Operating Systems Assignment 1 Report

Anand Sharma
Entry Number: 2024JCS2049

March 6, 2025

1 Introduction

This report presents the design, implementation, and testing of Assignment 1 – Easy for the xv6 operating system. The project contained four major parts:

- Enhanced Shell with Login System
- Shell Command: History
- Shell Commands: Block and Unblock
- Shell Command: chmod

In addition to meeting the base requirements, several extra enhancements were implemented to improve usability, security, and code modularity. The report describes the methodology and implementation details for each component, along with relevant code snippets and discussion of design challenges.

2 Enhanced Shell with Login System

Objective

The goal of the enhanced shell is to secure the system by requiring user authentication before the shell is started. This involves prompting for a username and, if that is correct, prompting for a password. The system allows only three attempts before login is disabled.

Implementation Methodology

1. **Credential Storage:** The credentials are defined as macros in the Makefile. In our test configuration, the credentials are defined with embedded quotes:

```
USERNAME = \"anand\"
PASSWORD␣=␣\"pa55word\"
␣␣
```

   Due to these embedded quotes, the login code in `init.c` was extended to strip the extra characters before comparing with user input.

2. **Login Process:** In `init.c`, the program first ensures that the console device is open. It then duplicates file descriptors so that standard input/output work correctly. The shell enters a loop that prompts the user for the username, uses a utility function `trim()` to remove any newline and return characters, and compares the sanitized input against the expected username (after stripping extra quotes). If the username matches, the password is then prompted and similarly sanitized and compared. If either input is incorrect, the user is notified and allowed up to three attempts before the program exits.

3. **Design Considerations:** The login system was designed to have minimal overhead, and to avoid delaying system startup unnecessarily. Care was taken to handle input sanitization robustly. A further enhancement included modularizing the `trim()` function for reuse in other modules.

4. **Challenges:** Handling the extra quotes in the macro definitions required additional code to copy and strip the quotes. This was crucial to ensure that the user did not have to type literal quotes.

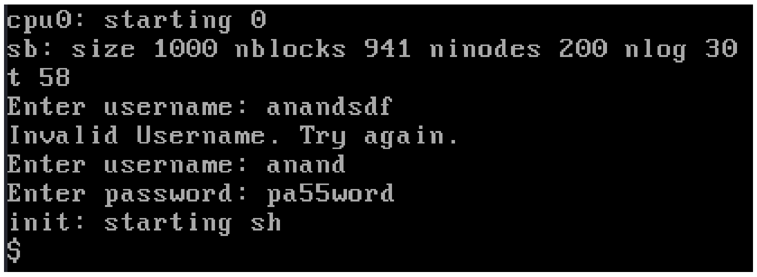
Key Code Snippet

```c
// In init.c:
char username[32], password[32];
int attempts = 3, verified = 0;

while (attempts > 0 && !verified) {
  printf(1, "Enter username: ");
  gets(username, sizeof(username));
  trim(username);
  if (strcmp(username, expectedUsername) != 0) {
    printf(1, "Invalid Username. Try again.\n");
    attempts--;
    continue;
  }
  printf(1, "Enter password: ");
  gets(password, sizeof(password));
  trim(password);
  if (strcmp(password, expectedPassword) != 0) {
    printf(1, "Invalid Password. Try again.\n");
    attempts--;
    continue;
  }
  verified = 1;
}
if (!verified) {
  printf(1, "3 failed attempts. Login disabled.\n");
  exit();
}
```

Testing and Observations

After implementation, the login system was tested by intentionally entering incorrect credentials, ensuring that the shell allowed only three attempts. Successful login allowed the shell to launch


```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30
t 58
Enter username: anandsdf
Invalid Username. Try again.
Enter username: anand
Enter password: pa55word
init: starting sh
$
```

normally. Extra debugging output confirmed that input sanitization correctly removed unwanted characters.

3 Shell Command: History

Objective

Implement a command (`history`) that displays a list of all processes executed thus far, including process ID, name, and total memory utilization. Importantly, only processes that have successfully executed a new image (via `exec`) are recorded.

