COMPUTER SYSTEMS AND ORGANIZATION

File Descriptors and Memory Errors

Daniel G. Graham Ph.D

UNIVERSITY of VIRGINIA | ENGINEERING

Contents

1. File Descriptors
2. Linux Permissions
3. STDIN/STDOUT/STDERROR
4. Memory Descriptions
5. Can you spot the memory error game

FIO EXAMPLE

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

int main()
{
    // file pointer variable to store the value returned by
    // fopen
    FILE* fptr;

    // opening the file in read mode
    fptr = fopen("filename.txt", "r");


    return 0;
}
```

FIO WRITE EXAMPLE

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

int main()
{
    // file pointer variable to store the value returned by fopen
    FILE* fptr;

    // opening the file in write mode
    fptr = fopen("filename.txt", "w");

    // Writing text to the file
    fprintf(fptr, "Hello, world! This is text being written to the file.\n");

    // closing the file
    fclose(fptr);

    return 0;
}
```

FILE DESCRIPTORS

A file descriptor is a non-negative integer that uniquely identifies an open file or I/O stream within a process. The operating system maintains a file descriptor table for each process to keep track of open files.

Draw the table

THE FCNTL.H HEADER

The fcntl.h header file in C stands for "file control." It is part of the C standard library and provides functions and symbolic constants for controlling file-related operations. The primary purpose of fcntl.h is to manipulate file descriptors and provide additional control over open files.

