Lab 2: Socket Programming and Serialization

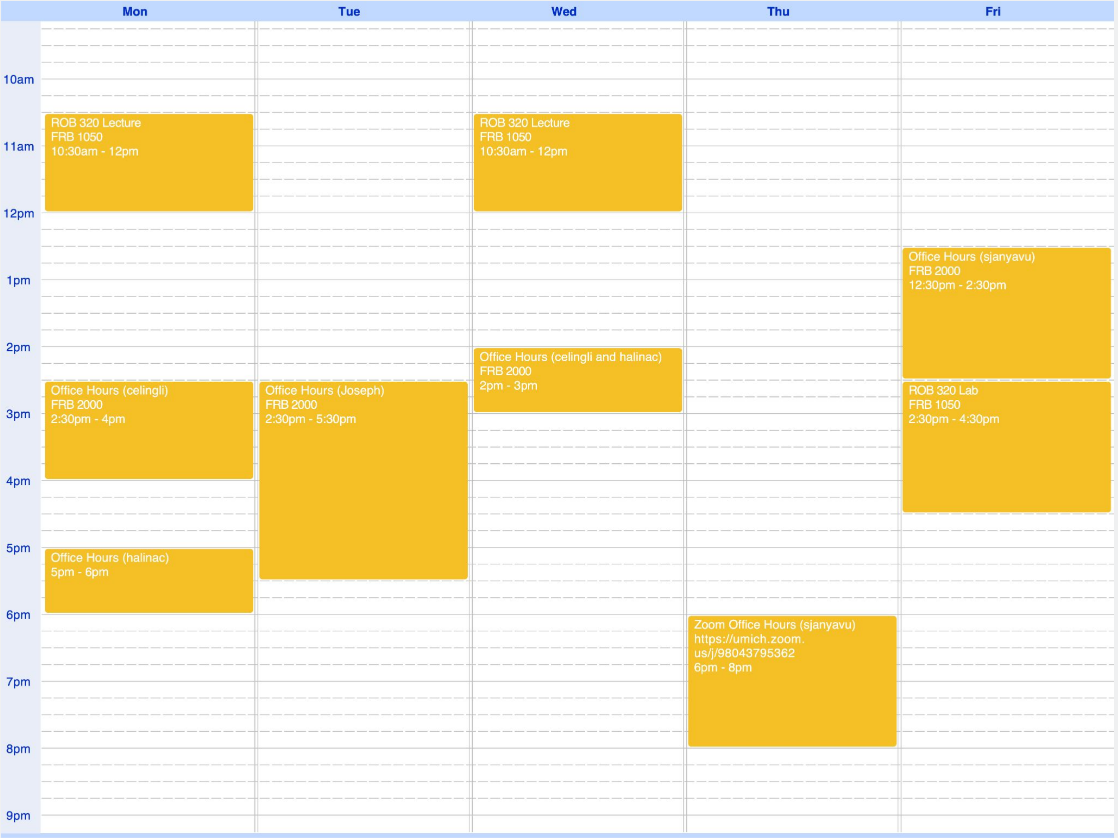
ROB 320: Robot Operating Systems

January 17, 2025

Agenda

- Office Hours
- Updates and Piazza Recap
- Discussion on Lab Workflow
- Lab Content and Assignment

