سیستم های عامل

تمرین سری دوم

مصطفی فضلی 9822803

استاد علیرضا تجری 1400/01/24

دانشگاه صنعتی شاهرود

سیستم های عامل

تمرین سری دوم

1) فراخوانی سیستمی نوشتن در فایل fs/read_write.c موجود در کاپایلر قرار گرفته است، این فایل بدین صورت است :

```
2)  SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
3)          size_t, count)
4)  {
5)      struct fd f = fdget_pos(fd);
6)      ssize_t ret = -EBADF;
7)
8)      if (f.file) {
9)          loff_t pos = file_pos_read(f.file);
10)         ret = vfs_write(f.file, buf, count, &pos);
11)         if (ret >= 0)
12)             file_pos_write(f.file, pos);
13)         fdput_pos(f);
14)     }
15)
16)     return ret;
17) }
```

در ابتدا با استفاده از تابع SYSCALL_DEFINE3 ماکروی تعریف شده در include/linux/syscalls.h هدر فایل و تعاریف موجود برای تعریف کارکرد تابع sys_name(...) را گسترش می دهد، این تابع بدین صورت تعریف شده است :

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)

#define SYSCALL_DEFINEx(x, sname, ...)              \
        SYSCALL_METADATA(sname, x, __VA_ARGS__)     \
        __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```

همانطور که مشاهده می شود، این فراخوانی به صورت define تعریف شده است و پارامتر name را می گیرد که بیانگر نام یک فراخوانی و تعداد متغیر پارامتر ها است. این ماکرو نیز به ماکروی SYSCALL_DEFINEx (ماکرویی که تعدا پارامتر های فراخوانی سیستم را میگیرد) گسترش می یابد. در مرحله بعد همین ماکرو به ماکروی دیگر به نام های زیر با عبارت define گسترش می یابد:

- SYSCALL_METADATA;
- __SYSCALL_DEFINEx.

پیاده سازی اولین ماکرو به CONFIG_FTRACE_SYSCALLS کرنل وابستگی دارد، از نام این گزینه آشکار است که ردیابی برای گرفتن فراخوانی های ورودی و خروجی به کار میرود. اگر این گزینه در پیکربندی کرنل فعال باشد، SYSCALL_METADATA مقدار دهی اولیه ساختمان syscall_metadata را که در فایل هدر include/trace/syscall.h تعریف شده است و شامل فیلد های کاربردی مختلف مانند نام فراخوانی سیستمی را انجام میدهد، تعداد فراخوانی سیستم ها در جدول فراخوانی سیستم ها دارای
پارامتر های متعددی از جمله تعدا یک فراخوانی سیستمی، نوع آن و... است که در ادامه میتوانید بخش از این جدول را مشاهده کنید :

سیستم های عامل

تمرین سری دوم

```
#define SYSCALL_METADATA(sname, nb, ...)                        \
    ...                                                         \
    ...                                                         \
    ...                                                         \
    struct syscall_metadata __used                              \
            __syscall_meta_##sname = {                          \
                .name           = "sys"#sname,                  \
                .syscall_nr     = -1,                           \
                .nb_args        = nb,                           \
                .types          = nb ? types_##sname : NULL,    \
                .args           = nb ? args_##sname : NULL,     \
                .enter_event    = &event_enter_##sname,         \
                .exit_event     = &event_exit_##sname,          \
                .enter_fields   = LIST_HEAD_INIT(__syscall_meta_##sname.enter_fields), \
            };                                                  \

    static struct syscall_metadata __used                       \
            __attribute__((section("__syscalls_metadata")))     \
            *__p_syscall_meta_##sname = &__syscall_meta_##sname;
```

هنگامی که CONFIG_FTRACE_SYSCALLS در کرنل فعال نباشد، ماکروی SYSCALL_METADATA تنها یک رشته خالی مانند زیر را تحویل می دهد :

```
#define SYSCALL_METADATA(sname, nb, ...)
```

