

Operating Systems Assignment 1 Report

Rohit Patidar
Roll No: 2024JCS2042

March 6, 2025

Contents

| | | |
|----------|---|----------|
| 1 | Introduction | 3 |
| 2 | Enhanced Shell for xv6 (10 Marks) | 3 |
| 2.1 | Objective | 3 |
| 2.2 | Steps Taken | 3 |
| 2.2.1 | Design the login function (Authentication): | 3 |
| 2.2.2 | Testing and Verification: | 4 |
| 3 | Shell Command: history (10 Marks) | 4 |
| 3.1 | Objective | 4 |
| 3.2 | Mechanism | 5 |
| 3.3 | System Call: sys-gethistory | 5 |
| 3.3.1 | Objective | 5 |
| 3.3.2 | Design Considerations | 5 |
| 3.4 | Kernel Modifications | 5 |
| 3.4.1 | Syscall Table and Headers | 5 |
| 3.5 | Global Data Structures | 5 |
| 3.6 | Shell Command Integration | 6 |
| 3.6.1 | Objective | 6 |
| 3.6.2 | Mechanism | 6 |
| 4 | Shell Command: block (10 Marks) | 6 |
| 4.1 | Introduction | 6 |
| 4.2 | Design and Implementation | 7 |
| 4.3 | System Call Handling | 7 |
| 4.4 | Kernel-Level Modifications | 7 |
| 4.5 | Shell-Level Modifications | 7 |
| 4.6 | Testing And Validation | 7 |

| | | |
|----------|--|----------|
| 5 | Shell Command: chmod (10 Marks) | 8 |
| 5.1 | Design Requirements | 8 |
| 5.1.1 | System Call Signature | 8 |
| 5.1.2 | Permission Storage | 9 |
| 5.1.3 | Error Handling | 9 |
| 5.2 | Implementation Details | 9 |
| 5.2.1 | Kernel-Level Implementation | 9 |
| 5.2.2 | Permission Checks in File Operations | 9 |
| 5.2.3 | User-Level Implementation | 10 |
| 5.3 | Filesystem Integration | 10 |

1 Introduction

This report covers the implementation methodology for Assignment 1 of the Operating Systems course, focusing on modifications and additions to the xv6 operating system.

2 Enhanced Shell for xv6 (10 Marks)

2.1 Objective

Implement an authentication mechanism in xv6 so that the shell is launched only after the user enters valid credentials (username and password)

- **Before Modification:** `init.c` directly launches the shell by calling `exec("sh", argv)`.
- **After Modification:** `init.c` first calls an authentication function (`login()`) to verify the username and password. If authentication is successful, the shell is executed; otherwise, the system does not exit `init` to avoid a kernel panic.

2.2 Steps Taken

2.2.1 Design the login function (Authentication):

- ```
int login(void) {
 char username[32];
 char password[32];
 int attempts = 0;

 while(attempts < ATTEMP) {
 printf(1, "Enter_username:");
 gets(username, sizeof(username));
 trim(username);

 if(strcmp(username, USERNAME) == 0) {
 printf(1, "Enter_password:");
 gets(password, sizeof(password));
 trim(password);

 if(strcmp(password, PASSWORD) == 0) {
 printf(1, "Login_successful\n");
 return 1; // Valid login
 } else {
 printf(1, "Incorrect_password\n");
 attempts++;
 }
 } else {
 printf(1, "Incorrect_username\n");
 }
 }
}
```

```
 attempts++;
 }
}
```

### 2.2.2 Testing and Verification:

A screenshot of a terminal window showing the boot process of the xv6 operating system. The text is as follows:  
Booting from Hard Disk...  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 log  
t 58  
Enter Username: abcdefg  
Incorrect username  
Enter Username: abcgrsegb  
Incorrect username  
Enter Username: rohit  
Enter Password: rohit  
Login successful  
init: starting sh  
\$ \_  
A mouse cursor is visible over the 'Login successful' line.

Figure 1: Enhanced Shell for xv6

## 3 Shell Command: history (10 Marks)

### 3.1 Objective

The objective is to implement an additional command history feature in the xv6 operating system. This feature will allow users to retrieve a list of all executed processes in chronological order based on their exit time. Each history entry will include the process ID (PID), the process name (or command), and its total memory utilization. This report outlines the process definition, design decisions, and system modifications required to integrate this feature.