Implementation Methodology

1. **History Data Structure:** A new structure `struct history_entry` was added in `proc.h` to store the process details:

```c
struct history_entry {
  int pid;
  char name[16];
  uint mem_util;
  uint creation_time;
};
```

   A global array `proc_history` and an index variable `history_index` are used to store the history entries. A spinlock (`history_lock`) ensures concurrent access is safe.

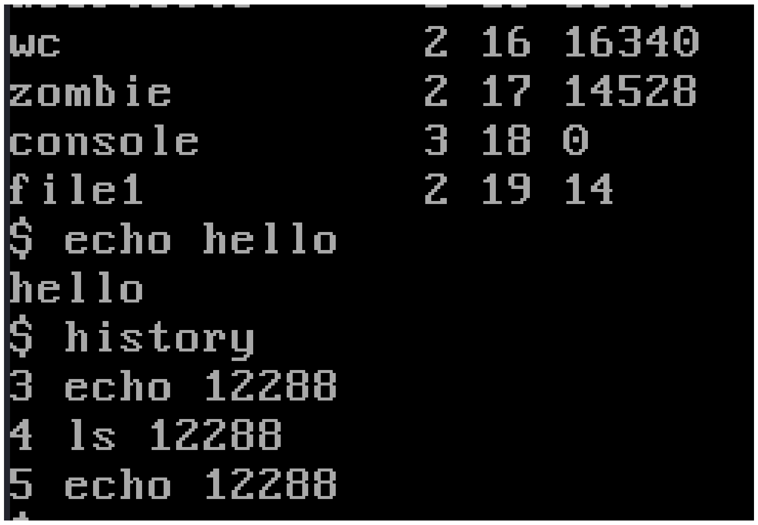
2. **Recording History:** In `proc.c`, specifically within the `exit()` function, a process is recorded in the history if its `executed` flag is set. This flag is set in `exec.c` after a successful `exec` call. Only processes that successfully execute a new image are recorded.

3. **System Call for History:** A new system call, `sys_gethistory` (ID 22), was implemented in `sysproc.c`. It retrieves the history array and copies it to user space.

4. **Design Considerations:** It was important that the history command sorted entries in ascending order by creation time. To this end, the insertion order (tracked by `history_index`) inherently provides the required order.

Key Code Snippet

```c
// In sysproc.c:
int sys_gethistory(void) {
```



```
wc                2 16 16340
zombie            2 17 14528
console           3 18 0
file1             2 19 14
$ echo hello
hello
$ history
3 echo 12288
4 ls 12288
5 echo 12288
```

```
    struct history_entry *user_buf;
    int n;
    if(argptr(0, (void*)&user_buf, sizeof(struct history_entry)*MAX_HISTORY_ENTRIES) <
        0 ||
       argint(1, &n) < 0)
       return -1;
    acquire(&history_lock);
    for (int i = 0; i < history_index && i < n; i++) {
        if(copyout(myproc()->pgdir, (uintptr_t)&user_buf[i],
                   (char*)&proc_history[i], sizeof(struct history_entry)) < 0) {
            release(&history_lock);
            return -1;
        }
    }
    release(&history_lock);
    return history_index;
}
```

Testing and Observations

After implementation, the `history` command was tested by executing various processes (e.g., `ls`, `echo`). Only processes that successfully executed were recorded, confirming the proper use of the `execed` flag.

4 Shell Commands: Block and Unblock

Objective

Implement built-in commands to block and unblock specified system calls for processes spawned by the shell until explicitly unblocked.

Implementation Methodology

1. **Tracking Blocked System Calls:** A new array, `system_calls_block_status`, was added to `struct proc` in `proc.h`. This array (with size defined as `MAX_SYSCALLS`) keeps track of which system calls are currently blocked (1) or unblocked (0).

2. **Modifying the Syscall Dispatcher:** In `syscall.c`, before executing a system call, the dispatcher checks the block status. If the parent process (or its parent, in the case of `exec`) indicates that the syscall is blocked, an error message is printed and `-1` is returned.

