPU when waiting for something to happen, perform some other I/O that is ready to be done instead
  - Back to webserver example: have a single thread service multiple client sockets
  - While waiting on eight clients to send requests, handle the ninth client's request
- This alleviates wasted time due to context switches (between threads/processes) and blocking





# Nonblocking sockets

- By default, sockets in Linux are blocking, which means that they do not return until the operating system has handled their associated request (or a signal interrupts)
  - If network data have not arrived yet, a call to `recv` will put the process to sleep until some data have arrived
  - If the OS send buffer is full, a call to `send` will put the process to sleep until it is no longer full
- You can change this behavior using `fcntl`
  - You can also bitwise OR the value `SOCK_NONBLOCK` in the `type` argument to `socket`, but this is less portable



# Making a socket nonblocking

- A single socket can be made blocking and nonblocking numerous times throughout its lifetime

```
int set_nonblocking(int sock) {
 int flags;
 /* Get flags for socket */
 if ((flags = fcntl(sock, F_GETFL)) == -1) {
 perror("fcntl get");
 return -1;
 }
 if (fcntl(sock, F_SETFL, flags | O_NONBLOCK) == -1) {
 perror("fcntl set nonblock");
 return -1;
 }
 return 0;
}
```

- Make a socket blocking by removing the O\_NONBLOCK flag



## Nonblocking sockets *(continued)*

- A nonblocking socket behaves just like a normal one, except that if an operation cannot currently be performed by the operating system then it will return immediately
  - If network data have not arrived yet, a call to `recv` will return -1 and `errno` will be set to `EAGAIN` or `EWOULDBLOCK`
  - If the OS send buffer is full, a call to `send` will return -1 and `errno` will be set to `EAGAIN` or `EWOULDBLOCK`
- This allows the program to continue executing, performing other tasks (such as handling data from a different client) while it waits for the OS to be able to service its request



# Using Nonblocking Sockets

- What is wrong with the following?

```
int got_data = 0;
int sock = socket(PF_INET, SOCK_STREAM, 0);
...
set_nonblocking(sock);
...
while (got_data == 0) {
 ret = recv(sock, buffer, bufsize, 0);
 if (ret == -1) {
 if (errno == EAGAIN || errno == EWOULDBLOCK) continue;
 }
 got_data = 1;
 ...
}
```

