# Operating Systems Assignment 1 Report

Anup Nayak
2022CS51827

March 6, 2025

# 1 Introduction

This report details the implementation of the following features in the xv6 operating system:

- **Login System**: A username-password-based login mechanism.
- **History Command**: A command to display the history of executed processes.
- **Block/Unblock Commands**: Commands to block and unblock system calls.
- **Chmod Command**: A command to modify file permissions.

The implementation methodology, relevant details, and code snippets are provided for each feature.

# 2 Login System

## 2.1 Implementation Methodology

The login system was implemented to protect the xv6 shell with a username-password-based authentication mechanism. The following steps were taken:

- Defined the username and password as macros in the `Makefile`.
- Modified `init.c` to prompt for a username and password before launching the shell.
- Limited the number of login attempts to three.

## 2.2 Relevant Details

- The username and password are hardcoded in the `Makefile` as follows:

```
USERNAME = <uname>
PASSWORD = <pwd>
CFLAGS += -DUSERNAME=\"admin\" -DPASSWORD=\"password\"
```

- The login logic is implemented in `init.c` as follows:

```
void login() {
    int attempts = 3;
    char uname[32], pass[32];

    while(attempts--) {
        printf(1, "Enter username:");
        gets(uname, sizeof(uname));
        uname[strlen(uname) - 1] = '\0';
        if(strcmp(uname, USERNAME) != 0) {
            continue;
        }
        printf(1, "Enter password:");
        gets(pass, sizeof(pass));
        pass[strlen(pass) - 1] = '\0';
        if(strcmp(pass, PASSWORD) == 0) {
            printf(1, "Login successful\n");
            return;
        }
    }
    while(1){
        sleep(100);
      }
}
```

# 3 History Command

## 3.1 Implementation Methodology

The history command was implemented to display a list of all executed processes, sorted by PID. The following steps were taken:
Created a global array to store process history.
Added a function to log processes to the history array.
Implemented a system call to retrieve and display the history.
Ensured that newly spawned shells inherit the same history.
Filtered out processes that executed unsuccessfully.
Sorted the history array by PID before displaying it.

## 3.2 Handling History Across Shells

To maintain a consistent history across shells, we ensured that whenever a new shell process was spawned using sh, it inherited the history from its parent shell. This was achieved by storing the history in a shared data structure that is accessible to child processes. By doing so, any new shell retains access to previously executed commands, ensuring continuity.

## 3.3 Filtering Unsuccessful Processes

For shell processes (sh), we filtered unsuccessful executions based on their memory size (sz). If a process named sh had a size different from 16384, it was considered unsuccessful and was not added to history. This prevents cluttering the history with failed shell executions.

The filtering logic was implemented as follows:

```
if(!(strcmp("sh", curproc->name) == 0 && curproc->sz != 16384)) {
add_to_history(curproc->pid, curproc->name, curproc->sz);
}
```

## 3.4 Sorting by PID for Chronological Order

Sorting by process ID (PID) is a valid method to maintain chronological ordering because each new process receives an incremented PID. Since the kernel assigns PIDs sequentially in the order of process creation, sorting by PID ensures that earlier processes appear first in history, effectively representing the timeline of process execution.

## 3.5 Relevant Details

- The history array is defined as follows:

```
struct proc_history {
  int pid;
  char name[16];
  uint memory_utilization;
};

#define MAX_HISTORY 100
struct proc_history history[MAX_HISTORY];
int history_count = 0;
```

- The `add_to_history` Function to add a process to the history:

```
void
add_to_history(int pid, char* name, uint memory_utilization)
{
  if (history_count < MAX_HISTORY) {
    history[history_count].pid = pid;
    safestrcpy(history[history_count].name, name,
    sizeof(history[history_count].name));
    history[history_count].memory_utilization = memory_utilization;
    history_count++;
  }
}
```

- The `gethistory` system call retrieves and displays the history:

```
int
gethistory(void)
{
  acquire(&ptable.lock);

  if (history_count == 0) {
    release(&ptable.lock);
    return -1;
  }

  for (int i = 0; i < history_count - 1; i++) {
    for (int j = 0; j < history_count - i - 1; j++) {
      if (history[j].pid > history[j + 1].pid) {
        struct proc_history temp = history[j];
        history[j] = history[j + 1];
        history[j + 1] = temp;
      }
    }
  }

  for (int i = 0; i < history_count; i++) {
    cprintf("%d-%s-%d\n", history[i].pid, history[i].name,
    history[i].memory_utilization);
  }

  release(&ptable.lock);
  return 0;
}
```

- The `exit` is modified to add history when process is successful:

```
if(!(strcmp("sh", curproc->name) == 0 && curproc->sz != 16384)){
  add_to_history(curproc->pid, curproc->name, curproc->sz);
}
```

# 4 Block/Unblock Commands

## 4.1 Implementation Methodology

The block and unblock commands were implemented to allow users to block and unblock specific system calls for processes spawned by the current shell. The following steps were taken:

- **Per-Process Blocking State**:
  - Each process maintains two arrays:
    - `blocked_syscalls`: Tracks which system calls are currently blocked for the process.

    * `pass_syscalls`: Tracks which system calls should be blocked for its child processes.
  - These arrays are stored in the `struct proc` structure.

- **System Call Dispatcher**:
  - The system call dispatcher (`syscall()`) checks if a system call is blocked before executing it.
  - If a system call is blocked, it prints a message and returns an error (`-1`).

