گزارشکار آزمایش دوم سیستم عامل

پارسا احمدی ناو، آریا عازم، مجید صادقی نژاد

پرسش ها

- پرسش ۱:

کتابخانه‌های سطح کاربر موجود جزئیات فراخوانی‌های سیستمی را از دید برنامه‌نویسان پنهان می‌کنند. این کار از طریق تعریف توابع پوشاننده (Wrapper Functions) انجام می‌شود که به عنوان واسط بین برنامه‌های کاربر و سیستم‌عامل عمل می‌کنند. برای مثال، به جای اینکه برنامه‌نویس مستقیماً شماره فراخوانی سیستمی (System Call Number) را مشخص کند و مقادیر را به رجیسترها منتقل کند، این توابع پوشاننده این کار را به صورت خودکار انجام می‌دهند. در این فرآیند: شماره فراخوانی سیستمی به صورت ثابت در کد تعریف شده و توسط توابع کتابخانه استفاده می‌شود. و آرگومان‌های فراخوانی سیستمی به صورت خودکار در مکان‌های مناسب (مانند رجیسترها یا استک) قرار داده می‌شوند. فراخوانی سیستمی از طریق دستورهای اسمبلی (مانند int یا syscall) انجام می‌شود. مقدار بازگشتی از فراخوانی سیستمی پردازش شده و به برنامه‌نویس بازگردانده می‌شود. استفاده از این انتزاع چندین مزیت دارد: (Portability): برنامه‌ها می‌توانند بدون تغییر یا با تغییرات کم روی سیستم‌عامل‌های مختلف اجرا شوند. افزایش امنیت: پنهان کردن جزئیات سطح پایین می‌تواند از بروز خطاهای امنیتی ناشی از استفاده نادرست از فراخوانی‌های سیستمی جلوگیری کند. سادگی توسعه: برنامه‌نویسان می‌توانند بدون نیاز به درک عمیق از جزئیات سیستم‌عامل، برنامه‌های خود را توسعه دهند.

- پرسش ۲:

(Virtual File Systems): برنامه‌های کاربر می‌توانند از طریق فایل‌های سیستمی مجازی مانند proc و sys به اطلاعات کرنل دسترسی پیدا کنند یا تنظیمات آن را تغییر دهند. این فایل‌ها به صورت پویا توسط کرنل تولید می‌شوند و امکان خواندن یا نوشتن اطلاعات مربوط به وضعیت سیستم را فراهم می‌کنند. کتابخانه‌های سطح کاربر: کتابخانه‌هایی مانند glibc توابعی را ارائه می‌دهند که به صورت غیرمستقیم با کرنل تعامل دارند. این توابع جزئیات فراخوانی‌های سیستمی را پنهان کرده و رابط‌های ساده‌تری برای برنامه‌نویسان فراهم می‌کنند. ماژول‌های کرنل (Kernel Modules): برنامه‌های کاربر می‌توانند از طریق ماژول‌های کرنل که به صورت پویا بارگذاری می‌شوند، با کرنل تعامل داشته باشند. این ماژول‌ها می‌توانند رابط‌های خاصی برای برنامه‌های کاربر فراهم کنند. همچنین برنامه‌های کاربر می‌توانند از طریق فایل‌های دستگاه (Device Files) که در دایرکتوری dev قرار دارند، با درایورهای

سخت‌افزار تعامل داشته باشند. این فایل‌ها واسطی برای ارتباط با سخت‌افزار یا شبیه‌سازی آن هستند.

- پرسش ۳:

در xv6، تنها از Trap Gate برای فراخوانی‌های سیستمی استفاده می‌شود، زیرا این نوع گیت امکان دسترسی از سطح کاربر (DPL_USER) را فراهم می‌کند. Interrupt Gate معمولاً برای مدیریت وقفه‌های سخت‌افزاری استفاده می‌شود و سطح دسترسی آن به گونه‌ای تنظیم شده است که فقط کرنل می‌تواند از آن استفاده کند. به همین دلیل، این گیت برای فراخوانی‌های سیستمی که از سطح کاربر انجام می‌شوند، مناسب نیست.همچنین Trap Gate برخلاف Interrupt Gate، وقفه‌ها را غیرفعال نمی‌کند. این ویژگی برای فراخوانی‌های سیستمی مهم است، زیرا کرنل ممکن است نیاز داشته باشد که وقفه‌ها در طول اجرای فراخوانی سیستمی فعال بمانند.

