COL 331 Operating Systems
Assignment 1 – Easy

Aditya Sahu 2022CS11113

1 Enhanced Shell for xv6

The goal of this part is to authenticate the user before it can gain access. The username and password are defined as macros in the Makefile. The login is implemented in the initial user process code (`init.c`). When `init.c` is run, it performs the following tasks:

1. Prompts the user for username.
2. If the entered username is correct then prompts the user for password.
3. If the password is correct then the shell is started by executing `sh`.
4. If the user fails to authenticate in 3 attempts, the login process is disabled by using a `while` loop.

2 Shell Command: history

The goal is to maintain a list of all processes executed in the system (in chronological order), displaying each entry's process ID, name, and total memory utilization. The history is displayed only for processes that have completed execution.

2.1 Data Structures

Listing 1: History Entry Data Structure in proc.h

```c
struct history_entry {
    int pid;            // Process ID
    char name[16];      // Process name (max 16 characters)
    uint mem_usage;     // Total memory utilization (in bytes)
    uint completed;     // Flag indicating if the process has finished (1 if
        completed, 0 otherwise)
};

extern struct history_entry history[MAX_HISTORY]; // Global history array
extern int history_count; // Number of history entries stored
```

The data structure holds the process ID, name, and memory usage of each process. Additionally, we include a flag `completed` to indicate whether the process has finished execution (set to 1 when completed).

2.2 Adding to History

In `proc.c` we define the following function:

```c
void add_to_history(struct proc *p) {
    // Only add history for processes with pid > 2.
    // This prevents system processes (e.g., init and the original shell)
    // as well as built-in commands (like history, block, unblock) from being recorded.
    if (p->pid <= 2)
        return;

    if (history_count >= MAX_HISTORY) {
        for (int i = 1; i < MAX_HISTORY; i++) {
            history[i - 1] = history[i];
        }
        history_count--;
    }

    history[history_count].pid = p->pid;
    safestrcpy(history[history_count].name, p->name, sizeof(p->name));
    history[history_count].mem_usage = p->sz;
    history[history_count].completed = 0;  // Process not yet completed
    history_count++;
}
```

This function is called in the `exec` function in `exec.c` after the process's name is updated and the new process image is committed to memory.

2.3 Marking Completion

In the process exit routine (the `exit` function in `proc.c`), after a process finishes execution, we mark its corresponding history entry as completed:

```c
for (int i = 0; i < history_count; i++) {
    if (history[i].pid == curproc->pid) {
        history[i].completed = 1;
        break;
    }
}
```

2.4 System Call for History

The system call that retrieves and displays the process history is implemented as follows:

```c
int sys_gethistory(void){
    for (int i = 0; i < history_count; i++) {
        if (history[i].completed == 1) {
            cprintf("%d %s %d\n", history[i].pid, history[i].name, history[i].mem_usage);
        }
    }
    return history_count; // Return the number of history entries
}
```

2.5 Extra Details

- Only processes that have completed execution are displayed. For example, if the shell spawns a new shell and, in that shell, a command like `ls` is executed, only the `ls` process is recorded in the history since the shell process remains running and is not marked as completed.

3 Shell Command: block

The goal was to implement two new system calls:

- `sys_block(int syscall_id)` – Blocks the specified system call.
- `sys_unblock(int syscall_id)` – Unblocks the specified system call.

3.1 Data Structures

We maintain an array `blocked_syscalls[MAX_SYSCALLS]` (declared as an `extern` variable in `syscall.c`). Each index of this array corresponds to a system call number. If the value is 1, that system call is blocked; if 0, it is allowed.