- **Handling New Shells and Child Processes**:
  - When a new shell (`sh`) is spawned, it inherits the `blocked_syscalls` state from its parent process's `pass_syscalls`.
  - For normal processes (e.g., `ls`, `echo`), the `blocked_syscalls` state is copied from the parent process.

- **Preventing Critical System Calls from Being Blocked**:
  - The `fork` and `exit` system calls cannot be blocked to ensure system stability.

## 4.2 Relevant Details

### 4.2.1 Data Structures

- The `struct proc` was extended to include `blocked_syscalls` and `pass_syscalls` arrays:

```
#define MAX_SYSCALL 64

struct proc {
  int pass_syscalls[MAX_SYSCALL];
  int blocked_syscalls[MAX_SYSCALL];
};
```

### 4.2.2 System Call Dispatcher

- The system call dispatcher checks if a system call is blocked before executing it:

```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  num = p->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    if (p->blocked_syscalls[num] && p->pass_syscalls[num]) {
      cprintf("syscall %d is blocked\n", num);
      p->tf->eax = -1;
      return;
```

```
    }
    p->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->tf->eax = -1;
  }
}
```

### 4.2.3 Block and Unblock System Calls

- The `sys_block` and `sys_unblock` system calls update the `blocked_syscalls` array.
- Ensuring that a child process cannot unblock a system call that their parent has blocked.

```
int sys_block(void) {
  int syscall_id;
  if (argint(0, &syscall_id) < 0)
    return -1;

  if (syscall_id == 1 || syscall_id == 2)
    return -1;

  struct proc *p = myproc();
  p->blocked_syscalls[syscall_id] = 1;

  return 0;
}

int sys_unblock(void) {
  int syscall_id;
  if (argint(0, &syscall_id) < 0)
    return -1;

  struct proc *p = myproc();
  if(p->pass_syscalls[syscall_id] == 1){
    return -1;
  }
  p->blocked_syscalls[syscall_id] = 0;
  return 0;
}
```

## 4.3 Handling New Shells and Child Processes

- When a new shell (`sh`) is spawned, it inherits the `blocked_syscalls` state from its parent process:

```
for (int i = 0; i < MAX_SYSCALLS; i++) {
```

```
    if(curproc->blocked_syscalls[i] &&
    curproc->pass_syscalls[i]==0){
      curproc->pass_syscalls[i] = 1;
    }
  }
```

### 4.4 Preventing Critical System Calls from Being Blocked

- The `fork` and `exit` system calls cannot be blocked to ensure system stability:

```
if (syscall_id == 1 || syscall_id == 2)
  return -1;
```

## 5 chmod Implementation

### 5.1 Implementation Methodology

The `chmod` command was implemented to allow users to change the permissions of files and directories. The following steps were taken:

- **Permission Checking in System Calls**:
  - Modified `sys_read`, `sys_write`, and `sys_exec` to check for appropriate permissions before performing operations.
  - If the required permission is not set, the system call prints an error message and returns `-1`.
- **Inode Structure Modifications**:
  - Added a `mode` field to the on-disk inode structure (`struct dinode`) in `fs.h`.
  - Added a `mode` field to the in-memory inode structure (`struct inode`) in `file.h`.
- **Initialization and Synchronization**:
  - Initialized the `mode` field to `7` (read, write, and execute permissions) in `mkfs.c` and `fs.c`.
  - Ensured that the `mode` field is synchronized between the on-disk and in-memory inode structures.

### 5.2 Relevant Details

#### 5.2.1 Data Structures

- The `struct dinode` and `struct inode` were extended to include the `mode` field:

```
// On-disk inode structure
struct dinode {
  ...
  int mode;
```

```
  int padding[15];
};

// in-memory copy of an inode
struct inode {
 ...
  uint mode;
};
```

### 5.2.2 Permission Checking in System Calls

- Modified `sys_read`, `sys_write`, and `sys_exec` to check for permissions:

```
int sys_read(void) {
  ...

  if (!(f->ip->mode & 0x1)) { // 0x1 = Read
    cprintf("Operation read failed\n");
    return -1;
  }

  ...
}

int sys_write(void) {
  ...

  if (!(f->ip->mode & 0x2)) { // 0x2 = Write
    cprintf("Operation write failed\n");
    return -1;
  }

  ...
}

int sys_exec(void) {
  ...

  struct inode *ip = namei(path);
  if (ip == 0)
    return -1;

  ilock(ip);

  if (!(ip->mode & 0x4)) { // 0x4 = Execute
    iunlockput(ip);
    cprintf("Operation execute failed\n");
    return -1;
  }
```

```
  iunlockput(ip);

  ...
}
```

### 5.2.3 Initialization and Synchronization

- Initialized the `mode` field to `7` in `mkfs.c` and `fs.c`:

```
// In mkfs.c
din.mode = 7;

// In fs.c
dip->mode = 7; // in ialloc
dip->mode = ip->mode; // in iupdate
ip->mode = dip->mode; // in ilock
```

### 5.2.4 Function to handle the chmod system call

```
int
sys_chmod(void)
{
  char *file;
  int mode;

  if (argstr(0, &file) < 0 || argint(1, &mode) < 0) {
    return -1;
  }

  if (mode < 0 || mode > 7) {
    return -1;
  }

  struct inode *ip = namei(file);
  begin_op();
  if (ip == 0) {
    end_op();
    return -1; // File not found
  }

  ilock(ip);
  ip->mode = mode & 0x7;
  iupdate(ip);
  iunlock(ip);
  end_op();
  return 0;
}
```