**CS311L OS Lab: The Implementation Guide**

This guide focuses on *how* to implement the concepts from labs 1-8, with minimal theory and maximum practical examples.

**Labs 1 & 2: Shell, Files & Permissions**

**Common Command Patterns**

|  |  |
| --- | --- |
| **Task** | **Command** |
| List contents with permissions & hidden files | ls -la |
| Navigate to your home directory | cd ~ or cd |
| Navigate one directory up | cd .. |
| Navigate to the root directory | cd / |
| Copy a file src.txt into backup dir | cp src.txt ./backup/ |
| Copy a whole directory Project to Project\_Backup | cp -r Project Project\_Backup |
| Rename old.txt to new.txt | mv old.txt new.txt |
| Move report.pdf into the Documents dir | mv report.pdf ./Documents/ |
| Create a new empty file | touch new\_file.log |
| Create a new directory | mkdir my\_app |
| Remove a single file (force, no prompt) | rm -f file.txt |
| Remove a directory and all its contents | rm -r my\_app |
| View the last 20 lines of a log file | tail -n 20 error.log |
| View the first 5 lines of a file | head -n 5 data.csv |
| Count lines in a file | wc -l file.txt |
| Get help for any command | man ls (press q to quit) |

**chmod: Changing Permissions**

**Use Case 1: Make a script executable (Most Common)**

# Gives owner (u) execute (x) permission

chmod u+x my\_script.sh

# OR (Set rwx for owner, r-x for group, r-x for others)

chmod 755 my\_script.sh

**Use Case 2: Make a file read-only for everyone**

# Removes write (w) permission for all (a)

chmod a-w config.ini

# OR (Set r-- for owner, r-- for group, r-- for others)

chmod 444 config.ini

**Use Case 3: Make a file private (owner only)**

# Set rw- for owner, and no permissions for group/others

chmod 600 private\_key.pem

**Wildcards: Matching Filenames**

|  |  |
| --- | --- |
| **Task** | **Command** |
| List all files ending in .c | ls \*.c |
| Remove all files starting with temp | rm temp\* |
| List files data1.txt, data2.txt, etc. | ls data?.txt |
| List files starting with a, b, or c | ls [abc]\* |
| List files starting with a number | ls [0-9]\* |

**Lab 3: Grep, Redirection & Piping**

**I/O Redirection Patterns**

* **File Descriptors:** 0 = stdin (keyboard), 1 = stdout (screen), 2 = stderr (errors to screen)

|  |  |
| --- | --- |
| **Task** | **Command** |
| Save command output to file.txt (overwrites) | ls -l > file.txt |
| Add command output to end of log.txt (appends) | date >> log.txt |
| Use file.txt as input for a command | sort < names.txt |
| Save **errors only** to errors.log | find / -name "f" 2> errors.log |
| Save **output** to out.txt, discard **errors** | find / -name "f" > out.txt 2>/dev/null |
| Save **output** to out.txt, save **errors** to err.txt | find / -name "f" > out.txt 2> err.txt |

**Piping | Patterns**

Piping sends the stdout of one command to the stdin of the next.

|  |  |
| --- | --- |
| **Task** | **Command** |
| Find the number of running chrome processes | `ps aux |
| Find all files in your home dir modified recently | `ls -l ~ |
| Sort a list of files by size (largest first) | `ls -l |

**grep Patterns (Searching *inside* files)**

|  |  |
| --- | --- |
| **Task** | **Command** |
| Find "error" (case-insensitive) in log.txt | grep -i "error" log.txt |
| Find lines that *do not* contain "DEBUG" | grep -v "DEBUG" log.txt |
| Count the number of warnings | grep -c "Warning" log.txt |
| Find lines that start with "User" | grep "^User" /etc/passwd |
| Find lines that end with "bash" | grep "bash$" /etc/passwd |
| Find the exact word "root" (not "rootkit") | grep -w "root" /etc/passwd |
| Find lines containing any number | grep "[0-9]" data.csv |

