Assignment 1: Easy Track

COL 331 OPERATING SYSTEMS

ANIKET CHATTOPADHYAY (2022CS11599)

1.2: ENHANCED SHELL FOR XV6

1.2.1 RELEVANT THEORY AND DETAILS

1.2.2 IMPLEMENTATION DETAILS

1.2.3 CODE SNIPPETS

1.3: SHELL COMMAND: history

1.3.1 RELEVANT THEORY AND DETAILS

1.3.2 IMPLEMENTATION DETAILS

1.3.3 CODE SNIPPETS

1.4: SHELL COMMAND: block / unblock

1.4.1 RELEVANT THEORY AND DETAILS

1.4.2 IMPLEMENTATION DETAILS

1.4.3 CODE SNIPPETS

1.5: SHELL COMMAND: chmod

1.5.1 RELEVANT THEORY AND DETAILS

1.5.2 IMPLEMENTATION DETAILS

1.5.3 CODE SNIPPETS

1.6: ADDITIONAL CODE SNIPPETS

1.2: ENHANCED SHELL FOR XV6

1.2.1 RELEVANT THEORY AND DETAILS

To do authentication, we need to ensure that both username and password match with USERNAME and PASSWORD (as defined in the Makefile) and the init process (i.e. the initial user-level process on boot) allows the user to enter the shell only on successful authentication within three tries

1.2.2 IMPLEMENTATION DETAILS

It is a simple command in which you take input from the user and compare it with other strings. The str commands are available in init, so they are used. Also, to make the console unresponsive after three unsuccessful tries, we have used the sleep () command to make the console unresponsive (shell not started)

1.2.3 CODE SNIPPETS

```
33  // --- Login system start ---
34  while(attempts < 3 && !loginSuccess){
35    printf(1, "Enter Username: ");
36    gets(username, sizeof(username));
37    // Remove trailing newline if any.
38    if(username[strlen(username)-1] == '\n')
39      username[strlen(username)-1] = '\0';
40
41    if(strcmp(username, USERNAME) == 0){
42      printf(1, "Enter Password: ");
43      gets(password, sizeof(password));
44      if(password[strlen(password)-1] == '\n')
45        password[strlen(password)-1] = '\0';
46
47      if(strcmp(password, PASSWORD) == 0){
48        printf(1, "Login successful\n");
49        loginSuccess = 1;
50        break;
51      } else {
52        printf(1, "Incorrect password\n");
53      }
54    } else {
55      printf(1, "Incorrect username\n");
56    }
57    attempts++;
58  }
59
60  if(!loginSuccess){
61    printf(1, "Maximum login attempts reached. Disabling login system.\n");
62    while(1)
63      sleep(100);
64  }
65  // --- Login system end ---
66
```

1.3: SHELL COMMAND: history

1.3.1 RELEVANT THEORY AND DETAILS

The history command is useful for maintaining the process history that are run in the system. The following conventions (and assumptions) have been adopted for ‘history’:
(1) It is executed ONLY from the shell.
(2) The process is logged into the ‘history’ in increasing order of their start times. Since no complex command using pipes are given, we can safely assume that the pids are in the order of the start time.
(3) The processes logged into history are visible only after the process has finished executing. Also, a process not spawned is not included in history

1.3.2 IMPLEMENTATION DETAILS

For every process, I have maintained a proc_hist structure, which maintains the history of the process (including pid, name, start time and memory on creation). Since it is given that memory does not change on execution, we can safely assume this. I have maintained a global history array of the pointers to proc_hist structures of each of the processes. In addition, I have used an additional field namely is_visible which tells whether to print this process history or not. If the process has not completed its execution or has been blocked (via system calls), it does not show up in the printed logs of history. If there is no output to be shown, the shell does not output anything.

1.3.3 CODE SNIPPETS

```
int sys_gethistory(void) {
  int count = 0;
  // Iterate over the history table.
  // Only processes with is_visible set to 1 (i.e. completed) are printed.
  for (int i = 0; i < history_count; i++) {
    if (history[i] != 0 && history[i]->is_visible == 1) {
      cprintf("%d %s %d\n", history[i]->pid, history[i]->name, history[i]->memsz);
      count++;
    }
  }
  return 0;
}
```

```c
struct proc_hist {
  int pid;
  char name[16];
  uint memsz;
  int is_visible;
};
```

```c
261
262   // Pass abandoned children to init.
263   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
264     if(p->parent == curproc){
265       p->parent = initproc;
266       if(p->state == ZOMBIE)
267         wakeup1(initproc);
268     }
269   }
270
271   // Update the history entry for the exiting process
272   for (int i = 0; i < history_count; i++) {
273     if (history[i] != 0 && history[i]->pid == curproc->pid) {
274       history[i]->is_visible = 1;  // Mark as completed
275       break;
276     }
277   }
278
```

```c
#define MAX_HIST_ENTRIES 128

extern struct proc_hist *history[MAX_HIST_ENTRIES];
extern int history_count;
```

1.4: SHELL COMMAND: block / unblock

1.4.1 RELEVANT THEORY AND DETAILS

The block/unblock command is useful for blocking system calls in the processes spawned by the child processes. The blocked processes are inherited and propagated to all the subsequent generation of processes as well. FORK and EXIT cannot be blocked as they are essential.