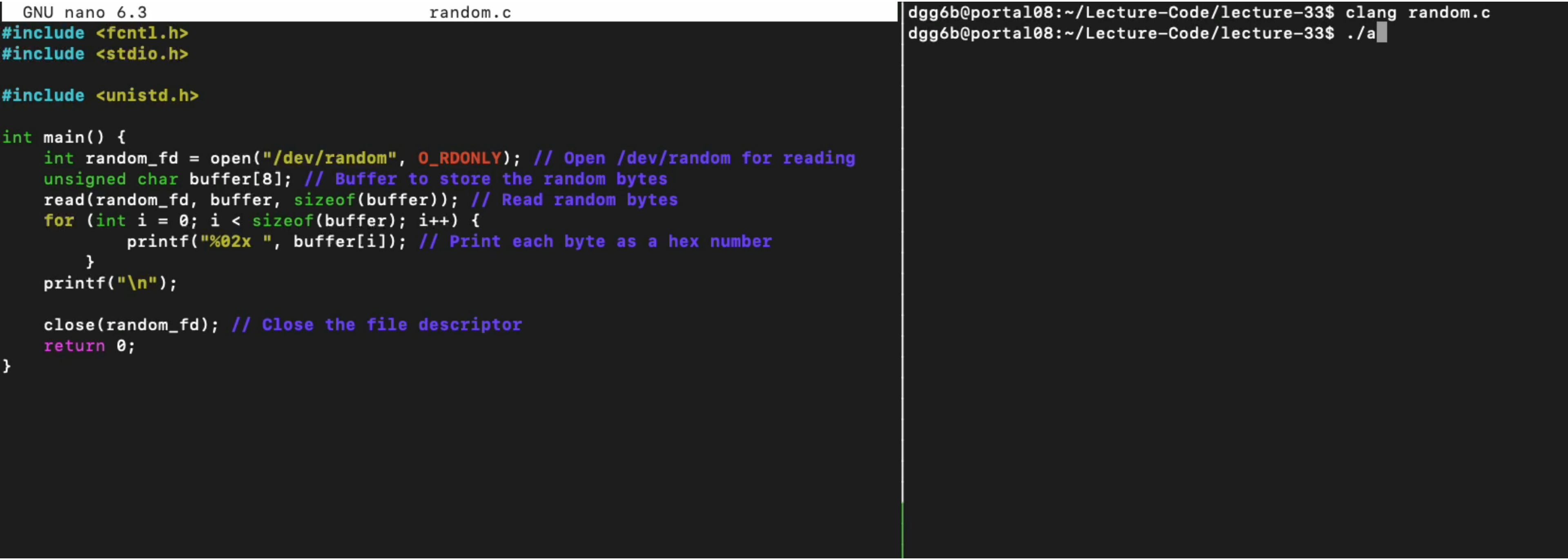
OPEN FILE WITH FILE DECRIPTOR

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

int main() {
    char buffer[1024];
    int file_descriptor = open("example.txt", O_RDONLY);
    bytesRead = read(input_fd, buffer, sizeof(buffer));
    close(file_descriptor); // Don't forget to close the file!
    return 0;
}
```

EVERYTHING IN LINUX IS A FILE (KINDA)

In Unix-like operating systems, including Linux, the philosophy that "everything is a file" refers to the fact that most resources, such as hardware devices, directories, and network sockets, are represented as file descriptors and can be interacted with using standard file I/O system calls.

DEV RANDOM



```
  GNU nano 6.3                       random.c
#include <fcntl.h>
#include <stdio.h>

#include <unistd.h>

int main() {
    int random_fd = open("/dev/random", O_RDONLY); // Open /dev/random for reading
    unsigned char buffer[8]; // Buffer to store the random bytes
    read(random_fd, buffer, sizeof(buffer)); // Read random bytes
    for (int i = 0; i < sizeof(buffer); i++) {
            printf("%02x ", buffer[i]); // Print each byte as a hex number
        }
    printf("\n");