- پرسش ۴:

در صورت تغییر سطح دسترسی، مقادیر SS و ESP روی پشته Push می‌شوند، زیرا این مقادیر نشان‌دهنده وضعیت پشته در سطح قبلی هستند و برای بازگشت به آن سطح ضروری‌اند. اما در غیر این صورت (یعنی زمانی که سطح دسترسی تغییر نمی‌کند)، نیازی به Push کردن این مقادیر نیست. دلیل این موضوع این است که در همان سطح دسترسی، پشته تغییری نمی‌کند و پردازنده نیازی به ذخیره این اطلاعات ندارد. به عبارت دیگر، SS و ESP فقط زمانی ذخیره می‌شوند که انتقال بین دو سطح مختلف (مانند User Mode به Kernel Mode) رخ دهد.

- پرسش ۵:

در سیستم‌عامل xv6، توابعی مانند argint و argptr برای بازیابی پارامترهای فراخوانی سیستمی از پشته کاربر استفاده می‌شوند. این توابع نقش مهمی در بررسی و اطمینان از صحت آرگومان‌های ورودی دارند. نقش تابع argptr: این تابع برای بازیابی یک اشاره‌گر (Pointer) از پشته کاربر استفاده می‌شود. همچنین بررسی می‌کند که آیا آدرس اشاره‌گر معتبر است و به یک بخش مجاز از حافظه اشاره می‌کند یا خیر. این بررسی برای جلوگیری از دسترسی غیرمجاز به حافظه کرنل یا سایر فرآیندها ضروری است. نقش تابع argint: این تابع برای بازیابی مقادیر عددی (Integer) از پشته کاربر استفاده می‌شود. این مقادیر معمولاً به عنوان آرگومان‌های ساده برای فراخوانی‌های سیستمی استفاده می‌شوند.اگر تابع `argptr` آدرس‌های ورودی را بررسی نکند، این مشکلات ممکن است رخ دهد: دسترسی به حافظه غیرمجاز، اجرای نادرست فراخوانی سیستمی، آسیب‌پذیری امنیتی

در فراخوانی سیستمی `read_sys`، کرنل داده‌ها را از یک فایل خوانده و در بافر کاربر ذخیره می‌کند. اگر آدرس بافر توسط `argptr` بررسی نشود: ممکن است آدرس بافر به یک بخش غیرمجاز از حافظه اشاره کند و داده‌ها به جای ذخیره در بافر کاربر، به حافظه دیگری نوشته شوند. این موضوع می‌تواند باعث خرابی داده‌ها یا افشای اطلاعات حساس شود.

بررسی گام های اجرای فراخوانی سیستمی توسط gdb

ابتدا برنامه ی تست را می نویسیم:

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

int main(void) {
    int pid = getpid();
    printf(1, "Process ID: %d\n", pid);
    exit();
}
```

سپس طبق خواسته ی گزارش مراحل را تا دستور bt طی می کنیم:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:150
150       struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:150
#1  0x80106efd in trap (tf=0x8dffefb4) at trap.c:43
#2  0x80106cac in alltraps () at trapasm.S:20
#3  0x8dffefb4 in ?? ()
```

دستورbt که مخفف backtrace است stack call برنامه در لحظه کنونی را نشان می دهد. هر تابع که صدا زد همی شود یک frame stack مخصوص به خودش را می گیرد که متغیرهای محلی و آدرس بازگشت و غیره در آن ذخیره می شود . خروجی این دستور در هر خط یک frame stack را نشان می دهد.