## 3.2 Mechanism

- Before transitioning to the ZOMBIE state, each process stores its details in a global history array (e.g., `history_list`).
- A check is performed to ensure that the array has not exceeded its maximum capacity, defined by a constant (e.g., `MAX_HISTORY`).
- The timestamp (or sequence number) is recorded (using a global tick counter) to maintain the order in which processes exit.

## 3.3 System Call: `sys-gethistory`

### 3.3.1 Objective

The objective is to retrieve recorded history entries from the global history array, sort them in ascending order based on their creation timestamp, safely copy the sorted data from kernel space to user space, and return `-1` if any error occurs.

### 3.3.2 Design Considerations

The system call must sort the history list by timestamp (or sequence number) to maintain chronological order, perform proper error handling to safely copy data to user space (returning `-1` if any error occurs), and return either a success indicator or the number of history entries on successful execution.

## 3.4 Kernel Modifications

### 3.4.1 Syscall Table and Headers

To integrate the new system call into the kernel:

- Assign a unique syscall identifier (e.g., 22) for `sys_gethistory` in `syscall.h`.
- Add the function prototype in the relevant header files (e.g., `sysproc.h`) so that it is accessible within the kernel.
- Update the syscall table (in `syscall.c`) to map the new syscall identifier to its corresponding implementation.

## 3.5 Global Data Structures

A global array, such as `history_list`, is used to store history entries, each containing the PID, process name, total memory utilization, and a creation timestamp (or sequence number). A global counter (e.g., `nhistory`) tracks the number of recorded entries.

## 3.6 Shell Command Integration

### 3.6.1 Objective

Enable a user-level command (e.g., `$ history`) to retrieve and display the history of executed processes by invoking the new system call.

### 3.6.2 Mechanism

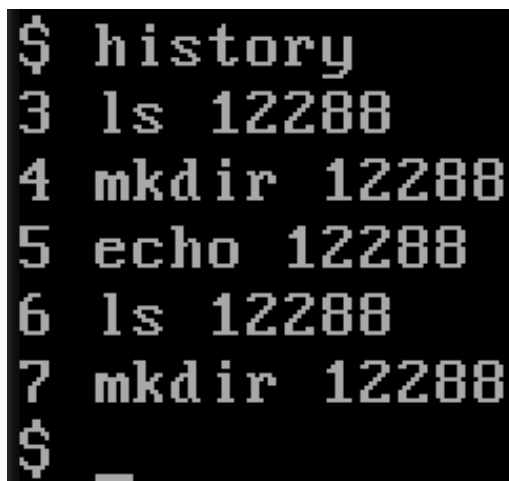
A terminal window with a black background and white text. The prompt is '\$'. The user has entered 'history'. The output shows a list of commands with their corresponding PID and memory usage. The commands are: 'ls 12288', 'mkdir 12288', 'echo 12288', 'ls 12288', and 'mkdir 12288'. The prompt is now '\$ \_'.

Figure 2: history command

When the shell receives the history command, it invokes the `sys_gethistory` system call, which returns the sorted history data. The shell then formats and prints the output in a user-friendly manner, typically displaying the PID, command, and memory usage.

**Testing:** This structured approach ensures that every executed process is properly recorded and that users can retrieve a complete and ordered history of commands

## 4 Shell Command: block (10 Marks)

Blocking and unblocking system calls:

### 4.1 Introduction

The aim is to implement two shell commands, **block** and **unblock**, in the xv6 operating system to allow a user to block or unblock specific system calls for all processes spawned by the current shell. This mechanism is implemented using two new system calls, `sys_block(int syscall_id)` to block and `sys_unblock(int syscall_id)` to unblock system calls, with

the system call identifiers defined in `syscall.h` as `SYS_block = 23` and `SYS_unblock = 24`.

## 4.2 Design and Implementation

### 4.3 System Call Handling

To support the blocking mechanism, a new field `blocked_syscalls` is added to the `proc` structure in `proc.h`. This array maintains the block state of system calls:

- `blocked_syscalls[syscall_id] = 1` → System call is blocked.
- `blocked_syscalls[syscall_id] = 0` → System call is unblocked.

The `syscall()` function in `syscall.c` is modified to check if a system call is blocked before execution:

- If blocked, it prints "syscall X is blocked" and returns -1.
- Otherwise, it executes normally.