و اما ماکروی دوم که __SYSCALL_DEFINEx نام داشت، بدین گونه پیاده سازی و گسترش یافته است :

```
#define __SYSCALL_DEFINEx(x, name, ...)                         \
    asmlinkage long sys##name(__MAP(x,__SC_DECL,__VA_ARGS__))   \
        __attribute__((alias(__stringify(SyS##name))));         \
                                                                \
    static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__));  \
                                                                \
    asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__));  \
                                                                \
    asmlinkage long SyS##name(__MAP(x,__SC_LONG,__VA_ARGS__))   \
    {                                                           \
        long ret = SYSC##name(__MAP(x,__SC_CAST,__VA_ARGS__));  \
        __MAP(x,__SC_TEST,__VA_ARGS__);                         \
        __PROTECT(x, ret,__MAP(x,__SC_ARGS,__VA_ARGS__));       \
        return ret;                                             \
    }                                                           \
                                                                \
    static inline long SYSC##name(__MAP(x,__SC_DECL,__VA_ARGS__))
```

سیستم های عامل

تمرین سری دوم

حال به بررسی این کد بالا می پردازیم:

تعریف ابتدایی sys##name یک تابع کنترل کننده با اسم sys_system_call_name است .
ماکروی __SC_DECL، ماکروی __VA_ARGS__ را گرفته و پارامتر های فراخوانی ورودی و نوع و اسم و... آن ها را ترکیب میکند...
حال ما با اطلاعاتی که داریم، میتوانیم به طریقه پیاده سازی فراخوانی سیستمی write بپردازیم :

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    struct fd f = fdget_pos(fd);
    ssize_t ret = -EBADF;

    if (f.file) {
        loff_t pos = file_pos_read(f.file);
        ret = vfs_write(f.file, buf, count, &pos);
        if (ret >= 0)
            file_pos_write(f.file, pos);
        fdput_pos(f);
    }

    return ret;
}
```

تعدادی از آرگومان های این کد بدین صورت است :

- fd - file descriptor;
- buf - buffer to write;
- count - length of buffer to write.

و اطلاعات را از روی فایل یا دستگاه ورودی گرفته و می نویسد، پارامتر buf به صورت __user تعریف شده است که هدف اصلی آن بررسی کد هسته لینوکس با کمترین استفاده است. آن در فایل هدر include/linux/compiler.h تعریف شده است و وابستگی به __CHECKER__ دارد.
همه این ها اطلاعت مفیدی برای فراخوانی سیستمی sys_write است، f که دارای ساختار fd است توصیفگر فایل را در هسته فرا میخواند و می تواند نتیجه را در تابع fdget_pos قرار دهیم. این تابع در فایل سورس کدی مشابه تابع __to_fd به صورت زیر تعریف شده است :

```
static inline struct fd fdget_pos(int fd)
{
        return __to_fd(__fdget_pos(fd));
}
```

هدف اصلی fdget_pos تبدیل فایل توصیفگر به ساختار fd است. همچنین با تابع file_pos_read می توانیم f_pos را به صورت زیر تعریف کنیم :

سیستم های عامل

تمرین سری دوم

```
static inline loff_t file_pos_read(struct file *file)
{
	return file->f_pos;
}
```

و تابع vfs_write که در فایل سورس fs/read_write.c تعریف شده است را نوشتن بافر داده شده استفاده کنیم.در این تابه به جزئیات پرداخته نمی شود زیرا این تابع در system call مفهومی ضعف دارد ولی درباره فایل مجازی استفاده می شود. پس از آنکه vfs_write کار خود را به اتمام رساند ما نتیجه را بررسی می کنیم و موفقیت را بررسی میکنیم :

```
if (ret >= 0)
   file_pos_write(f.file, pos);
```

که تنها f_pos موقعیت فایل داده را شده را به صورت زیر بررسی میکند :

```
static inline void file_pos_write(struct file *file, loff_t pos)
{
	file->f_pos = pos;
}
```

و در آخر فراخوانی سیستمی write با تابع زیر :