**Labs 4 & 5: C Compilation, Args & Memory**

**GCC Compilation Patterns**

|  |  |
| --- | --- |
| **Task** | **Command** |
| Simple compile | gcc my\_program.c -o my\_program |
| Compile with a library (e.g., math) | gcc my\_calc.c -o my\_calc -lm |
| Compile with Pthreads library | gcc my\_threads.c -o my\_threads -lpthread |
| Compile main.c and utils.c together | gcc main.c utils.c -o my\_app |
| Compile separately (good for large projects) | gcc -c main.c (creates main.o)  gcc -c utils.c (creates utils.o)  gcc main.o utils.o -o my\_app |

**Command Line Arguments (int main(int argc, char \*argv[]))**

#include <stdio.h>

#include <stdlib.h> // for atoi()

// To run: ./a.out 10 25

int main(int argc, char \*argv[]) {

// argc is 3

// argv[0] is "./a.out"

// argv[1] is "10"

// argv[2] is "25"

if (argc != 3) {

fprintf(stderr, "Usage: %s <num1> <num2>\n", argv[0]);

return 1; // Exit with error

}

int num1 = atoi(argv[1]); // Convert string "10" to int 10

int num2 = atoi(argv[2]); // Convert string "25" to int 25

printf("Sum: %d\n", num1 + num2);

return 0;

}

**Error Handling (errno and perror)**

Always check return values of system calls (open, fork, malloc, etc.)!

#include <stdio.h>

#include <errno.h> // for errno

#include <string.h> // for strerror

FILE \*f = fopen("non\_existent\_file.txt", "r");

if (f == NULL) {

// 'errno' is set automatically by fopen()

// Method 1: perror (easiest)

perror("Error opening file");

// Prints: "Error opening file: No such file or directory"

// Method 2: strerror (more control)

fprintf(stderr, "Error %d: %s\n", errno, strerror(errno));

return 1;

}

**Dynamic Memory (malloc & free)**

#include <stdlib.h> // for malloc() and free()

#include <stdio.h>

int main() {

int n = 10;

int \*arr;

// 1. Allocate memory for 10 integers

arr = (int\*) malloc(n \* sizeof(int));

// 2. ALWAYS check for failure

if (arr == NULL) {

perror("malloc failed");

return 1;

}

// 3. Use the memory

arr[0] = 5;

printf("%d\n", arr[0]);

// 4. ALWAYS free the memory when done

free(arr);

return 0;

}

**Structs: Stack vs. Heap (Dynamic)**

#include <stdlib.h>

#include <string.h>

typedef struct {

char name[50];

int id;

} Student;

int main() {

// --- On the STACK (automatic) ---

Student s1;

s1.id = 101;

strcpy(s1.name, "Alice");

// Use . (dot) to access members

printf("Stack student: %s\n", s1.name);

// --- On the HEAP (dynamic) ---

// 1. Allocate

Student \*s2 = (Student\*) malloc(sizeof(Student));

if (s2 == NULL) { return 1; }

// 2. Use

s2->id = 102; // Use -> (arrow) to access members of a pointer

strcpy(s2->name, "Bob");

printf("Heap student: %s\n", s2->name);

// 3. Free

free(s2);

return 0;

}

**Lab 6: I/O System Calls (Low Level)**

* **Headers:** <fcntl.h>, <unistd.h>
* **Identifier:** int fd (File Descriptor).
* **Constants:** STDIN\_FILENO (0), STDOUT\_FILENO (1), STDERR\_FILENO (2)

**open() Patterns (Returns fd, or -1 on error)**

#include <fcntl.h>

#include <stdio.h> // for perror

// Pattern 1: Open for reading

int fd\_read = open("file.txt", O\_RDONLY);

if (fd\_read == -1) {

perror("open (read) failed");

}

// Pattern 2: Create for writing (overwrite/truncate)

// 0644 = permissions (rw-r--r--)

int fd\_write = open("out.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd\_write == -1) {

perror("open (write) failed");

}

// Pattern 3: Append to a log file (create if new)

int fd\_log = open("app.log", O\_WRONLY | O\_CREAT | O\_APPEND, 0644);

if (fd\_log == -1) {

perror("open (log) failed");

}

**read() / write() / close() Pattern (File Copy)**

This is the standard way to read a file in a loop until it's done.

