COL331 Operating Systems Assignment 1

Abhinav Rajesh Shripad **Entry Number: 2022CS11596**

March 5, 2025

Enhanced Shell for xv6

- Implemented a `login` function in `init.c`, which reads the user input for the username and password. It then verifies the credentials against the actual password. If the maximum number of login attempts is exceeded, the function terminates the program by calling `exit()`.
- Declared few Macros like `MAX_ATTEMPT`, `USERNAME`, `PASSWORD`.
- Used inbuilt function `strcmp` to compare the strings.

Shell Commands (General)

To implement a shell command, modifications are typically required in specific files responsible for command parsing and execution. The key steps involved are as follows:

- Defining the System Call (if needed): If the command requires kernel-level functionality, a new system call must be added. This involves modifying:
  - `syscall.c` – Add an entry for the new system call.
  - `syscall.h` – Define a unique syscall number.
  - `sysproc.c` – Implement the system call logic.
  - `user.S` – Add an assembly wrapper to enable user-space access.
  - `user.h` – Declare the function prototype for the new system call so that user-space programs can invoke it.

Listing 1: Updating `main` in `proc.c`

```
int main(void){
    ...
    if (strncmp(buf, "history", 7) == 0) {
        gethistory();
        continue;
    }
    if (strncmp(buf, "block␣", 5) == 0) {
        int syscall_id = atoi(buf + 6);
        block(syscall_id);
        continue;
    }
    if (strncmp(buf, "unblock␣", 7) == 0) {
        int syscall_id = atoi(buf + 8);
        unblock(syscall_id);
        continue;
    }
    if (strncmp(buf, "chmod␣", 6) == 0) {
        ...
        if (arg1 && arg2 && *arg2) {
            int mode = atoi(arg2);
            chmod(arg1, mode);
        }
        ...
    }
    ...
```

This process ensures that the shell command is properly recognized, executed, and integrated within the xv6 environment.

Shell Command: history

Followed the same general procedures as shown above. Few more specific implementation details are:

- Created a `history_entry` in `proc.h`, which has attributes to store the name, PID, and memory usage of the processes.
- Created a function `log_process(struct proc* p)` that records each process as it executes. This function populates the `history` array with the data.
- Modified `proc.c` to maintain a buffer for storing the most recent process executions, ensuring older entries are overwritten when the buffer is full.
- Implemented a new system call `sys_gethistory()` in `sysproc.c` that allows user-space programs to retrieve the stored process history.

Listing 2: Definition of `history_entry` and `log_process` in proc.h

```
struct history_entry {
    int pid;
    char name[16];
    int memory;
};
extern struct history_entry history[MAX_HISTORY];
extern int history_count;
void log_process(struct proc *p);
```

Listing 3: Implementation of `log_process` in proc.c

```
struct history_entry history[MAX_HISTORY]; // Define history array here
int history_count = 0; // Define history count here
int blocked_syscalls[MAX_SYSCALLS] = {0};

void log_process(struct proc *p) {
    if (history_count < MAX_HISTORY) {
        if (strncmp(p->name, "block", 5) == 0)
            return;
        if (strncmp(p->name, "unblock", 7) == 0)
            return;
        if (strncmp(p->name, "history", 7) == 0)
            return;
        history[history_count].pid = p->pid;
        safestrcpy(history[history_count].name, p->name, sizeof(p->name));
        history[history_count].memory = p->sz; // Total memory usage
        history_count++;
    }
}
```

Listing 4: Implementation of `sys_gethistory` in `proc.c`

```
int sys_gethistory(void) {
    for(int i = 0; i < history_count; i++) {
        cprintf("%d␣%s␣%d␣\n", history[i].pid, history[i].name, history[i].memory);
    }
    return history_count;
}
```

Shell Command: block and unblock

Followed the same general procedures as shown above. Few more specific implementation details are:

- Created a 2 attributes `int blocked_syscalls_self[MAX_SYSCALLS]` and `int blocked_syscalls_child[MAX_SYSCALLS]` inside the `struct proc` in `proc.h`, to store if a process inherits which syscall blocks and which syscalls are blocked for his children.

- Updated the `fork` function in `proc.c`, to properly transfer the information of blocked syscalls from parent to its child.
- Implemented a new system call `sys_block(int syscall_id)` and `sys_unblock(int syscall_id)` in `sysproc.c` that updates the `block_syscalls` information for the process.

Listing 5: Updated Definition of `struct proc` in `proc.h`

```
#define MAX_SYSCALLS 30
...
struct proc {
  ...
  int blocked_syscalls_self[MAX_SYSCALLS];
  int blocked_syscalls_child[MAX_SYSCALLS];
};
```

Listing 6: Updated `fork` Method in `proc.c`

```
int fork(void) {
    ...
    /// np is child process, curproc is parent process
    for(int i = 0; i < MAX_SYSCALLS; i++){
        np->blocked_syscalls_self[i] = curproc->blocked_syscalls_self[i] | curproc->
            blocked_syscalls_child[i];
        np->blocked_syscalls_child[i] = curproc->blocked_syscalls_self[i] | curproc->
            blocked_syscalls_child[i];
    }
    ...
}
```

Listing 7: Implementation of `sys_block` and `sys_unblock`

```
int sys_block(void) {
    ...
    struct proc *p = myproc();
    p->blocked_syscalls_child[syscall_id] = 1;
    return 0;
}

int sys_unblock(void) {
    ...
    struct proc *p = myproc();
    p->blocked_syscalls_child[syscall_id] = 0;
    return 0;
}
```

Shell Command: chmod

Followed the same general procedures as shown above. Few more specific implementation details are:

- Created a attributes `int mode` inside the `struct inode` in `file.h`, to store if file modes.
- Updated the `sys_read`, `sys_write` and `sys_exec` function in `sysfile.c`, to check `int mode` for a file before executing.
- Implemented a new system call `sys_chmod(...)` in `sysfile.c` that updates the `int mode` information for the file.

Listing 8: Definition of `struct file` in `file.h`

```
struct file {
    ...
    short mode;
    ...
};
```

Listing 9: Implementation of System Calls

```
int sys_chmod(void) {
    ...
    ip->mode = mode & 0x7; // Store only the last 3 bits
    iupdate(ip);
    ...
}

int sys_read(void) {
    ...
    if (!(f->ip->mode & (1 << READ_BIT))) {
        cprintf("Operation read failed\n");
        return -1;
    }
    ...
}


int sys_write(void) {
    ...
    if (!(f->ip->mode & (1 << WRITE_BIT))) {
        cprintf("Operation write failed\n");
        return -1;
    }
    ...
}

int sys_exec(void) {
    ...
    if (!(ip->mode & (1 << EXEC_BIT))) {
        iunlockput(ip);
        end_op();
        cprintf("Operation execute failed\n");
        return -1; // Permission denied
    }
    ...
}
```