۱. تابع syscall

این تابع در فایل syscall.c قرار دارد و مسئول پردازش فراخوانی‌های سیستمی است. زمانی که یک برنامه کاربر فراخوانی سیستمی مانند getpid را اجرا می‌کند، شماره فراخوانی سیستمی (System Call Number) در رجیستر eax ذخیره می‌شود.این تابع شماره فراخوانی سیستمی را از `Trap Frame` (`tf->eax`) می‌خواند و سپس تابع مربوطه را از جدول فراخوانی‌های سیستمی (`[]syscall`) اجرا می‌کند. در این مرحله، کرنل تشخیص می‌دهد که کدام فراخوانی سیستمی باید اجرا شود.

۲. تابع trap

این تابع در فایل trap.c قرار دارد و مسئول مدیریت وقفه‌ها (Interrupts) و تله‌ها (Traps) است. زمانی که یک برنامه کاربر فراخوانی سیستمی را درخواست دهد CPU به حالت کرنل (Kernel Mode) سوئیچ می‌کند و این تابع فراخوانی می‌شود. آرگومان `Trap Frame` (`tf`) شامل اطلاعاتی مانند مقادیر رجیسترها (از جمله `eax`) و آدرس بازگشت (Return Address) است. اگر وقفه مربوط به یک فراخوانی سیستمی باشد (کد وقفه `T_SYSCALL`)، این تابع، کنترل را به تابع `syscall` منتقل می‌کند.

۳. تابع alltraps

این تابع در فایل اسمبلی trapasm.S تعریف شده است و نقطه ورود (Entry Point) برای تمام وقفه‌ها و تله‌ها است. زمانی که CPU یک وقفه یا تله را شناسایی می‌کند، اجرای برنامه به این تابع منتقل می‌شود. این تابع مقادیر رجیسترها را در یک ساختار `trapframe` ذخیره می‌کند و سپس تابع `trap` را فراخوانی می‌کند. به عبارت دیگر، این تابع نقش آماده‌سازی و انتقال کنترل به تابع `trap` را بر عهده دارد.

۴. آدرس بازگشت

این آدرس نشان‌دهنده محل بازگشت به برنامه کاربر پس از اتمام فراخوانی سیستمی است. زمانی که اجرای فراخوانی سیستمی کامل شد، کرنل از این آدرس برای بازگشت به برنامه کاربر استفاده می‌کند.

سپس دستور down را اجرا می کنیم:

```
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
```

از آنجایی که در حال حاضر در درونی ترین لایه frame قرار داریم، نیاز به وارد کردن دستور down که به یک لایه درونی تر میرود، نداریم. لذا با وارد کردن down به ارور بالا بر میخوریم

حال مقدار eax را در هیت شدن های مختلف بریک پوینت چاپ می کنیم:

در ابتدا مقدار آن ۵ است که نشان دهنده ی سیستم کال read است زیر ابتدا از ترمینال مقدار ورودی ما خوانده می شود

```
(gdb) print ((struct trapframe *)0x8dffefb4)->eax
$1 = 5
```

سپس به ترتیب سیستم کال های fork, wait, sbrk, exec اجرا می شوند

```
(gdb) print ((struct trapframe *)0x8dffefb4)->eax
$7 = 1
```

```
(gdb) print ((struct trapframe *)0x8dffefb4)->eax
$8 = 3
```

و در نهایت به سیستم کال getpid می رسم که شماره ی آن ۱۱ است:

```
(gdb) print ((struct trapframe *)0x8dfc6fb4)->eax
$36 = 11
```

Next palindrome

تعریف شماره فراخوانی سیستمی: در فایل syscall.h، یک شماره جدید برای فراخوانی سیستمی تعریف می‌کنیم. این شماره برای شناسایی فراخوانی سیستمی در کرنل استفاده می‌شود.زیرا هر فراخوانی سیستمی در xv6 با یک شماره یکتا شناسایی می‌شود. این شماره در رجیستر `eax` ذخیره می‌شود و کرنل از آن برای شناسایی فراخوانی استفاده می‌کند.

```
#define SYS_getcmostime 28
#define SYS_next_palindrome 29
```