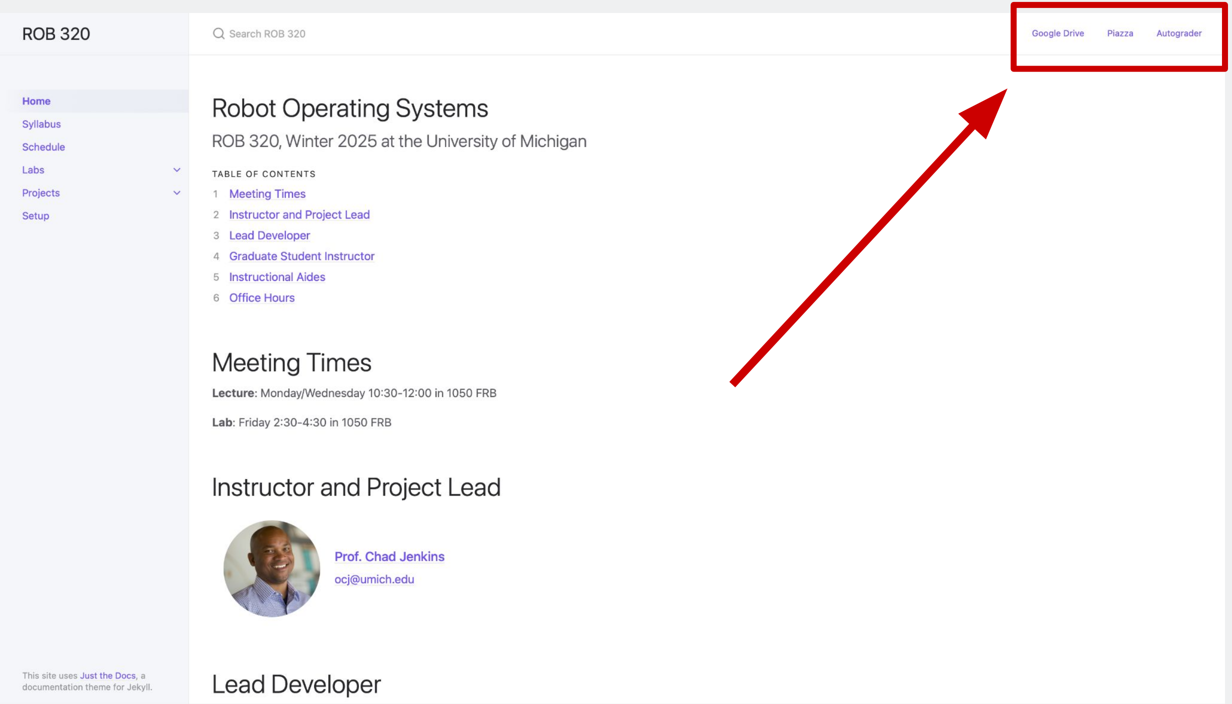
Office Hours



Mon	Tue	Wed	Thu	Fri
ROB 320 Lecture FRB 1050 10:30am - 12pm Office Hours (celingli) FRB 2000 2:30pm - 4pm Office Hours (halinac) 5pm - 6pm	Office Hours (Joseph) FRB 2000 2:30pm - 5:30pm	ROB 320 Lecture FRB 1050 10:30am - 12pm Office Hours (celingli and halinac) FRB 2000 2pm - 3pm	Zoom Office Hours (sjanyavu) https://umich.zoom.us/j/98043795362 6pm - 8pm	Office Hours (sjanyavu) FRB 2000 12:30pm - 2:30pm ROB 320 Lab FRB 1050 2:30pm - 4:30pm

- Brody office hours available by appointment 10am to 3pm on weekdays in FRB 2150 or Zoom
- Schedule them here

Updates



Robot Operating Systems

ROB 320, Winter 2025 at the University of Michigan

TABLE OF CONTENTS

Meeting Times

Lecture: Monday/Wednesday 10:30-12:00 in 1050 FRB

Lab: Friday 2:30-4:30 in 1050 FRB

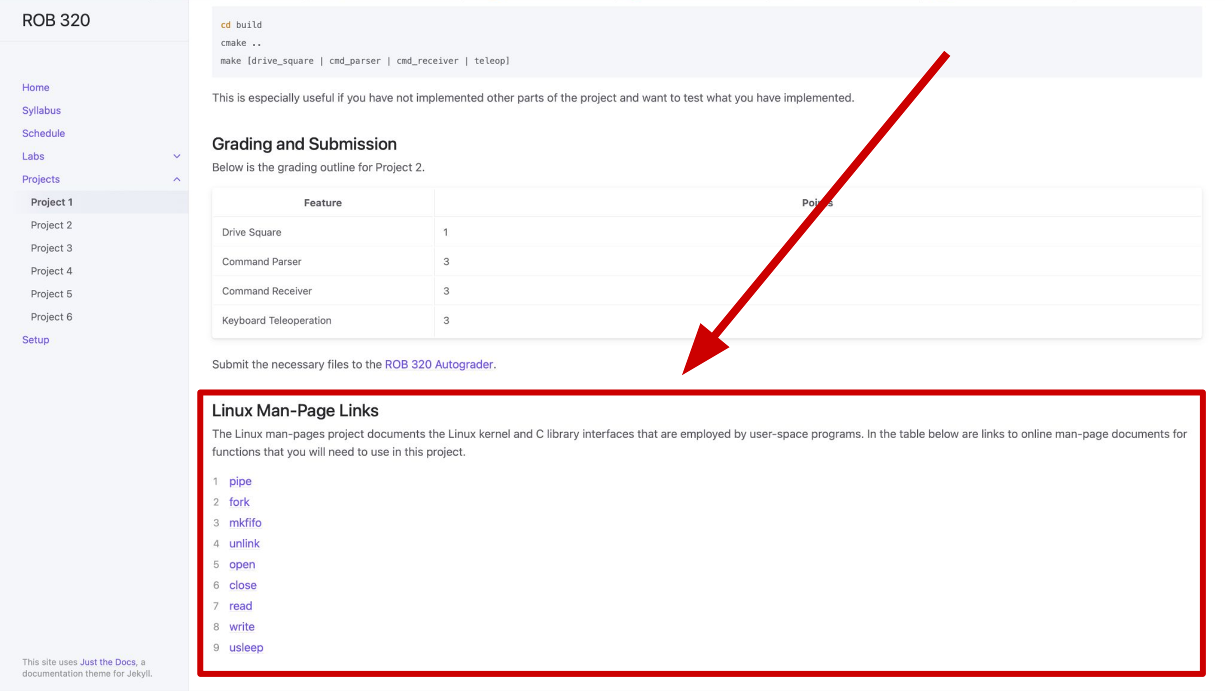
Instructor and Project Lead

Prof. Chad Jenkins
ocj@umich.edu

Lead Developer

This site uses Just the Docs, a documentation theme for Jekyll.

Updates



ROB 320

```
cd build
cmake ..
make [drive_square | cmd_parser | cmd_receiver | teleop]
```

This is especially useful if you have not implemented other parts of the project and want to test what you have implemented.

Grading and Submission

Below is the grading outline for Project 2.

Feature	Points
Drive Square	1
Command Parser	3
Command Receiver	3
Keyboard Teleoperation	3

Submit the necessary files to the ROB 320 Autograder.

