پیاده سازی سیستم کال تاریخچه

فاطمه راق(۴۰۰۱۳۰۴)

مقدمه

سیستم کال چیست؟

سیستم عامل در دو حالت عمل می‌کند: حالت هسته و حالت کاربر. دستورات اختصاصی مانند IN و OUT تنها در حالت هسته اجرا می‌شوند، در حالی که دستورات غیراختصاصی یا عادی در حالت کاربر اجرا می‌شوند. فرآیندهای کاربر در فضای آدرس پایین با امتیازات کاربر قرار دارند و کد هسته در فضای آدرس بالا با امتیازات هسته قرار دارد، که جلوگیری از دسترسی مستقیم توسط فرآیندهای کاربر را فراهم می‌کند. با این حال، برای وظایفی مانند دسترسی به سخت‌افزار، برنامه‌های کاربر نیاز به ارتباط با سیستم عامل دارند که به دلیل مسائل امنیتی منابع را مدیریت می‌کند. سیستم کالها به عنوان یک پل عمل می‌کنند که به برنامه‌های کاربر اجازه درخواست دسترسی به سخت‌افزار و منابع دیگر را می‌دهند. سیستم کالها توسط سیستم عامل پشتیبانی می‌شوند و به کاربران اجازه انجام وظایف مختلف از طریق آنها را می‌دهند.در این پروژه قصد پیاده سازی دو سیستم کال تاریخچه و top command را بر روی سیستم عامل xv6 داریم.

سیستم عامل xv6 به منظور اهداف آموزشی در دوره مهندسی سیستم‌های عامل دانشگاه MIT و همچنین در دوره طراحی سیستم‌های عامل دانشگاه ژورجیا تک و در بسیاری از دیگر مؤسسات استفاده می‌شود. xv6 یک پیاده‌سازی دوباره از Unix Version 6 (v6)) دنیس ریچی و کن تامپسون است. xv6 به طور میانبر ساختار و سبک v6 را دنبال می‌کند، اما برای یک چندپردازنده مبتنی بر x86 با استفاده از ANSI C پیاده‌سازی شده است.

بخش اول - سیستم کال history

Syscall.c

. این فایل شامل آرایه‌ای از اشاره‌گرهای تابع برای تمام توابع سیستم کال است. سیستم کال هیستوری را به موارد زیر اضافه می کنیم

پیاده سازی این سیستم کال در این فایل انجام نمی شود بلکه فقط پروتوتایپ آن را در این فایل اضافه می کنیم.

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_history(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
```

```
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_history] sys_history,
};
```

Syscall.h

در فایل syscall.c یک آرایه از اشاره‌گرهای تابع وجود دارد. برای اینکه به این آرایه دسترسی داشته باشیم، تعداد تماس‌های سیستمی را در فایل syscall.h تعریف خواهیم کرد. این عدد برای indexing در آرایه اشاره‌گرهای تابع استفاده شود.

```
define SYS_mkdir  20#
define SYS_close  21#
define SYS_history  22#
```

sysproc.c

پیاده سازی سیستم کال هیستوری را در این فایل انجام می دهیم .

```
int

sys_history(void)

{
```

```
  int historyId;
  if (argint(0, &historyId) < 0)
      return -1;
  if (historyId < 0 || historyId > MAX_HISTORY -1) {
      cprintf("Error: Invalid historyId. It should be between 0 and
%d\n", MAX_HISTORY-1);
      return -1;
  }
  return history(historyId);
}
```

استخراج آرگومان: از تابع argint برای استخراج historyId از آرگومان‌های تماس سیستمی استفاده می‌کند. در صورت عدم موفقیت در استخراج، یک خطا را بازمی‌گرداند.

بررسی می‌کند که historyId در محدوده معتبر قرار دارد یا خیر. اگر نه، یک پیام خطا را چاپ کرده و بازمی‌گرداند.

مدیریت دستورات تاریخ: اگر historyId برابر با 1 نباشد(تا خود دستور history را ذخیره نکند)، تابع saveToHistory را صدا می‌زند تا دستور "history" را ذخیره کند.

تماس تابع: تابع history را با historyId استخراج شده فرا می‌خواند و نتیجه آن را بازمی‌گرداند.

پیاده سازی این سیستم کال به شکل کامل نمی تواند در sysproc انجام شود زیرا نیاز به تغییر تابع consolintr و افزودن historylock داریم.

فایل console را بررسی می کنیم.