```
fdput_pos(f);
```

mutex f_pos_lock را که از موقعیت پرونده در هنگام نوشتن همزمان از موضوعاتی که توصیف کننده فایل را به اشتراک می گذارند ، محافظت می کند.

2. این یک بحث کتبی درباره هسته لینوکس و معماری هسته آن به صورت عمومی بود که آقای Tanenbaum خالق مینیکس در سال 1992 آغاز کرد و استلال ایشان این بود که ریز هسته ها به هسته های یکپارچه برتری دارند. پس از آن افراد دیگری به این بحث اضافه شدند و موضوعات مختلف دیگری نیز پا به عرصه وجود گذاشتند. در ابتدا آقای تاننباوم انتقاد خود را با اشاره به چونگی طراحی یکپارچه هسته و ضرر های آن بیان کرد و با اینکه برای آن دلایل محکمی نمی آورد اما با استدلال گره خوردن زیاد لینوکس به پردازنده های x86 پیشنهاد به قابل حمل بودن آن کرد. فردای آن روز آقای توروالدز پساخ مستقیم داد که minix دارای نقض ذاتی طراحی است و تصدیق میکند که او از نظر تئوری و زیبایی شناسی هسته، ریز هسته برتری دارد.
او نیز به به این اشاره کرد که در اوقات فراغت هسته لینوکس را توسعه میدهد و رایگان است و نباید به وی اعتراض کند. او نیز اشاره کرد که به دلیل یادگیری لینوکس را به طور خاص برای پردازنده Intel 80386 توسعه داده است . وی نیز اظهار داشت که linuxقابل حمل تر از minix است.
علی رغم این بحث توروالدز و تاننباوم با هم صحبت خوبی دارند و در آخر تاننباوم در با دفاع از توروالدز در مصاحبه ای اعلام کرد که میخواهد چند باور غلط را پاک و موضوع را به اتمام برساند و سعی نمیکرده است که جای گنو را بگیرد و رفتاری طبق شغل و حرفه خود و برای بهبود این زمینه انجام داده است.

3. مولتیکس از نخستین سیستم های عامل اشتراک زمانی(اشتراک گذاشتن منابع مختلف رایانه میان چند کاربر با بهره گرفتن از روش های چندبرنامه ای یا چند وظیفه ای) بود که در سال 1964 در شهر کمبریج آغاز شد. آخرین سستم مولتیکس در سال 2000 در سازمان وزارت دفاع کانادا خاموش شد.
این پروژه در اصل به رهبری ITB و همکاری General Electricو آزمایشگاه های بل بود. این پروژه چنان بلندپروازانه بود که تبدیل به پروژه ای پیچیده و خارج از دور شد و آزمایشگاه های بل از آن کناره گیری کردند.
بعد ها چندی از محققان نظیر Thompson و Ritchie با استفاده از تجربیاتی که از پروژه مولتیکس کسب کرده بودند، بعد ها سیستم عامل دیگری به نام یونیکس بسیار موفق بود را پایه گذاری کردند که نوعی بازی به کلمات محسوب می شود .

سیستم های عامل

تمرین سری دوم

4.فراخوان های سیستمی رابطی بین سیستم عامل و برنامه های دیگر هستند. این فراخوانی ها را معمولا به چند دسته:
مدیریت فایل ها و فهرست ها: ایجاد و حذف، باز و بسته کردن، خواندن و نوشتن و تغییر صفات فایل ها و...
مدیریت وسایل: درخواست اتصال و رهاسازی eject ، وسیله، خواندن و نوشتن در وسیله و ...
مدیریت پردازش ها: ایجاد و پایان دادن پردازش، بارگذاری و اجرای پردازش در سیستم عامل، تخصیص و آزادکردن حافظه و...
به دست اوردن اطلاعات: خواندن زمان استفاده از سیستم توسط کاربر، تعداد کاربرانف میزان فضای آزاد حافظه یا دیسک و نسخه سیستم عامل و خواندن زمان و تاریخ و ...
تقسیم می کنند .
این فراخوانی ها را گاها فراخوانی هسته ای نیز می نامند زیرا که در اکثر پردازنده های مدرن برای انجام فراخوانی سیستمیردازنده باید در مد هسته باشد. بدین معنا که تنها با استفاده از سد سیستم عامل می توان به سخت افزار با فراخوانی های مربوط به آن ارتباط برقرار کرد.
این فراخوانی سبب می شود که برنامه نویس یا کاربر بدون درگیر شدن با مسائل جزئی و ریز سخت افزاری بتوانند با آن ها ارتباط برقرار کنند.

5. تعداد فراخوانی های سیستمی در آخرین نسخه از هسته لینوکس که در اختیار داریم(5.11.13)، دارای 442عدد فراخوانی سیستمی می باشد که فراخوانی در سیستم های 32 بیتی دارای دو قسمت off_t و loff_t است و در سیستم های 64 بیتی تنها ورژن off_t مشاهده می شود. از این 422 عدد، 244 دستور مخصوص arch هستند .

تمامی این پاسخ ها به صورت دستی و کلمه به کلمه نوشته شده اند و تنها برای ترجمه برخی لغات از مترجم استفاده شده است و ممکن دارای غلط های املایی باشند، همچنین برای هر سوال سعی شده است پاسخ ها به صورت کامل و واضح ارائه شوند.