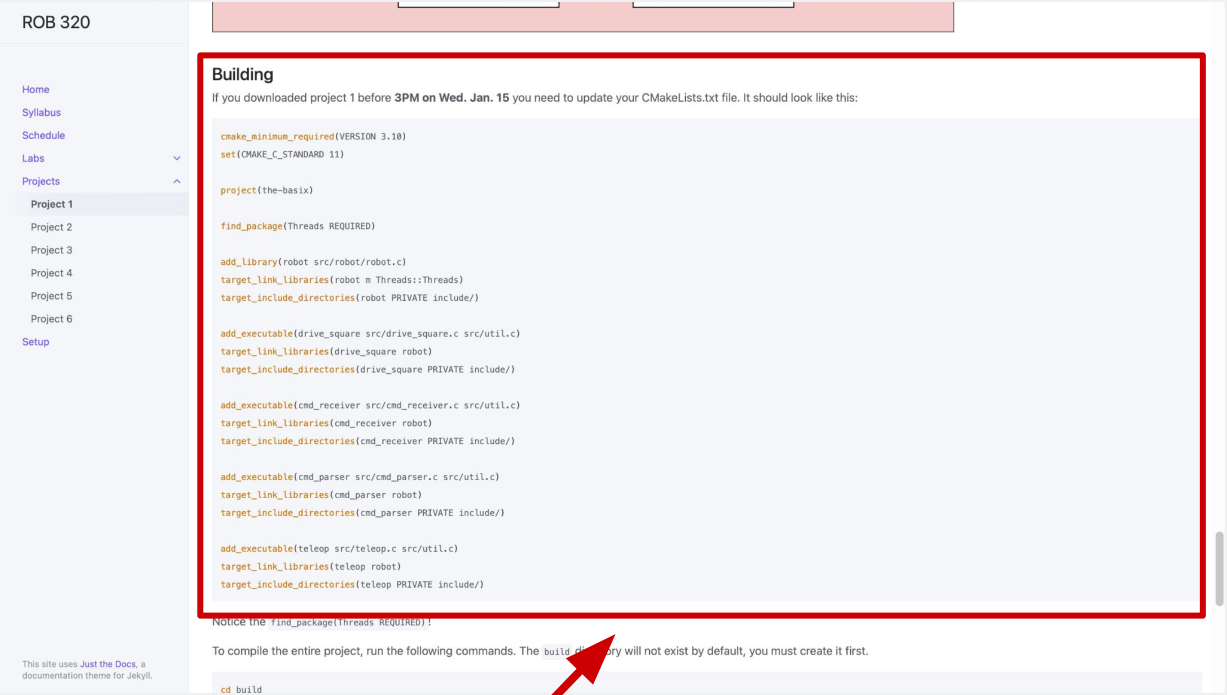
Linux Man-Page Links

The Linux man-pages project documents the Linux kernel and C library interfaces that are employed by user-space programs. In the table below are links to online man-page documents for functions that you will need to use in this project.

1. pipe
2. fork
3. mkfifo
4. unlink
5. open
6. close
7. read
8. write
9. usleep

This site uses Just the Docs, a documentation theme for Jekyll.

Updates



ROB 320

- Home
- Syllabus
- Schedule
- Labs
- Projects
  - Project 1
  - Project 2
  - Project 3
  - Project 4
  - Project 5
  - Project 6
- Setup

This site uses Just the Docs, a documentation theme for Jekyll.

Building

If you downloaded project 1 before **3PM on Wed. Jan. 15** you need to update your CMakeLists.txt file. It should look like this:

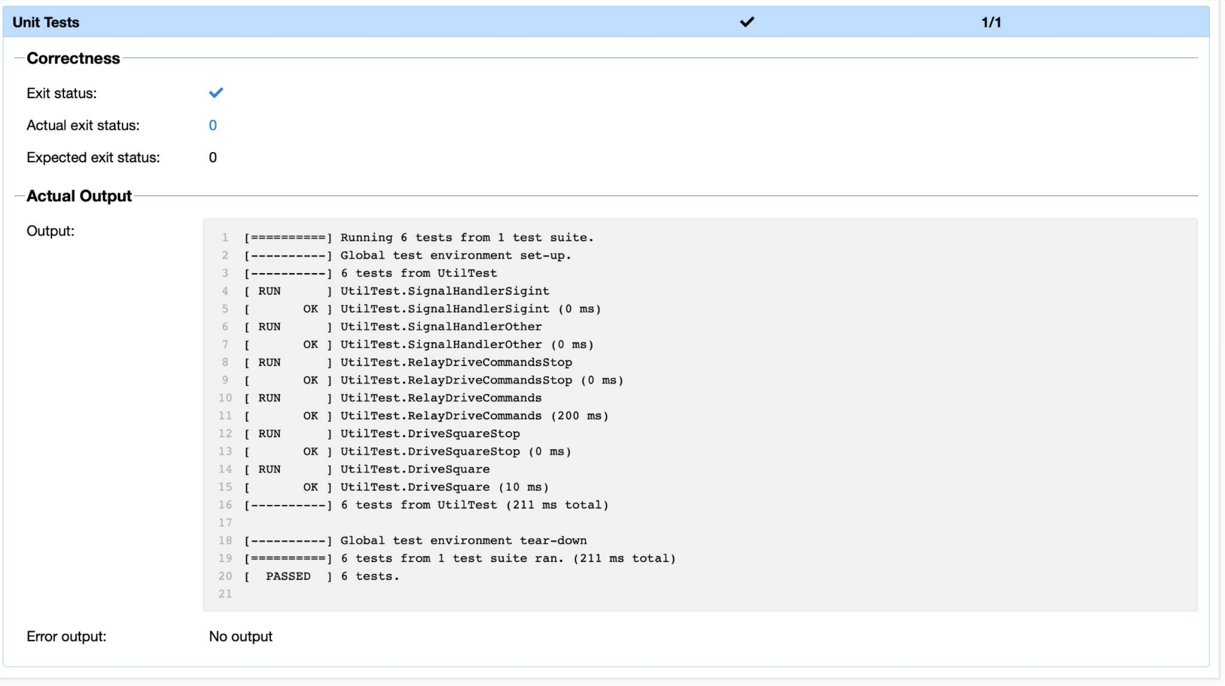
```
cmake_minimum_required(VERSION 3.10)
set(CMAKE_C_STANDARD 11)

project(the-basix)

find_package(Threads REQUIRED)

add_library(robot src/robot/robot.c)
target_link_libraries(robot m Threads::Threads)
target_include_directories(robot PRIVATE include/)

add_executable(drive_square src/drive_square.c src/util.c)
target_link_libraries(drive_square robot)
target_include_directories(drive_square PRIVATE include/)

add_executable(cmd_receiver src/cmd_receiver.c src/util.c)
target_link_libraries(cmd_receiver robot)
target_include_directories(cmd_receiver PRIVATE include/)

add_executable(cmd_parser src/cmd_parser.c src/util.c)
target_link_libraries(cmd_parser robot)
target_include_directories(cmd_parser PRIVATE include/)

add_executable(teleop src/teleop.c src/util.c)
target_link_libraries(teleop robot)
target_include_directories(teleop PRIVATE include/)
```