```
void
consoleintr(int (*getc)(void))
        // save command to history only when Enter is pressed
```

```
          if (c == '\n') {
            char command[INPUT_BUF];
            int i, j = 0;
            for (i = input.r; i < input.w; i++) {
              command[j++] = input.buf[i % INPUT_BUF];
            }
            command[j] = '\0';
            saveToHistory(command);
          }
        }
      }
      break;
    }
  }
  release(&cons.lock);
  if(doprocdump) {
    procdump();  // now call procdump() wo. cons.lock held
  }
}
```

`consoleintr` یک تابع است که با وقوع وقفه‌های ورودی کنسول سر و کار دارد. این تابع یک اشاره‌گر تابع به نام `getc` را به عنوان آرگومان می‌پذیرد، که یک تابع برای دریافت یک کاراکتر از ورودی کنسول است.

پس از پردازش کاراکترها، قفل را آزاد می‌کند تا قسمت‌های دیگر سیستم بتوانند به منابع مشترک دسترسی داشته باشند.

هنگامی که کلید Enter فشرده می‌شود (`c == '\n'`)، دستور وارد شده را در تاریخچه ذخیره می‌کند با کپی کردن آن از بافر ورودی به یک آرایه جداگانه به نام `command` و سپس فراخوانی تابع `saveToHistory`.

در ادامه توابع مربوط به تاریخچه را در فایل console بررسی می کنیم

```
struct HistoryBuffer {
    char bufferArray[MAX_HISTORY][INPUT_BUF];
    uint lengthArray[MAX_HISTORY];
    uint lastCommandIndex;
    int numOfCommandsInMem;
    int currentHistory;
};
struct spinlock history_lock;
struct HistoryBuffer historyBufferArray;
// Initialize the history lock
void history_init(void) {
    initlock(&history_lock, "history");
    historyBufferArray.numOfCommandsInMem = 0;
    historyBufferArray.currentHistory = -1;
}
int skipHistoryCommand(char* command) {
    char* prefix = "history";
    int prefixLength = strlen(prefix);
    return (strncmp(command, prefix, prefixLength) == 0) ? 1 :-1 ;
}
void saveToHistory(char* command) {
    // skip saving if command contains 'history' word(prefix)
    if (skipHistoryCommand(command)>0){
      return;
    }
    // acquire lock
    acquire(&history_lock);
```

```
    // Increment the circular buffer index
    historyBufferArray.lastCommandIndex =
(historyBufferArray.lastCommandIndex + 1) % MAX_HISTORY;
    // Store the new command in the circular buffer
strncpy(historyBufferArray.bufferArray[historyBufferArray.lastCommandIndex
], command, INPUT_BUF);
    historyBufferArray.lengthArray[historyBufferArray.lastCommandIndex] =
strlen(command);
// Update the number of commands in memory
    if (historyBufferArray.numOfCommandsInMem < MAX_HISTORY) {
        historyBufferArray.numOfCommandsInMem++;
    }
    // Update the current history position
    historyBufferArray.currentHistory = -1;
    // realease lock
    release(&history_lock);
}
// Function to print the command history
void printHistory(void) {
    int i;
    for (i = 0; i <= historyBufferArray.numOfCommandsInMem; i++) {
        cprintf("Command %d: %s\n", i + 1,
historyBufferArray.bufferArray[i]);
    }
}
void printSortedHistory(void) {
    int i;
    for (i = 0; i < historyBufferArray.numOfCommandsInMem ; i++) {
```