    close(random_fd); // Close the file descriptor
    return 0;
}
```

```
dgg6b@portal08:~/Lecture-Code/lecture-33$ clang random.c
dgg6b@portal08:~/Lecture-Code/lecture-33$ ./a
```

STDIN/STDOUT/STDERROR FILE DESCRIPTOR

0 (stdin): Standard input file descriptor. It is used for reading input from the keyboard or other input sources.

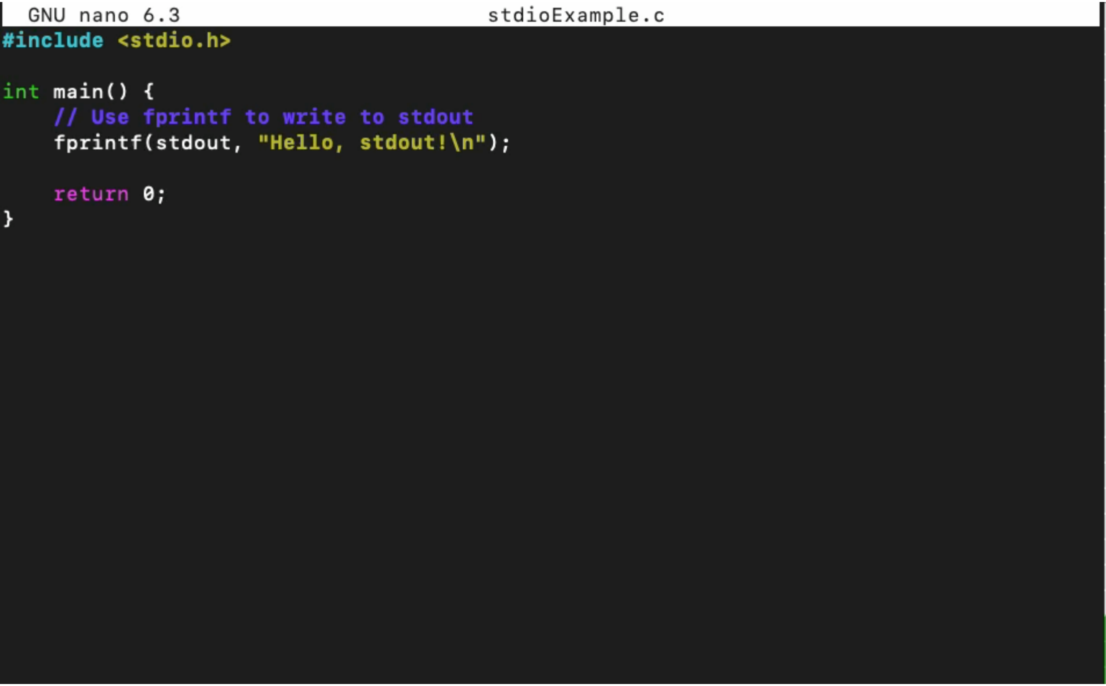
1 (stdout): Standard output file descriptor. It is used for writing normal output, which is typically the console.

2 (stderr): Standard error file descriptor. It is used for writing error messages or diagnostic information to the console.

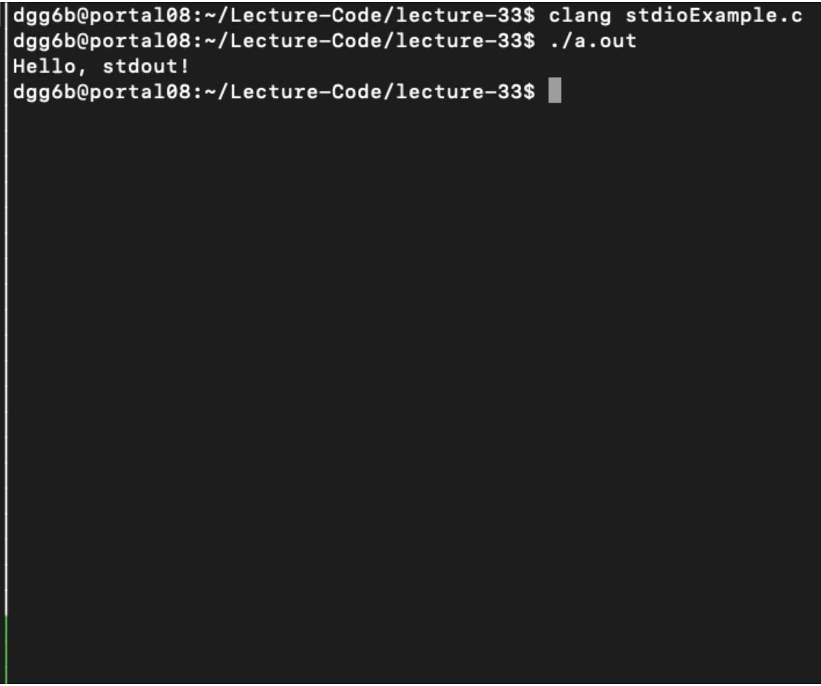
WHAT DO WE THINK

```
#include <stdio.h>

int main() {
    // Use fprintf to write to stdout
    fprintf(stdout, "Hello, stdout!\n");

    return 0;
}
```

STDIO



```
  GNU nano 6.3                     stdioExample.c
#include <stdio.h>

int main() {
    // Use fprintf to write to stdout
    fprintf(stdout, "Hello, stdout!\n");

    return 0;
}
```



```
dgg6b@portal08:~/Lecture-Code/lecture-33$ clang stdioExample.c
dgg6b@portal08:~/Lecture-Code/lecture-33$ ./a.out
Hello, stdout!
dgg6b@portal08:~/Lecture-Code/lecture-33$ 
```

FGETS

```
char *fgets(char *s, int size, FILE *stream);
```

EXAMPLE FILE DESCRIPTOR PROGRAM

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

int main() {
    // Assuming stdin and stdout as file descriptors
    FILE *input_stream = fdopen(0, "r");  // File descriptor 0 is stdin

    // Read a line of text from the user
    char buffer[1024];
    printf("Enter a line of text: ");
    fgets(buffer, sizeof(buffer), input_stream);

    // Write the entered line back to stdout
    printf("You entered: %s", buffer);

    // Close the file descriptor (not necessary, but good practice)
    fclose(input_stream);

    return 0;
}
```

