Chapter 1: Linux/UNIX Process and File Management

Gene Cooperman

1 Chapter: Linux/UNIX Process and File Management

In this chapter, we will demystify how the kernel manages execution and files using simple, familiar data structures: **arrays of structs**. By looking at the `proc.h` header from early UNIX-like systems (such as UNIX V6), we can see exactly how the operating system tracks every task.

The subsections in this chapter are:

1.1 Preliminaries

1.1.1 Program vs. Process

Before diving into the kernel's tables, it is important to distinguish between a **program** and a **process**:

- **The Program**: This is a passive entity—a file stored on your disk containing a set of instructions.
- **The Process**: This is an active entity—a "program in execution." Every process is currently running a program.
- **Multiple Processes, One Program**: It is entirely possible for two different processes to run the same program simultaneously. For example, if two different users both run the `grep` command, the kernel creates two separate process entries, even though they are executing the same set of instructions from the same file.

1.1.2 Understanding System Calls and Documentation

To understand how programs interact with a computer, it is helpful to think of the operating system as a service provider.

1.1.3 What is a System Call?

At its simplest level, a **system call (or syscall)** is just a **function call to the operating system**.

When your program needs to do something "privileged"—like reading a file from the disk, starting a new process, or sending data over a network—it cannot do it directly due to security restrictions. Instead, it calls a specific function that asks the **Operating System Kernel** to perform the task on its behalf.

1.1.4 Navigating the Documentation (`man` pages)

In Linux and Unix-like systems, you can look up the documentation for any of these functions using the `man` (manual) command. These manuals are divided into numbered sections to distinguish between different types of tools.

1.1.4.1 Section 2: System Calls

Section 2 is dedicated to **system calls** that go directly to the operating system kernel. * **Command:** `man 2 <name>` * **Example:** `man 2 fork` * **When to use:** Use this when you are looking for low-level operations that interface directly with the hardware or core OS management.

1.1.4.2 Section 3: Library Functions

Section 3 is used for **higher-level utilities** and library functions. These are often easier for programmers to use, but they frequently call Section 2 syscalls "under the hood" to do their work. * **Command:** `man 3 <name>` * **Example:** `man 3 printf` * **Comparison:** While `printf()` is a Section 3 library function, it eventually triggers the `write()` system call (Section 2) to actually put text on your screen.

1.1.4.3 Quick Reference Table

If you aren't sure which section a function belongs to, use this guide:

Section	Category	Purpose	Example
1	User Commands	Standard shell commands.	`man 1 ls`
2	**System Calls**	Functions that call the OS Kernel directly.	`man 2 open`
3	**Library Functions**	Higher-level C library functions.	`man 3 malloc`

Pro Tip: If you don't know which section a command is in, try typing `whatis <name>` (e.g., `whatis printf`) in your terminal to see all available manual sections for that keyword.

Pro Tip: If you want to also search in the one-line descriptions of all commands, try `man -k print` or `apropos print`. You can also scan the results more carefully by doing things like `apropos print | less`, or `apropos print | grep printf`.

Pro Tip: Some syscalls will use pointer arguments or return pointers. The '&myvariable' (ampersand for "address of") will produce a pointer to the address in memory of `myvariable`. If `myvariable` is already of pointer type, then you can use `*myvariable` to *dereference* it (i.e., to return the value of the address in memory that the 'myvariable' pointer points to. There is a nice description of pointers for C in the book, The C Book, by Mike Banahan, Declan Brady and

Mark Doran.

1.2 The Process Table: The Kernel's Master List

The kernel maintains a global **Process Table** to manage every program currently in execution. In the UNIX V6/xv6 model, this table is an **array of structs**, where each element is a `struct proc`.

1.2.1 The PID as an Index

In a standard C array, you access data using an index (e.g., `array[5]`). In an operating system, the **index into the process table** is what we call the **PID (Process ID)**. When you use a command like `kill 1234`, you are telling the kernel to look at index `1234` in its global array and update that specific struct.

