CS111 Midterm Review

Jay Chauhan & Poojan Pandya
Winter ‘25

1

Unix V6

Unix V6

- Disks are **sector-addressable**. Filesystems read **one block** at a time. (A block is 1 or more sectors. For Unix v6, it's one sector).
- An **inode** stores metadata about a single file or folder.
- Inodes are stored in blocks in the **inode table** starting at block 2 (after boot block and superblock). Inodes are 1 indexed
- If a file is large, it may use more blocks than can be stored in a single inode. In this case, we use **indirect addressing**.
- **Directories are stored as "files" too**.
  - The root directory is always inode number ("inumber") 1.
  - In Unix V6, the block contains an array of 16-byte **directory entries**.
  - Each directory entry stores a **14-byte name** and a **2-byte inumber**.
- Directory entries **map filenames to inode numbers**

Unix V6

If block sizes are **1024 (or 2^10) bytes** and inodes are **32 (or 2^5) bytes**, what percentage of the storage device should be allocated for the inode table if we never want to run out of inodes? Your answer can be approximate, and the 50-words-or-less defense of your answer should include the necessary math. Assume that all files require at least one block of payload, and assume a minimum storage device size of 1 terabyte (or 2^40 bytes).

Unix V6

If block sizes are **1024 (or 2^10) bytes** and inodes are **32 (or 2^5) bytes**, what percentage of the storage device should be allocated for the inode table if we never want to run out of inodes? Your answer can be approximate, and the 50-words-or-less defense of your answer should include the necessary math. Assume that all files require at least one block of payload, and assume a minimum storage device size of 1 terabyte (or 2^40 bytes).

One block of inodes can lead to at most 2^5 files, each with at least one block. **1 out of every 2^5 blocks must store inodes**. That's about 3%. (Arguments involving 16 bytes of dirent structure are also good, but remember that answer only needed to be approximate.)

2

File Manipulation

Manipulating Files

- Privileged operations (like operating on the file system) are done via the operating system, rather than in your programs. These are called system calls ("**syscalls**").
- Syscalls sometimes create **file descriptors**, which are merely indices into the **file descriptor table**.
  - You can think of file descriptors as "ticket numbers" that refer to files or other open resources (e.g. pipes) the current program has **open()**ed.
  - You can use them in the **read, write**, and **close** syscalls to read and manipulate files.
- The **file descriptor table** is stored *per process* (running program), while the **open file table** is global

Syscalls

- **Syscalls** have different function call semantics from normal functions, since they're handled by the system.
    - (This distinction prevents vulnerabilities in *your* code from granting privileged access to the entire system).
- You can use the **open, read, write,** and **close** syscalls to manipulate files and file descriptors.

File Manipulation

Write a function, writeBuf that uses the write function to print an entire buffer to a file descriptor (you are *not* allowed to use the dprintf function for this question).

```
/* Function: writeBuf
 * ------------------
 * Writes the contents of buf with size len to the file descriptor fdOut
 * @param fdOut - the file descriptor where the data will be written
 * @param buf - the buffer to write
 * @param len - the number of bytes in the buffer
 */
void writeBuf(int fdOut, char *buf, ssize_t len) {
    // TODO
}
```

File Manipulation

```
/* Function: writeBuf
 * ------------------
 * Writes the contents of buf with size len to the file descriptor fdOut
 * @param fdOut - the file descriptor where the data will be written
 * @param buf - the buffer to write
 * @param len - the number of bytes in the buffer
 */
void writeBuf(int fdOut, char *buf, ssize_t len) {
    ssize_t bytesWritten = 0;
    while (bytesWritten < len) {
        bytesWritten += write(fdOut, buf + bytesWritten,
                              len - bytesWritten);
    }
}
```

3

Multiprocessing & Pipes

Multiprocessing

- **Multiprocessing** allows us to spawn other processes that run at the same time.
- You can create a clone of your program by running **fork**.
  - This creates a full copy of your process - right down to the current line of code!
  - The original is the "parent" and the new one is the "child".
  - The only difference is the return value of **fork**: 0 if you're the child, and the PID (process ID) of the child if you're the parent.
  - **Both the parent and the child run at the same time, and there is no guarantee about the order in which they run**.
- You can use **waitpid** to stall until a child quits, and observe how it exited (i.e. see if it crashed).