اضافه کردن فراخوانی سیستمی به جدول فراخوانی‌ها: در فایل syscall.c، تابع مربوط به فراخوانی سیستمی (`sys_next_palindrome`) را به آرایه `[]syscalls` اضافه می‌کنیم. زیرا این آرایه شماره فراخوانی سیستمی را به تابع مربوطه در کرنل نگاشت می‌کند. وقتی شماره فراخوانی سیستمی دریافت می‌شود، کرنل از این آرایه برای اجرای تابع مناسب استفاده می‌کند.

```
[SYS_getcmostime] sys_getcmostime,
[SYS_next_palindrome] sys_next_palindrome,
};
```

پیاده‌سازی تابع کرنل: در فایل sysproc.c، تابع `sys_next_palindrome` را پیاده‌سازی می‌کنیم. این تابع آرگومان ورودی را از کاربر دریافت می‌کند و تابعی را برای محاسبه عدد بعدی که یک پالیندروم است، فراخوانی می‌کند.این تابع مستقیماً توسط کرنل اجرا می‌شود و وظیفه پردازش ورودی و بازگرداندن خروجی به برنامه کاربر را بر عهده دارد.

```
int
sys_next_palindrome(void)
{
    int num;
    if (argint(0, &num) < 0)
        return -1;
    return next_palindrome(num);
}
```

اضافه کردن تابع کمکی: تابعی برای محاسبه عدد پالیندروم بعدی پیاده‌سازی می‌کنیم.

```c
int
next_palindrome(int num)
{
    num++;
    while (1) {
        int reversed = 0, original = num;
        while (original > 0) {
            reversed = reversed * 10 + original % 10;
            original /= 10;
        }
        if (reversed == num)
            return num;
        num++;
    }
}
```

اضافه کردن فراخوانی سیستمی به فضای کاربر: در فایل user.h، یک declaration برای فراخوانی سیستمی `next_palindrome` اضافه می‌کنیم. و در فایل usys.S، ورودی مربوط به `next_palindrome` را اضافه می‌کنیم زیرا مسئول ایجاد یک wrapper برای فراخوانی سیستمی است که برنامه‌های کاربر بتوانند به راحتی از آن استفاده کنند.

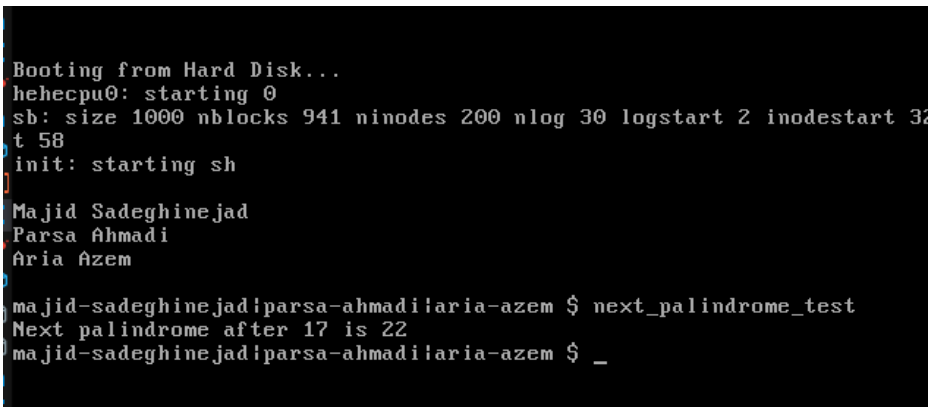
```c
int getcmostime(struct rtcdate*);
int next_palindrome(int num);
```

```
SYSCALL(set_sleep)
SYSCALL(getcmostime)
SYSCALL(next_palindrome)
```

نوشتن برنامه ی سطح کاربر برای تست: در نهایت یک برنامه سطح کاربر برای تست فراخوانی سیستمی طراحی شده می نویسیم و آن را اجرا می کنیم.