1.2.2 Inside the Process Struct

Using the `struct proc` from early UNIX as a reference, we see several key fields that define a process:

- `pid`: The unique integer (the array index) identifying the process.
- `state`: An enum indicating if the process is `UNUSED`, `SLEEPING`, `RUNNING`, or `ZOMBIE`.
- `parent`: A pointer to the `struct proc` of the process that created this one.
- `name`: A simple 16-character string used to identify the program being run.
- `cwd`: A pointer to an `inode` representing the process's **Current Working Directory**.
- `context`: A sub-structure storing the **CPU registers** (such as `%edi`, `%esi`, `%ebx`, `%ebp`, and `%eip`) so the kernel can save and resume the process later.
- `ofile`: An inline array of pointers to open files. This is the process's local File Descriptor Table.

Understanding the `ofile` Array: In `struct proc`, the field `struct file *ofile[NOFILE]` is an inline array of pointers. In this context, a "pointer to an open file" refers to an abstract data structure that points to a specific entry in the **Global Open File Table** (detailed in Section 1.3). Essentially, this local array acts as a bridge between a specific process and the system-wide list of all open files.

1.2.3 Process Lifecycle: Fork and Exec

The relationship between the Process Table and system calls is best seen during process creation and transformation.

1.2.3.1 Creating Processes with `fork()`

When a program calls `fork()`, the kernel finds an empty slot in the **Process Table** array:

- **Duplication**: It creates a new child entry by copying the data from the parent's `struct proc`.
- **Inheritance**: The child's `ofile` array (file descriptors) is copied from the parent, so both now point to the same entries in the **Global Open File Table**.
- **Differentiation**: The kernel assigns the child its own unique `pid` (index) and sets its `parent` pointer to the original process.

1.2.3.2 Transforming Processes with `execvp()`

The `execvp()` call runs a new program without creating a new table entry:

- **Modification**: It modifies the *existing* process entry in the table.
- **The "Name" Field**: The kernel updates the `name` field to the new program's name.
- **Persistence**: Almost all other fields, including the `pid`, `parent`, and the `ofile` array, remain the same.

1.3 The Global Open File Table and Descriptors

While each process has its own `ofile` list, the kernel tracks every open file system-wide in the **Global Open File Table**.

- **The Global Table**: This is a global array of structs where each entry tracks information about an open resource.
- **The File Descriptor (FD)**: An FD is simply an index into the local `ofile` array inside a process's `struct proc`.

1.3.1 The Purpose of a Global Open File Table Entry

The entry in this table serves as a universal interface for I/O. The underlying data structure of an entry may point to:

- A physical file on a disk.
- A hardware device, such as a terminal or a network socket.
- A **pipe** used for communication between processes.

Furthermore, the entry stores **access modes** (defining if it can be used for reading, writing, or both) and the **file offset**.

1.3.2 Managing Files with System Calls

How the kernel manipulates these tables depends on the system call used:

- `open`: The kernel creates a **new entry in the Global Open File Table** pointing to the file or device. It returns an integer (`fd`) and modifies `ofile[fd]` in the process table to point to that new global entry.
- `close(fd)`: The kernel resets the `ofile[fd]` entry in the current process's table to **NULL**.
- `dup2(oldfd, newfd)`: The pointer in `ofile[oldfd]` is **copied** into `ofile[newfd]`.
- `dup(oldfd)`: The kernel finds the first available index and copies the pointer from `ofile[oldfd]` into it.

Crucially, the **`open` syscall is the only one** that can create a new entry in the Global Open File Table. Other syscalls like `close`, `dup`, or `dup2` simply modify the file descriptors (the `ofile` array) within the Process Table.

1.3.3 Reference Counting

An entry in the Global Open File Table is a struct containing a field called **`ref`**, used for a **reference count**.

1. When a syscall causes another file descriptor to point to an entry (such as `dup` or `dup2`), the `ref` field is incremented.

2. When `close(fd)` is called, the corresponding entry's `ref` field is decremented.
3. When `fork` is called, the child process copies all file descriptors from the parent; therefore, the `ref` count for every corresponding entry in the Global Open File Table is incremented.

When the reference count for an entry in the Global Open File Table is reduced to **0**, the operating system knows that no processes are currently using that resource. At this point, the kernel will remove that entry from the Global Open File Table to free up the space in the array for future `open` calls.