FWRITE

```
size_t fwrite(const void *ptr, size_t size,
                          size_t nmemb, FILE *stream);
```

EXAMPLE FILE DESCRIPTOR

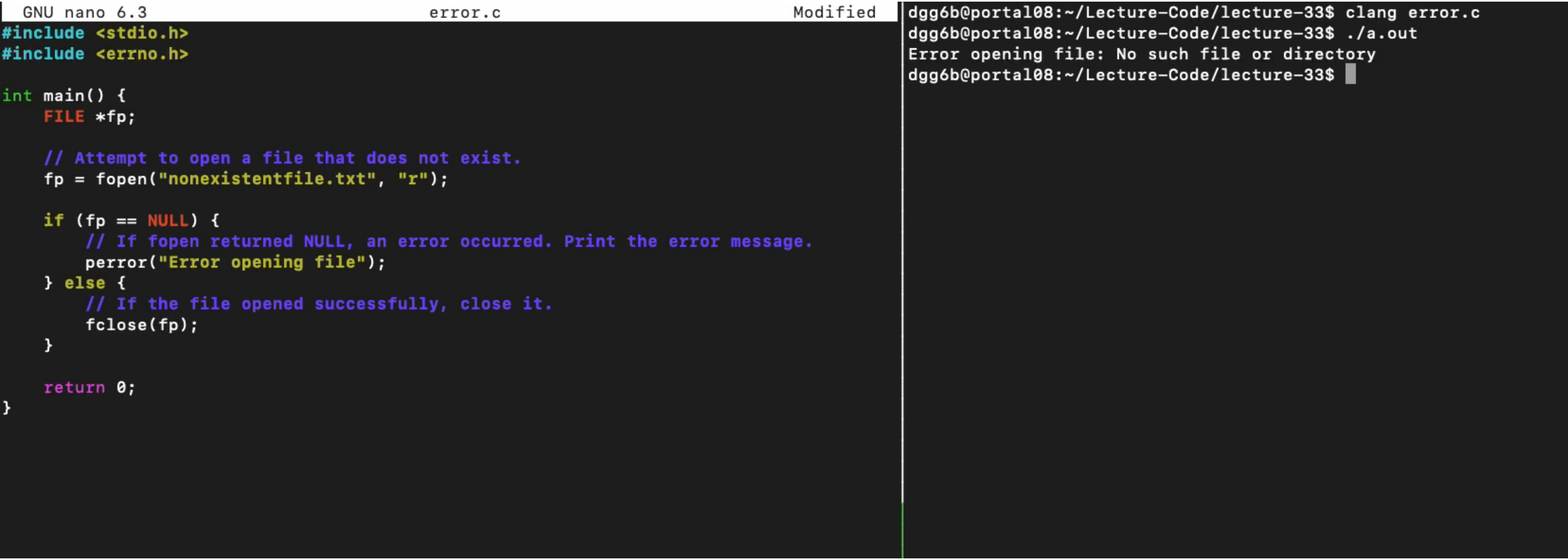
```c
#include <stdio.h>
#include <unistd.h>

int main() {
    // Assuming stdout as a file descriptor
    FILE *output_stream = fdopen(1, "w");  // File descriptor 1 is stdout

    // Message to be written
    const char *message = "Hello, stdout!\n";

    // Use fwrite to write to stdout
    size_t message_length = strlen(message);
    fwrite(message, sizeof(char), message_length, output_stream);

    // Close the file descriptor (not necessary, but good practice)
    fclose(output_stream);

    return 0;
}
```

PERROR

perror is a library function in C that prints a descriptive error message to the standard error output (stderr). The message is based on the global variable **errno**, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

