Easy Assignment – 1
Disha (2022CS11118)

Part 2: Enhanced Shell

Objective: Implement a secure login system for the xv6 shell, requiring a username and password. The system allows three login attempts, after which it locks out the user.

Implementation: (init.c)

1. **Username Prompt & Validation:**

   - The user is prompted to enter a username.
   - The input is checked against the predefined `USERNAME`. If incorrect, the user is asked to try again until the correct username is entered.

```
while (strcmp(input_username, USERNAME) != 0) {
    printf(1, "Invalid username. Try again.\n");
    printf(1, "Enter Username: ");
    gets(input_username, sizeof(input_username));
    input_username[strlen(input_username) - 1] = '\0';
}
```

Figure 1: init.c

2. **Password Prompt & Validation:**

   - After a correct username, the user is prompted to enter a password.
   - The input password is compared to the predefined `PASSWORD`.
   - If incorrect, the user has three attempts to enter the correct password.

```
while (attempts < MAX_ATTEMPTS) {
    printf(1, "Enter Password: ");
    gets(input_password, sizeof(input_password));
    input_password[strlen(input_password) - 1] = '\0';

    if (strcmp(input_password, PASSWORD) == 0) {
        printf(1, "Login successful\n");
        return 1;
    } else {
        attempts++;
        printf(1, "Incorrect password. Attempt %d of %d\n", attempts, MAX_ATTEMPTS);
    }
}
printf(1, "Login failed. System locked.\n");
return 0;
```

Figure 2: init.c

3. **Attempts Limitation:**

   - If the user exceeds three incorrect password attempts, the system locks the login process.

```
if (login() == 0) {
    exit();
}
```

Figure 3: init.c

4. **Success:**

    - Upon correct username and password entry, the user gains access, and the shell proceeds.

```
Enter Username: 2022CS11118
Enter Password: disha
Login successful
init: starting sh
$ 
```

Part 3: History

Objective: To implement an additional command history that needs to display a list of all the processes that have been executed until now

Implementation :

1. `history_entry` structure is defined to store the required process details (PID, name, total memory).

```
#define HISTORY_SIZE 100

struct history_entry {
  int pid;
  char name[16];
  uint total_mem;
};

struct history_entry history[HISTORY_SIZE];
int history_count = 0;


void
```

Figure 4: proc.c

2. maintained an array of `history_entry` structures and a count to track how many processes are stored in the history

```
if (history_count < HISTORY_SIZE) {
  history[history_count].pid = curproc->pid;
  safestrcpy(history[history_count].name, curproc->name, sizeof(curproc->name));
  history[history_count].total_mem = curproc->sz;
  history_count++;
}
```

Figure 5: proc.c : exit()

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

Figure 6: proc.c (helper function to print the history)

Part 4 : block and unblock system calls

Objective:need a way to keep track of which system calls are blocked for each process spawned by the shell.
Implementation:Used a global array to store blocked system calls for processes associated with the shell.

```
  char name[16];               // Process name (debugging)
  int blocked_syscalls[MAX_SYSCALLS];   //Array for blocking syscalls for this process
  int blocked_child_syscalls[MAX_SYSCALLS];
};
```

Figure 7: proc.h : struct proc

```
int block(int syscall_id) {
  if ((syscall_id < 0 || syscall_id >= MAX_SYSCALLS) || is_restricted(syscall_id)) {
    return -1;
  }
  struct proc *curproc = myproc();
  curproc->blocked_child_syscalls[syscall_id] = 1;
  return 0;
}
```

Figure 8: proc.c : block.c

similarly 'int unblock(int syscall_id)' is implemented

```
static int restricted_syscalls[] = {1, 2}; //cannot block fork and exit

int is_restricted(int syscall_id) {
  int i;
  for (i = 0; i < sizeof(restricted_syscalls) / sizeof(restricted_syscalls[0]); i++)
      if (restricted_syscalls[i] == syscall_id) {
        return 1;  //
      }
  }
  return 0;
}
```

