Operating Systems Assignment 1 - Easy

Aditya Jha
2022CS11102

1 Enhanced Shell for xv6

First, added the username and password to **Makefile** and then added them in the CFLAGS.

```
// Makefile
# Login creds
USERNAME=aditya
PASSWORD=something
CFLAGS += -DUSERNAME=\"$(USERNAME)\" -DPASSWORD=\"$(PASSWORD)\"
```

Used a function for login that checks for the username and password and called it just before the execution of **sh** command.

```
// init.c
#define MAX_ATTEMPTS 3
#define BUFFER_LEN 32

int login(void){ // returns 1 for success and 0 for failure
  char username[BUFFER_LEN], password[BUFFER_LEN];
  int attempts = 0;
  while(attempts < MAX_ATTEMPTS){
    ... check for username and password
  }
  return 0;
}

... (inside the main function)
      if(!login()){
        printf(2, "Login Unsuccessful\n");
        while(1) sleep(100);
        exit();
      }
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
...
```

Hence, the shell is started only after successful authentication.

2 Shell Command: history

First, defined the syscall identifier for history and mapped it to its corresponding system call.

```
// syscall.h
#define SYS_gethistory 22

// syscall.c
extern int sys_gethistory(void);

... (*syscalls)
[SYS_gethistory] sys_gethistory,
...

// usys.S
SYSCALL(gethistory)
```

Then defined the respective kernel-space and user-space wrapper functions.

```
// defs.h
int gethistory(void);

// user.h
int gethistory(void);
```

To store the history of processes, created a structire **history_entry** for each process and created an array of these structures called **history**.

```
// proc.h
#define MAX_HISTORY 100

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

extern struct history_entry history[MAX_HISTORY];
extern int history_count; // initialised to 0
```

After that created a procedure for adding the successfully exited processes to history. This procedure is called from the exit() function. When, a user-process exits successfully, the add_proc() procedure is called and this adds the details of the process to the history.

```
// proc.c
void add_proc(int pid, char *name, uint size){
  if(history_count >= MAX_HISTORY)
    panic("Out of Memory!");

  history[history_count].pid = pid;
  safestrcpy(history[history_count].name, name,
        sizeof(history[history_count].name));
  history[history_count].mem_usage = size;
  ++history_count;
}

... (exit() function)
  add_proc(curproc->pid, curproc->name, curproc->sz);
...
```

Whenever history command is entered in the shell, the shell calls the user-space gethistory() wrapper function, which invokes the syscall (sys_gethistory). This calls the kernerl-space wrapper function gethistory().

```
// sh.c
... (main() function)
  if(// history is entered){
    if(gethistory() < 0)
        printf("could not retrieve history\n");
  }
...

// sysproc.c
int sys_gethistory(void){
  return gethistory();
}


// proc.c
int gethistory(void){
  sort_history(); // sort accordint to time
  for (int i = 0; i < history_count; ++i) {
    cprintf("%d %s %d\n",
        history[i].pid, history[i].name, history[i].mem_usage);
  }
  return 0;
}
```

3 Shell Commands: block & unblock

First few steps are similar to the history command. (creating syscall identifiers and wrapper functions).

For storing the blocked syscalls, added arrays in each process and they are updated when the syscalls are made. .

```
// proc.h
struct proc { ... other fields
  int my_blocked[NUMCALLS + 1];
  int ch_blocked[NUMCALLS + 1];
}

// proc.c
... block for process p
  p->ch_blocked[syscall_id] = 1;

... unblock for process p
  p->ch_blocked[syscall_id] = 0;
```

When forked, the blocked arrays are copied to the child and when executed the child's my_blocked is updated to 1 if the parent's ch_blocked is 1.

```
// proc.c ... in fork() function
    np->my_blocked[i] = curproc->my_blocked[i];
    np->ch_blocked[i] = curproc->ch_blocked[i];

// exec.c
      curproc->my_blocked[i] = 1;
      // when parent's ch_blocked is 1.
```

Whenever a user-process tries to invoke a syscall_id which is blocked, it returns an error message.

```
// syscall.c
... (syscall() function)
    if(curproc->my_blocked[num]) // here num is the syscall_id
      cprintf("syscall %d is blocked\n", num);
      return;
...
```

4 Shell Command: chmod

Most of the initial steps are similar to other syscalls. First, changed the in-memory copy of and the on-disk nodes for the files.

```
// proc.h
// in-memory copy of an inode
struct inode {
  ... other fields
  int mode;
};

// fs.h
// On-disk inode structure
struct dinode {
  ... other fields
  int mode;
};
```

Then added how theses nodes to be initialised with all permissions. These are also updated when files are written to memory.

```
// fs.c
struct inode* ialloc(uint dev, short type){ ...
    dip->mode = 7; // initial permission
}

//mkfs.c
uint ialloc(ushort type){ ...
  din.mode = 7;
}
```

When the syscall is invoked, the mode of the file is changed to the one entered by the user.

```
int sys_chmod(void){... in sysfile.c
  ip->mode = mode;
}
```

Whenever a file is read/written/executed, the corresponing permission is checked.

```
  if(!(file->ip->mode & 1)) ... mode 2: write & 4: execute
    cprintf("Operation read failed\n");
```

5 Additional commands

I have created a user program called **clear**, which when entered in the shell clears the screen (similar to UNIX shells).

```
// clear.c
#include "user.h"
int main(){
    printf(1, "\033[H\033[J");
        // clears screen and starts at the top of the screen
    exit();
}

// Makefile
UPROGS : _clear\
EXTRA  : clear.c
```