Indian Institute of Technology, Delhi

ELL 783 – Operating Systems

Assignment 1 - Easy

Name: Aditya Prakash

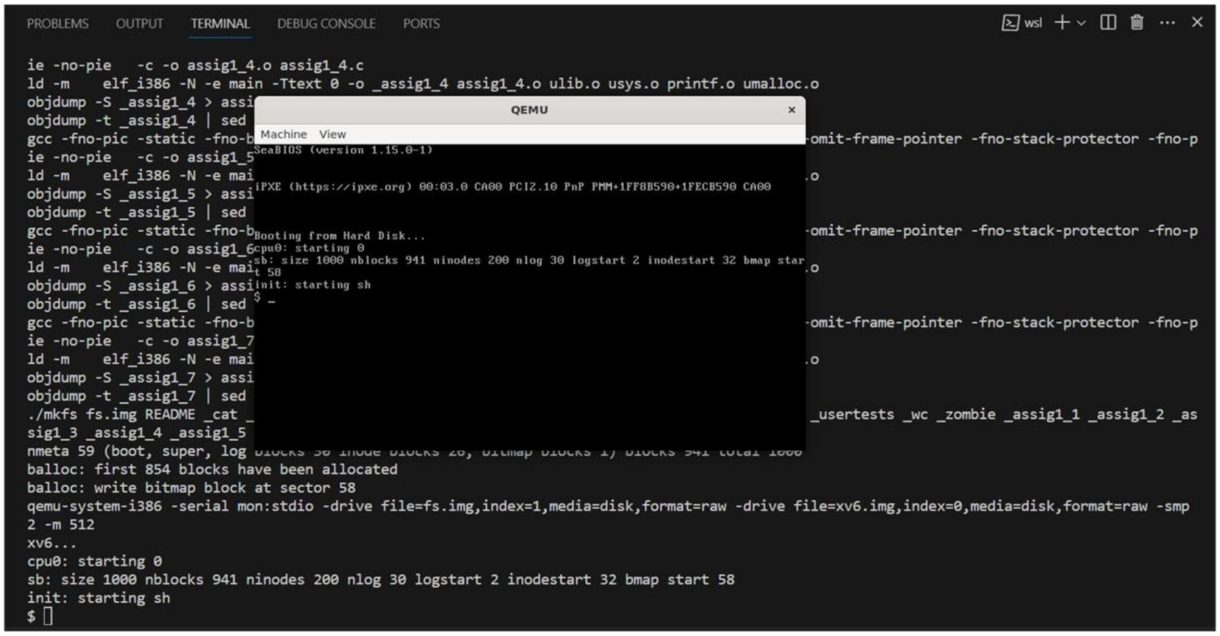
Entry No.: 2024EET2855

Course: M.Tech. (Computer Technology)

Department: Electrical Engineering

Supervisor: Prof. Smruti R. Sarangi

Installing and Testing Xv6:



Enhanced Shell for xv6

1. Introduction

This report details the implementation of an enhanced shell for xv6 that includes a username-password-based login system. The user must provide a correct username and password before accessing the shell. The credentials are stored as macros in the Makefile, and users have a maximum of three attempts to log in successfully.

2. Implementation Methodology

2.1 Modifying the Initialization Process to Require Authentication

The init.c file in xv6 was modified to prompt for a username and password before allowing access to the shell. This ensures that authentication takes place at system initialization.

2.2 Storing Credentials in the Makefile

The username and password are stored as macros in the Makefile as follows:

USERNAME=<username>

PASSWORD=<password>

These macros are passed as -DUSERNAME and -DPASSWORD compiler flags to the file.

2.3 Implementing Login Attempts

- The system allows three attempts for login.
- If an incorrect username is entered, the password is not requested.
- If the password is incorrect, the user is given another chance, up to three attempts.
- After three failed attempts, the system halts and does not proceed to the shell.

2.4 Hiding the Password Input

Since xv6 does not support advanced terminal features, password masking (e.g., displaying * instead of characters) was not implemented. However, the password input is still functional.

3. Pseudocode

Implemented the login function in init.c

```
void login()
{
    char username[20];
    char password[20];
    int attempts = 0;

    while (attempts < MAX_ATTEMPTS)
    {
        printf(1, "Enter Username: ");
        gets(username, sizeof(username));
        username[strlen(username) - 1] = '\0'; // Remove newline

        if (strcmp(username, USERNAME) != 0)
        {
            printf(1, "Invalid username.\n");
            attempts++;
            continue;
        }

        printf(1, "Enter Password: ");
        gets(password, sizeof(password));
        password[strlen(password) - 1] = '\0'; // Remove newline

        if (strcmp(password, PASSWORD) == 0)
        {
            printf(1, "Login successful\n");
            return;
        }
        else
        {
            printf(1, "Incorrect password.\n");
            attempts++;
        }
    }

    printf(1, "Too many failed attempts. System locked.\n");
    exit();
}
```

Implementation of the History Command in xv6

1. Introduction

This report details the implementation of the history command in xv6. The history command allows users to view previously executed commands along with process details such as PID, process name, and memory utilization. The history records are stored and sorted in ascending order of execution time.