Multiprocessing II

- Parents should *always* wait on child processes, because waiting cleans up ("reaps") information about children.
- **waitpid** returns the PID of the process it waited for.
  - You can wait on *any* **one child** by passing in -1 for "pid"
  - You can get information by using the "status" parameter.
  - We can check if a process **segfault**ed with **WIFSIGNALED** and **WTERMSIG**.

Multiprocessing III

- Most commonly, you'll want to create a **child process** that **runs a totally different program** to do something for you.
- You can achieve this by **fork**ing, and then using **execvp**.
- **execvp** (exec + vp; v for argv, p for path) takes in the path of a program, and a **null-terminated** array of arguments.
    - execvp never returns (the process is cannibalized)

Pipes

- **Pipes** allows processes to communicate!
- **Pipes** are a construct that allow you to read/write data like you would with a file, without needing to manage a file.
    - Relies on the fact that **file descriptors are kept when execvp'ing**.
        - Remember to pipe before you fork if you're sharing a pipe between a parent and a child!

Pipelines

- Create them using **pipe** or **pipe2**.
  - Returns 0 on success, -1 on error
  - populates the 2-elem fds array with 2 file descriptors
  - fds[0] is the "read end" of the pipe, fds[1] is the "write end".
    - "*You learn to read **before** you learn to write.*"
- Beware: Make sure to close the file descriptors you're not using - may lead to various issues, such as pipe stalling.
  - Child reads from pipe, but parent waits for child to finish before writing - Child will stall.
  - Child reads until there is nothing left, but write end of pipe is not closed everywhere - program will stall, read will only finish once it detects write is closed.
- We can also use **kill** to terminate another process (using its PID).

Pipeline and I/O Redirection

- **dup2** supports fd redirection - duplicates an open resource session from one file descriptor number to another - i.e. both fd will point to the same file table entry.
- We can use this feature to accomplish:
  - **I/O Redirection**: If fd 0/1/2 are changed, we can redirect STDIN/STDOUT/STDERR to be something else without the program knowing.
  - **Pipelines**: If we rewire a program's sequence of input/outputs so that a program B reads its inputs from program A's outputs, we create pipelines - i.e. the previous program feeds the input of the next program.

Multiprocessing

Believe it or not, sorting is still an active research field, and computer scientists develop new sorting algorithms frequently. Some sorting algorithms are better than others. For this problem, you will write sleep sort, a sorting algorithm literally first posted on an online internet forum.

Here is how the sleep sort algorithm works: the function loops through an unsorted vector of integers, forks a process, and sleeps, in seconds, equal to the integer itself. After the sleep is complete, it prints the number to stdout. Convince yourself that this will, indeed, print out a sorted list of integers, and then convince yourself that it is an incredibly inefficient way to sort a list of integers.

```
void sleepsort(vector<int> numbers) {

        // TODO

}
```

Multiprocessing

```
void sleepsort(vector<int> numbers) {
    for (int n : numbers) {
        pid_t pid = fork();
        if (pid == 0) {
            sleep(n);
            printf("%d\n", n);
            exit(0);
        }
    }

    for (size_t i=0; i < numbers.size(); i++) {
        waitpid(-1, NULL, 0);
    }
}
```

Multiprocessing

Consider the following C program and its execution. Assume all processes run to completion, all system and printf calls succeed, and that all calls to printf are atomic. Assume nothing about scheduling or time slice durations.

List all possible outputs.

```
int main(int argc, char *argv[]) {
  pid_t pid;
  int counter = 0;
  while (counter < 2) {
    pid = fork();
    if (pid > 0) break;
    counter++;
    printf("%d", counter);
  }

  if (counter > 0) printf("%d", counter);

  if (pid > 0) {
    waitpid(pid, NULL, 0);
    counter += 5;
    printf("%d", counter);
  }

  return 0;
}
```

Multiprocessing

Consider the following C program and its execution. Assume all processes run to completion, all system and printf calls succeed, and that all calls to printf are atomic. Assume nothing about scheduling or time slice durations.