### 4.4 Kernel-Level Modifications

The new system calls are implemented in `sysproc.c`:

- `sys_block(int syscall_id)`: Sets `blocked_syscalls[syscall_id] = 1`.
- `sys_unblock(int syscall_id)`: Sets `blocked_syscalls[syscall_id] = 0`.

The initialization of the `blocked_syscalls` array is done in the `allocproc()` function inside `proc.c`.

### 4.5 Shell-Level Modifications

Since `block` and `unblock` are shell-specific commands, they are implemented inside `sh.c` in a similar manner to the `cd` command. Screen shot of Block and unblock command.

### 4.6 Testing And Validation

: Execution:

- `$ block 22` → Calls `sys_block(22)`, blocking syscall 22.
- `$ unblock 22` → Calls `sys_unblock(22)`, unblocking syscall 22.

```
Enter Username: rohit
Enter Password: rohit
Login successful
init: starting sh
$ echo hii
hii
$ echo hii bro
hii bro
$ history
3 echo 12288
4 echo 12288
$ block 22
syscall 22 is now blocked
$ history
syscall 22 is blocked
history: -1 error retrieving history
$ unblock 22
syscall 22 is now unblocked
$ history
3 echo 12288
4 echo 12288
$
```

Figure 3: block And unblock command

## 5 Shell Command: chmod (10 Marks)

subsectionOverview This report outlines the design and implementation of a new shell command `chmod` that modifies file permissions via the system call `sys_chmod` defined as `int sys_chmod(const char* file, int mode);`. File permissions are encoded as a 3-bit integer where bit 0 (0x1) allows read, bit 1 (0x2) allows write, and bit 2 (0x4) allows execution.

### 5.1 Design Requirements

#### 5.1.1 System Call Signature

The system call `sys_chmod` is defined as `int sys_chmod(const char* file, int mode);` and takes two parameters: `file` (the name of the file whose permissions are modified) and `mode` (a 3-bit integer specifying the desired permissions).



### 5.1.2 Permission Storage

Instead of adding a new field for file permissions, the implementation reuses the existing `minor` field of an inode. This approach is viable as long as it does not conflict with the functionality of device files.

### 5.1.3 Error Handling

If any error occurs (e.g., file not found, improper arguments), the system call returns -1. Additionally, subsequent file operations (read, write, and execute) verify the permission bits and print an error message if the operation is not permitted:

Operation read failed , Operation write failed , Operation execute failed

## 5.2 Implementation Details

### 5.2.1 Kernel-Level Implementation

The `sys_chmod` System Call In `sysproc.c`, the system call is implemented as follows:

```
int
sys_chmod(void)
{
 char *path;
 int mode;
 struct inode *ip;

 if(argstr(0, &path) < 0 || argint(1, &mode) < 0)
 return -1;

 begin_op();
 if((ip = namei(path)) == 0){
 end_op();
 return -1;
 }
 ilock(ip);
 ip->minor = mode; // Store permissions in the minor field.
 iupdate(ip);
 iunlockput(ip);
 end_op();
 return 0;
}
```

### 5.2.2 Permission Checks in File Operations

To enforce file permissions, modifications were made in key system calls. In `sys_read`, the file's read permission is verified by checking if bit 0 of the inode's `minor` field is set; if not, it prints "Operation read failed" and returns -1. Similarly, `sys_write` checks bit 1 for write permission, and `exec` checks bit 2 for execute permission, printing the corresponding

error messages ("Operation write failed" or "Operation execute failed") and aborting the operation if the required permission is not present.

### 5.2.3 User-Level Implementation

The system call is exposed to user space by making concise modifications across several files: in `syscall.h`, `SYS_chmod` is defined as 25; in `usys.s`, the system call stub is added via the `SYSCALL(chmod)` macro; and in `user.h`, the prototype `int chmod(const char *file, int mode);` is declared. These changes collectively ensure that the new `chmod` system call is accessible from user-level applications.

## 5.3 Filesystem Integration

The inode structure in `file.h` and the on-disk inode in `fs.h` remain largely unchanged, except that the `minor` field is now used for permission bits. During inode allocation (e.g., in `ialloc` within `fs.c`), the `minor` field should be initialized with a default permission value (for example, 0x7 to allow all operations). Below is the execution of `chmod`.

```
$ echo "hello rohit" > file1
$ cat file1
"hello rohit"
$ chmod file1 0
$ cat file1
Operation read failed
cat: cannot open file1
$ chmod file1 7
$ cat file1
"hello rohit"
$ _
```

Figure 4: `chmod` command