3.2 New System Calls: `sys_block` and `sys_unblock`

Listing 2: Implementation of sys_block

```c
int sys_block(void) {
    int syscall_id;
    // Retrieve the syscall id from the user argument.
    if(argint(0, &syscall_id) < 0)
        return -1;
    // Validate the syscall id and ensure critical syscalls cannot be blocked.
    if (syscall_id < 0 || syscall_id >= MAX_SYSCALLS || syscall_id == 1 ||
       syscall_id == 2)
        return -1;
    // Mark the syscall as blocked.
    blocked_syscalls[syscall_id] = 1;
    return 0;
}
```

These functions are registered in the syscall table in `syscall.c` using their assigned syscall numbers.

3.3 Enforcement in the Kernel's syscall Dispatcher

The kernel's syscall dispatcher is modified to enforce system call blocking based on the global array `blocked_syscalls`. The following code snippet shows the key logic:

Listing 3: Syscall Enforcement in syscall.c

```
if (blocked_syscalls[num] && curproc->pid > 2) {
    // Additional mechanism: if the process's "to_be_blocked" flag is set, the syscall is denied.
    if (curproc->to_be_blocked == 1) {
        cprintf("syscall %d is blocked\n", num);
        curproc->tf->eax = -1;  // Return -1 to indicate the syscall is blocked.
        return;
    }
}
// For new child processes, when they call exec (syscall 7), set the flag to indicate
// they should be blocked later.
if (curproc->to_be_blocked == 0 && num == 7)
    curproc->to_be_blocked = 1;
```

In this code:

- `blocked_syscalls[num]` checks if the specific syscall (by its number) is marked as blocked.
- We exempt system processes such as the shell (pid 2) and the init process (pid 1) by checking that the current process's pid is greater than 2.

In the finer details we cannot block the syscalls that are directly spawned by the parent shell with pid 2. This means blocking the syscall 7 should not block commands like ls,echo because these are processes with pid 3 and name sh when they reach the syscall function. To manage this what I have done is that I have made a variable `to_be_blocked` . This is an attribute of the proc struct. For processes with pid greater than 2 if the syscall is blocked and if the `curproc -> to_be_blocked` is 1, then the syscall is blocked. The `to_be_blocked` variable is set as 1 when there is a sycall with num 7 for that process. This is because any child process of the parent process shell process calls exec syscall for its name to be changed and memory to be allotted. At that time we set the variable to be one indicating that any further syscalls if blocked by the user will actually get blocked.

3.4 User-Level Command Integration

In the shell (implemented in `sh.c`), the commands `block` and `unblock` are handled as follows:

Listing 4: User-Level Command Handling in sh.c

```
if (startswith(buf, "block ")) {
    int id = my_atoi(buf + 6);
    block_command(id); // block_command calls the system call wrapper for block.
    continue;
}
if (startswith(buf, "unblock ")) {
    int id = my_atoi(buf + 8);
    unblock_command(id); // Similarly for unblock.
```

```
    continue;
}
```

Here:

- The shell checks if the input command begins with "block " or "unblock ".
- It extracts the syscall ID using a custom conversion function (e.g., my_atoi()).
- Then, it calls the appropriate user-level function (block_command or unblock_command), which in turn invokes the corresponding system call.

4 Shell Command: chmod

The goal is to implement a chmod command for xv6 that modifies file permissions based on a 3-bit mode.

4.1 Data Structures

We extended the on-disk inode structure (struct dinode in fs.h) and the in-memory inode structure (in file.h) to include a permission field.

Listing 5: On-disk inode structure (fs.h)

```
struct dinode {
    short type;            // File type
    short major;           // Major device number (T_DEV only)
    short minor;           // Minor device number (T_DEV only)
    unsigned short nlink:13;  // 13 bits for link count (max 8191)
    unsigned short perm:3;    // 3 bits for permissions (0-7)
    uint size;             // Size of file (bytes)
    uint addrs[NDIRECT+1];  // Data block addresses
} __attribute__((packed));
```

Listing 6: In-memory inode structure (file.h)

```
struct inode {
    uint dev;              // Device number
    uint inum;             // Inode number
    int ref;               // Reference count
    struct sleeplock lock; // Protects all fields below
    int valid;             // Indicates if the inode has been read from disk?

    // Fields copied from the on-disk inode
    short type;
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT+1];
    uint perm;             // Permission bits
};
```

4.2 Inverted Permission Convention

The initial permission of all files should be 7 (i.e., read, write, and execute allowed). However, instead of initializing `perm` to 7 for every file, we change the convention as follows:

- A bit **set** in the `perm` field indicates that the corresponding permission is *not allowed*.
- A bit **not set** (i.e., 0) indicates that the permission is allowed.