List all possible outputs.

Possible Output 1: 112265
Possible Output 2: 121265
Possible Output 3: 122165

```
int main(int argc, char *argv[]) {
  pid_t pid;
  int counter = 0;
  while (counter < 2) {
    pid = fork();
    if (pid > 0) break;
    counter++;
    printf("%d", counter);
  }

  if (counter > 0) printf("%d", counter);

  if (pid > 0) {
    waitpid(pid, NULL, 0);
    counter += 5;
    printf("%d", counter);
  }

  return 0;
}
```

4

Multithreading

Threads

- Threads allow you to **run multiple functions simultaneously, within the same process.**
- This means that **all threads share the same address space**.
- Create threads using the **thread** constructor, which creates a **thread** object.
  - Example: **thread(functionName, arg1, ref(referenceArg2), ...)**
  - To pass arguments by reference, you must use the **ref** function.
- You can wait for a thread to finish using the **join** method, e.g. **myThread.join()**;

Race Conditions

- With everything sharing memory space, **many threads can compete for access to the same memory at once**, which causes **undefined behavior** (BAD).
- To avoid these "race conditions", you can use **mutex**es, which allow only **one** thread through it at a time!
  - Use these to block off "Critical Sections" of your code.
  - "Critical Section" = "Only one thread may execute this code at a time".
- **Deadlock** is when multiple threads get into a situation where they're **permanently waiting on shared resources**. Examples:
  - Thread A waiting on a mutex that is never unlocked.
  - Thread A and Thread B mutually waiting on each other to complete (cyclic waiting).

Race Condition Checklist

1. **Identify shared data that may be modified concurrently**. What shared data is used across threads, passed by reference or globally?
2. **Document and confirm an ordering of events that causes unexpected behavior**. What assumptions are made in the code that can be broken by certain orderings?
3. **Use concurrency directives to force expected orderings and add constraints**. How can we use mutexes, atomic operations, or other constraints to force the correct ordering(s).

5

Crash Recovery

Block Cache

- Since accessing the disk can be expensive, the **file system often caches blocks**
- These blocks are written back to the disk later (**delayed writes**), which leads to **better performance**, but **can cause data loss** if a crash occurs before a block is written back
- Cache has the freedom to **write back blocks in any order**

Crash Recovery Mechanisms

- fsck: runs on boot after a crash to scan & repair inconsistencies
  - Can take a very long time
  - Limited in what kinds of inconsistencies it can resolve

- Ordered-Writes: Doing operations in specific orders can avoid certain classes of problems
  - Example: initialize inode before adding it to a directory entry
  - General rules:
    - Initialize targets before creating references
    - Remove references before freeing resources

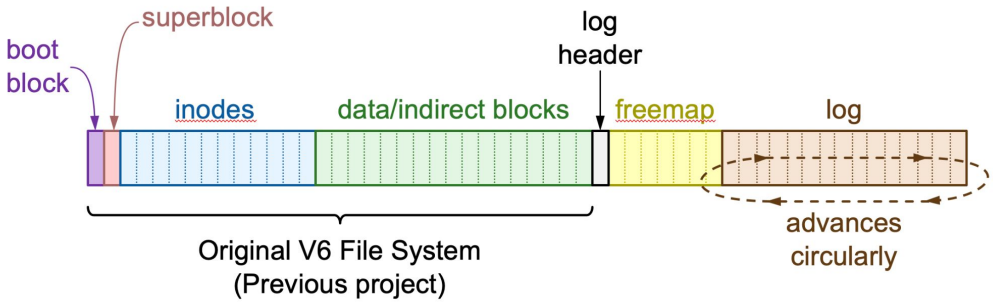
- Write-Ahead Logging (more info on next slide)

Write-Ahead Logging

- General idea: write **metadata** operations to a log **before** executing them
  - Example: "Adding block 4267 to inode 27" is written to log before performing the operation.

- **Importantly, the log does not track payload data!**

- Common A2 error: the free list tracks **blocks**, not **inodes**



Questions?