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

int main(void) {
    int num = 17;
    int result = next_palindrome(num);
    printf(1, "Next palindrome after %d is %d\n", num, result);
    exit();
}
```



فراخوانی سیستمی Logs

برای اضافه کردن این سری فراخوانی‌ها، ابتدا باید در ادامه فایل syscall.h برای هر فراخوانی جدید یک آیدی عدد اختصاص دهیم. چون در این بخش ۴ تا فراخوانی اضافه می‌کنیم، اعداد را از ۲۲ شروع کرده و تا ۲۵ ادامه می‌دهیم:

```
23
24   // I added:
25   #define SYS_make_user 22
26   #define SYS_login     23
27   #define SYS_logout    24
28   #define SYS_get_log   25
```

همچنین در فایل syscall.c این اعداد را مانند syscallهای دیگر به mapping اضافه می‌کنیم تا بتوانیم تابع مورد نظر را به وسیله این عدد فراخوانی کنیم:

```
// I added:
[SYS_make_user] sys_make_user,
[SYS_login]     sys_login,
[SYS_logout]    sys_logout,
[SYS_get_log]   sys_get_log,
```

به همین طریق اسم توابع را بدون پیشوند sys به usys.S اضافه می‌کنیم:

```
SYSCALL(make_user)
SYSCALL(login)
SYSCALL(logout)
SYSCALL(get_log)
SYSCALL(diff)
SYSCALL(set_sleep)
SYSCALL(getcmostime)
```

فایل جدیدی به نام user.c ایجاد کرده و این توابع را در این فایل پیاده‌سازی می‌کنیم. در فایل proc.h دو استراکت به نام systemcall و user ایجاد می‌کنیم و در آن‌ها اطلاعات مورد نیاز را نگهداری می‌کنیم.

```c
#define MAX_USERS 16
#define MAX_PASSWORD_LEN 32
#define MAX_LOG_ENTRIES 50

struct user {
  int user_id;
  char password[MAX_PASSWORD_LEN];
  int valid; // 1 if this user slot is in use
};

struct syscall_log {
  int syscall_num;
  int pid;
};
```

در اول فایل user.c آرایه‌های مربوط به ذخیره‌سازی یوزرها و log ها را تعریف می‌کنیم. یک آرایه برای ذخیره‌سازی systemcall های گلوبال اختصاص می‌دهیم. یک آرایه دو بعدی برای ذخیره فراخوانی‌های اختصاصی هر کاربر می‌سازیم.

یک متغیر درنظر می‌گیریم که در صورت login بودن کاربر، نشان‌دهنده id کاربر است. در غیر این صورت منفی یک را نشان می‌دهد. برای پیاده‌سازی login و logout ابتدا باید id کاربر و index ذخیره شده آن را پیدا کنیم و در صورت موفقیت‌آمیز بودن ورود، این عدد را به آیدی کاربر تغییر می‌دهیم. برای پیاده‌سازی make_user هم به همین صورت باید عمل کنیم و وارد جزییات پیاده‌سازی آن نسبت به login و logout نمی‌شویم.

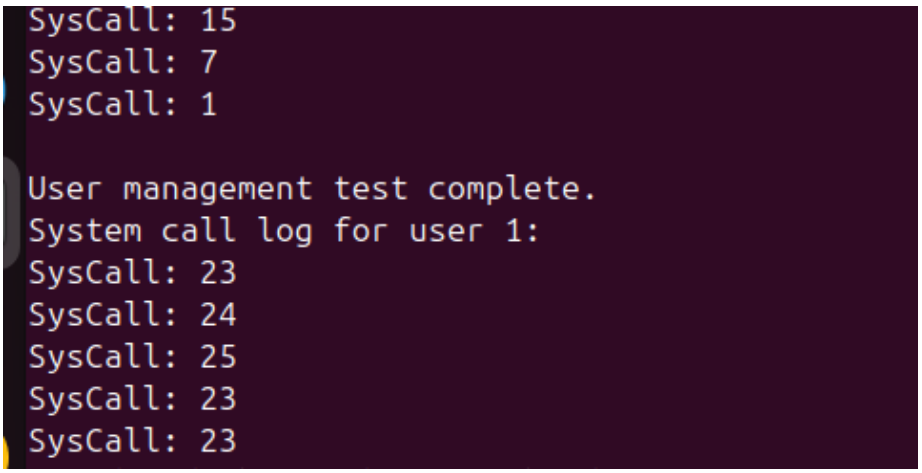
به یک تابع نیاز داریم که هر فراخوانی را log کند. به این صورت که این تابع، شماره فراخوانی را به آرایه global systemcall اضافه می‌کند. اگر کاربری هم وارد شده باشد، این شماره را به سیستم کال‌های آن کاربر هم اضافه می‌کند. این تابع را در تابع systemcall در فایل systemcall.c اضافه میکنیم تا فراخوانی‌های معتبر لاگ شوند. getlog هم به این صورت است که بسته به login بودن کاربر، فراخوانی‌های مناسب را چاپ می‌کند.

در فایل make_user_test این توابع را تست می‌کنیم. با نوشتن دستور make_user_test می‌توان نتیجه اجرای این توابع را مشاهده کرد. در این بخش زدن رمز اشتباه و لاگین اشتباه و ... تست شده است. نمونه اجرا را در پایین می‌توان مشاهده کرد:

```
majid-sadeghinejad|parsa-ahmadi|aria-azem $ make_user_test
Creating users...
Create user 1: SUCCESS
Create user 2: SUCCESS
Create duplicate user: FAILED (expected to fail)

Testing login...
Login user 1: SUCCESS
Login while logged in: FAILED (expected to fail)

Getting system call log for current user...
System call log for user 1:
SysCall: 23
SysCall: 23
```



Diff syscall

1. تخصیص شماره فراخوان سیستم

- یک شماره فراخوان سیستم جدید با نام `SYS_diff` و مقدار 26 در فایل `syscall.h` تعریف شده است.
  این تخصیص به سیستم اجازه می‌دهد که این فراخوان را از سایر فراخوان‌های موجود متمایز کند.

2. اعلان فراخوان سیستم

- در فایل `syscall.c` تابع `sys_diff()` به عنوان فراخوان سیستم اعلام و به ثبت رسیده است.

- همچنین، از طریق اعلان تابع `int diff(const char* file1, const char* file2)` در فایل `user.h` این فراخوان سیستم در برنامه‌های کاربری در دسترس قرار گرفته است.

3. پیاده‌سازی فراخوان سیستم

- در فایل `sysfile.c` پیاده‌سازی فراخوان سیستم `diff` به گونه‌ای صورت گرفته که یک ابزار مقایسه جامع بین محتویات دو فایل ایجاد می‌کند. از ویژگی‌های این پیاده‌سازی می‌توان به موارد زیر اشاره کرد:

- **گرفتن آرگومان‌های فایل:** دو نام فایل به‌عنوان ورودی از طریق تابع `argstr()` دریافت می‌شود.

- **بازکردن فایل‌ها و بررسی خطا:** ابتدا فایل‌ها باز شده و از وجود آنها، غیر بودن نوع دایرکتوری و سایر خطاهای ممکن بررسی می‌شود.

- **مقایسه خط به خط:** فایل‌ها به صورت خط به خط خوانده شده و در صورت بروز تفاوت، شماره خط مربوطه گزارش می‌شود.

- **تشخیص تعداد خطوط متفاوت:** در صورتیکه تعداد خطوط فایل‌ها با یکدیگر تفاوت داشته باشد، این مورد نیز شناسایی و گزارش می‌شود.

- **نتیجه مقایسه:** در صورتی که هیچ تفاوتی بین دو فایل یافت نشود، مقدار 0 و در غیر این صورت مقدار -1 برگردانده می‌شود.

4. پیاده‌سازی برنامه کاربری

- برنامه آزمایشی `difftest.c` ایجاد شده است تا فراخوان سیستم `diff` را مورد آزمایش قرار دهد.

- این برنامه با دریافت دو نام فایل از طریق آرگومان‌های خط فرمان، فراخوان سیستم `diff` را فراخوانی می‌کند.

- در صورتی که نتیجه فراخوان برابر با 0 باشد، پیغام «Files are identical» (فایل‌ها یکسان هستند) به کاربر نمایش داده می‌شود.

- در ضمن، وظیفه گزارش تفاوت‌ها به عهده فراخوان سیستم است که اطلاعات دقیق در مورد تفاوت‌ها را در صورت وجود ارائه می‌دهد.