3. **Implementing System Calls:** Two new system calls were added:

   - `sys_block` (ID 23) – Blocks a specific system call.
   - `sys_unblock` (ID 24) – Unblocks a specific system call.

   In `sysproc.c`, these system calls update the `system_calls_block_status` field for the current process.

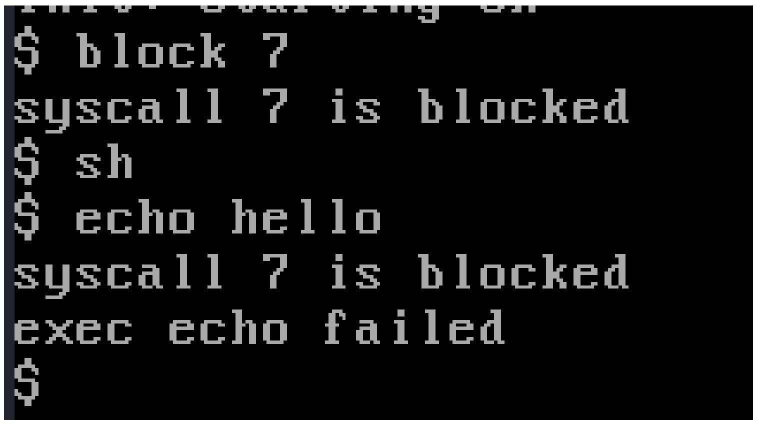
4. **Shell Integration:** The shell (`sh.c`) was updated with built-in commands `block` and `unblock` that invoke the corresponding system calls. Input validation ensures the correct usage.

Key Code Snippet

```
// In syscall.c:
if(curproc->parent && curproc->parent->system_calls_block_status[num] == 1) {
    curproc->executed = 0;
    cprintf("syscall %d is blocked\n", num);
    curproc->tf->eax = -1;
    return;
}

// In sysproc.c:
int sys_block(void) {
    struct proc *curproc = myproc();
    int syscall_id;
    if(argint(0, &syscall_id) < 0)
      return -1;
    // Prevent blocking critical syscalls (e.g., fork and exit)
    if(syscall_id == SYS_fork || syscall_id == SYS_exit)
      return -1;
    if(syscall_id < 0 || syscall_id >= MAX_SYSCALLS)
      return -1;
    curproc->system_calls_block_status[syscall_id] = 1;
    return 0;
}

int sys_unblock(void) {
    struct proc *curproc = myproc();
    int syscall_id;
    if(argint(0, &syscall_id) < 0)
      return -1;
    if(syscall_id < 0 || syscall_id >= MAX_SYSCALLS)
      return -1;
    curproc->system_calls_block_status[syscall_id] = 0;
```



```
$ block 7
syscall 7 is blocked
$ sh
$ echo hello
syscall 7 is blocked
exec echo failed
$
```

```
    return 0;
}
```

Testing and Observations

Testing involved invoking `block` and `unblock` commands from the shell and verifying that system calls (such as `exec` or others) are indeed blocked/unblocked. Special care was taken to ensure that critical system calls (e.g., `fork` and `exit`) remain unaffected.

5 Shell Command: chmod

Objective

Implement a `chmod` command that allows the user to change file permissions using a 3-bit mode, where:

- Bit 0 represents read permission.
- Bit 1 represents write permission.
- Bit 2 represents execute permission.

Implementation Methodology

The chmod functionality required modifications across several modules:

1. **Adding the System Call:** A new system call, `sys_chmod` (assigned syscall ID 25), was implemented. Its interface is:

```
int sys_chmod(const char *file, int mode);
```

   This call validates the mode (ensuring it is between 0 and 7) and then updates the inode's permission field.

