p CA_2 OS_LAB

پرسش اول :

کتابخانه های سطح کاربر در xv6 به عنوان یک wrapper عمل می کنند .

پنهان سازی جزئیات : وقتی یک برنامه نویس در سطح کاربر تابعی را فراخوانی می کند در واقع یک تابع پوشاننده که در usys.S توسط ماکروی SYSCALL فراخوانی می کند .

این ماکرو به طور خودکار کارهای سطح پایین که به آن اشاره می شود انجام می دهد :

شماره مربوط به فراخوانی سیستمی را در رجیستر eax می گذارد .

آرگومان های ارسال شده به تابع C را بر روی استک قرار می دهد.

اجرای trap : دستور int 64 را اجرا می کند تا کنترل به هسته منتقل شود

پس از اتمام کار هسته و بازگشت به سطح کاربر مقدار بازگشتی را که در eax قرار دارد به عنوان خروجی تابع باز می گرداند.

دلایل استفاده برای موارد زیر :

ساده سازی توسعه برنامه : نیازی نیست که برنامه نویس جزئیات پیچیده اسمبلی و شماره های فراخوانی و نحوه اجرای دستور int را بداند و فقط یک تابع معمولی را فراخوانی می کند .

افزایش قابلیت حمل : جزئیات مربوط به فراخوانی های سیستمی مانند شماره ها یا حتی دستورالعمل های مورد استفاده را بین معماری های مختلف و سیستم عامل های مختلف متفاوت است و توابع پوشاننده این تفاوت ها را هندل می کنند و همین موضوع ها باعث می شود که کد برنامه سطح کاربر مستقل از این جزئیات باشد و می توانیم آنرا رو سیستم های مختلف کامپایل و اجرا کنیم .

افزایش امنیت : این لایه انتزاعی یک نقطه ورودی استاندار و کنترل شده را به هسته فراهم می کند و تضمین می کند که برای انتقال به مد هسته از یک روش ثابت و پیش بینی شده انجام می شود و از ایجاد خطا ها برای مواردی مثل تنظیم حالت پردازنده با استک جلوگیری می کند.

پرسش دوم :

مقایسه دستور int در xv6 دستورات جدیدی مانند sysexit/sysenter در لینوکس از نظر موارد زیر :

نحوه پیاده سازی و عملکرد :

int : وقتی این دستور اجرا می شود پردازه جدول IDT (جدول توصیفگر وقفه) را چک می کند و گیت مربوطه را پیدای می کند و سطح دسترسی را چک می کند و سپس با ذخیره کامل وضعیت فعلی روی استک هسته به کد هسته می پرد و یک عملیات سخت افزای پیچیده مربوط می شود .

sysexit/sysenter : دستورات جدید تری هستند که به طور خاص برای فراخوانی های سیستمی سریع طراحی شدند و از مکانیزم های مربوط به وقفه و IDT استفاده نمی کنند به جای آن هسته در زمان بوت رجیستر های مخصوص را (MSRs) را تنظیم می کنند که آدرس های دقیق کد هسته و استک هسته و سگمنت های مد نظر را به CPU اطلاع می دهد . sysenter به سرعت به آدرس مورد نظر می پرد و sysexit بازگشت سریع به سطح کاربر را انجام می دهد.

کارایی و سربار :

int :

مشکل اصلی این روش سربار مستقیم بالای آن است که باعث مصرف چرخه های پردازشی زیاد برای جست و جو در IDT و ذخیره سازی کامل وضعیت می شود اما دستور های مربوط به دستورات جدید سربار انتقال بسیار کمتری دارد و دلیل این سربار کمتر حذف کردن مراحل اضافی است و استفاده از مسیر مستقیم و بهینه می باشد . دستور int یک مکانیزم همه کاره برای هندل کردن هر نوع وقفه و استثنایی دارد و اما دستور های sysenter به طور خاص فقط برای فراخوانی های سیستمی طراحی شده

کاربرد های مختلف :

دستور int در سیستم عامل های قدیمی تر یا سیستم عامل های xv6 به دلیل سادگی پیاده سازی می شود اما دستور های sysexit , sysenter برای سیستم عامل های مدرن استفاده می شود تا سربار مستقیم را کاهش دهند و کارایی به بیشینه مقدار خود برسد.