PERROR

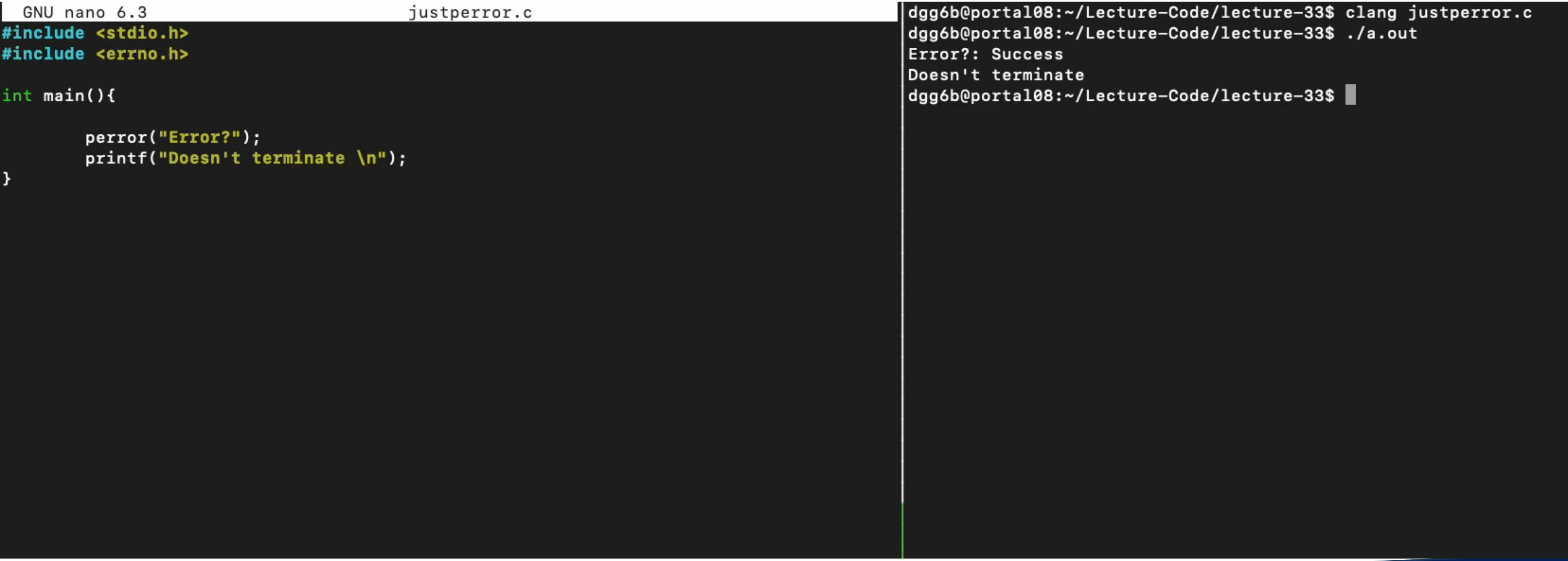
```c
#include <stdio.h>
#include <errno.h>

int main() {
    FILE *fp;

    // Attempt to open a file that does not exist.
    fp = fopen("nonexistentfile.txt", "r");

    if (fp == NULL) {
        // If fopen returned NULL, an error occurred. Print the error message.
        perror("Error opening file");
    } else {
        // If the file opened successfully, close it.
        fclose(fp);
    }

    return 0;
}
```



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

int main() {
    FILE *fp;