Figure 9: proc.c : some system calls cannot be blocked

```
  // blocked syscalls for the child
  for(int i = 0; i < MAX_SYSCALLS; i++){
    np->blocked_syscalls[i] = curproc->blocked_child_syscalls[i];
    np->blocked_child_syscalls[i] = curproc->blocked_child_syscalls[i];
  }
```

Figure 10: proc.c : child processes inheriting blocked syscalls from the parent, and the syscalls for the child processes of that child are also blocked.

```c
struct proc *curproc = myproc();

num = curproc->tf->eax;
if (!(curproc->pid == 2 || strncmp(curproc->name, "sh", sizeof("sh") - 1) == 0)) {
  if (curproc->blocked_syscalls[num] == 1) {
    cprintf("syscall %d is blocked\n", num);
    return;
  }
}

if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
```
Figure 11: syscall.c

system calls are not blocked for the current shell, they are blocked only for the child processes

part 5: chmod

Objective: Implement the `chmod` system call to modify file permissions using a 3-bit integer (read, write, execute) in the xv6 operating system.

```c
int chmod(const char* file, int mode) {

  struct inode *ip = namei((char*)file);

  begin_op();

  if ((ip = namei((char*)file)) == 0) {
    end_op();
    return -1;
  }

  ilock(ip);

  ip->mode = mode & 0b111;

  iupdate(ip);
  iunlock(ip);
  end_op();

  return 0;
}
```

Figure 12: proc.c

Implemention:

1. **Define the `chmod` Function:**
    - The function takes two arguments: a `file` (filename) and `mode` (3-bit permission value).
    - The file is located using the `namei()` function, which returns the corresponding inode.
2. **Check File Existence:**
    - If the file does not exist, the function returns `-1` and terminates.

```c
// On-disk inode structure
struct dinode {
  short type;           // F
  short major;          // M
  short minor;          // M
  short nlink;          // N
  uint size;            // S
  uint addrs[NDIRECT+1];   /
  uint mode;               //
  char padding[60];  // Padd
};
```

Figure 13: fs.h

Created a member “mode” to store the mode in disk. Also padding was done.

```c
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];

  uint mode;
};
```

Figure 14: file.h – struct inode

Created a member “mode” to store the mode of the file in memory.

3. **Lock the Inode:**

- If the file is found, the inode is locked using `ilock()`. This ensures no other process can modify the inode while it's being updated.

```
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquiresleep(&ip->lock);

  if(ip->valid == 0){
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    ip->mode = dip->mode;  //disk to memory
    brelse(bp);
    ip->valid = 1;
```

Figure 15: fs.c

4. **Update Permissions:**

- The permission bits are masked with `0b111` to ensure only the last three bits are set (read, write, and execute permissions).
- The mode is then updated in the inode.

```
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  dip->mode = ip->mode;  //memory to disk
  log_write(bp);
  bwrite(bp);
  brelse(bp);
}
```

Figure 16: fs.c

5. **Write Back to Disk:**

- After modifying the mode, the inode is updated on disk using `iupdate()`.

```c
uint
ialloc(ushort type)
{
  uint inum = freeinode++;
  struct dinode din;

  bzero(&din, sizeof(din));
  din.type = xshort(type);
  din.nlink = xshort(1);
  din.size = xint(0);
  din.mode = 0x7;
  winode(inum, &din);
  return inum;
}
```

Figure 18: mkfs.c

```c
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum, sb));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      dip->mode = 0x7;
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

Figure 17: fs.c

```c
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;

  if(!(f->ip->mode & 0x2)){
    cprintf("Operation write failed\n");
    cprintf("mode = %d\n", f->ip->mode);
    return -1;
  }

  return filewrite(f, p, n);
}
```

Figure 19: sysfile.c - Checking the permissions before performing read, write, execute operations.

Similarly for sys_read and sys_exec