Notice the `find_package(Threads REQUIRED)`!

To compile the entire project, run the following commands. The `build` directory will not exist by default, you must create it first.

```
cd build
```

Updates



Unit Tests ✓ 1/1

Correctness

Exit status: ✓

Actual exit status: 0

Expected exit status: 0

Actual Output

Output:

```
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from UtilTest
[ RUN      ] UtilTest.SignalHandlerSigint
[       OK ] UtilTest.SignalHandlerSigint (0 ms)
[ RUN      ] UtilTest.SignalHandlerOther
[       OK ] UtilTest.SignalHandlerOther (0 ms)
[ RUN      ] UtilTest.RelayDriveCommandsStop
[       OK ] UtilTest.RelayDriveCommandsStop (0 ms)
[ RUN      ] UtilTest.RelayDriveCommands
[       OK ] UtilTest.RelayDriveCommands (200 ms)
[ RUN      ] UtilTest.DriveSquareStop
[       OK ] UtilTest.DriveSquareStop (0 ms)
[ RUN      ] UtilTest.DriveSquare
[       OK ] UtilTest.DriveSquare (10 ms)
[----------] 6 tests from UtilTest (211 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (211 ms total)
[  PASSED  ] 6 tests.

```

Error output: No output

Updates

- You now have 4 submissions per day for Project 1.
  - CMake error prevented some students from building on CAEN
  - Unit Test output was not available
  - Did not have clear instructions on drive square and teleop

Project 1 Piazza Recap

- There are several ways to implement read_line.
- The purpose is to return an array of bytes from the start of the stream to the first occurrence of a new line character (\n).
- We don’t care about the runtime of this function.

Project 1 Piazza Recap

- Autograder tests no longer private
- All tests in this course will tell you pass/fail + exit value of the program

Project 1 Piazza Recap

- drive_cmd_t utime would typically hold the epoch time in microseconds when the object was last modified
- It should be set to 0 for this project

Project 1 Piazza Recap

- Teleop commands
  - w: +vx
  - a: +vy
  - s: -vx
  - d: -vy
  - q: +wz
  - e: -wz
  - ‘ ‘: 0

- Drive square commands
  - N: +vx
  - E: -vy
  - S: -vx
  - W: +vy

Project 1 Piazza Recap

- relay_drive_commands should send a "stop" command once SIGINT is received.
  - vx: 0, vy: 0, wz: 0 once you break out of the loop
  - This is a safety feature so the MBot will not continue driving if your last command was nonzero
- drive_square should also send a "stop" command once the square is completed.