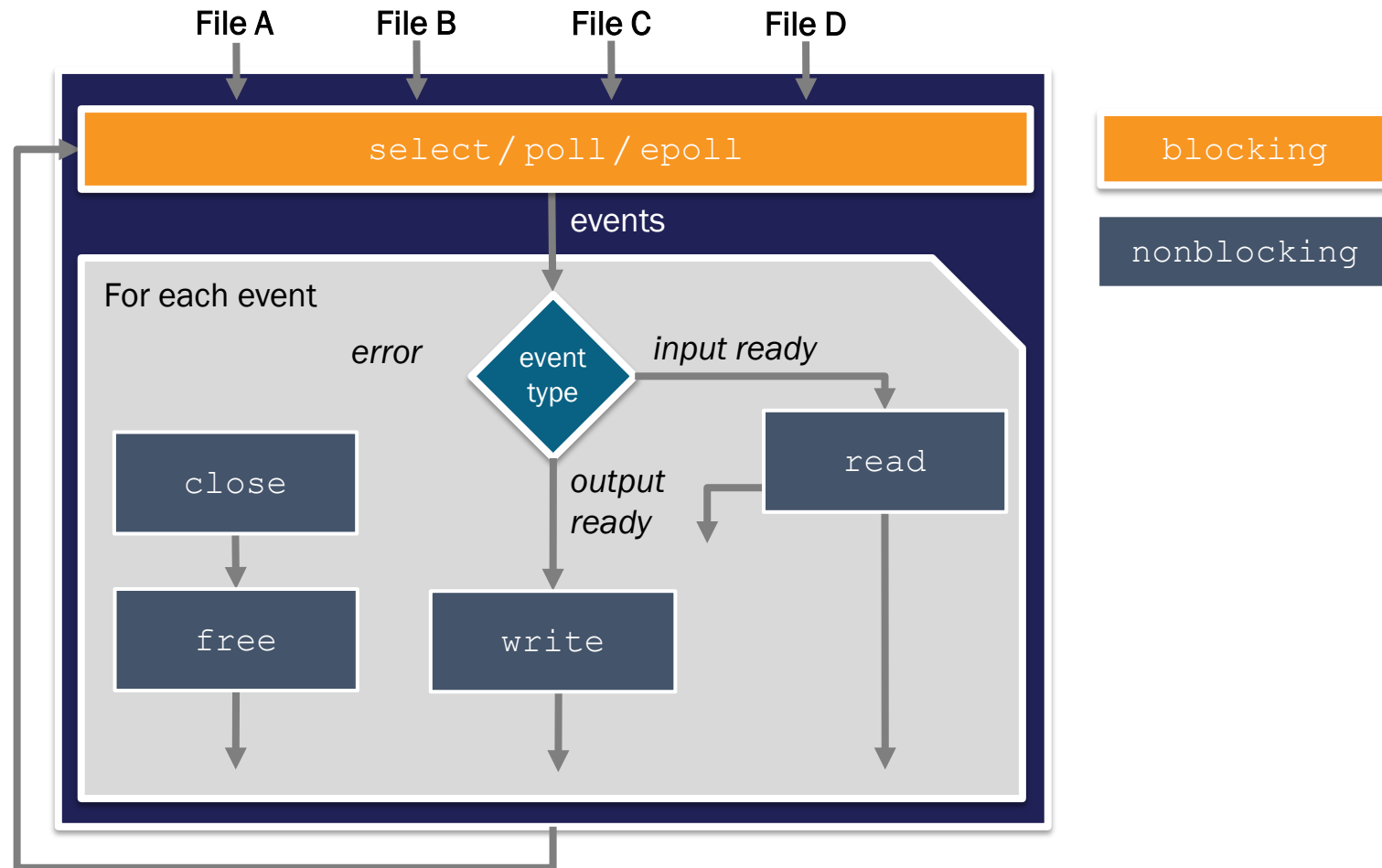


# Event Loops

- It is still useful to block, as waiting allows other processes to run and doesn't waste CPU cycles
- The key is to block **only** when you have **nothing else to do**
- Event loops make this easy
- Event loops
  1. Wait on a set of file descriptors for some event (ready for reading, ready for writing, disconnect, error, etc.)
  2. Wake up after the OS indicates that some events have occurred on one **or more** of the files, or after a timeout
  3. Loop through the file descriptors that have associated events, handling each of them appropriately with **nonblocking calls**
  4. Go back to step 1



# Event Loop Structure Example



# Polling Mechanisms

- Numerous mechanisms can serve as the blocking element of the event loop
- `select`
  - This function is the simplest, and most cross-platform (even Windows supports it)
  - Handles descriptors in a linear fashion, so it gets slower the more descriptors you monitor
  - Only supports 1024 file descriptors without workarounds
- `poll`
  - Also linear handling of descriptors
  - Supports more than 1024 file descriptors by default



# Polling Mechanisms *(continued)*

- `epoll`
  - The most advanced option on Linux
  - Constant time access to all events that occur
  - Extremely flexible, supports virtually unlimited file descriptors and lots of event types
  - Most complex to use
- Note: Libevent is a cross-platform, easy-to-use wrapper around `select`, `poll`, `epoll`, and `kqueue` with many additional features
- See more about the benefits and drawbacks of each approach, as well as other architectures using threads: <http://www.kegel.com/c10k.html>
- In the real world, most modern servers use event-driven, non-blocking I/O with a multithread/multiprocess architecture
  - This takes efficient advantage of all available cores





# Using Select

- To use `select`, you first have to create a bitmask that represents the file descriptors you are interested in
- The `fd_set` type holds this bitmask
  - Initialize this with `FD_ZERO`
  - Set the descriptors you are interested in using `FD_SET`
- You can tell `select` which actions you are interested for your descriptors using its parameters (any of these can be `NULL`)
  - 2<sup>nd</sup> parameter – `fd_set` for read events
  - 3<sup>rd</sup> parameter – `fd_set` for write events
  - 4<sup>th</sup> parameter – `fd_set` for error events
  - 5<sup>th</sup> parameter – a timeout value (if no events occur during the specified duration then `select` will return)



## Using Select *(continued)*

- The `select` function itself actually modifies the `fd_sets` you pass in, so you must maintain a copy if you want to call `select` again (and in an event-loop, you do)
- After `select` returns, you have to check to see which file descriptors have been made ready
  - If you have a lot of file descriptors, iterate through the `fd_sets`, checking `FD_ISSET` as you go
    - File descriptors are just numbers, and assigned sequentially, so you can just loop through them with an integer
  - Or, if you only have a handful of descriptors, just check them individually using `FD_ISSET`
- If `FD_ISSET` returns true, then you can perform the associated action on that file (`read`, `write`, `recv`, `send`, etc.)