پرسش سوم :

سایر تله‌ها (وقفه‌های سخت‌افزاری و استثناها) را نمی‌توان با سطح دسترسی DPL_USER فعال نمود.

دلیل این امر امنیتی است. فراخوانی سیستمی یک درخواست عمدی و قانونی از طرف برنامه سطح کاربر برای دریافت خدمات از هسته است؛ بنابراین، DPL_USER به عنوان "مجوز ورود" عمل می‌کند تا کاربر بتواند این درخواست را صادر کند.

اما وقفه‌های سخت‌افزاری یا استثناها (مانند تقسیم بر صفر یا خطاهای حفاظتی)، رویدادهای داخلی و حساس سیستمی هستند. اگر DPL آن‌ها نیز "User" بود، یک برنامه مخرب یا معیوب می‌توانست به راحتی این تله‌های بحرانی را به صورت مصنوعی فراخوانی کند. این کار به کاربر اجازه می‌داد تا مستقیماً به کدهای مدیریتی هسته دسترسی یابد و امنیت و پایداری کل سیستم را به خطر بیندازد.

بنابراین، DPL آن‌ها روی 0 (سطح هسته) تنظیم می‌شود تا اطمینان حاصل شود که این رویدادها هرگز توسط کاربر قابل فراخوانی نیستند و مدیریت آن‌ها منحصراً در اختیار هسته باقی می‌ماند.

پرسش چهارم :

هر پردازه در xv6 دو پشته مجزا دارد: یک پشته سطح کاربر (که در حالت کاربر استفاده می‌شود) و یک پشته سطح هسته (که در حالت هسته استفاده می‌شود).

1. در صورت تغییر سطح دسترسی (مانند انتقال از کاربر به هسته):
   - هنگامی که یک فراخوانی سیستمی رخ می‌دهد، پردازنده از حالت کاربر (Ring 3) به حالت هسته (Ring 0) تغییر سطح دسترسی می‌دهد. در این لحظه، پردازنده باید استفاده از پشته کاربر را متوقف کرده و به پشته هسته سوئیچ کند. برای اینکه پردازنده بتواند پس از اتمام کار در هسته، دقیقاً به همان نقطه و همان پشته در سطح کاربر بازگردد، باید آدرس پشته سطح کاربر را ذخیره کند.به همین دلیل، پردازنده به طور خودکار مقادیر SS (سگمنت پشته کاربر) و esp (اشاره‌گر پشته کاربر) را بر روی پشته جدید (پشته هسته) پوش (Push) می‌کند تا پس از اتمام کار، بتواند آن‌ها را بازیابی کرده و به سطح کاربر بازگردد.
2. در صورت عدم تغییر سطح دسترسی (مانند انتقال از هسته به هسته):
   - اگر یک وقفه (مثلاً وقفه تایمر) زمانی رخ دهد که پردازنده از قبل در حالت هسته بوده است، هیچ تغییری در سطح دسترسی رخ نمی‌دهد.پردازنده از قبل در حال استفاده از پشته هسته بوده و به استفاده از همان پشته ادامه می‌دهد.از آنجایی که پشته‌ای عوض نمی‌شود، مقادیر SS و esp تغییری نکرده‌اند و نیازی به ذخیره‌سازی مجدد آن‌ها وجود ندارد.

پرسش پنجم :

۱. نقش توابع argint و argptr:

این توابع، ابزارهای کمکی در هسته هستند که برای واکشی (Fetch) امن آرگومان‌های فراخوانی سیستمی از پشته سطح کاربر به کار می‌روند. زمانی که کنترل به هسته منتقل می‌شود، آرگومان‌ها هنوز در حافظه کاربر قرار دارند.