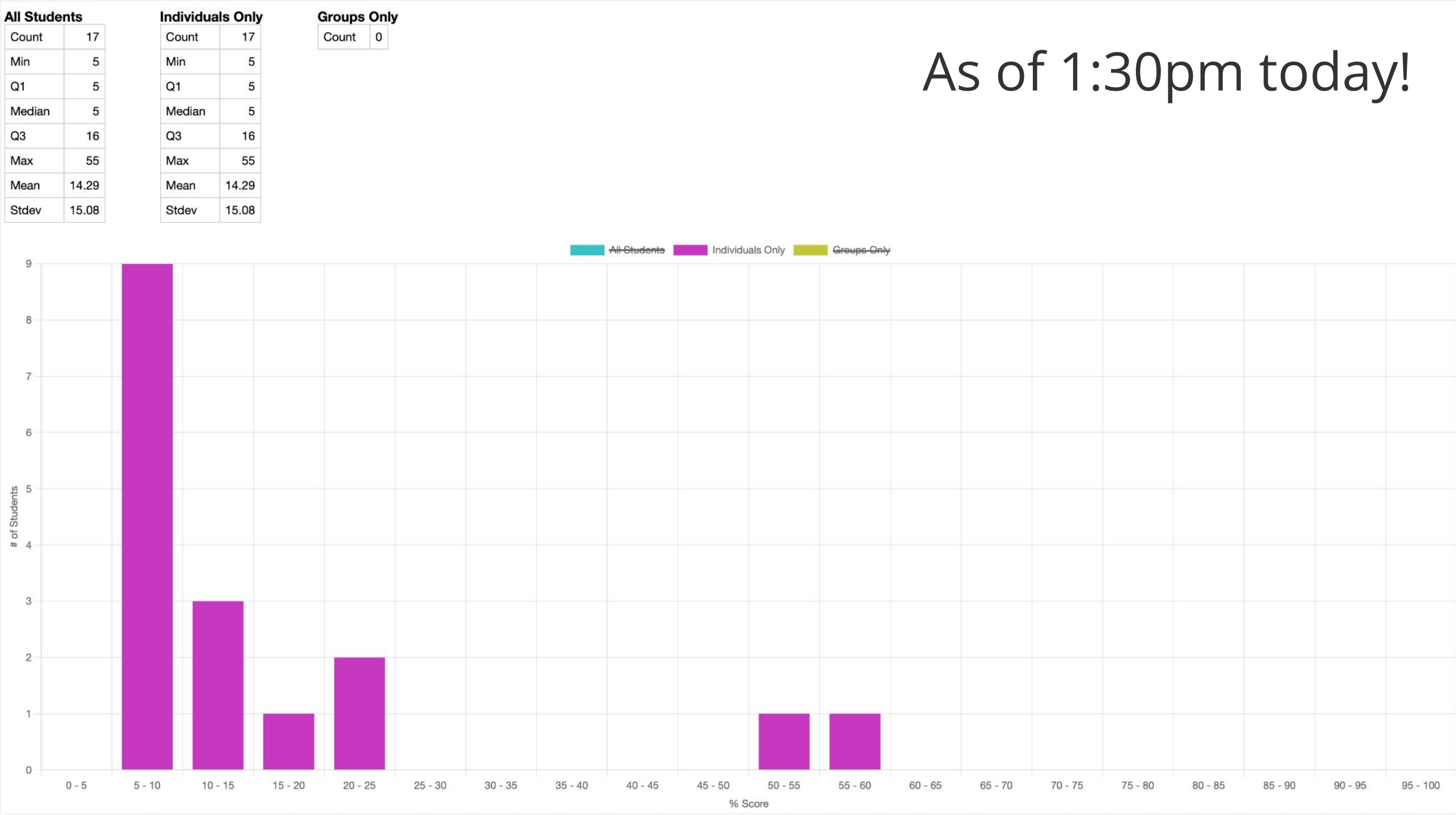
Project 1 Piazza Recap

- ./drive_square 0 N 0.5
  - utime: 0, vx: 0.5, vy: 0, wz: 0
  - utime: 0, vx: 0, vy: -0.5, wz: 0
  - utime: 0, vx: -0.5, vy: 0, wz: 0
  - utime: 0, vx: 0, vy: 0.5, wz: 0
  - utime: 0, vx: 0, vy: 0, wz: 0 (from drive_square)
  - utime: 0, vx: 0, vy: 0, wz: 0 (from relay_drive_commands)
-
  ```
  drive_cmd_t cmd = {0};
  ```

Project 1 Piazza Recap

- Autograder tests based on data sent to stdout
- Make sure you are not printing any debug information!

Project 1 Submissions



All Students

Count	17
Min	5
Q1	5
Median	5
Q3	16
Max	55
Mean	14.29
Stdev	15.08

Individuals Only

Count	17
Min	5
Q1	5
Median	5
Q3	16
Max	55
Mean	14.29
Stdev	15.08

Groups Only

Count	0

As of 1:30pm today!

Discussing Lab 1

- Difficulty level?
- Did you feel adequately prepared?
- Did you attend OH? Were staff helpful?
- Other suggestions?

Lab Content Overview

- Linux Man-Pages
- Files
- C Standard Library Functions
- Serialization
- POSIX Sockets

Linux Man-Pages

- Comprehensive documentation for Unix and Linux commands and functions.
- Accessed by entering `man [command]` in the terminal.
- Generally installed on all Unix-like systems for offline access.
- Also available online: https://man7.org/linux/man-pages/

Unix File Abstraction

- Everything is a file!
  - Central design philosophy in Unix systems.
  - Devices, sockets, and pipes are treated as files.
- Unified set of system calls
  - `open`, `read`, `write`, `close` work for all file types.
- Abstraction layer for the programmer's convenience

Unix File Abstraction

- Files have a unique id (index node number): `ls -i`
  - Index node (inode) is a data structure that contains type, permissions, size, etc: `stat [file]`
- File Types
  - **Regular Files**: Contain data, such as text, binaries.
  - **Directories**: Special files that contain lookup tables mapping filenames to inode numbers.
  - **Device Files**: Represent hardware devices; can be character (/dev/tty) or block (/dev/sda).
  - **Pipes and Sockets**: Enable IPC.

File Descriptors

- An integer handle used to access and manage files and I/O resources.
- Acts as an index into a file descriptor table that is maintained per process by the OS.
- By default, each process has three:
  - 0: Standard Input (stdin): Receives input data.
  - 1: Standard Output (stdout): Outputs data.
  - 2: Standard Error (stderr): Handles error messages.