# Select Example

```
fd_set active_fd_set;
fd_set read_fd_set;
int socks[500]; /* 500 sockets */
... /* initialize and connect sockets */
FD_ZERO(&active_fd_set);
...
for (i = 0; i < 500; i++) {
 FD_SET(socks[i], &active_fd_set);
}
while (1) {
 read_fd_set = active_fd_set;
 if (select(FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) == -1) {
 perror("select");
 exit(EXIT_FAILURE);
 }
 for (i = 0; i < FD_SETSIZE; i++) {
 if (FD_ISSET(i, &read_fd_set)) {
 printf("We got data on socket %d\n", i);
 recv(i, buffer, bufsize, 0);
 }
 }
}
```

Zero out bitmask of files we are interested in

Assign mask for all of our 500 sockets

Copy our `fd_set` so we can loop and use it again

Wait for one or more of our sockets to be ready for reading (write and error sets are `NULL`)

Iterate the file descriptor set, check to see if any of our sockets have data to receive. If they do, call `recv`.



# Using epoll

- epoll is an advanced I/O multiplexing mechanism that facilitates event-driven programming
- You create an epoll with `epoll_create`, register various file descriptors with the epoll, and then call `epoll_wait`
- You can register a file descriptor with an epoll, and an associated set of events and data pointer
  - The `epoll_ctl` function can add and remove file descriptors from the epoll, as well as change the events you're interested in for a given file descriptor
- You can even register an epoll with an epoll for more advanced functionality



## Using epoll *(continued)*

- Numerous events can be associated with a file descriptor before it is registered with an epoll
  - EPOLLIN – available for reading
  - EPOLLOUT – available for writing
  - EPOLLRDHUP – peer shut down writing side of socket
  - \*EPOLLHUP – Hang up occurred
  - \*EPOLLERR – Error occurred
  - EPOLLET – Edge Triggering mode - **not an event**
  - And more (see `man 2 epoll_ctl`)
- Bitwise OR these together to wait on multiple event types

*\* These events are always waited for, for all registered file descriptors, so you do not need to add them manually*



## Using epoll *(continued)*

- `epoll_ctl` takes a file descriptor and an `epoll_event` structure as parameters
  - The `epoll_event` holds the set of events you want to associate with the file descriptor
  - The `epoll_event` also holds data of your choice
    - Either a pointer to some data, or integer

```
typedef union epoll_data {
 void *ptr;
 int fd;
 uint32_t u32;
 uint64_t u64;
} epoll_data_t;

struct epoll_event {
 uint32_t events; /* Epoll events */
 epoll_data_t data; /* User data variable */
};
```



# Edge Triggering vs Level Triggering

- By default, all events registered with an epoll are level-triggered
- This means that once the event condition is met, any call to `epoll_wait` will result in that event being set, until the condition is no longer true
  - For example, if you call `epoll_wait` for data to be available for reading, and you don't read any data, all future calls to `epoll_wait` will indicate data is available for reading, until you drain that data with a call to `recv/read`
  - This means you can opt to only partially handle data, and rest assured that you will receive another event notification later to handle whatever remains



# Edge Triggering vs Level Triggering

- Using the event bitfield `EPOLLET`, you can request that a given event be edge triggered
- This means that once the event condition is met, `epoll` will set the event once, and it will not be set again until the condition reoccurs
  - For example, if you call `epoll_wait` for data to be available for reading, and you don't read any data, calling `epoll_wait` again will not indicate that data is available
  - This means that you must call the corresponding I/O operation (`recv`, `send`, etc.) repeatedly, until it returns `EWOULDBLOCK` or `EAGAIN`, before calling `epoll_wait` again
- If any of this is confusing, stick with the default level-triggering for now