- argint(n): این تابع برای خواندن آرگومان Nاُم (که انتظار می‌رود یک عدد صحیح (int) باشد) استفاده می‌شود. همانطور که در تصویر توضیح داده شده، این تابع آدرس آرگومان را بر اساس اشاره‌گر پشته کاربر (esp) محاسبه می‌کند (مثلاً esp + 4 + 4*n) و سپس ۴ بایت را از آن آدرس در فضای کاربر می‌خواند.
- argptr(n, ...): این تابع برای خواندن آرگومان Nاُم (که انتظار می‌رود یک اشاره‌گر (pointer) باشد) استفاده می‌شود. این تابع نه تنها خودِ آدرس را می‌خواند، بلکه وظیفه بسیار مهم‌تری نیز بر عهده دارد.

۲. ضرورت بررسی بازه‌های آدرس در argptr:

اشاره‌گری که از سطح کاربر به هسته ارسال می‌شود، کاملاً غیرقابل اعتماد است. یک پردازه در حالت کاربر باید محدود به فضای حافظه خود باشد. argptr باید بازه‌های آدرس ورودی را به دلایل امنیتی حیاتی زیر بررسی کند:

1. جلوگیری از دسترسی به حافظه هسته: یک برنامه مخرب می‌تواند اشاره‌گری را به هسته ارسال کند که به فضای حافظه خودِ هسته اشاره دارد (مثلاً آدرس جدول پردازه‌ها یا کد هسته).
2. جلوگیری از دسترسی به حافظه سایر پردازه‌ها: برنامه می‌تواند آدرسی را ارسال کند که متعلق به یک پردازه دیگر در حال اجرا روی سیستم است.

۳. مشکلات امنیتی در صورت عدم بررسی (مثال sys_read):

فراخوانی سیستمی read(int fd, char *buffer, int size) را در نظر بگیرید که یک اشاره‌گر buffer برای نوشتن داده‌ها دریافت می‌کند.

- سناریوی حمله: یک برنامه مخرب می‌تواند read را با یک آدرس buffer که به یک بخش حساس از حافظه هسته اشاره دارد، فراخوانی کند.
- نتیجه فاجعه‌بار (در صورت عدم بررسی): هسته (که در حالت ممتاز Ring 0 اجرا می‌شود) به این آدرس اعتماد کرده و داده‌های خوانده شده از فایل (یا کیبورد) را مستقیماً روی داده‌های حیاتی خود رونویسی (Overwrite) می‌کند.
- خطر: این کار می‌تواند فوراً باعث سقوط (Crash) سیستم شود یا بدتر از آن، یک آسیب‌پذیری افزایش سطح دسترسی (Privilege Escalation) ایجاد کند که در آن کاربر کنترل کامل سیستم را به دست می‌گیرد. همانطور که در تصویر هشدار داده شده، هسته "فریب می‌خورد" تا به روندی که نباید، دسترسی پیدا کند و حافظه را تخریب کند.

۴. امکان حذف بررسی یا ارسال بازه اشتباه:

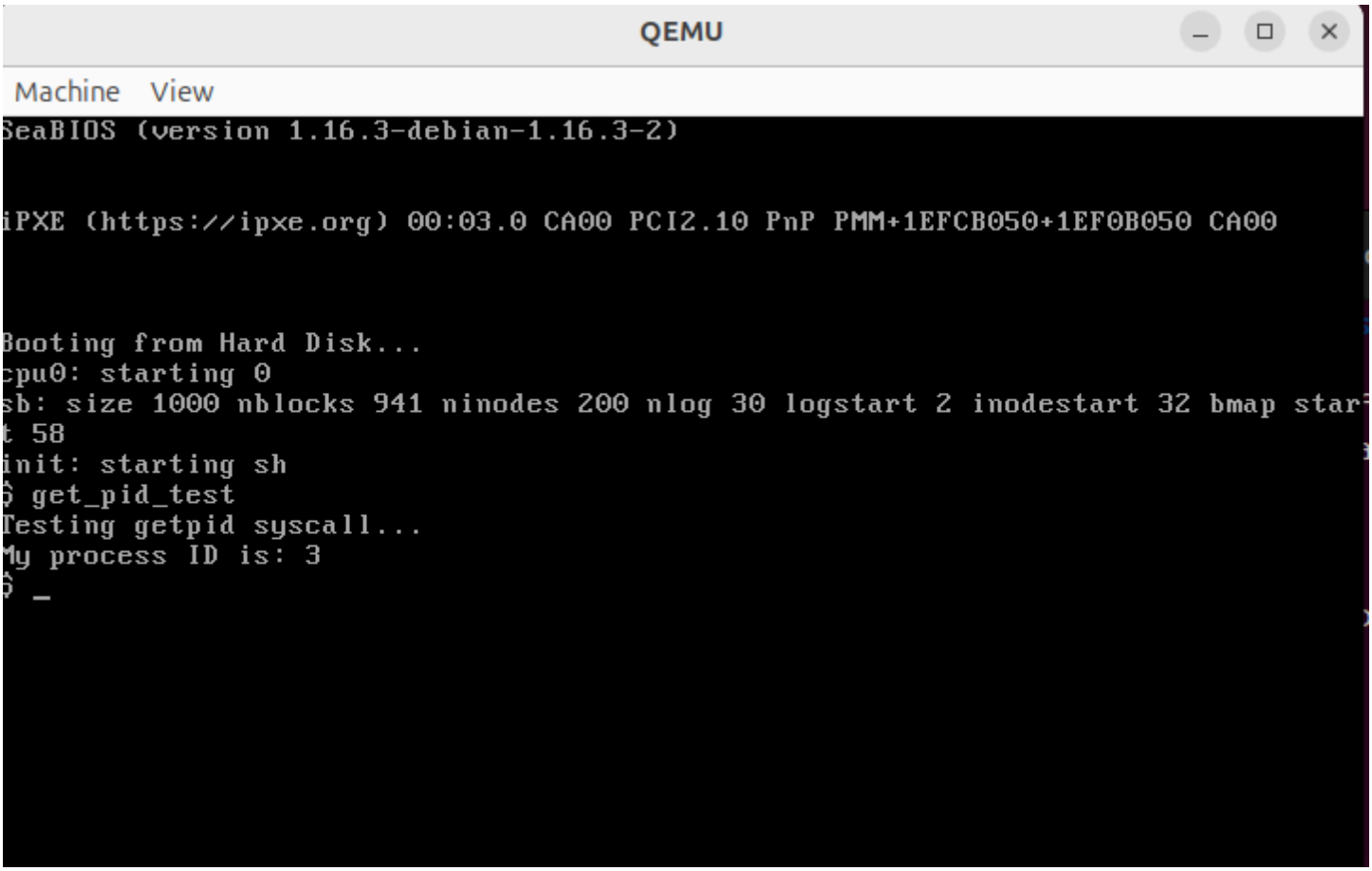
- آیا حذف بررسی ممکن است؟ خیر، به هیچ وجه. این بررسی‌ها ستون فقرات مدل امنیتی سیستم‌عامل و اصل جداسازی حافظه (Memory Isolation) هستند. حذف آن‌ها سیستم‌عامل را کاملاً ناامن می‌کند.
- آیا می‌توان بازه اشتباه وارد کرد؟ بله، یک برنامه سطح کاربر می‌تواند (و برنامه مخرب عمداً) بازه آدرس اشتباه ارسال کند. وظیفه argptr دقیقاً این است که این تلاش را شناسایی کرده و فراخوانی سیستمی را با برگرداندن یک کد خطا (مانند -1) رد (Reject) کند و اجازه ندهد هیچ آسیبی به سیستم وارد شود.