File Descriptors

- Can represent sockets, pipes, and devices in Unix-like systems.
- File descriptors are obtained using system calls like `open`, `socket`, or `pipe`.
- Systems impose limits on the number of open file descriptors per process: `ulimit -n`.
- Descriptors should be closed using `close` to free resources and avoid leaks.

File Pointers

- A file descriptor is an integer handle used to identify an opened file at the kernel level
  - Used for system calls (`open`, `read`, `write`, `close`, etc.)
- A file pointer (`FILE*`) is struct that contains a file descriptor
  - Adds buffering and other features to make I/O easier
  - Used for C standard library functions (`fopen`, `fread`, `fwrite`, `fclose`, etc.)
  - Convert from file descriptor to pointer using `fdopen`

C Header Files

- C Standard Library
  - stdio.h
  - stdlib.h
  - stdint.h
  - string.h
  - signal.h
  - time.h
  - errno.h
  - math.h

- C POSIX Library
  - unistd.h
  - fctnl.h
  - pthread.h
  - termios.h
  - sys/socket.h
  - arpa/inet.h
  - netdb.h

libc: stdlib.h

- malloc
- calloc
- free
- atoi
- atof

```c
#include <stdlib.h>

int main(void) {
   int *int_ptr = (int *)malloc(sizeof(int));
   *int_ptr = atoi("123");

   float *float_arr = (float *)calloc(5, sizeof(float));
   float_arr[2] = atof("3.14");

   free(int_ptr);
   free(float_arr);

   return 0;
}
```

libc: stdio.h

- fopen
- fread
- fwrite
- fclose

```c
#include <stdio.h>
#define SIZE 5
int main() {
   const double a[SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0};
   FILE *fp = fopen("test.bin", "wb"); // wb for writing in binary mode
   fwrite(a, sizeof *a, SIZE, fp);
   fclose(fp);

   double b[SIZE];
   fp = fopen("test.bin","rb");
   fread(b, sizeof(b[0]), SIZE, fp);
   fclose(fp);

   for (int i = 0; i < SIZE; i++) {
       printf("%f\n", b[i]);
   }
}
```

libc: stdio.h

- printf
- scanf
- fprintf
- fscanf
- sprintf
- sscanf

```c
#include <stdio.h>

int main(void) {
   char name[50];
   int age;
   char buffer[100];
   FILE *file;

   printf("Enter your name: ");
   scanf("%s", name);
   printf("Enter your age: ");
   scanf("%d", &age);
   sprintf(buffer, "%s is %d years old!\n", name, age);

   file = fopen("output.txt", "w");
   fprintf(file, "%s", buffer);
   fclose(file);

   file = fopen("output.txt", "r");
   fscanf(file, "%s is %d years old!\n", name, &age);
   fclose(file);
   printf("Read from file: Name: %s, Age: %d\n", name, age);

   sscanf(buffer, "%s is %d years old!\n", name, &age);
   printf("Read from buffer: Name: %s, Age: %d\n", name, age);
   return 0;
}
```

libc: string.h

- strcmp
- strcpy
- strncpy

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

int main(void) {
   char str1[20];
   strcpy(str1, "Hello");
   printf("strcpy: %s\n", str1);

   char str2[20];
   strncpy(str2, "World", 3);
   str2[3] = '\0'; // null-terminate the string
   printf("strncpy: %s\n", str2);

   int cmp1 = strcmp("abc", "abc");
   int cmp2 = strcmp("abc", "def");
   printf("strcmp: %d, %d\n", cmp1, cmp2);

   return 0;
}
```

libc: string.h

- memcpy
- memmove
- memset

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

void print_array(int* arr, int size) {
   for (int i = 0; i < size; i++) {
       printf("%d ", arr[i]);
   }
   printf("\n");
}

int main(void) {
   int arr1[5] = {1, 2, 3, 4, 5};
   int arr2[5] = {0};

   // Copy the contents of arr1 to arr2
   memcpy(arr2, arr1, sizeof(arr1));
   printf("arr2: ");
   print_array(arr2, 5);

   // Move the contents of arr2 to the right by 1 element
   memmove(arr2 + 1, arr2, sizeof(arr2) - sizeof(arr2[0]));
   printf("arr2: ");
   print_array(arr2, 5);

   // Set the contents of arr2 to 0
   memset(arr2, 0, sizeof(arr2));
   printf("arr2: ");
   print_array(arr2, 5);
}
```

libc: signal.h

- signal

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

int flag = 0;

void signal_handler(int signum) {
   printf("Received signal %d\n", signum);
   flag = 1;
}

int main(void) {
   signal(SIGINT, signal_handler);
   signal(SIGTERM, signal_handler);
   signal(SIGUSR1, signal_handler);

   printf("Waiting for signal...\n");
   while (!flag);
}
```

libc: errno.h

- errno
- perror (stdio.h)

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

int main(void) {
   FILE *file = fopen("non_existent_file.txt", "r");
   if (file == NULL) {
       printf("fopen failed with error: %d\n", errno);
       perror("fopen");
       return 1;
   }
}
```

posix: unistd.h

- open
- read
- write
- close

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

int main(void) {
   char name[16] = {0};
   char buffer[64] = {0};
   int fd;

   write(1, "Enter your name: ", 17);
   read(0, name, 50);

   fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   write(fd, "Hello, ", 7);
   write(fd, name, 16);
   write(fd, "Welcome to ROB320!\n", 20);
   close(fd);

   fd = open("output.txt", O_RDONLY);
   read(fd, buffer, 64);
   close(fd);

   write(1, buffer, 64);
   return 0;
}
```

posix: unistd.h

- pipe
- fork

```c
#include <unistd.h>

int main(void) {
   int fd[2];
   pipe(fd);

   int pid = fork();

   if (pid == 0) {
       // Child process
       close(fd[0]);
       write(fd[1], "Hello, world!", 13);
       close(fd[1]);
   } else {
       // Parent process
       close(fd[1]);
       char buffer[13];
       read(fd[0], buffer, 13);
       close(fd[0]);
       write(1, buffer, 13);
   }
}
```

posix: sys/stat.h

- mkfifo
- unlink (unistd.h)