# epoll Example



```
int epoll_fd;
int nfds;
struct epoll_event ev;
struct epoll_event events[MAX_EVENTS];
int socks[500]; /* 500 sockets */
... /* initialize and connect sockets */
epoll_fd = epoll_create1(0);
...
for (i = 0; i < 500; i++) {
 ev.events = EPOLLIN | EPOLLOUT;
 ev.data.ptr = socks[i]; /* this can be any data */
 epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socks[i], &ev);
 ...
}
while (1) {
 nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
 ...
 for (n = 0; n < nfds; n++) {
 if (events[n].events & EPOLLIN) {
 printf("Data to read on socket %d!\n", events[n].data.ptr);
 recv(events[n].data.ptr, buffer, bufsize, 0);
 ...
 }
 ...
 }
}
```

← *epoll descriptor*

← *Register for input and output events for all of our 500 sockets*

← *In this case the data we associate with each event is just the socket descriptor*

← *Wait for conditions to be met on one or more of our sockets*

← *Handle each input event by printing and calling `recv`*





## Lab 15

I/O Multiplexing

## Description

---

- You will create a chat client that operates over TCP
- The chat server will broadcast all chats sent to all clients connected
- The chat client will simultaneously wait for input on both its TCP socket and standard input from the user
- When `stdin` becomes ready for reading the chat client will send the data to the server
- When the TCP socket becomes ready for reading the chat client will print out the data sent to `stdout`



# Tasks

- Take a username from `argv` and then prefix all content sent with that name
  - Example: `Bob: Hello, everyone!`
- Connect your client to port 8080 of the instructor's IP
- Use `select` with fd sets to continually wait on input from both your socket and standard input
- Do not worry about messages from others being printed out at the same time you are typing your own message and garbling up your terminal
  - Bonus: handle this using the `ncurses` library or other mechanism



# Tasks

- Now create the server portion of the chat server
- Use epoll and nonblocking sockets this time
- Feel free to use level or edge triggering
- You will need to maintain a list of clients that are currently connected so that you can broadcast received messages to all of them
  - Keep this list updated as new clients connect (EPOLLIN on your listening socket) and old ones disconnect (EPOLLHUP)



# LINUX CNO Programming



## Daemons and Services



# Daemons

- A daemon is a program that runs in the background to provide a service for an indefinite amount of time
- “In the background” means without a terminal
- Because they run in the background and do not have GUIs, interaction with a daemon is usually done using signals
  - Example: `SIGSTOP` (stop), `SIGCONT` (continue), `SIGHUP` (restart)
- Some programs have a “daemonize” option that makes the program run in the background
  - Typically this makes the program fork on startup, close standard file descriptors, exit the parent, and continue the child, adopted by init. Signal handlers for communication with the process may be installed too.
- Adding a `&` at the end of a shell command will run it in the background
- There’s even a `daemonize` command-line tool that can run a program as a Linux daemon (`man 1 daemonize`)



# Daemons as Services

- There are a variety of ways to start and schedule daemons in a Linux system
- We will talk about two:
  - Cronjobs
  - Systemd services





# Cronjobs



- Cronjobs are tasks which are run at some specified degree of frequency
- The tasks can be any command-line string
  - Example: `cd ~/jobs/ && ./do_daily param1 param2`
- Cronjobs and their scheduling are controlled by the crontab (cron table)
  - Each user has their own crontab
  - Running `crontab -e` will bring up the crontab for the current user in the default text editor
  - To add or modify a cronjob, just edit this file, save, and exit



## Cronjobs *(continued)*

- A crontab file uses # for single-line comments
- Each cronjob occupies a single line of the crontab
- The first five values of a cronjob dictate when it is run
  1. Minute (0 – 59)
  2. Hour (0 – 24)
  3. # of Day of Month (1 – 31)
  4. # of month (1 – 12)
  5. Day of week (0 – 6, 0 is Sunday)
- Any of these values can be an asterisk (\*) to represent all values
  - \* in the minute field means every minute