    // Attempt to open a file that does not exist.
    fp = fopen("nonexistentfile.txt", "r");

    if (fp == NULL) {
        // If fopen returned NULL, an error occurred. Print the error message.
        perror("Error opening file");
    } else {
        // If the file opened successfully, close it.
        fclose(fp);
    }

    return 0;
}
```

```
dgg6b@portal08:~/Lecture-Code/lecture-33$ clang error.c
dgg6b@portal08:~/Lecture-Code/lecture-33$ ./a.out
Error opening file: No such file or directory
dgg6b@portal08:~/Lecture-Code/lecture-33$
```

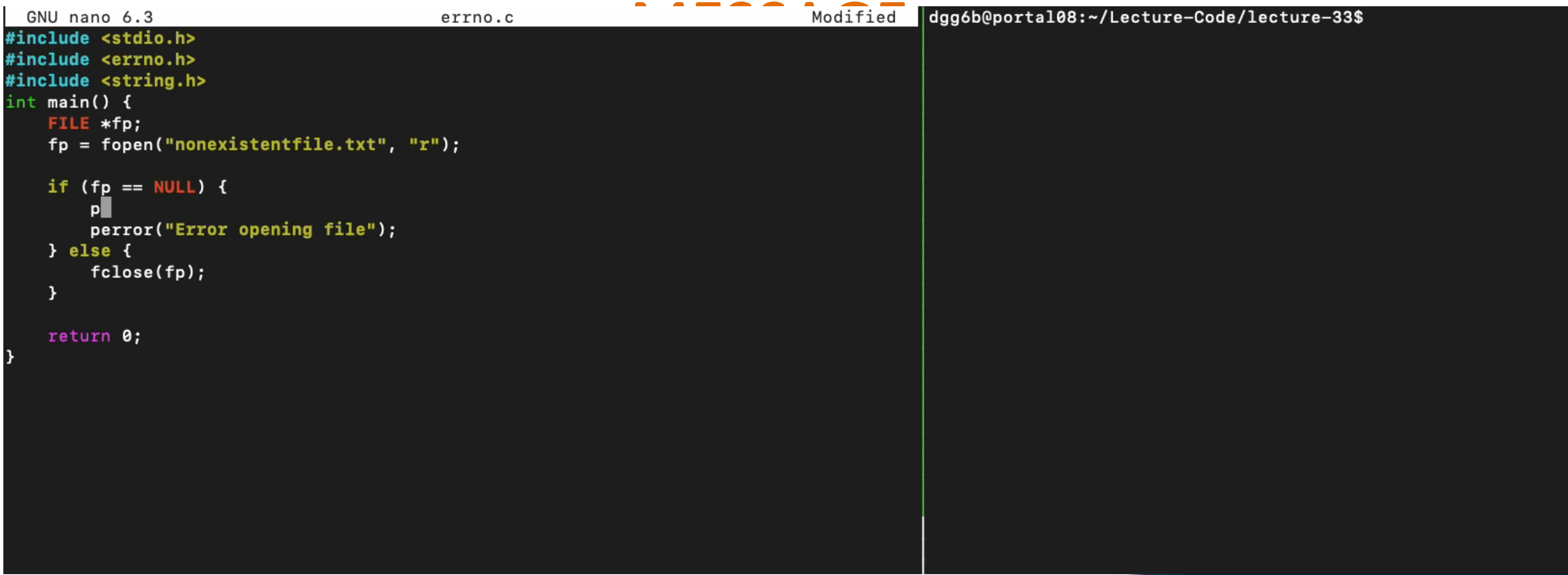
PERROR SUCCESS CASE



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

int main(){

        perror("Error?");
        printf("Doesn't terminate \n");
}
```

```
dgg6b@portal08:~/Lecture-Code/lecture-33$ clang justperror.c
dgg6b@portal08:~/Lecture-Code/lecture-33$ ./a.out
Error?: Success
Doesn't terminate
dgg6b@portal08:~/Lecture-Code/lecture-33$ 
```

WHAT IS THE MESSAGE AFTER PERROR



```
  GNU nano 6.3                    errno.c                         Modified
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main() {
    FILE *fp;
    fp = fopen("nonexistentfile.txt", "r");

    if (fp == NULL) {
        p
        perror("Error opening file");
    } else {
        fclose(fp);
    }

    return 0;
}
```

```
dgg6b@portal08:~/Lecture-Code/lecture-33$
```

IMPLEMENTING PERROR

Take a second talk to your neighbor could you implement perror?