```c
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
   int fd;
   char buffer[32] = {0};

   mkfifo("fifo", 0666);
   fd = open("fifo", O_RDONLY);
   read(fd, buffer, sizeof(buffer));
   close(fd);
   unlink("fifo");// Remove the FIFO file

   write(1, buffer, sizeof(buffer));
}
```

posix: fcntl.h

- fcntl
- F_GETFL
- F_SETFL
- O_RDONLY,
  O_WRONLY,
  O_RDWR,
  O_CREAT,
  O_TRUNC
- O_NONBLOCK

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

int main(void) {
   int fd;
   int flags;

   fd = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);

   flags = fcntl(fd, F_GETFL);
   printf("Current file status flags: %d\n", flags);

   fcntl(fd, F_SETFL, flags | O_NONBLOCK);

   flags = fcntl(fd, F_GETFL);
   printf("New file status flags: %d\n", flags);

   close(fd);
   return 0;
}
```

Serialization

- Converting a data object into a format that can be stored or transmitted, and reconstructed later.
- Common method: JSON

```
let student = {};
student.name = "Jane Doe";
student.gpa = 3.25;
student.is_happy = true;
student.fav_courses = ["ROB320", "ROB102"]

let encoded = JSON.stringify(student);
console.log(encoded);
let decoded = JSON.parse(encoded);

> {"name":"Jane Doe","gpa":3.25,"is_happy":true,"fav_courses":["ROB320","ROB102"]}
```

Serialization

- In this course, we make assumptions about the data objects that we will serialize
  - **Contiguous**: Data is sequential and uninterrupted
  - **Statically allocated**: Fixed size known at compile time

- In C and C++, this allows us to serialize our objects by casting to a byte pointer `(uint8_t*)`.
- Deserialize by casting to the struct
  - Only if the byte array is the same size!

Serialization

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>

#pragma pack(push, 1)
typedef struct {
   char name[16];
   float gpa;
   int is_happy;
   char fav_courses[3][7];
} Student;
#pragma pack(pop)

int main() {
   Student student = {0};
   strcpy(student.name, "Jane Doe");
   student.gpa = 3.25;
   student.is_happy = 1;
   strcpy(student.fav_courses[0], "ROB320");
   strcpy(student.fav_courses[1], "ROB102");

   uint8_t *encoded = (uint8_t*)&student;
   for (size_t i = 0; i < sizeof(Student); i++) {
       printf("%02x ", encoded[i]);
   }
   Student *decoded = (Student*)encoded;
}
```

```
4a 61 6e 65 20 44 6f 65 00 00 00 00 00 00
00 00 00 00 50 40 01 00 00 00 52 4f 42 33
32 30 00 52 4f 42 31 30 32 00 00 00 00 00
00 00 00
```

Serialization

- Pros
  - Fast, casting pointers has negligible overhead, O(1) operation
  - Memory efficient, no data is added to the encoded array
- Cons
  - Constrained to contiguous data objects, no dynamic arrays or strings
  - Constrained to statically allocated data objects, size must be constant

Sockets

- A socket contains an **IP address** and a **port number**
  - IP address is a unique numerical identifier assigned to a device on a network
  - The port number specifies which application or service on a device should receive incoming data
- An IP address is to an apartment, as a port number is to a unit number.

posix: sys/sockets.h

- socket
- setsockopt
- sockaddr_in, sockaddr (arpa/inet.h)
- htons
- ntohs
- inet_ntoa
- bind
- listen
- accept
- recv

```c
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>

int main(void) {
   struct sockaddr_in addr, client_addr;
   int fd = socket(AF_INET, SOCK_STREAM, 0);  // SOCK_STREAM for TCP
   int opt = 1;
   setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

   addr.sin_family = AF_INET;
   addr.sin_addr.s_addr = INADDR_ANY;  // Bind to all available interfaces
   addr.sin_port = htons(8000);        // Bind to port 8000
   bind(fd, (struct sockaddr*)&addr, sizeof(addr));

   listen(fd, INT16_MAX);
   socklen_t client_addr_len = sizeof(client_addr);
   int client_fd = accept(fd, (struct sockaddr*)&client_addr, &client_addr_len);
   printf("Connection socket %s:%d\n", inet_ntoa(client_addr.sin_addr),
           ntohs(client_addr.sin_port));

   char buffer[256];
   size_t bytes_read = recv(client_fd, buffer, sizeof(buffer), 0);
   buffer[bytes_read] = '\0';
   printf("Received %zu bytes: %s\n", bytes_read, buffer);
   close(fd);
}
```

posix: sys/sockets.h

- inet_aton
- connect
- send

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>

int main(void) {
   struct sockaddr_in server_addr;

   int fd = socket(AF_INET, SOCK_STREAM, 0);  // SOCK_STREAM for TCP
   server_addr.sin_family = AF_INET;
   server_addr.sin_addr.s_addr = inet_aton("127.0.0.1"); // Server's IP
   server_addr.sin_port = htons(8000); // Server's port
   connect(fd, (struct sockaddr*)&server_addr, sizeof(server_addr));

   const char buffer[] = "Hello, server!";
   size_t bytes_sent = send(fd, buffer, sizeof(buffer), 0);
   printf("Sent %zu bytes", bytes_sent);

   close(fd);
}
```

Non-blocking I/O

- A blocking operation halts the program until it has completed
- A non-blocking operation will return immediately, without waiting for the operation to complete

- When would this be useful?