1.4 The Power of Indirection

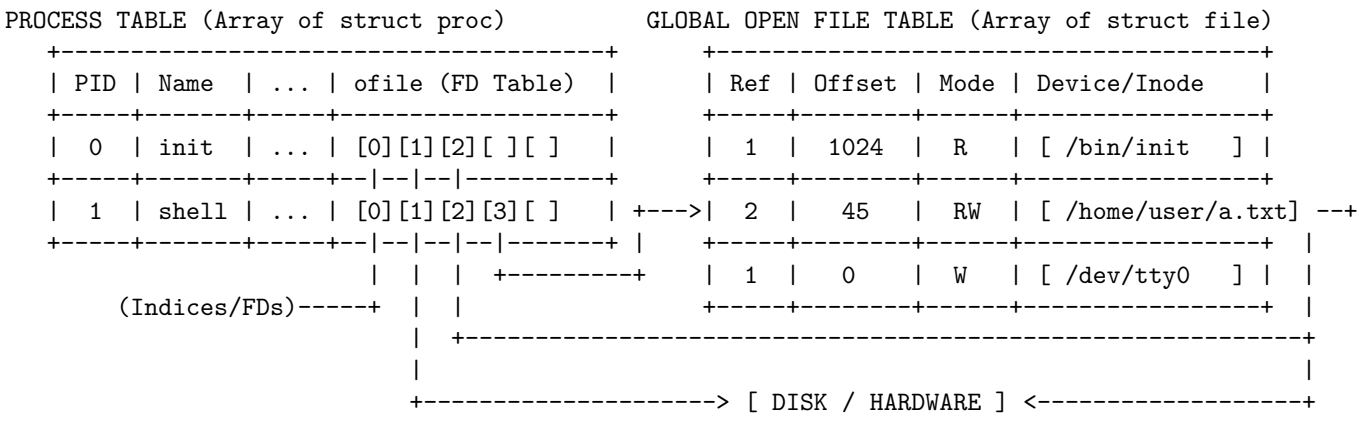
This entire architecture is a perfect illustration of David Wheeler's famous quote:

> **"All problems in computer science can be solved by another level of indirection."**

Technically, the kernel could have been designed so that a file descriptor in the process table pointed directly to a specific block on the disk. However, by adding the **Global Open File Table** as a level of indirection between the process and the hardware, the system becomes incredibly flexible. This indirection allows the kernel to easily manage many important programs, such as the **shell** (the program that reads your commands and prints output in a terminal window). Because the shell can manipulate pointers in the Process Table to point to different entries in the Global Open File Table, it can perform complex tasks like redirecting output from the screen to a file without the running program ever knowing the difference.

1.5 Architecture Diagram for the Operating System Kernel Tables

(This diagram is a work-in-progress. It was generated by an AI, with some errors.)



1.6 Review Questions

1. **Table Structure:** If the Process Table is an array of structs, what specifically does the `pid` represent in the context of that array?

2. **State Management:** Why must the kernel store a `context` (CPU registers) inside the `struct proc`? What happens to these values when a process is "sleeping"?
3. **Process Creation:** After a `fork()`, the parent and child have different PIDs but identical `ofile` arrays. If the parent calls `read()` and moves the file offset forward, will the child see this change? Why or why not?
4. **Program Execution:** When you call `execvp()`, the program changes but the file descriptors usually stay the same. Which field in the `struct proc` is explicitly updated to reflect the new program?
5. **System-Wide Tracking:** Why is the `open` system call unique compared to `dup` or `dup2` regarding the Global Open File Table?
6. **Resource Cleanup:** Explain the lifecycle of a `ref` count in the Global Open File Table. If a process has a file open and then calls `fork()`, what is the minimum value the `ref` count can be for that file entry?
7. **Identity:** If two different processes are running the same `.exe` file, how many entries exist in the Process Table? How many different `name` fields in those entries will be the same?
8. **Indirection:** Based on Wheeler's quote, how does having a Global Open File Table make it easier to implement output redirection (e.g., `ls > files.txt`)?