Operating Systems
Assignment 1 – Easy

Nikhil Gupta (2024JCS2611)

6 March 2025

1 Enhanced Shell Login in xv6

Implemented username-password authentication for the xv6 shell. The credentials are defined using macros in the Makefile as follows:

```
USERNAME=<username>
PASSWORD=<password>
```

1.1 Input Validation

Validation ensures that usernames and passwords contain only letters and numbers. The function below rejects any special characters or spaces:

```
int isInputValid(char *input)
{
    int i = 0;
    while (input[i] != '\0')
    {
        if (!((input[i] >= 'A' && input[i] <= 'Z') ||
              (input[i] >= 'a' && input[i] <= 'z') ||
              (input[i] >= '0' && input[i] <= '9')))
        {
            return 0;
        }

        i++;
    }

    return 1;
}
```

1.2 Login Functionality

The login procedure allows for a maximum of three attempts. An incorrect username restarts the loop without asking for the password. The password is requested only if the username is correct. The credentials are compared with the macros defined in the Makefile (‘USERNAME‘ and ‘PASSWORD‘):

```
for (int i = 0; i < TRIES; i++)
{
    printf(1, "Enter username: ");
    int usernameLength = read(0, username, MAX_LENGTH - 1);
    if (usernameLength > 0)
        username[usernameLength - 1] = '\0';
    if (!isInputValid(username)
        || strcmp(username, USERNAME))
    {
        printf(1, "Invalid Username\n");
        continue;
    }

    printf(1, "Enter password: ");
    int passwordLength = read(0, password, MAX_LENGTH - 1);
    if (passwordLength > 0)
        password[passwordLength - 1] = '\0';
    if (!isInputValid(password)
        || strcmp(password, PASSWORD))
    {
        printf(1, "Invalid Password\n");
        continue;
    }
    else
    {
        printf(1, "Login successful\n");
        return 1;
    }
}

printf(1, "Too many failed attempts!!! Login disabled\n");
return 0;
```

1.3 Shell Initialization

After successful authentication, the shell process is executed. If authentication fails entirely, further attempts are blocked by entering an infinite loop:

```
if(!login())
    while(1) sleep(100);

for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
        printf(1, "init: fork failed\n");
        exit();
    }
    if(pid == 0){
        exec("sh", argv);
        printf(1, "init: exec sh failed\n");
        exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)
        printf(1, "zombie!\n");
}
```

This implementation provides secure access control to xv6's shell environment.

2 Shell Command: history

Implemented the `history` command in xv6 to display completed processes, showing their PID, process name, and memory utilization. Introduced a new system call `sys_gethistory()` with identifier 22 (defined in `syscall.h`).

2.1 Data Structures

Define a struct `proc_entry` and a circular buffer to maintain the process history:

```
#define MAX_HISTORY 100

struct proc_entry {
  int pid;
  char name[16];
  uint mem_used;
  int completed;
};

struct proc_entry history[MAX_HISTORY];
int history_start = 0, history_end = 0;
```

2.2 Recording Process Information

In `exec.c`, right after process creation and before setting CPU's current address space to the newly created process's virtual memory space (`switchuvm(curproc)`), freeing the old page directory (`freevm(oldpgdir)`) and returning success (`return 0`), the new process details are recorded in the history buffer. Each entry is initially marked incomplete:

```
// in exec.c before "return 0"
if((history_end + 1) % MAX_HISTORY == history_start)
  history_start = (history_start + 1) % MAX_HISTORY;

history[history_end].pid = curproc->pid;
safestrcpy(history[history_end].name,
           curproc->name,
           sizeof(curproc->name));
history[history_end].mem_used = curproc->sz;
history[history_end].completed = 0;

history_end = (history_end + 1) % MAX_HISTORY;
```

When a process terminates (in `proc.c`, within the `exit()` function), the corresponding history entry is marked completed:

```
// proc.c inside exit()
for(int i = 0; i < MAX_HISTORY; i++){
  if(history[i].pid == curproc->pid
    && !history[i].completed){
    history[i].completed = 1;
    break;
  }
}
```

2.3 System Call `sys_gethistory()`

The system call copies completed process entries from kernel space to user space:

```
// sysproc.c
int sys_gethistory(void)
{
  struct proc_entry *temp_history;
  if(argptr(0, (void *)&temp_history,
            sizeof(struct proc_entry)*MAX_HISTORY) < 0)
    return -1;

  int history_size = 0, i = history_start;

  while(i != history_end){
    if(history[i].completed){
      if(copyout(myproc()->pgdir,
         (uint)temp_history + history_size * sizeof(struct proc_entry),
         (char *)&history[i], sizeof(struct proc_entry)) < 0)
        return -1;

      history_size++;
    }
    i = (i + 1) % MAX_HISTORY;
  }

  return history_size;
}
```