```
        cprintf("%s",
historyBufferArray.bufferArray[(historyBufferArray.lastCommandIndex - i +
MAX_HISTORY) % MAX_HISTORY]);
    }
}
int history(int historyId) {
    acquire(&history_lock);
    // Check if the historyId is within a valid range
    if (historyId < 0 || historyId >
historyBufferArray.numOfCommandsInMem-1) {
        cprintf("Error: Invalid historyId. It should be between 0 and
%d\n", historyBufferArray.numOfCommandsInMem-1);
        release(&history_lock);
        return -1;
    }
    printSortedHistory();
    // Find the index of the requested historyId in the circular buffer
    int index = (historyBufferArray.lastCommandIndex - historyId +
MAX_HISTORY) % MAX_HISTORY;
    // Print the requested command
    cprintf("requested command: %s\n",
historyBufferArray.bufferArray[index]);
    release(&history_lock);
    return 0;
```

این کد یک سیستم ساده‌ای از بافر تاریخچه تعریف می‌کند که تاریخچه دستورات را ذخیره و مدیریت می‌کند. اجازه دهید اجزای کلیدی و عملکردهای آن را توضیح دهم:

1. **ساختار HistoryBuffer:**

- `struct HistoryBuffer`: ساختار بافر تاریخچه را نمایان می‌کند. این شامل موارد زیر است:

- `bufferArray[MAX_HISTORY][INPUT_BUF]`: یک بافر دایره‌ای برای ذخیره رشته‌های دستور.

- `lengthArray[MAX_HISTORY]`: یک آرایه برای ذخیره طول هر دستور در بافر.
- `lastCommandIndex`: یک ایندکس به آخرین دستور ذخیره شده در بافر دایره‌ای.
- `numOfCommandsInMem`: تعداد فعلی دستورات ذخیره شده در حافظه.
- `currentHistory`: ایندکسی که موقعیت فعلی در تاریخچه را نشان می‌دهد و به -1 مقداردهی اولیه شده است.

- `struct spinlock history_lock`: یک قفل چرخشی برای همگام‌سازی دسترسی به بافر تاریخچه.

- `struct HistoryBuffer historyBufferArray`: یک نمونه از بافر تاریخچه.

2. **تابع `history_init`:**
- قفل تاریخچه را با استفاده از `initlock` مقداردهی اولیه می‌کند.
- مقدارهای اولیه برای اعضای `numOfCommandsInMem` و `currentHistory` را تعیین می‌کند.

3. **تابع `skipHistoryCommand`:**
- بررسی می‌کند که آیا یک دستور داده شده باید رد یا ذخیره شود بر اساس یک پیشوند ("history").
- اگر دستور باید رد شود، 1 را برمی‌گرداند و در غیر این صورت -1.

4. **تابع `saveToHistory`:**
- یک دستور را در بافر تاریخچه ذخیره می‌کند و در صورتی که حاوی پیشوند مشخص شده باشد، از ذخیره‌سازی آن خودداری می‌کند.
- قفل تاریخچه را به دست می‌آورد.
- ایندکس بافر دایره‌ای را افزایش می‌دهد.
- دستور جدید را در بافر دایره‌ای ذخیره می‌کند.
- طول آرایه و تعداد دستورات در حافظه را به‌روزرسانی می‌کند.
- قفل تاریخچه را آزاد می‌کند.

5. **تابع `printHistory`:**
- تاریخچه کامل دستورات را چاپ می‌کند.
- از طریق بافر تاریخچه حرکت کرده و هر دستور را چاپ می‌کند.

6. **تابع `printSortedHistory`:**

- تاریخچه دستورات را به ترتیب معکوس چاپ می‌کند.
- از طریق بافر تاریخچه به صورت معکوس حرکت کرده و هر دستور را چاپ می‌کند.

7. **تابع `history`:**

- دستور درخواستی را از بافر تاریخچه بر اساس شناسه تاریخچه چاپ می‌کند.
- قفل تاریخچه را به دست می‌آورد.
- بررسی می‌کند که آیا شناسه تاریخچه داده شده در محدوده معتبر است یا خیر.
- تاریخچه مرتب شده و دستور درخواستی را چاپ می‌کند.
- قفل تاریخچه را آزاد می‌کند.

Usys.s

برای اینکه برنامه‌های کاربری قادر به فراخوانی این تماس سیستمی شوند، نیاز به افزودن یک رابط کاربری وجود دارد. این رابط در فایل usys.S افزوده شده است. (پسوند فایلهای اسمبلی .S)

```
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(history)
```

user.h

اکنون، نیاز است تا پروتوتایپ تابعی که برنامه‌های کاربری آن را صدا می‌زنند، افزوده شود. این پروتوتایپ در فایل user.h افزوده شده است.

```
int sleep(int);
int uptime(void);
int history(int);
```

این تابع به تماس سیستمی نگاشته شده است از آرایه تماس‌های سیستمی که در فایل syscall.c با اندیس 22 تعریف شده‌اند و این اندیس در syscall.h مشخص شده است.

اکنون که با موفقیت تماس سیستمی را به xv6 اضافه کردیم، نیاز است یک برنامه‌ی کاربری کوچک بنویسیم تا این تماس سیستمی را صدا بزنیم.

Makefile

آخرین مرحله برای اجرای این برنامه‌ی کاربری، نیاز است که فایل Makefile را اصلاح کنیم. نام فایل برنامه‌ی کاربری را بدون پسوند فایل در بخش UPROGS فایل Makefile اضافه کنید.

```
UPROGS=\
    _cat\
     _grep\
  _zombie\
    _history\
```

برای هر سیستم کال یک فایل در کرنل نیز ایجاد شده است پس برای سیستم کال هیستوری نیز یک فایل به شکل زیر می سازیم , و در قسمت extra makefile اضافه می کنیم.

```
EXTRA=\
    mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c\
    history.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

History.c

```
#include "types.h"
#include "user.h"
```

```
int
main(int argc, char **argv)
{
  if(argc < 1 ){
    printf(2, "usage: history history_id\n");
    exit();
  }

  int historyId = atoi(argv[1]);

  if (history(historyId)<0){
    printf(2, "Error: uable to retrive history with id %s\n", argv[1]);
  }

  exit();
}
```

این برنامه یک برنامه کاربری سطح کاربر می باشد که یک دستور history را با توجه به id history ارائه شده بازیابی و چاپ کند. این برنامه انتظار دارد که history id را به عنوان یک آرگومان خط فرمان دریافت کند و در صورت عدم موفقیت در بازیابی تاریخ، یک پیام خطا چاپ کند. پیام استفاده نادرست نمایش داده می‌شود اگر تعداد صحیح آرگومان‌ها به برنامه ارائه نشده باشد.