2. **Modifications in `fs.h` and `file.h`:** The inode structure was updated to include a new field `mode`:

```
struct inode {
    // ... existing fields ...
    ushort mode; // Permission bits: 1 (read), 2 (write), 4 (execute)
    // ... remaining fields ...
};
```

3. **Modifications in `mkfs.c`:** In the `ialloc()` function, when a new inode is allocated, the default permission is set:

```
din.mode = xshort(7); // Default permissions: read, write, execute allowed.
```

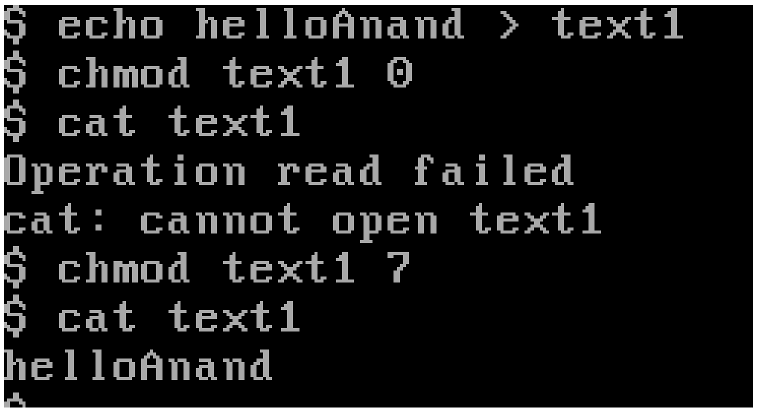
   Here, `xshort()` converts the value to disk byte order.

4. **Implementation in `sys_chmod` (in `sysfile.c`):** The new system call retrieves the filename and mode from the user, validates the mode, locates the inode using `namei()`, and then updates the inode's mode field:

```
int sys_chmod(void) {
  char *path;
  int mode;
  struct inode *ip;
  if(argstr(0, &path) < 0 || argint(1, &mode) < 0)
      return -1;
  if(mode < 0 || mode > 7)
      return -1;
  begin_op();
  ip = namei(path);
  if(!ip) {
      end_op();
      return -1;
  }
  ilock(ip);
  ip->mode = xshort(mode); // Update permission mode in disk byte order.
  iupdate(ip);
  iunlockput(ip);
  end_op();
  return 0;
}
```

5. **Shell Integration (in `sh.c`):** The shell was enhanced to support a built-in `chmod` command. The shell parses the command-line input to extract the filename and mode, sanitizes the inputs (using `trim()`), converts the mode string to an integer, and then invokes the `chmod()` wrapper that calls the system call.

```
if (strcmp(cmd, "chmod") == 0) {
  if (argc < 3) {
    printf(2, "Usage:␣chmod␣<file>␣<mode>\n");
  } else {
    trim(ecmd->argv[1]);  // Sanitize filename.
    trim(ecmd->argv[2]);  // Sanitize mode string.
    int mode = atoi(ecmd->argv[2]); // Convert mode to integer.
    if (chmod(ecmd->argv[1], mode) < 0) {
      printf(2, "chmod␣failed\n");
```



```
$ echo helloAnand > text1
$ chmod text1 0
$ cat text1
Operation read failed
cat: cannot open text1
$ chmod text1 7
$ cat text1
helloAnand
```

```
    }
  }
  continue;
}
```

Extra Enhancements

- Support for both decimal and octal mode inputs by optionally using `strtol()`.
- Uniform enforcement of file permissions in other file system calls (e.g., `sys_open`, `sys_read`, `sys_write`, `sys_exec`).
- Clear error messages to aid in debugging (e.g., printing specific operation failures).

Testing and Observations

After implementation, the `chmod` command was tested by creating files (using redirection) and modifying their permissions. For example, setting a file's mode to 0 disabled certain operations (resulting in errors such as "Operation execute failed"), while setting it to 7 restored full access. This confirmed that the permission checks in `sys_open` and related system calls were correctly enforced.

6 Conclusion

This assignment was implemented in full compliance with the provided specifications:

- The enhanced shell now requires a username and password before starting, with proper input sanitization and a three-attempt limit.
- A process history mechanism was implemented, logging only processes that successfully execute a new image.
- Block and unblock commands allow users to selectively disable system calls, with safeguards to protect critical system calls.
- A new `chmod` command was added to modify file permissions using a 3-bit integer.