```
majid-sadeghinejad|parsa-ahmadi|aria-azem $ echo "This is a test file" > test1.t
xt
majid-sadeghinejad|parsa-ahmadi|aria-azem $ echo "This is not a test file" >test
2.txt
majid-sadeghinejad|parsa-ahmadi|aria-azem $ echo "This is a tetost
"This is a tetost
majid-sadeghinejad|parsa-ahmadi|aria-azem $ echo "This is a test file" > test3.t
xt
majid-sadeghinejad|parsa-ahmadi|aria-azem $ difftest test1.txt test2.txt
Line 1: files differ
< "This is a test file"
> "This is not a test file"
majid-sadeghinejad|parsa-ahmadi|aria-azem $ difftest test1.txt test3.txt
Files are identical
majid-sadeghinejad|parsa-ahmadi|aria-azem $
```

Set sleep syscall

در ادامه گزارشی جامع به زبان فارسی ارائه شده است که مراحل پیاده‌سازی فراخوانی سیستمی `sys_set_sleep` را شرح می‌دهد:

۱. ثبت فراخوانی سیستمی (System Call Registration)

- **تعریف شماره فراخوانی سیستمی:**
  فراخوانی سیستمی `sys_set_sleep` به عنوان فراخوانی شماره ۲۷ در فایل `syscall.h` ثبت شده است.

- **تعریف رابط کاربری:**
  این فراخوانی در فایل `user.h` به صورت تابع `int set_sleep(int)` تعریف شده است.
  در فایل اسمبلی `usys.S` نیز یک نقطه ورود از فضای کاربری ایجاد می‌شود که با استفاده از ماکروی SYSCALL فراخوانی اصلی متصل می‌شود.

۲. پیاده‌سازی در فایل `sysproc.c`

در فایل `sysproc.c`، پیاده‌سازی اصلی فراخوانی سیستمی `sys_set_sleep` صورت گرفته است.

۳. تفاوت‌های اصلی نسبت به فراخوانی‌های خواب معمولی

- **مدیریت قفل‌ها (Lock Handling):**
  پیاده‌سازی `sys_set_sleep` با دقت به ترتیب دریافت و آزادسازی قفل‌ها توجه ویژه‌ای دارد.
  به‌طور مثال، قفل `tickslock` پیش از دریافت قفل `ptable.lock` آزاد می‌شود تا از بن‌بست جلوگیری شود.

- **بررسی مجدد زمان:**
در هنگام دریافت مجدد قفل‌ها، بررسی می‌شود که آیا زمان خواب به پایان رسیده است یا خیر و در صورت اتمام زمان، فرایند بلافاصله از حلقه خارج می‌شود.

- **مدیریت وضعیت فرآیند:**
فرآیند با تنظیم وضعیت به SLEEPING علامت‌گذاری شده و کانال انتظار آن به متغیر `ticks` تنظیم می‌شود. سپس از طریق فراخوانی `sched()` به زمانبندی تحویل داده می‌شود تا CPU به فرآیندهای دیگر اختصاص یابد.

۴. کاربرد فراخوانی سیستمی در برنامه‌ها

- **آزمایش عملکرد:**
برنامه‌هایی مانند `sleep_test.c` از این فراخوانی سیستمی برای تست و اطمینان از عملکرد صحیح سیستم خواب استفاده می‌کنند.

- **تعیین مدت زمان:**
مدت زمان خواب به صورت تیک‌های زمانی (timer ticks) تعیین می‌شود و نه به ثانیه.

۵. مدیریت وضعیت فرآیند (Process State Management)

- **علامت‌گذاری فرآیند:**
فرآیند با تنظیم `p->state = SLEEPING` علامت‌گذاری می‌شود و کانال انتظار آن به `&ticks` تنظیم می‌گردد تا عملیات بیدارسازی به شکل صحیح انجام شود.

- **بیدارسازی فرآیند:**
زمانی که رویداد وقفه تایمر (`ticks` افزایش پیدا می‌کند)، فرآیندهای در حالت خواب که بر روی این کانال منتظر هستند، بیدار می‌شوند و توسط زمانبندی دوباره وارد صف اجرایی می‌شوند.