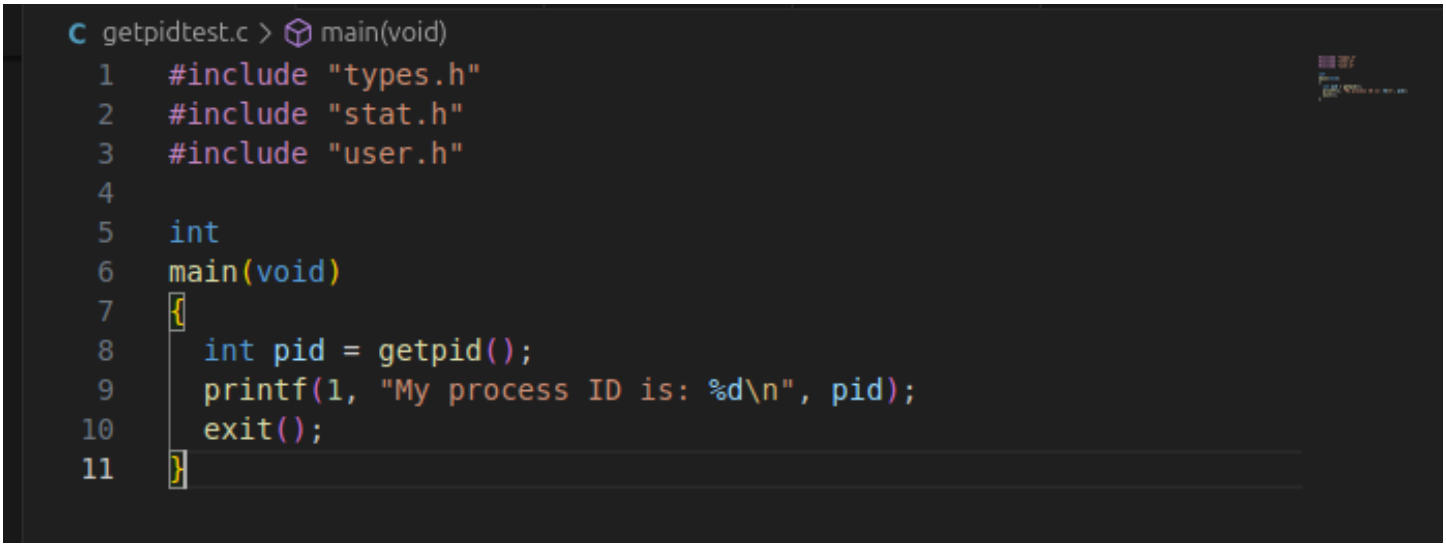
پرسش ششم :

این موضوع بستگی به نحوه فراخوانی سیستمی از سمت کاربر دارد :

1. اگر کاربر از glibc که کتابخانه استاندارد است استفاده کند هیچ مشکلی پیش نمی آید زیرا توابع پوشاننده glibc پارامتر هارا از استک دریافت می کنند و سپس قبل از اجرای دستور sysenter آنهارا در رجیستر های درست کپی می کند پس اگر کاربر قبل از فراخوانی مقدار رجیستر را دستی تغییر دهد تابع پوشاننده glibc به سادگی مقدار را Overwrite می کند و مقدار درست را قبل ورود به هسته در رجیستر قرار میدهد.
2. اگر کاربر glibc را دور بزند و مستقیما با اسمبلی فراخوانی کند مشکلات جدی رخ می دهد اگر کاربر رجیستر ها را با مقادیر نادرست پر کند و مستقیما به هسته برود مشکلات امنیتی که در پاسخ پرسش 5 به آن اشاره شده رخ می دهد و هم چنین در لینوکس برخی از رجیستر ها بای پس از بازگشت از فراخوانی سیستمی مقدار اولیه خود را حفظ کنند اما اگر کاربر مستقیما syscall را اجرا کند این ذخیره و بازیابی فراموش می شود و هسته ممکن مقادیر آن رجیستر ها را تغییر دهد و پس از بازگشت به سطح کاربر برنامه با رجیستر های تخریب شده مواجه میشود که باعث محاسبات اشتباه یا crash می شود .