2.4 Shell Command

The `history` command was integrated into the shell (`sh.c`) by adding it within the while loop (where `getcmd()` reads user input (commands) from the console into a buffer), allowing history to be executed as a shell command. The shell invokes the `gethistory()` function and prints the retrieved processes details.

```
// sh.c (inside getcmd loop)
if(!strcmp(buf, "history\n")){
  struct proc_entry history[MAX_HISTORY];
  int history_size = gethistory(history);

  if(history_size < 0)
    printf(2, "Error retrieving history\n");
  else{
    for (int i = 0; i < history_size; i++)
      printf(1, "%d %s %d\n",
             history[i].pid,
             history[i].name,
             history[i].mem_used);
  }
}
```

3 Shell Command: block

Implemented the `block` and `unblock` commands in xv6, which allow a shell to block and unblock specific system calls for all processes created by that shell. Two new system calls were added, with their identifiers defined in syscall.h:

- `sys_block` (identifier 23)
- `sys_unblock` (identifier 24)

3.1 Process and Shell Tracking

Each process has a `sid` (Shell ID) field to track the shell it belongs to. The `proc` structure was updated as follows:
In `proc.h`, the following were added:

```
struct proc {
  ...
  int sid;  // Shell ID
  ...
};
```

Each shell has an entry in `shells[]`, where `blocked_syscalls[i] = 1` indicates that syscall `i` is blocked. Helper function `is_blockable_syscall()` checks whether a syscall is blockable.

```
#define MAX_SYSCALLS 26

struct shell_entry {
  int blocked_syscalls[MAX_SYSCALLS];
};

extern struct shell_entry shells[NPROC];
extern const int non_blockable_syscalls[];

extern int is_blockable_syscall(int);
extern int is_syscall_blocked(struct proc *, int);
```

In `allocproc()`, `sid` is initialized to `-1`. This applies to the very first shell and its descendants until updated:

```
p->sid = -1;
```

For forked processes, the `sid` is copied from the parent, ensuring all child processes share the same shell ID.
In `exec.c`, before `return 0`, each process checks its parent chain to find the nearest shell (named `"sh"`) and assigns its PID as the `sid`:

```
// Set Shell ID
struct proc *temp_proc = curproc->parent;
while(temp_proc && strncmp(temp_proc->name, "sh", 2))
  temp_proc = temp_proc->parent;
if(!strncmp(temp_proc->name, "sh", 2))
  curproc->sid = temp_proc->pid;
```

Processes with `sid = -1` do not have any blocked syscalls.

3.2 Checking for Blocked Calls

Blocked syscalls are checked inside the `syscall()` function before executing any system call. If a syscall is blocked, it prints a message and sets the system call's return value to -1 (indicating an error).

```
if(is_syscall_blocked(curproc, num))
{
  cprintf("syscall %d is blocked\n", num);
  curproc->tf->eax = -1;
  return;
}
```

The function `is_syscall_blocked()` checks if a syscall is blocked for a process by walking up the parent chain and verifying if any parent has a valid `sid`, and whether the corresponding shell's blocking table has marked the syscall as blocked. If any shell in the chain has blocked the syscall, it is considered blocked for the process.

```
int
is_syscall_blocked(struct proc *curproc, int syscall_id)
{
  while(curproc) {
    if(curproc->sid >= 0
        && shells[curproc->sid].blocked_syscalls[syscall_id])
      return 1;
    curproc = curproc->parent;
  }
  return 0;
}
```

3.3 Blocking and Unblocking Syscalls

The `sys_block()` and `sys_unblock()` system calls update the `blocked_syscal-ls[]` array of the calling shell in the `global shells[]` table. In `sys_block()`, the syscall ID is marked as blocked (1), and in `sys_unblock()`, it is marked as unblocked (0). Both check if the syscall is valid and not part of the critical, non-blockable syscalls.

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

  if(syscall_id > 0
     && syscall_id < MAX_SYSCALLS
     && is_blockable_syscall(syscall_id))
  {
    shells[myproc()->pid].blocked_syscalls[syscall_id] = 1;
    return 0;
  }

  return -1;
}

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

  if(syscall_id > 0
     && syscall_id < MAX_SYSCALLS
     && is_blockable_syscall(syscall_id))
  {
    if(shells[myproc()->pid].blocked_syscalls[syscall_id] == 0)
      return -1;

    shells[myproc()->pid].blocked_syscalls[syscall_id] = 0;
    return 0;
  }

  return -1;
}
```

3.4 Shell Commands

The `block` and `unblock` commands are handled in the main `getcmd()` loop of `sh.c` to allow blocking and unblocking syscalls as shell commands.