برنامه ی سطح کاربر

تفاوت جزئی عمدتاً به دلیل آن است که دو روش اندازه‌گیری زمان، مبتنی بر مکانیزم‌ها و وضوح‌های متفاوتی هستند که هر کدام دارای سربار و بروزرسانی خاص خود می‌باشند :

1. **زمان‌بندی مبتنی بر تیک در مقابل ساعت واقعی**
فراخوانی سیستمی که پیاده‌سازی کرده‌اید از متغیر سراسری ticks استفاده می‌کند که در فواصل

زمانی ثابت (معمولاً مرتبط با وقفه‌های تایمر سخت‌افزاری) بروزرسانی می‌شود. این مکانیزم زمان را به صورت گسسته (تیک‌ها) اندازه‌گیری می‌کند. در مقابل، فراخوانی سیستمی cmostime ساعت واقعی سیستم را می‌خواند که ممکن است دارای وضوح و مکانیزم به‌روزرسانی متفاوتی باشد. این اختلاف مبانی اندازه‌گیری می‌تواند به تفاوت‌های جزئی در بازه‌های اندازه‌گیری شده منجر شود.

2. **وضوح و فرکانس به‌روزرسانی**
   تیک‌ها در فواصل زمانی ثابت شمارش می‌شوند و ممکن است دقت ثبت یک تیک نسبت به زمان واقعی کمی نامعلوم باشد. به عنوان مثال، اگر یک پردازه درست پیش از به‌روزرسانی تیک زمان‌بندی شده باشد، مدت زمان انتظار ممکن است اندکی بیشتر از حد انتظار شود. از سوی دیگر، ساعت واقعی ممکن است با فرکانس یا تأخیر بروزرسانی متفاوتی عمل کند، که باعث تفاوت در اندازه‌گیری زمان می‌شود.

3. **سربار فراخوانی سیستم و تأخیرهای زمان‌بندی**
   هرگونه تأخیر پردازشی اضافی ناشی از مکانیزم فراخوانی سیستمی—مانند جابجایی زمینه، پردازش وقفه‌ها و سایر سربارهای زمان‌بندی کرنل—می‌تواند دقت زمان‌بندی را تحت تأثیر قرار دهد. این تأخیرها ممکن است باعث شوند زمان پایان ثبت شده توسط cmostime کمی از لحظه دقیق پایان دوره خواب مبتنی بر تیک فاصله بگیرد.

4. **تفاوت در مبانی اندازه‌گیری**
   از آنجا که شمارنده تیک و ساعت واقعی به‌طور مستقل نگهداری می‌شوند، مبداهای شروع آنها ممکن است کاملاً هماهنگ نباشد. حتی اگر هر دو در دراز مدت دقیق باشند، تفاوت‌های کوچک در مبنا یا شیوه به‌روزرسانی آن‌ها می‌تواند باعث بروز اختلاف‌های جزئی بین مدت زمان خواب مورد انتظار و اختلاف زمان اندازه‌گیری شده با استفاده از cmostime شود.

به طور خلاصه، اگرچه هر دو روش هدف اندازه‌گیری زمان را دنبال می‌کنند، تفاوت‌های ذاتی در نحوه به‌روزرسانی و سربارهای پردازشی مرتبط باعث می‌شود تفاوت‌های جزئی در اختلاف زمانی مشاهد شده به وجود آید. این عوامل—افزایش تدریجی تیک‌ها، وضوح به‌روزرسانی و سربار سیستم—تفاوت مشاهده شده را توضیح می‌دهند.

```
majid-sadeghinejad¦parsa-ahmadi¦aria-azem $ sleep_test 10000
Putting process to sleep for 10000 ticks...
Starting sleep at tick: 3788, time: 17:36:34
Woke up at tick: 13789, time: 17:38:14

--- Results ---
Requested sleep duration: 10000 ticks
Actual sleep duration: 10001 ticks
Elapsed real time: 100 seconds
Estimated ticks per second: 100

Note: Any difference between requested and actual sleep time
may be due to scheduler latency and timer resolution.
majid-sadeghinejad¦parsa-ahmadi¦aria-azem $ _
```