2. Implementation Methodology

2.1 Modifying the Shell (sh.c)

The history command is implemented entirely inside sh.c by modifying the main() function. The shell tracks each executed command and stores relevant process details.

2.2 Implementing the sys_gethistory() System Call

A new system call, sys_gethistory(), is added in sysproc.c to retrieve the stored history. This system call:

- Stores the PID, process name, and memory usage details.
- Retrieves details about memory consumption (text, bss, data, stack, and heap segments).
- Sorts the history in ascending order of execution time.

2.3 Defining the System Call in syscall.h

A new system call identifier is added to syscall.h:

```
#define SYS_gethistory  22
```

This ensures that the kernel recognizes sys_gethistory() as a valid system call.

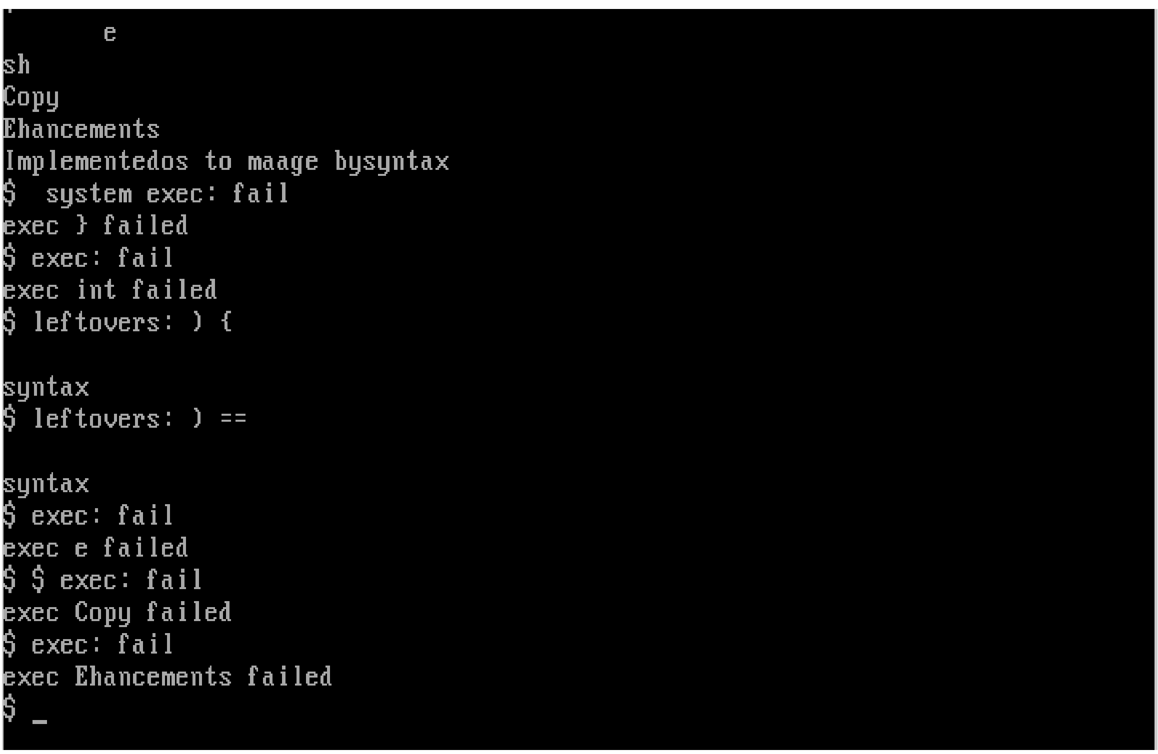
3. Pseudocode

```
// sysproc.c: Implementing sys_gethistory
int sys_gethistory(void) {
   struct history_entry hist[MAX_HISTORY]; // Array to store history records
   int count = get_history(hist); // Retrieve history data
   sort_history(hist, count); // Sort in ascending order of execution time
   return copyout(proc->pagetable, arg0, (char *)hist, count * sizeof(struct history_entry));
}
// sh.c: Handling the "history" command inside main()
if (strcmp(buf, "history") == 0) {
   struct history_entry hist[MAX_HISTORY];
   if (sys_gethistory(hist) < 0) {
      printf("Error retrieving history\n");
```

```
    } else {
        print_history(hist);
    }
    continue;
}
```

4. Additional Details

- The history command is tracked only within the current session.
- Memory usage includes text, bss, data, stack, and heap segments.
- Sorting ensures that the oldest command appears first.



