Operating Systems Assignment-1 Report

Name: Pola Venkata Revanth
Entry Number: 2022CS51650

March 6, 2025

1 Enhanced Shell with Login System

1.1 Objective

Implement username/password authentication for xv6 shell with 3-attempt limit.

1.2 Implementation

- **Login Flow in init.c**:

```
while (attempts < 3 && !logged_in){
    printf(1, "Enter username: ");
    memset(username, 0, sizeof(username));
    gets(username, sizeof(username));

    if (strcmp(username, USERNAME) != 0){
        attempts++;
        continue;
    }

    printf(1, "Enter password: ");
    memset(password, 0, sizeof(password));
    gets(password, sizeof(password));

    if (strcmp(password, PASSWORD) != 0){
        attempts++;
        printf(1, "Invalid password. Attempts left: %d\n", 3 - attempts);
    }else{
        logged_in = 1;
    }
    if (!logged_in) {
        printf(1, "Maximum login attempts reached. Login disabled.\n");
        while(1) sleep(100);
    }
}
```

The login mechanism prompts the user for a username and password, allowing up to three attempts. If the username is incorrect, the attempt counter increases, and the user is prompted again. Upon entering the correct username, the system requests the password. If the password is incorrect, it notifies the user of the remaining attempts. If the correct password is provided, the logged_in flag is set, granting access. If all three attempts fail, the program displays a "Login disabled" message and exits, preventing further login attempts. This ensures basic security by limiting incorrect login attempts and restricting unauthorized access.

2 history Command

2.1 Objective

Log executed processes (PID, name, memory) after termination.

2.2 Implementation Details

2.2.1 Creating the System Call

The `history` command uses a custom system call `gethistory()`. The following process shows how the system call is implemented and managed.

- **user.h**: Declare user-space interface for system call, such that it tells user programs that gethistory exists.
  ```
  int gethistory(void);
  ```

- **usys.S**: Generate assembly code to invoke the syscall.
  ```
  SYSCALL(gethistory)
  ```

  When invoked, this macro:

  - Pushes syscall number (22) into %eax
  - Triggers software interrupt (int T_SYSCALL)
  - Transitions to kernel mode

- **syscall.h**: Assign a unique system call number
  ```
  #define SYS_gethistory 22
  ```

- **syscall.c**: Map to kernel function
  ```
  extern int sys_gethistory(void);
  [SYS_gethistory] sys_gethistory,
  ```

- **sysproc.c**: Kernel implementation
  ```
  int sys_gethistory(void) {
      // Implementation logic
      return phistory_count;
  }
  ```

2.2.2 Locking Mechanism

A spinlock protects the history array from race conditions:

```
1 // main.c
2 initlock(&phistory_lock, "phistory");
```

Why Needed: Prevents concurrent access when:

- Multiple processes exit simultaneously
- History command reads while processes update

2.2.3 Logging Process History

The struct used was:

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

The mem_usage and start_time were calculated in ‘exec.c‘ at the time of calling the ‘exec()‘ function. This following code was added to ‘exit()‘ in ‘proc.c‘ before calling the ‘sched()‘ function:

```
  acquire(&phistory_lock);
  if (phistory_count < MAX_HISTORY) {
    struct proc_history *ph = &phistory[phistory_count];
    ph->pid = curproc->pid;
    safestrcpy(ph->name, curproc->name, sizeof(ph->name));
    ph->mem_usage = curproc->init_mem;
    ph->start_time = curproc->start_time;
    phistory_count++;
  }
  release(&phistory_lock); // Release lock
```

2.2.4 Retrieving History

Before returning and printing the stored process history, the data is sorted based on `start_time` using Bubble Sort:

```
int sys_gethistory(void) {
acquire(&phistory_lock);
// .....
  for (int i = 0; i < count - 1; i++) {
    for (int j = 0; j < count - i - 1; j++) {
      if (phistory[j].start_time > phistory[j + 1].start_time) {
        // Swap the entries
        struct proc_history temp = phistory[j];
        phistory[j] = phistory[j + 1];
        phistory[j + 1] = temp;
      }
    }
  }
// ......
release(&phistory_lock);
}
```

After sorting, the system prints the process details, including `pid`, `name`, and `mem_usage`, before returning the history count.
Features:

- Only shows **exited** processes (logged in exit())
- Automatically skips system processes (init and main shell)
- Atomic operations protected by spinlock

3 block/unblock Commands

3.1 Objective

Block/unblock syscalls for child processes post-`exec`.

3.2 Implementation

- **Initialization**:

```
uint blocked_calls[MAX_SH] = {0};
int current_sh = 0;
```

The array `blocked_calls` is initialized to store blocked system calls, assuming a maximum of 80 processes. The variable `current_sh` keeps track of the number of currently running processes.

- **Blocking and Unblocking of System Calls**:

```
int block(int id) {
  if (id == 1 || id == 2 || id < 0 || id >= MAX_SYSCALLS) {
    return -1;
  }
  struct proc *curr_proc = myproc();
  if (strncmp(curr_proc->name, "sh", 2) != 0) return -1;
  blocked_calls[current_sh] |= (1U << id);
  return 0;
}

int unblock(int id) {
  // ... similar to block()
  blocked_calls[current_sh] &= ~(1U << id);
  return 0;
}
```

Each time a block or unblock system call is invoked, the corresponding bit in `blocked_calls` is set or cleared accordingly, and blocking of init(pid = 1) and primary shell (pid = 2) returns -1.