SNPRINTF

```
int snprintf ( char * s, size_t n, const char * format, ... );
```

Instead of writing printing the console it prints to a buffer specified by a pointer.

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

void my_perror(const char *prefix) {
     char buffer[256];
     snprintf(buffer, sizeof(buffer), "%s: %s\n", prefix, strerror(errno));
     fwrite(buffer, strlen(buffer), 1, stderr);
}

int main() {
    FILE *fp = fopen("nonexistentfile.txt", "r");
    if (fp == NULL) {
        my_perror("Error opening file");
    } else {
        fclose(fp);
    }

    return 0;
}
```

LINUX PERMISSIONS

Three types of permissions:

- Read (r) - 4
- write (w) – 2
- Execute (x) -1

Normally three sets of permissions are associated with a file

User

Group

Other

Only the owner of the file can change permissions.

Permission String

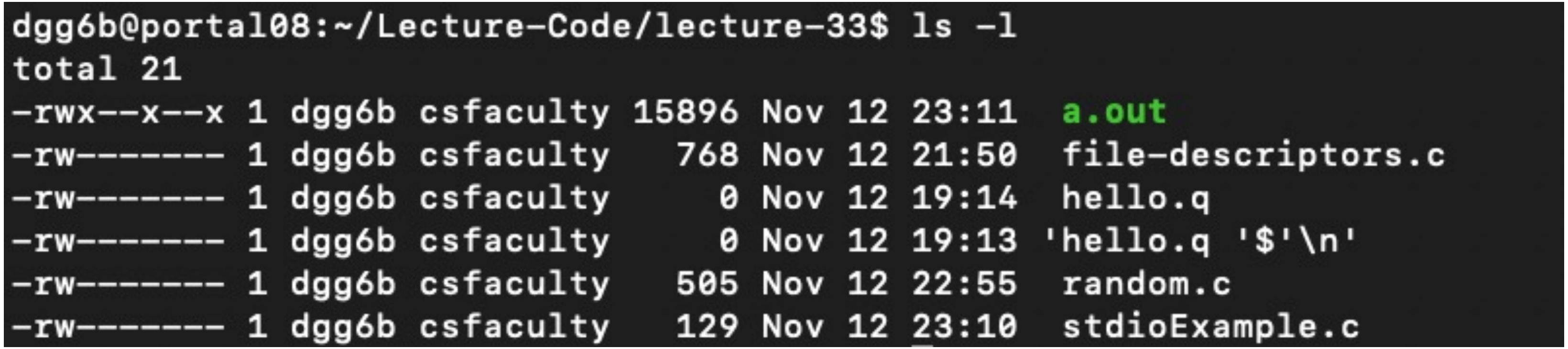
-rw-r--r--

Linux Permission as a number

6 4 4

First, the system checks to see if the user owns the file, if they are not then the system checks if they are a group own of the file. If not apply the other permissions.

EXAMPLE PERMISSIONS



```
dgg6b@portal08:~/Lecture-Code/lecture-33$ ls -l
total 21
-rwx--x--x 1 dgg6b csfaculty 15896 Nov 12 23:11  a.out
-rw------- 1 dgg6b csfaculty   768 Nov 12 21:50  file-descriptors.c
-rw------- 1 dgg6b csfaculty     0 Nov 12 19:14  hello.q
-rw------- 1 dgg6b csfaculty     0 Nov 12 19:13 'hello.q '$'\n'
-rw------- 1 dgg6b csfaculty   505 Nov 12 22:55  random.c
-rw------- 1 dgg6b csfaculty   129 Nov 12 23:10  stdioExample.c
```

user group size Last updated

MEMORY ERRORS

See if can spot the code errors in the following code segment

PUZZLE 1

```c
#include <stdio.h>

int main() {
    char *ptr;
    printf("%c\n", *ptr);
    return 0;
}
```

PUZZLE 1

