Operating Systems Assignment 1– Easy

Name : Jitesh Shamdasani

Entry Number : 2024JCS2043

Login Flow

1. **Startup and Console Setup:**
    - The `init` process starts and ensures that the console device exists and is open.
    - Standard input, output, and error are duplicated (via `dup`) so the shell can use them.
2. **Infinite Loop and Forking:**
    - In an infinite loop, `init` forks a child process.
    - The parent waits for the child to exit (handling zombie processes).
3. **Child Process – Login Procedure:**
    - In the child (`pid == 0`), the `login()` function is invoked.
    - **Login Function Steps:**
        - Prompts the user for a username.
        - Reads input via `gets()` and trims the newline.
        - Compares the input to a defined `USERNAME`.
        - If the username is incorrect, prints an error and increments the attempt counter.
        - If the username is correct, it proceeds to prompt for the password.
        - Reads and trims the password input.
        - Compares the password against the defined `PASSWORD`.
        - If the password matches, the function returns, allowing the login to succeed.
        - After three failed attempts, `login()` delays further execution by calling `sleep()` for a long interval.
    - After a successful login, it calls `exec("sh", argv)` to replace the current process image with that of the shell.
4. **Outcome:**
    - Only authenticated users reach the shell. The shell runs in the child process, while the original `init` process continues to monitor and fork new children as needed.

History System Call

1. Global Definitions and Structure

- **Header File: `history_struct.h`**
  Defines the `history_struct` and global variables used for history tracking.
  - **Structure Definition:**
    - `int pid` – Process ID of the process whose history is recorded.
    - `char name[16]` – Process name (using 16 chars for consistency with other parts of xv6).
    - `int totalMemory` – Memory usage of the process (usually from `proc->sz`).
    - `int creationTime` – Increase count at the process creation (for timestamping issues, if used).
  - **Global Variables Declared:**
    - `hist_arr[MAX_LIMIT]` – Array that holds up to `MAX_LIMIT` history entries.
    - `hist_count` – A counter that tracks the number of history entries recorded.
    - `hist_lock` – A spinlock to protect updates to the history data.

---

2. Recording History in the Kernel (in `proc.c`)

- **Initialization:**
  In the `pinit(void)` function, the history lock is initialized:
- **Recording a History Entry:**
  When a process is created or at appropriate points (likely during fork or process creation), the current process's information is recorded.

3. Retrieving History Data via System Call

In `exec.c` (Actual Implementation)

- **Function: `gethistory_actual`**
- The function acquires `hist_lock` to ensure exclusive access while reading the global history.
- It computes the number of bytes to copy based on `hist_count`.
- Uses `copyout()` to transfer history entries into the user-space buffer.
- Releases the lock and returns the number of history entries.

In `sysproc.c` (System Call Wrapper)

- **Function: `sys_gethistory`**
- `sys_gethistory` obtains the user pointer via `argptr()` and then calls `gethistory_actual()` to do the actual data copy.
- **System Call Number:**
  In `syscall.h`, `SYS_gethistory` is defined as `22`, ensuring that when the shell calls `gethistory()`, this system-call gets invoked.

---

4. Handling the History Command in the Shell (`sh.c`)

- **Detection and Execution of `history` Command:** In the shell's main loop, the code checks if the input command starts with `"history"`

- The shell trims the input.

- It declares a local array to hold the retrieved history.

- Calls the system call `gethistory()`, which internally maps to `sys_gethistory()`.

- Finally, it iterates through the returned history entries and prints them.

6. Overall Flow Summary

1. **Process Creation and History Logging:**

- During process creation (in `proc.c`), the process's details (PID, name, memory usage) are recorded into `hist_arr` under a lock.

2. **User Command for History:**

- A user types `history` at the shell prompt.
- The shell recognizes it as a built-in command and calls the system call `gethistory()`.

3. **System Call Execution:**

- `sys_gethistory()` in `sysproc.c` validates the user's buffer and calls `gethistory_actual()`.
- `gethistory_actual()` in `exec.c` copies the history entries from the kernel's `hist_arr` into the user-provided buffer.

4. **Display in Shell:**

- The shell prints out the history entries (format: PID, process name, total memory).

5. **Block/Unblock (Separate but Similar Flow):**

- Block and unblock commands update the per-process `blocked_syscalls` array.
- The kernel checks this array in `syscall()` to enforce blocking behavior before a system call is executed.

This cohesive flow ensures that history tracking and retrieval work seamlessly, while also integrating dynamic control over system call execution via the block/unblock mechanisms.

Block & Unblock

1. User-Space Interface

usys.S

The user library macros make the system calls available to user programs.

```
SYSCALL(unblock)
SYSCALL(block)
```

These macros generate the appropriate trap instructions to invoke the kernel functions for block and unblock.

user.h

The user-level prototypes are declared so that user programs (including the shell) can call This ensures that when the shell's code calls `block(…)` or `unblock(…)`, the call is routed to the corresponding system calls.

2. Kernel System Call Wrappers (sysproc.c)

Block Wrapper

In `sysproc.c`, the `sys_block` function extracts the syscall number from user space and checks that critical syscalls (e.g., fork and exit) are not blocked. It then calls `block_actual` (defined in exec.c) to do the actual work.

Unblock Wrapper

Similarly, `sys_unblock` extracts the syscall number and then calls `unblock_actual`:

3. System Call Number Mapping

syscall.h

The syscall numbers for block and unblock are defined so that the kernel's syscall dispatcher can correctly route the calls:

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

syscall.c

The system call array in `syscall.c` maps these numbers to their implementations When a process invokes the block or unblock system call, the kernel dispatcher calls the corresponding wrapper in `sysproc.c`.

