Cop3402-System-Software-Final-Exam-Cheat-Sheet

Final Review System Software

With Paul Gazzillo

All Multiple Choice

```
Thursday, 12/04/2025, 10:00 a.m. - 12:00 p.m.

32 Multiple Choice Questions

14 Questions Midterm (Old Content)
18 Questions After Midterm (Compiler) (New Content)

12 Pages of notes - 24 Sides of Pages
Laptop with only exam open
Blank Scratch Paper and Pencil
```

Refresher

File Systems

1️⃣ What makes the Unix file system "hierarchical"?

> A hierarchical file system organizes files in a **tree-like structure** with a single root directory `/`.
>
> /
> ├── home
> │ ├── alice
> │ └── bob
> └── etc

Navigation

1️⃣ What is the working directory and how do you display it?

> The **working directory** is the directory you are currently "in" while using the terminal. It determines the starting point for relative paths.

Command to display it:

```
pwd # pwd = print working directory
```

2️⃣ What is the Unix standard command to rename a file?

```
mv old_filename.txt new_filename.txt
```

3️⃣ What Unix standard command will show you the text of a file?

```
cat filename.txt    # Display the full file
less filename.txt   # View file page by page
more filename.txt   # Similar to less
```

4️⃣ What does `grep` do?

> `grep` searches for text patterns in files.
> With no < input it will read from stdin

```
grep "hello" file.txt
```

5️⃣ How do you change the working directory to your home directory?

```
cd # No arguments will also take to home directory
# or
cd ~ # Takes to home directory
```

6️⃣ What is the Unix command to delete a file?

```
rm filename.txt
rm -rf directory_path # can delete a full directory
```

7️⃣ How does the implementation of deleting a file work? Does it remove the file's contents from the storage medium?

> - When you delete a file in Unix:
>
>   1. The **directory entry** pointing to the file is removed.

> 2. The **file's inode** (metadata) is marked as free.
>
> 3. The **storage blocks** may not be immediately overwritten.
>
> - The file content **remains on the storage medium** until reused.
> - This is why deleted files can sometimes be recovered with special tools.

Processes / Advanced Processes

1️⃣ How do you redirect standard output or standard input of a bash command to a file?

Redirect standard output (stdout) to a file:

```
grep "pattern" file.txt > grep.txt
```

Redirect standard input (stdin) from a file:

```
grep "pattern" < file.txt
```

Example combining input and output:

```
grep "pattern" < input.txt > grep.txt
```

2️⃣ How do you redirect standard output from one command to another command's standard input?

Use a pipe (`|`) to connect commands:

```
find . -type f | wc -l
```

Editor (Emacs)

1️⃣ How do you edit files in Emacs?

To open a file in Emacs, use:

```
emacs filename.txt
```

2️⃣ How do you save a file in Emacs?

While in Emacs:

`C-x C-s`

- Hold `Ctrl` and press `x`, then press `Ctrl` + `s`
- Saves the current buffer to its file

3️⃣ How do you quit Emacs?

To quit Emacs:

`C-x C-c`

- Hold `Ctrl` and press `x`, then press `Ctrl` + `c`
- Emacs will ask to save any unsaved changes before quitting

Version Control (Git)

1️⃣ What git command copies commits from the local repository to the remote repository?

```
git push
```

> Sends your local commits to the remote repository (e.g., GitHub, GitLab)

```
Syntax: git push <remote> <branch>
Example: git push origin main
```

2️⃣ What git command copies commits from the remote repository to the local repository?

`git pull`

- Fetches commits from the remote repository and merges them into your local branch

- Syntax: `git pull <remote> <branch>`
  Example: `git pull origin main`

3️⃣ What git command stages a new file?

`git add filename.txt`

- Adds the file to the **staging area** in preparation for a commit
- To stage all files: `git add .`

4️⃣ What git command creates a log of the change to a staged file to the local repository?

`git commit -m "Commit message"`

- Commits the staged changes to the local repository
- The `-m` flag allows you to include a short commit message describing the changes

---

File Syscalls

1️⃣ Using the open syscall to open a path given in the string `char *filepath` variable.