#include <unistd.h>

#include <fcntl.h>

#include <stdio.h>

char buffer[1024];

ssize\_t bytes\_read; // 'ssize\_t' is a signed int, can hold -1 for error

int fd\_in = open("source.txt", O\_RDONLY);

if (fd\_in == -1) { perror("open source"); return 1; }

int fd\_out = open("dest.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd\_out == -1) { perror("open dest"); return 1; }

// Loop: read returns > 0 bytes, 0 at end-of-file, -1 on error

while ( (bytes\_read = read(fd\_in, buffer, 1024)) > 0 ) {

// Write the exact number of bytes that were read

write(fd\_out, buffer, bytes\_read);

}

close(fd\_in);

close(fd\_out);

return 0;

**Labs 7 & 8: Process Management**

**fork(): The Core Implementation Pattern**

This if/else block is the standard way to manage parent/child execution.

#include <stdio.h>

#include <unistd.h> // for fork(), getpid(), getppid(), execvp()

#include <sys/wait.h> // for wait()

#include <stdlib.h> // for exit()

pid\_t pid = fork(); // One process becomes two

if (pid < 0) {

// --- ERROR ---

perror("fork failed");

exit(1);

} else if (pid == 0) {

// --- CHILD PROCESS CODE ---

printf("I am the CHILD. My PID is %d, my parent's PID is %d\n",

getpid(), getppid());

// ... child does its work ...

exit(0); // Child must exit!

} else {

// --- PARENT PROCESS CODE ---

printf("I am the PARENT. My PID is %d, my child is %d\n",

getpid(), pid);

// ... parent often waits for child ...

wait(NULL); // Pause here until child terminates

printf("Parent: My child has finished.\n");

}

**exec(): How to Use It (in the Child)**

exec() **replaces** the child process. It *only* returns if it fails.

// --- INSIDE THE CHILD (pid == 0) BLOCK ---

// Pattern 1: execlp (list of args, searches PATH)

// First arg is program, last arg is NULL

execlp("ls", "ls", "-l", "/home", NULL);

// Pattern 2: execvp (array of args, searches PATH)

// char \*my\_args[] = {"ls", "-l", "/home", NULL};

// execvp(my\_args[0], my\_args);

// This line is ONLY reached if exec fails:

perror("exec failed");

exit(1); // Exit child with an error

**wait(): Reaping Zombies**

wait() is crucial for cleaning up child processes.

// --- INSIDE THE PARENT (pid > 0) BLOCK ---

// Pattern 1: Simple wait (don't care about exit status)

wait(NULL);

// Pattern 2: Get child's exit status

int status;

waitpid(pid, &status, 0); // Wait for a specific child (pid)

if (WIFEXITED(status)) {

// WIFEXITED checks if child terminated normally

int exit\_code = WEXITSTATUS(status);

printf("Parent: Child exited with code %d\n", exit\_code);

}

**How to Create Zombie vs. Orphan (for Demos)**

* **To create a ZOMBIE:**
* if (fork() == 0) {
* // CHILD
* printf("Child: I am exiting now.\n");
* exit(0);
* } else {
* // PARENT
* printf("Parent: I am sleeping, not waiting...\n");
* sleep(10); // Child is a zombie during this time
* wait(NULL); // Clean up zombie
* printf("Parent: I just cleaned up the zombie.\n");
* }
* **To create an ORPHAN:**
* if (fork() == 0) {
* // CHILD
* printf("Child: Parent is about to die. My PPID is %d\n", getppid());
* sleep(2); // Live longer than parent
* printf("Child: Parent is dead. My new PPID is %d\n", getppid());
* exit(0);
* } else {
* // PARENT
* printf("Parent: I am exiting immediately.\n");
* exit(0); // Die before child
* }