4. Shell Command Parsing (sh.c)

The shell (in `sh.c`) recognizes when the user types a block or unblock command

This code parses the command line, extracts the target syscall ID, and calls the user-level wrappers, which in turn invoke the kernel system calls.

5. Process Structure and Fork Inheritance (proc.h and proc.c)

In proc.h

The `struct proc` is extended with fields for blocking

In proc.c

During a fork, the child process inherits the block/unblock settings from the parent

This ensures that the blocking settings are preserved across process creation.

---

6. Actual Implementation of Block/Unblock (exec.c)

block_actual and unblock_actual Functions

All this perform the actual update of the `blocked_syscalls` array for the current process

hese functions ensure that:

- The provided syscall_id is within bounds.
- Critical syscalls (like fork and exit) are never blocked.
- The per-process `blocked_syscalls` array is updated correctly.

Overall Flow Summary

1. **User Interface:**

    - The shell uses the `block` and `unblock` commands, parsed in `sh.c`.
    - The user's command string is converted (using `atoi`) into a syscall number.

2. **System Call Invocation:**

    - The system calls `block()` and `unblock()` are invoked (via usys.S and declared in user.h).
    - These calls enter the kernel and are dispatched through the syscall table (`syscall.h` and `syscall.c`).

3. **Kernel Wrappers:**

    - `sys_block` and `sys_unblock` in `sysproc.c` retrieve the syscall number from user space, perform preliminary checks (e.g., disallowing block of critical syscalls), and then call `block_actual()` or `unblock_actual()`.

4. **Actual Implementation:**

    - In `exec.c`, `block_actual()` and `unblock_actual()` update the current process's `blocked_syscalls` array.
    - These changes are inherited by child processes in `fork()` (as defined in `proc.c`).

This complete flow ensures that system calls can be dynamically blocked or unblocked at runtime by updating a per-process array, and the corresponding checks in the syscall dispatcher (in `syscall.c`) will use these settings to either allow or deny the execution of particular system calls.

Chmod System Call

1. User-Space Shell Command (sh.c)

- **Command Parsing:**
  - The shell (in `sh.c`) checks if the command begins with `"chmod "` by comparing the first few characters.
  - It removes the trailing newline.
  - It then skips any extra whitespace and extracts:
    - The target filename (`fileName`).
    - The mode string (`modeStr`).
  - The mode string is converted into an integer using `atoi()`.
- **Invocation:**
  - The shell then calls the user-level function:

    ```
    if(chmod(fileName, mode) < 0)
      printf(2, "chmod %s failed\n", fileName);
    else
      printf(1, "chmod %s succeeded\nnew mode is %d\n", fileName, mode);
    ```

  - This `chmod()` function is declared in `user.h` and invokes the system call via `usys.S`.

---

2. User-Level System Call Interfaces

- **Prototype in user.h:**

This ensures that when the shell calls `chmod(fileName, mode)`, the call will trap into the kernel.

- **System Call Macro in usys.S:** This macro generates the necessary trap instruction to transfer control to the kernel.

---

3. Kernel Dispatch: Syscall Mapping (syscall.h and syscall.c)

- **Syscall Number Definition:**
- In `syscall.h`, the syscall number is defined:

  ```
  #define SYS_chmod  25
  ```

- **Syscall Table Entry:**
- In `syscall.c`, the system call number 25 is mapped to the kernel function `sys_chmod`

```
[SYS_chmod] sys_chmod,
```

---

4. System Call Wrapper (sysproc.c)

- **Wrapper Function:**
- The function `sys_chmod` acts as a wrapper that extracts the arguments from user space
- It uses `argstr()` and `argint()` to fetch the filename and mode values passed from the user.
- Then, it calls the actual implementation function `chmod_actual(file, mode)`.

5. Actual Implementation (exec.c)

- **Core Functionality:**
- The function `chmod_actual` in `exec.c` performs the permission modification
- **Execution Steps:**
    1. **Validation:**
        - Checks if the target file is "chmod" (to prevent modifying itself).
        - Validates that the mode is within the allowed range `[0, 7]`.
    2. **File System Operation Start:**
        - Calls `begin_op()` to initiate a file system transaction.
    3. **Inode Lookup:**
        - Uses `namei(file)` to obtain a pointer to the inode corresponding to the file.
        - If not found, the operation is terminated.
    4. **Inode Locking/Updating:**
        - Acquires a lock on the inode (`ilock(ip)`).
        - Updates the `mode` field of the inode.
        - Writes changes to disk using `iupdate(ip)` and then unlocks/releases the inode with `iunlockput(ip)`.
    5. **Finalization:**
        - Calls `end_op()` to finish the file system operation.
        - Returns `0` on success or `-1` on any error.

6. Flow Summary

1. **User Action:**

- The user types `chmod <file> <mode>` in the shell.

2. **Shell Processing (sh.c):**

- Parses the command to extract `fileName` and `mode`.
- Invokes the user-level `chmod(fileName, mode)` function.

3. **User-Level to Kernel Transition:**

- The system call interface (via usys.S) traps into the kernel.

4. **Kernel Dispatch:**

- The syscall number `SYS_chmod` (25) is matched and routed to `sys_chmod` (in sysproc.c).

5. **Wrapper Function (sysproc.c):**

- `sys_chmod` extracts the arguments and calls `chmod_actual(file, mode)`.

6. **Actual File Operation (exec.c):**

- `chmod_actual` validates inputs, locates the file's inode, updates its mode, and finalizes the file system transaction.

7. **Result:**

- On success, the new file mode is set, and a success message is printed by the shell. Otherwise, an error message is displayed.