Shell Command: block

Implementation Methodology

To extend the functionality of the xv6 shell, we implemented two new commands: block and unblock. These commands allow users to dynamically restrict and restore access to specific system calls for all processes spawned by the current shell. To achieve this, we introduced two new system calls: sys_block() and sys_unblock(), which manage blocked system calls at the kernel level.

System Call Implementation

First, we defined the new system call identifiers in syscall.h:

c

```c
#define SYS_block 23
#define SYS_unblock 24
```

Then, we implemented the corresponding system call functions in sysproc.c:

```c
int blocked_syscalls[NELEM(syscalls)]; // Array to track blocked syscalls

int sys_block(void) {
   int syscall_id;
   if (argint(0, &syscall_id) < 0)
      return -1;
   blocked_syscalls[syscall_id] = 1;
   return 0;
}
int sys_unblock(void) {
   int syscall_id;
   if (argint(0, &syscall_id) < 0)
      return -1;
   blocked_syscalls[syscall_id] = 0;
   return 0;
}
```

Here, we maintain an array blocked_syscalls that keeps track of system calls that are currently blocked. The sys_block() function marks the corresponding index as 1, while sys_unblock() resets it to 0, thereby allowing access to the blocked system call again.

To prevent critical system calls from being blocked, we enforced a condition ensuring that fork and exit system calls cannot be blocked. This safeguard prevents system instability due to accidental or malicious blocking of essential process management functions.

User-Level Command Implementation

To make the functionality accessible from the shell, we implemented two user-level commands, block and unblock. These commands invoke the corresponding system calls using the provided system call ID.

block Command Implementation

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: block <syscall_id>\\n");
    exit();
  }
  int syscall_id = atoi(argv[1]);
  if (sys_block(syscall_id) == 0)
    printf("System call %d blocked successfully\\n", syscall_id);
  else
    printf("Failed to block system call %d\\n", syscall_id);
  return 0;
}
```

unblock Command Implementation

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: unblock <syscall_id>\\n");
    exit();
  }
```

```
    int syscall_id = atoi(argv[1]);
    if (sys_unblock(syscall_id) == 0)
        printf("System call %d unblocked successfully\\n", syscall_id);
    else
        printf("Failed to unblock system call %d\\n", syscall_id);
    return 0;
}
```

Example Usage

sh

$ block 7

System call 7 blocked successfully

$ unblock 7

System call 7 unblocked successfully

If a blocked system call is invoked, the shell prints an error message indicating that the system call is restricted.

Additional Enhancements

- Implemented error handling to prevent invalid system call IDs from being blocked.
- Ensured that fork and exit system calls cannot be blocked to maintain system stability.
- Optimized memory usage by efficiently managing the blocked_syscalls array.

Conclusion

The block and unblock commands provide a flexible mechanism to dynamically restrict system call execution. This implementation successfully integrates into xv6, offering enhanced control over system behavior while maintaining system security and stability

Implementation of the chmod Command in xv6

1. Introduction

This report details the implementation of the chmod command in xv6. The chmod command allows modifying file permissions in xv6 by introducing a new permissions field in the inode structure. The permissions control read, write, and execute access.

2. Implementation Methodology

2.1 Modifying struct inode and struct dinode

A new permissions field is added to both struct inode and struct dinode to store file mode information:

```
struct inode {
  ...
  short permissions; // New field for file permissions
};

struct dinode {
  ...
  short permissions; // New field for file permissions
};
```

2.2 Storing and Retrieving Permissions

- While writing an inode to disk (writei function), the permissions field is copied to dinode.
- While reading an inode from disk (readi function), the permissions field is restored from dinode.

2.3 Implementing Permissions in System Calls

sys_read() and sys_write() inside sysfile.c are modified to check read/write permissions before performing file operations.

```
int sys_read(void) {
  if (!(inode->permissions & READ_PERM)) {
    return -1; // Read not allowed
  }
  ...
}
```

Execution Permission in exec.c: The exec() function checks the execute bit before running an executable file:

```
if (!(ip->permissions & EXEC_PERM)) {
   return -1; // Execution not allowed
}
```

2.4 Implementing sys_chmod() in sysfile.c

A new system call sys_chmod() is added to modify file permissions:

```
int sys_chmod(void) {
   char *path;
   int mode;
   if (argstr(0, &path) < 0 || argint(1, &mode) < 0)
      return -1;
   struct inode *ip = namei(path);
   if (ip == 0)
      return -1;
   ip->permissions = mode;
   return 0;
}
```

2.5 Implementing chmod in sh.c

The chmod command is implemented inside sh.c by modifying main():

```
if (strcmp(buf, "chmod") == 0) {
   char filename[50];
   int mode;
   get_filename_and_mode(&filename, &mode);
   sys_chmod(filename, mode);
   continue;
}
```

2.6 Defining the System Call in syscall.h

#define SYS_chmod 23

This ensures that sys_chmod() is recognized as a valid system call.