- **Syscall Enforcement**:

```
void syscall(void) {
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax; // This contains the system call number
  int present_top = current_sh;

  if (num == 7) { // exec() system call
    current_sh += 1;
    if (current_sh < MAX_SH) {
      blocked_calls[current_sh] = blocked_calls[current_sh - 1];
    }
    if (curproc->parent->pid > 2) present_top += 1;
  }

  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    if (present_top > 0 && (blocked_calls[present_top - 1] & (1U << num))) {
      cprintf("syscall %d is blocked\n", num);
      curproc->tf->eax = -1;
      return;
    }
    curproc->tf->eax = syscalls[num]();
  }
}
```

As seen in the `syscall()` function, whenever the `exec()` system call (i.e., `num == 7`) is called, a new process is assumed to be created. The variable `current_sh` is incremented, and it is decremented only when the `exit()` function in `proc.c` is executed. Additionally, the blocked system calls of the new process are inherited from the parent process.

According to the assignment specification, *"Block and unblock will only be enforced on the child process after it has been created (after the exec call by the primary shell (PID 2))."* Therefore, when the `current_pid` is greater than 2, the blocked system calls are checked from the corresponding shell. This is managed through using the variable present-top.

4 chmod Command Implementation

4.1 Inode Structure Modifications

4.1.1 On-Disk Inode (dinode)

Added permissions storage to persist across reboots:

```
struct dinode {
  // ...
  uchar mode;  // 3-bit permissions (rwx)
  uchar padding[/*calculated size*/];
};
```

Key Changes:

- `mode` field stores permissions (read/write/execute bits)
- Padding ensures fixed 128-byte structure size for block alignment

4.1.2 In-Memory Inode (struct inode)

Mirrors disk structure with matching `mode` field:

```
struct inode {
  // ...
  uchar mode;
};
```

4.2 File System Layer Changes

4.2.1 File System Operations

- **ialloc()** (fs.c): Initialize new inodes with default permissions
  ```
  dip->mode = 0b111; // Default: rwx for all
  ```
- **iupdate()** (fs.c): Sync memory state to disk
  ```
  dip->mode = ip->mode; // Persist permissions
  ```
- **ilock()** (fs.c): Load permissions from disk
  ```
  ip->mode = dip->mode; // Cache permissions
  ```

4.2.2 Disk Formatting (mkfs.c)

Set default permissions for root directory:

```
din.mode = 0b111; // Initial files get rwx
```

4.2.3 Permission Enforcement

Modified system calls to check inode permissions:

- **Permission Checks for System Calls implemented in sysfile.c and exec.c**:

exec() (Process Execution)

```
if(!(ip->mode & 0b100)) { // Execute permission check
    cprintf("Operation execute failed\n");
    return -1;
}
```

sys_read() (File Reading)

```
if(!(f->ip->mode & 0b001)) { // Read permission check
    cprintf("Operation read failed\n");
    return -1;
}
```

sys_write() (File Writing)

```
if(!(f->ip->mode & 0b010)) { // Write permission check
    cprintf("Operation write failed\n");
    return -1;
}
```

4.2.4 sys_chmod System Call

Added to modify file permissions via kernel interaction:

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

    // Get arguments from user space
    if(argstr(0, &path) < 0 || argint(1, &mode) < 0)
        return -1;

    // Validate 3-bit mode
    if(mode < 0 || mode > 7) return -1;

    struct inode *ip;
    begin_op();
    if((ip = namei(path)) == 0) { // Find file's inode
        end_op();
        return -1;
    }

    ilock(ip);
    ip->mode = mode;       // Update in-memory permissions
    iupdate(ip);          // Persist to disk
    iunlock(ip);
    end_op();

    return 0;
}
```

When a user executes the chmod command, the system call sys_chmod is triggered to modify file permissions. The function first retrieves the filename and mode arguments from user space and validates the mode to ensure it falls within the correct range (0-7). It then locates the file's inode using namei(path), which resolves the filename to an inode structure. Once the inode is found, it is locked using ilock(ip) to prevent concurrent

modifications. The mode field of the inode is updated in memory, and iupdate(ip) ensures the changes are written to disk. Finally, the inode is unlocked and released, and the function returns control to the user space, completing the permission change process. This modification affects how system calls like exec(), read(), and write() enforce access permissions based on the updated mode.

Summary of Chmod

- Validates mode range (0-7) using bitmask checks
- Uses `namei()` to resolve path to inode
- Atomic update sequence: lock → modify → persist → unlock
- Commits changes to disk via `iupdate()`

5 Extra Features

As an additional feature, I implemented a new system call, `sys_getprocinfo`, to aid in debugging by identifying blocked system calls for the current process.

5.1 Implementation of `sys_getprocinfo`

The system call retrieves and displays the name and process ID of the currently executing process. It then iterates through the system call table to check and print any blocked system calls.

```
int sys_getprocinfo(void) {
  struct proc *curr_proc = myproc();
  cprintf("Current process: %s (PID: %d)\n", curr_proc->name, curr_proc->pid);

  for (int i = 0; i < MAX_SYSCALLS; i++) {
    if (curr_proc->blocked_normal_calls[i] == 1) {
      cprintf("Blocked system call: %d\n", i);
    }
  }

  return 0;
}
```