Non-blocking I/O

- Useful when our program has other things to do!
  - Server should not wait for a message when there are other clients trying to connect
  - Server should not wait for a connection when there are clients trying to send data
  - What if a signal is handled and the program state changes?

Lab Assignment

- `discovery`
  - Server that maintains a list of logs that have registered/deregistered.
  - Sends this list to a client upon request.
- `chat`
  - Client application that will request the discovery server.
  - Prompts the user to select a registered log and enter a message
  - Sends this message to the registered log server
- `log`
  - Server application that registers with the discovery server on boot and deregisters on exit.
  - Accepts connections from chat clients, receives a message, and prints it to stdout.

Lab Assignment

Discovery (35.3.7.155:8000)

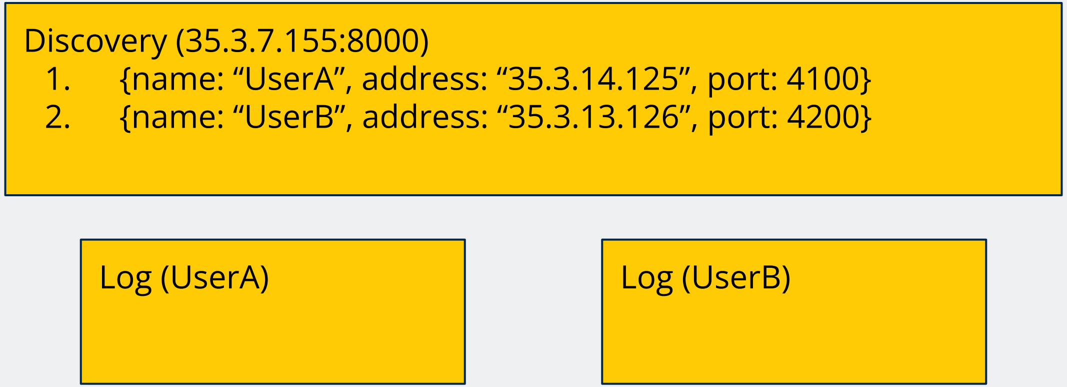
Lab Assignment

Discovery (35.3.7.155:8000)

1. {name: “UserA”, address: “35.3.14.125”, port: 4100}

Log (UserA)

Lab Assignment



Discovery (35.3.7.155:8000)

1. {name: “UserA”, address: “35.3.14.125”, port: 4100}
2. {name: “UserB”, address: “35.3.13.126”, port: 4200}

Log (UserA)

Log (UserB)

Lab Assignment

```
Discovery (35.3.7.155:8000)
  1.     {name: “UserA”, address: “35.3.14.125”, port: 4100}
  2.     {name: “UserB”, address: “35.3.13.126”, port: 4200}
```

```
Log (UserA)
```

```
Log (UserB)
UserA : “Hello!”
```

```
Chat (UserA)
Chatters:
[0] : UserA (35.3.14.125:4100)
[1] : UserB (35.3.13.126:4200)
Select a chatter to send a message to: 1
Enter message: Hello!
```

Lab Assignment

```
Discovery (35.3.7.155:8000)
  1.    {name: “UserA”, address: “35.3.14.125”, port: 4100}
  2.    {name: “UserB”, address: “35.3.13.126”, port: 4200}
```

```
Log (UserA)
UserB : “Hey!”
```

```
Log (UserB)
UserA : “Hello!”
```

```
Chat (UserB)
Chatters:
[0] : UserA (35.3.14.125:4100)
[1] : UserB (35.3.13.126:4200)
Select a chatter to send a message to: 0
Enter message: Hey!
```

Lab Assignment

Discovery (35.3.7.155:8000)

1. {name: “UserA”, address: “35.3.14.125”, port: 4100}
2. {name: “UserB”, address: “35.3.13.126”, port: 4200}

Log (UserA)
UserB : “Hey!”

Log (UserB)
UserA : “Hello!”

Lab Assignment

Discovery (35.3.7.155:8000)

1. {name: “UserA”, address: “35.3.14.125”, port: 4100}

Log (UserA)
UserB : “Hey!”

Lab Assignment

Discovery (35.3.7.155:8000)

Lab Assignment

- util.h
  - Write functions that handle non-blocking socket calls
- messages.h
  - Write serialization functions for structs
- chat.c
  - Fill in the socket calls for the chat application
- log.c
  - Fill in the socket calls for the log application
- discovery.c
  - This program is done for you, use it to test `chat` and `log`

Lab 2 Notes

- Spend less time talking about file descriptors and file pointers (ideally move to Lab 1)
- Move C standard library and posix library to lab 1
- Go over main socket functions more in depth (socket, bind, listen, accept, connect, send, and recv) also setsockopt and fcntl
- Add discussion about UDP if time permits (if some content is moved to Lab 1, this would work)