```
if(buf[0] == 'b' && buf[1] == 'l' && buf[2] == 'o' &&
   buf[3] == 'c' && buf[4] == 'k' && buf[5] == ' '){
  struct execcmd* ecmd = (struct execcmd*)parsecmd(buf);

  if(ecmd->argv[1] != 0){
    if (block(atoi(ecmd->argv[1])) < 0)
      printf(2, "Error: block syscall failed\n");
  } else
    printf(2, "Usage: block <syscall_id>\n");

  continue;
}

if(buf[0] == 'u' && buf[1] == 'n' && buf[2] == 'b' &&
   buf[3] == 'l' && buf[4] == 'o' && buf[5] == 'c' &&
   buf[6] == 'k' && buf[7] == ' '){
  struct execcmd* ecmd = (struct execcmd*)parsecmd(buf);

  if(ecmd->argv[1] != 0){
    if (unblock(atoi(ecmd->argv[1])) < 0)
      printf(2, "Error: unblock syscall failed\n");
  } else
    printf(2, "Usage: unblock <syscall_id>\n");

  continue;
}
```

3.5 Clearing Blocked Syscalls on Shell Exit

When a shell exits, its blocked syscalls are cleared:

```
// proc.c inside exit()
if(!strncmp(curproc->name, "sh", 2))
  for(int i = 0; i < MAX_SYSCALLS; i++)
    shells[curproc->pid].blocked_syscalls[i] = 0;
```

4 Shell command : chmod

Implemented the chmod command in xv6 to modify file permissions of files. Introduced a new system call `sys_chmod()` with identifier 25 (defined in `syscall.h`).

4.1 Modifications in `dinode` and `inode` structures

To support file permissions, a new `mode` field was added in both the `dinode` and `inode` structures.
While adding this field, it was necessary to ensure that the `dinode` size remains exactly 64 bytes so that it evenly divides the block size. Originally, the `type` field was declared as `short` (2 bytes), but since the file type requires only a few

values, it was reduced to `__UINT8_TYPE__` (1 byte). The remaining 1 byte was used for the new `mode` field (also 1 byte), allowing us to store the permission bits without exceeding the 64-byte size limit of the structure.
Changes in `fs.h`:

```
struct dinode {
  __UINT8_TYPE__ type;    // File type (1 byte)
  __UINT8_TYPE__ mode;    // Permission bits (1 byte)
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

Changes in `file.h`:

```
struct inode {
  uint dev;
  uint inum;
  int ref;
  struct sleeplock lock;
  int valid;

  __UINT8_TYPE__ type;   // File type (1 byte)
  __UINT8_TYPE__ mode;   // Permission bits (1 byte)
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

4.2 System Call and Inversion of Permission Bits

The `sys_chmod()` system call is responsible for changing the permissions of a file. When the system call is invoked, the following steps occur:

1. The arguments are fetched from the user using `argstr()` for the filename and `argint()` for the mode.
2. A file system operation is started with `begin_op()`.
3. The inode of the file is located using the `namei()` function.
4. If the file does not exist, the operation is ended with `end_op()` and `-1` is returned.

5. If the file exists, the inode is locked with `ilock()`.
6. The user-supplied mode is inverted using:

   ```
   file_inode->mode = ~(mode) & 0b111;
   ```

   This inversion ensures internal consistency, treating bit `0` as full access internally, even though the user sets bit `1` to grant permission. This was necessary because all existing xv6 files initially have `mode = 0`, and we must avoid blocking all access to legacy files.
7. The inode is updated on disk using `iupdate()`.
8. The inode is unlocked with `iunlock()`.
9. The file system operation is completed with `end_op()`.
10. Finally, the system call returns `0` to indicate success.

This inversion ensures:

- User provides `7` (full access) → stored as `0` (no permissions blocked).
- User provides `0` (no access) → stored as `7` (all permissions blocked).

The complete implementation of `sys_chmod()` in `sysfile.c` is as follows:

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

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

  struct inode *file_inode;
  begin_op();

  file_inode = namei(file);
  if(file_inode == 0)
  {
    end_op();
    return -1;
  }

  ilock(file_inode);
  file_inode->mode = ~(mode) & 0b111;
  iupdate(file_inode);
  iunlock(file_inode);

  end_op();

  return 0;
}
```

4.3 Permission Checks in System Calls

Permission enforcement is added directly into the following system calls:
In `sys_read()` (`sysfile.c`):

```
if(f->ip && (f->ip->mode & 1)){
    cprintf("Operation read failed\n");
    return -1;
}
```

In `sys_write()` (`sysfile.c`):

```
if (f->ip && (f->ip->mode & 2)) {
    cprintf("Operation write failed\n");
    return -1;
}
```

In `sys_exec()` (`sysfile.c`):

```
struct inode *ip;
begin_op();
if ((ip = namei(path)) == 0){
    end_op();
    return -1;
}
ilock(ip);

if (ip->mode & 4){
    cprintf("Operation execute failed\n");
    iunlock(ip);
    end_op();
    return -1;
}

iunlock(ip);
end_op();
```

4.4 Shell Command

The `chmod` command was added in `sh.c` to handle the user input and invoke the system call:

```
if(buf[0] == 'c' && buf[1] == 'h' && buf[2] == 'm' &&
   buf[3] == 'o' && buf[4] == 'd' && buf[5] == ' '){
    struct execcmd* ecmd = (struct execcmd*)parsecmd(buf);

    if(ecmd->argv[1] != 0 && ecmd->argv[2] != 0){
      if(chmod((const char*)ecmd->argv[1],
            atoi(ecmd->argv[2])) < 0)
        printf(2, "Error: chmod syscall failed\n");
    } else
      printf(2, "Usage: chmod <file> <mode>\n");

    continue;
}
```