1.4.2 IMPLEMENTATION DETAILS

To implement this, I have added two bitmasks (uint is 32 bits, so each bit corresponds to whether the process is blocked or not). One is blocked_syscall, which maintains the syscalls blocked in the current shell. The other is blocked_syscall_child, which maintains the syscalls blocked in any new process spawned by it. The blocked_syscall_child contains the system calls for which the block command has been invoked, in addition to the already blocked_syscall. This ensures that blocking is maintained. The reverse process is done for unblocking.

1.4.3 CODE SNIPPETS

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  uint blocked_syscalls;       // Bitmask of blocked syscalls for current process
  uint blocked_syscalls_child; // Bitmask of blocked syscalls for child processes
};
```

```
int sys_block(void) {
  int syscall_id;
  if (argint(0, &syscall_id) < 0)
    return -1;
  if (syscall_id == 1 || syscall_id == 2)
    return -1; // Prevent blocking critical syscalls
  myproc()->blocked_syscalls_child |= (1 << syscall_id);
  return 0;
}
```

```c
int sys_unblock(void) {
  int syscall_id;
  if (argint(0, &syscall_id) < 0)
    return -1;
  myproc()->blocked_syscalls_child &= ~(1 << syscall_id);
  return 0;
}
```

```c
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Set the blocked syscalls fields
  np->blocked_syscalls = curproc->blocked_syscalls;
  np->blocked_syscalls_child = curproc->blocked_syscalls_child;

  // Clear %eax so that fork returns 0 in the child
```

1.5: SHELL COMMAND: chmod

1.5.1 RELEVANT THEORY AND DETAILS

chmod command is useful for maintaining permissions for a file. These permissions include the permissions for read, write and execute

1.5.2 IMPLEMENTATION DETAILS

To implement chmod, I store the permission flags in the file metadata. The file metadata is stored in the disk, in a dinode struct, which is then transferred to an inode struct in the working memory of the system. The ialloc() and iupdate() functions maintain the consistency between the file permissions in the working memory and the disk. Also, since we are adding additional fields, the struct is now > 64 bits and this causes issues as to prevent more cache misses the cache line is a multiple of the structs. Hence, to satisfy this requirement, we need to pad a few more bits to reach the next factor of 512 (64 to 128). Also, we need to initialise with a value of 1 for all flags, so that the read/write/execute works.

1.5.3 CODE SNIPPETS

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];

  // for simplification, let's first implement using different fields : we'll use a bitmask later
  uint read_perm;
  uint write_perm;
  uint exec_perm;
};
```

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

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

  return chmod(path, mode);
}
```

```

      begin_op();
      ilock(f->ip);
      // Check the inode's write permission:
      if(f->ip->write_perm == 0){
        cprintf("Operation write failed\n");
        iunlock(f->ip);
        end_op();
        return -1;
      }
      if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
        f->off += r;
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
  // for simplification, let's first implement using different fields : we'll use a bitmask later
  uint read_perm;
  uint write_perm;
  uint exec_perm;
  char pad[52]; //bad idea really but let's keep it for now
};
```

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;
  struct inode* ip;

  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }

  begin_op();
  if((ip = namei(path)) == 0){
    end_op();
    return -1;
  }
  ilock(ip);
  if(ip->type != T_FILE){
    iunlockput(ip);
    end_op();
    return -1;
  }
  if(ip->exec_perm == 0){
    cprintf("Operation execute failed\n");
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
```

1.6: ADDITIONAL CODE SNIPPETS

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_gethistory 22
#define SYS_sys_block  23
#define SYS_sys_unblock 24
#define SYS_chmod 25
```

```
// Built-in command: history
if(my_strncmp(buf, "history", 7) == 0){
  buf[strlen(buf)-1] = 0; // remove trailing newline
  gethistory();
  continue;
}

// Built-in command: block
if(my_strncmp(buf, "block ", 6) == 0){
  buf[strlen(buf)-1] = 0; // Remove newline
  int syscall_id = atoi(buf+6);
  if(sys_block(syscall_id) < 0)
    printf(2, "Failed to block system call %d\n", syscall_id);
  else
    printf(1, "System call %d blocked\n", syscall_id);
  continue;
}

// Built-in command: unblock
if(my_strncmp(buf, "unblock ", 8) == 0){
  buf[strlen(buf)-1] = 0; // Remove newline
  int syscall_id = atoi(buf+8);
  if(sys_unblock(syscall_id) < 0)
    printf(2, "Failed to unblock system call %d\n", syscall_id);
  else
    printf(1, "System call %d unblocked\n", syscall_id);
  continue;
}

//Built in command: chmod
if(my_strncmp(buf, "chmod", 5) == 0){
  char file[64];
  int mode;
  char *p = buf + 5;

  // Skip spaces after "chmod"
  while(*p == ' ')
    p++;

  // Check if a filename was provided
  if(*p == '\0'){
    printf(2, "usage: chmod <filename> <mode>\n");
    continue;
  }
```

One of the code snippets for ialloc (initialisation of disk inode permissions to 1)

```
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.exec_perm = 1;
  din.read_perm = 1;
  din.write_perm = 1;
  winode(inum, &din);
  return inum;
}
```