# Cronjob Examples

- `0 0 * * 1 ~/myjob.sh`
  - Run every Monday at midnight
- `1 1 1 1 * ~/myjob.sh`
  - Run every January 1<sup>st</sup> at 1:01 am
- `0 * * * * ~/myjob.sh`
  - Run every hour at the top of the hour
- `59 14 10 1 6 ~/myjob.sh`
  - Run every January 10<sup>th</sup> at 2:59 pm if it is a Saturday
- See `man 5 crontab` for more information



# Systemd services

- Systemd services are background tasks that boot up with the system
- Systemd starts these services (among other things), and hosts configuration files that you can use to create and manage them
- You can control services using `systemctl`
  - `systemctl start servicename` (start now)
  - `systemctl stop servicename` (stop now)
  - `systemctl enable servicename` (start on boot)
  - `systemctl disable servicename` (disable on boot)
- To create a service, create a `.service` file (e.g., `mysvr.service`)
- Place this file in the `/etc/systemd/system/` directory



# Systemd Service Options

- The `.service` file can have many different sections, each with their own set of options to set
- The `[Service]` option is the most important. Use it to
  - Specify the service type
  - Working directory
  - Executable (or script)
  - Restart behavior
- See `man systemd.service` for more info
- View the status of system services using `systemctl status`



# Systemd Service File Example



```
[Unit]
Description=My Service

[Service]
Type=simple
User=student
WorkingDirectory=/home/student
ExecStart=/home/student/my_srv param1 param2
Restart=on-failure

[Install]
WantedBy=multi-user.target
```



# LINUX CNO Programming



## Packaging

# Packaging Overview

- Linux-based operating systems install software in precompiled “packages”
- Packages are archives containing
  - Software
  - Configuration files
  - Information about dependencies
- Packages are handled by software called package managers, like
  - dpkg
  - yum
  - dnf
  - rpm





# Package Managers

- Common features of a package manager:
  - Package downloading (interestingly enough)
  - Dependency resolution
  - Standardization of software package format
  - Common installation/config locations
  - Updates/upgrades
- Without package managers, users would
  - Hunt down dependencies
  - Compile software from source code
  - Manage configuration
  - Stay on top of updates



# Package Managers – Fedora

- Dandified yum (dnf) commands
  - `dnf install package-name`
  - `dnf erase package-name`
  - `dnf check-update`
  - `dnf upgrade` (run this after update)
  - `dnf list`
  - `dnf history`
  - `dnf autoremove` (removes unwanted/orphaned packages)



## .rpm files

---

- RPM Package Manager (RPM)
- Software package format created for use in Red Hat Linux



# Package Managers – Debian

- Advanced Packaging Tool (APT) commands
  - `apt-get install package-name`
  - `apt-get remove package-name`
  - `apt-get clean`
    - Removes package files for already installed software
  - `apt-get update`
  - `apt-get upgrade` (run this after update)
- APT is front-end software for a powerful tool called `dpkg`
  - `dpkg --i package-file-name.deb`
    - Installs a .deb file
  - `dpkg --get-selections package-name`
  - `dpkg --configure package-name`
  - `dpkg-reconfigure package-name`



## .deb files

- Software package format for Debian and derivative distros
- Dpkg can be used to install and manipulate packages in .deb files
- Has two file archives
  - Control information archive (control.tar)
    - Package meta-information
    - Maintainer scripts
  - Installable data archive (data.tar)
    - Contains actual installable files



# .deb files

- Important vocabulary
  - Upstream tarball
    - Software written by someone else, the “upstream developer”
  - Source package (control archive)
    - Simplest version has
      - Upstream tarball
      - Debian directory with changes made to upstream source plus files required by debian packages
      - Description file (.dsc)
  - Binary package (data archive)
    - This is what is actually installed



## Further Reading

---

- Fedora
  - [https://fedoraproject.org/wiki/Package\\_management\\_system](https://fedoraproject.org/wiki/Package_management_system)
- Debian
  - <https://www.debian.org/doc/manuals/debian-reference/ch02.en.html>



# LINUX CNO Programming



## Series Conclusion



# Windows Analogs



| WINDOWS              | LINUX                      |
|----------------------|----------------------------|
| Win32 / MSVCRT       | POSIX / glibc              |
| Registry             | Config files (/etc)        |
| MSDN documentation   | Man-pages                  |
| Blue Screen of Death | Kernel Panic / Kernel Oops |
| WinDbg               | GDB                        |