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

// open file for reading only
int fd = open(filepath, O_RDONLY);
if (fd == -1) {
    perror("open");  // print error if open fails
}
```

```
O_RDONLY opens the file in read-only mode
```

2️⃣ How do you check for and terminate the program on an error with opening a file?

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

int fd = open(filepath, O_RDONLY);
if (fd == -1) {
    perror("open"); // print error message
    exit(EXIT_FAILURE); // terminate program
}
```

- `perror` prints a descriptive error
- `exit(EXIT_FAILURE)` terminates the program with a failure status

3️⃣ Using the read syscall, you already have an open file with descriptor `fd` . Read the first 200 bytes and print it to stdout.

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

char buffer[201]; // 200 bytes + null terminator
ssize_t bytesRead = read(fd, buffer, 200);

if (bytesRead == -1) {
    perror("read");
} else {
    buffer[bytesRead] = '\0'; // null-terminate string
    printf("%s", buffer); // print to stdout
}
```

- `read` returns the number of bytes actually read
- Always check for errors (`-1`)

4️⃣ What syscall can you use to find the size and number of hard-links of a file?

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

struct stat st;
if (stat("filename.txt", &st) == 0) {
    printf("Size: %ld bytes\n", st.st_size);
    printf("Hard links: %ld\n", st.st_nlink);
}
```

> `stat` retrieves metadata (size, permissions, timestamps, hard links, etc.)

5️⃣ What syscall can you use to find the name of a file?

- To read directory entries, use:

```
#include <dirent.h>

DIR *dir = opendir(".");
struct dirent *entry;
while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);  // prints file/directory name
}
closedir(dir);
```

- `d_name` contains the filename
- `readdir` reads one entry at a time

6️⃣ Print all files in a given directory, except "." and ".."

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

DIR *dir = opendir(".");
struct dirent *entry;

while ((entry = readdir(dir)) != NULL) {
    if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
        printf("%s\n", entry->d_name);
    }
}

closedir(dir);
```

- `strcmp` is used to skip `"."` and `".."` entries
- This prints only actual files or directories in the folder

Process, Pipe, Syscalls

1️⃣ Write code that uses Unix standard syscalls to create a new process that runs the `ls` command

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

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        execlp("ls", "ls", NULL);
        perror("execlp"); // only reached if exec fails
        exit(EXIT_FAILURE);
    } else {
        // Parent process waits for child
        wait(NULL);
    }
    return 0;
}
```

---

2 Write code that creates a new process, where the original process writes "parent" and the new process writes "child" to stdout

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

int main() {
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        write(1, "child\n", 6); // 1 = stdout
    } else {
        // Parent process
        write(1, "parent\n", 7);
    }
    return 0;
}
```

---

3 Write code that replaces the current process with the `stat` or `ls` command

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

int main() {
    execlp("ls", "ls", NULL); // replaces current process with ls
    perror("execlp");          // only reached if exec fails
    exit(EXIT_FAILURE);
}
```

- `execlp` replaces the current process image with a new program

4️⃣ Write a program that opens a pipe and reads from and writes to it

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

int main() {
    int pipefd[2];
    char buffer[20];
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process writes
        close(pipefd[0]);           // close read end
        write(pipefd[1], "Hello\n", 6);
        close(pipefd[1]);
    } else {
        // Parent process reads
        close(pipefd[1]);           // close write end
        read(pipefd[0], buffer, 6);
        write(1, buffer, 6);        // stdout
        close(pipefd[0]);
    }

    return 0;
}
```

5️⃣ Write a program that redirects the standard output to a file called `output.txt`

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

int main() {
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Redirect stdout to file
    if (dup2(fd, 1) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
    }

    close(fd);

    // Anything written to stdout goes to output.txt
    printf("Hello, redirected output!\n");

    return 0;
}
```

- `dup2(fd, 1)` replaces **stdout (file descriptor 1)** with the file descriptor `fd`
- All subsequent writes to `stdout` go to the file

Extras

> Directories are files, but they map filenames to the i_node number of the file in the system.
> How do you reference parent directories <- ".."
>
> open() <- Syscall
> fopen() <- C library call
>
> Version control
> git pull : Remote -> Local
> git push: Local -> Remote
>
> fork() create a new process in the unix world
> exec() Executes a new program
> dup2() is easy way to redirect stdout

Second Half Material

Source Code to Processes (Toolchain)

1️⃣ Know each phase of the toolchain, what it does, and their ordering

The typical **C/C++ toolchain** has the following phases:

Phase	Description	Input	Output
1. Preprocessing	Handles `#include`, `#define`, and macros	`.c` or `.cpp` source file	Expanded source code (`.i`)
2. Compilation	Translates preprocessed code to assembly	Preprocessed code (`.i`)	Assembly code (`.s`)
3. Assembly	Converts assembly code to machine code (object code)	Assembly code (`.s`)	Object file (`.o`)
4. Linking	Combines object files and libraries into an executable	Object files (`.o`) and libraries	Executable (`a.out` or custom name)
5. Loading	Loads the executable into memory and starts execution	Executable file	Running process in memory

Ordering:
Source code (.c/.cpp)
↓ Preprocessing
Preprocessed code (.i)
↓ Compilation
Assembly code (.s)
↓ Assembly
Object code (.o)
↓ Linking
Executable
↓ Loading
Running process

> **Preprocessing** happens first, handling macros and includes
> **Compilation** generates assembly instructions
> **Assembly** produces machine-readable object code
> **Linking** resolves references between files and libraries
> **Loading** places the program into memory for execution

Compilation Pipeline

```
# = Gotten Files

# C Source (.c)

Goes through C Preprocessor (cpp)

# Preprocessed C (.i)

Goes through C Compiler (gcc)

# Assembly (.s)

Goes through Assembler (as)

# Object File (.o)

Goes through Linker (id) -> Links 2
   > object files into 1 object file and
   > ensures the calls and callees match up

# Executable File

Goes through Loader (execve)

# Running File
```

Arithmetic

1️⃣ Understand and write both SimpleIR and assembly code for arithmetic

- **Arithmetic operations** include: addition, subtraction, multiplication, division, and modulo.
- Both SimpleIR and assembly represent these operations, but the syntax differs.

2️⃣ What assembly instructions perform arithmetic?

Operation	x86-64 Assembly Instruction
Addition	`add dest, src`
Subtraction	`sub dest, src`
Multiplication	`imul dest, src`
Division	`idiv src` (divides accumulator by src)
Modulo	`idiv src` (remainder stored in `rdx` after division)

- `dest` is typically a register or memory location

- Most arithmetic instructions update processor flags for comparisons

---

3️⃣ Write equivalent assembly code

Example: Compute `c = a + b`

```
mov rax, [a]    ; load value of a into rax
add rax, [b]    ; add value of b to rax
mov [c], rax    ; store result into c
```

Example: Compute `c = a * b`

```
mov rax, [a]
imul rax, [b]
mov [c], rax
```

4️⃣ Write equivalent SimpleIR code

Example: Compute `c = a + b`

```
c := a + b
```

Example: Compute `c = a * b`

```
c := a * b
```

Bonus Stuff

```
add
sub
imul
idiv <- Integer Division
```

Be able to write assembly code

x = x/7
x := x / 7

```
Access local variables with -offset(%rbp)
ex. x -> -16(%rbp)
```

Insert Operation from Codegen

Branching

1️⃣ Understand and write both SimpleIR and assembly code for branching

- **Branching** changes the flow of execution based on a condition.
- Both **assembly** and **SimpleIR** can implement conditional or unconditional branches.
- Typical use cases: `if` statements, loops, and comparisons.

2️⃣ What assembly instructions perform branching?

ASM Op	SimpleIR Op	Jump if...
je	=	Equal
jne	!=	Not equal
jl	<	Less than
jle	<=	Less than or equal
jg	>	Greater than
jge	>=	Greater than or equal
jmp		Unconditional jump

- Conditional jumps rely on **flags set by a prior comparison** (`cmp` instruction).

3️⃣ Write equivalent assembly code

Example:

```
if (a == b) {
    c = 1;
} else {
    c = 0;
}
```

Assembly (x86-64):

```
mov rax, [a]
cmp rax, [b]       ; compare a and b
je equal_label      ; jump if equal
mov rbx, 0
jmp end_label
```

```
equal_label:
mov rbx, 1
end_label:
mov [c], rbx
```

---

4 Write equivalent SimpleIR code

Example:
`if (a == b) { c = 1; } else { c = 0; }`

```
t1 := a
t2 := b
if t1 == t2 goto true_label
c := 0
goto end_label
true_label:
c := 1
end_label:
```

Pointers

1 Understand and write both SimpleIR and assembly code for pointers

- **Pointers** store the memory address of a variable.
- Assembly uses **register indirect addressing** to read/write via a pointer.
- SimpleIR can represent pointers using `load` and `store` operations with addresses.