Since `perm` is automatically initialized to 0, all permissions are allowed by default. When the user enters a mode via `chmod`, we convert it by taking XOR with 7. For example, if the user enters the mode `"010"`, then after XOR with 7 it becomes `"101"`. This conversion marks the read and execute permissions as disallowed (bits set to 1) while allowing write permission.

4.3 User-Level Command in Shell (sh.c)

The shell is modified to detect when the user enters a `chmod` command. The code extracts the filename and the mode from the input and then calls the user-level wrapper `chmod()`, which in turn invokes the corresponding system call to update the permission field in the inode.

4.4 Kernel Implementation: sys_chmod

The `sys_chmod` system call is implemented to modify file permissions based on a 3-bit mode. The implementation is as follows:

Listing 7: sys_chmod implementation

```
int sys_chmod(void) {
    char *file;
    int mode;
    struct inode *ip;
    // Retrieve the filename from the user.
    if(argstr(0, &file) < 0)
        return -1;
    // Retrieve the mode argument.
    if(argint(1, &mode) < 0)
        return -1;
    // Validate that mode is a 3-bit integer (0 to 7).
    if(mode < 0 || mode > 7)
        return -1;
    begin_op();
    if((ip = namei(file)) == 0) {
        end_op();
        return -1;
    }
    ilock(ip);

    // Invert the bits: if user enters mode, we XOR it with 7.
    mode ^= 7;
    // Update the permission field, ensuring only the lower 3 bits are stored.
```

```
    ip->perm = mode & 7;
    iupdate(ip);
    iunlockput(ip);
    end_op();
    return 0;
}
```

4.5 Permission Checks in System Calls

Permission checks are performed in the functions `sys_exec`, `sys_write`, and `sys_read` because the execution, read, and write operations ultimately rely on these functions. For example, in `sys_exec`, the following code snippet enforces the execute permission:

Listing 8: Permission Check in sys_exec

```
if ((ip->perm & 0x4)) { // Execute bit disallowed (bit is set)
    cprintf("Operation execute failed\n");
    iunlockput(ip);
    end_op();
    return -1;
}
```

Similar checks are added in `sys_open` and `sys_write` to deny operations if the corresponding read (0x1) or write (0x2) bits are set.

4.6 Extra Points and Design Considerations

When we attempted to add an extra permission field to the `dinode` structure, the overall size of the on-disk inode increased. The file system code depends on a fixed inode size (typically 64 bytes) to maintain a consistent disk layout, ensuring that `BSIZE` is an exact multiple of `sizeof(struct dinode)`. Adding a new full-sized field would disrupt this fixed size.

To resolve this, we observed that the existing `nlink` field only requires 13 bits to represent the link count (which is sufficient for our purposes). Thus, we split the 16-bit storage for `nlink` into two bit-fields:

```
unsigned short nlink:13;
unsigned short perm:3;
```

This technique preserves the overall size of the `dinode` while providing storage for the new `perm` attribute.

Moreover, in xv6 the on-disk inode (`struct dinode`) and the in-memory inode (`struct inode`) must remain coherent. Initially, when we added the permission attribute, updates to the permission bits were not consistently reflected between the two. To maintain coherence, we modified the update and locking routines (such as `iupdate()` and `ilock()`) so that the permission field is explicitly synchronized between the on-disk dinode and the in-memory inode.