```
#include <stdio.h>

int main() {
    char *ptr;
    printf("%c\n", *ptr);
    return 0;
}
```

Uninitialized Pointer

PUZZLE 2

```
#include <stdlib.h>

void function_that_forgets_to() {
    int *ptr = malloc(sizeof(int) * 100);
    *ptr = 123;
}

int main() {
    function_that_forgets_to();
    return 0;
}
```

PUZZLE 2

```c
#include <stdlib.h>

void
function_that_forgets_to() {
    int *ptr =
malloc(sizeof(int) * 100);
    *ptr = 123;
}

int main() {

function_that_forgets_to();
    return 0;
}
```

Memory Leak

Forgot to free

PUZZLE 3

```c
#include <string.h>

int main() {
    char small_buffer[5];
    strcpy(small_buffer, "This string is too");
    return 0;
}
```

PUZZLE 3

```
#include <string.h>

int main() {
    char small_buffer[5];
    strcpy(small_buffer, "This string is too");
    return 0;
}
```

Buffer Overflow

PUZZLE 4

```
#include <stdlib.h>
#include <stdio.h>
int main() {
    int *ptr = malloc(sizeof(int));
    *ptr = 10;
    free(ptr);
   fprintf(“Something fun”);
   free(ptr);
    return 0;
}
```

PUZZLE 4

```
#include <stdlib.h>
#include <stdio.h>
int main() {
    int *ptr = malloc(sizeof(int));
    *ptr = 10;
    free(ptr);
   fprintf(“Something fun”);
   free(ptr);
    return 0;
}
```

Double free

This can lead to corruption of memory management structures

PUZZLE 5

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 10;

    printf("Value: %d\n", *ptr);
    free(ptr);

    printf("Value after free: %d\n", *ptr);

    return 0;
}
```

PUZZLE 5

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 10;  // Initialize allocated memory with a
value

    printf("Value: %d\n", *ptr); // Correct usage
    free(ptr);  // Free the allocated memory

    // Use after free - undefined behavior!
    printf("Value after free: %d\n", *ptr);

    return 0;
}
```

PUZZLE 6

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

int main() {
    char *buffer = (char *)malloc(10 * sizeof(char));
    snprintf(buffer, 17, "The Good old song”;
    free(buffer);
    return 0;
}
```

PUZZLE 7

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

#define ARRAY_SIZE 2   // Size of the array of pointers
#define ALLOC_SIZE 10  // Size of each dynamically allocated array

int main() {
      int *pointerArray[ARRAY_SIZE];
      for (int i = 0; i < ARRAY_SIZE; ++i) {
              pointerArray[i] = (int *)malloc(ALLOC_SIZE * sizeof(int));
              if (pointerArray[i] == NULL) {
                  perror("Memory allocation failed");
                   for (int j = 0; j < i; ++j) {
                      free(pointerArray[j]);
                  }
                  return 1; //EXIT_FAILURE

             }
      }

    free(pointerArray);
    return 0; //EXIT_SUCCESS
}
```

PUZZLE 7

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

#define ARRAY_SIZE 2   // Size of the array of pointers
#define ALLOC_SIZE 10  // Size of each dynamically allocated array

int main() {
      int *pointerArray[ARRAY_SIZE];
      for (int i = 0; i < ARRAY_SIZE; ++i) {
              pointerArray[i] = (int *)malloc(ALLOC_SIZE * sizeof(int));
              if (pointerArray[i] == NULL) {
                  perror("Memory allocation failed");
                   for (int j = 0; j < i; ++j) {
                      free(pointerArray[j]);
                  }
                  return 1; //EXIT_FAILURE

             }
      }

    free(pointerArray);  //Can’t free stack allocated variables
    return 0; //EXIT_SUCCESS
}
```

Tricky two errors
Didn’t free the
malloced arrays