---

2 Write equivalent assembly code

Example: C code:

```c
int a = 5;
int *p = &a;
*p = 10;
```

Assembly (x86-64) using register indirect addressing:

```
mov rax, [a] ; optional: load a into rax (not needed here)
lea rbx, [a] ; load address of a into rbx
mov dword [rbx], 10 ; store 10 at the memory address stored in rbx
```

Explanation:

- `lea rbx, [a]` loads the **address** of `a` into `rbx`
- `[rbx]` dereferences the pointer
- `mov [rbx], 10` writes `10` into the location pointed to by `rbx`

3️⃣ Write equivalent SimpleIR code

Example: C code:

```
int a = 5;
int *p = &a;
*p = 10;
int x = *p;
```

Simple IR:

```
p := &a; store the address of a in p
*p := 10; store 10 at the address pointed to by p
x := *p; load the value from the address pointed to by p into x
```

Bonus Stuff

> **Register Copy** (Load from Memory) mov (%rax) %rbx (%rax) <- Look up address
> Parenthesis mean look up address of this register

Dereference %rbp at -48 offset
mov -48(%rbp), %rax

Working with address
mov %rbp, %rax
add -48, %rax

> **Register Indirect mov**
> mov %rbx, (%rax) <- Store the value of rbx into
> the address of rax

Functions and Their Implementation (x86-64, System V ABI)

1️⃣ What are the contents of the stack frame in the x86-64 System V ABI?

A **stack frame** (activation record) typically contains:

- **Return address** – the instruction to return to after the function call
- **Saved base pointer** (`rbp`) – points to the previous stack frame
- **Local variables** – stored on the stack
- **Saved registers** – registers that the function must preserve according to the calling convention
- **Function parameters** (if not passed in registers)

Diagram:

```
High Memory
Function parameters (if spilled)
Saved registers
Local variables
Saved RBP
Return address
Low Memory (stack grows downward)
```

---

2 How are parameters passed in the x86-64 System V ABI?

- **First 6 integer/pointer arguments:** passed in registers: `rdi, rsi, rdx, rcx, r8, r9`
- **Additional arguments:** passed on the stack

Example:

```c
int add(int a, int b, int c);
```

- `a` → `rdi`
- `b` → `rsi`
- `c` → `rdx`

3 How are variables represented and accessed in memory?

- **Local variables** → stored on the stack (relative to `rbp`)
- **Global/static variables** → stored in data or BSS segment
- **Pointers** → store memory addresses, accessed via dereference (`[rbp-offset]` or `[register]`)

Example:

```
mov rax, [rbp-8]   ; load local variable at rbp-8 into rax
mov [rbp-16], rbx  ; store rbx into local variable at rbp-16
```

4️⃣ How is the stack frame managed?

- **Stack grows downward** (toward lower addresses)
- **Push/pop** instructions or `sub rsp, <size>` allocate space for local variables
- **RBP** is used as a stable frame pointer to reference local variables and function parameters

Typical steps:

1. Save previous `rbp`
2. Set `rbp` = `rsp`
3. Allocate space for locals (`sub rsp, size`)
4. Access locals via `[rbp-offset]`

5️⃣ What do the prologue and epilogue do?

Prologue: sets up the stack frame at the start of the function

```
push rbp        ; save caller's base pointer
mov rbp, rsp    ; set base pointer for this function
sub rsp, N      ; allocate space for local variables
```

Epilogue: cleans up the stack frame before returning

```
mov rsp, rbp    ; restore stack pointer
pop rbp         ; restore caller's base pointer
ret             ; return to caller
```

6️⃣ How are values returned from a function in the x86-64 System V ABI?

- **Integer/pointer return values:** `rax`
  Example:

```
mov rax, 42     ; return 42 from function
ret
```

- Caller reads the return value from `rax`

Bonus Stuff

Stackframes are important
What information is stored in stackframe

6 Parameters in Register, the rest in stack

Some Parameters are stored in Stackframe

Caller puts Parameters in Registers and Stackframe

Return Address is on the Stackframe <- Caller Puts it there

Base Pointer [**RBP** or **rbp**] is stored on Stack <- pointing at the current active function's stackframe, Pointing at the address of the old functions base pointer.

> Local variables are stored on the stack
>
> %rip <- Current line of execution

When making a new Function call, we make a new stack frame so we always have access to the previous stack frame from the previous functional call.

How are values returned from functions

> call f
>
> - First saves the return address
> - Then branches to f
>
> ret
>
> - ret <- Opcode pops retrives the return address
> - branches back to the return address