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb


QEMU
Machine View
SeaBIOS (version 1.16.3-debian-1.16.3-2)
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ get_pid_test
Testing getpid syscall...
My process ID is: 3
$ _


getpidtest.c > main(void)
#include "types.h"
#include "stat.h"
#include "user.h"
int
main(void)
{
int pid = getpid();
printf(1, "My process ID is: %d\n", pid);
exit();
}

```
(gdb) target remote localhost:26000
Remote debugging using localhost:26000
0x0000fff0 in ?? ()
(gdb) b syscall
Breakpoint 1 at 0x80104b90: file syscall.c, line 139.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:139
139       struct proc *curproc = myproc();
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x80105d3d in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$1 = 7
(gdb) bt
#0  syscall () at syscall.c:139
#1  0x80105d3d in trap (tf=0x8dffffb4) at trap.c:43
#2  0x80105aed in alltraps () at trapasm.S:20
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:139
139       struct proc *curproc = myproc();
```

```
(gdb) up
#1  0x80105d3d in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$2 = 15
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:139
139       struct proc *curproc = myproc();
(gdb) up
#1  0x80105d3d in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$3 = 17
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:139
139       struct proc *curproc = myproc();
(gdb) up
#1  0x80105d3d in trap (tf=0x8dffffb4) at trap.c:43
43          syscall();
(gdb) p tf->eax
$4 = 15
```

دستور bt یکی از دستورات اساسی در GDB است که تاریخچه فراخوانی توابع یا (Call Stack) را نمایش می‌دهد.

این دستور به ما نشان می‌دهد که برنامه چگونه به نقطه‌ی توقف فعلی رسیده است؛ به عبارت دیگر، کدام تابع، کدام تابع را فراخوانی کرده تا به اینجا رسیده

در تصویری که داریم ، پس از توقف برنامه در نقطه توقف (Breakpoint)، دستور bt اجرا شده و GDB پشته‌ی فراخوانی فعلی را چاپ کرده است. این خروجی به ما مسیر دقیق رسیدن به تابع syscall را نشان می‌دهد:

لایه‌ی 0 : syscall () at syscall.c:139

این لایه‌ی فعلی است. به ما می‌گوید که برنامه در حال حاضر در تابع syscall (در فایل syscall.c، خط ۱۳۹) متوقف شده است.

لایه‌ی 1 : 0x80105d3d in trap ... at trap.c:43

این لایه‌ی قبلی یا فراخواننده (Caller) است. به ما می‌گوید که تابع syscall مستقیماً توسط تابع trap (در فایل trap.c، خط ۴۳) فراخوانی شده است.

لایه‌ی 2 : 0x80105aed in alltraps () at trapasm.S:20

این لایه‌ی دوم است. به ما می‌گوید که تابع trap نیز به نوبه‌ی خود توسط یک روتین سطح پایین اسمبلی به نام alltraps فراخوانی شده است.

نتیجه‌گیری:

خروجی bt به طور واضح مسیر اجرای یک فراخوانی سیستمی در xv6 را تأیید می‌کند: یک تله (trap) از سطح کاربر رخ داده، ابتدا وارد کد اسمبلی alltraps شده، سپس تابع C به نام trap را صدا زده و در نهایت تابع syscall (جایی که ما متوقف شده‌ایم) فراخوانی شده است.

دستور down در GDB برای حرکت **به سمت پایین** (یا داخلی‌تر) در پشته فراخوانی (Call Stack) استفاده می‌شود. "پایین" در GDB به معنای رفتن به یک تابع جدیدتر (تابعی که اخیراً فراخوانی شده) است.

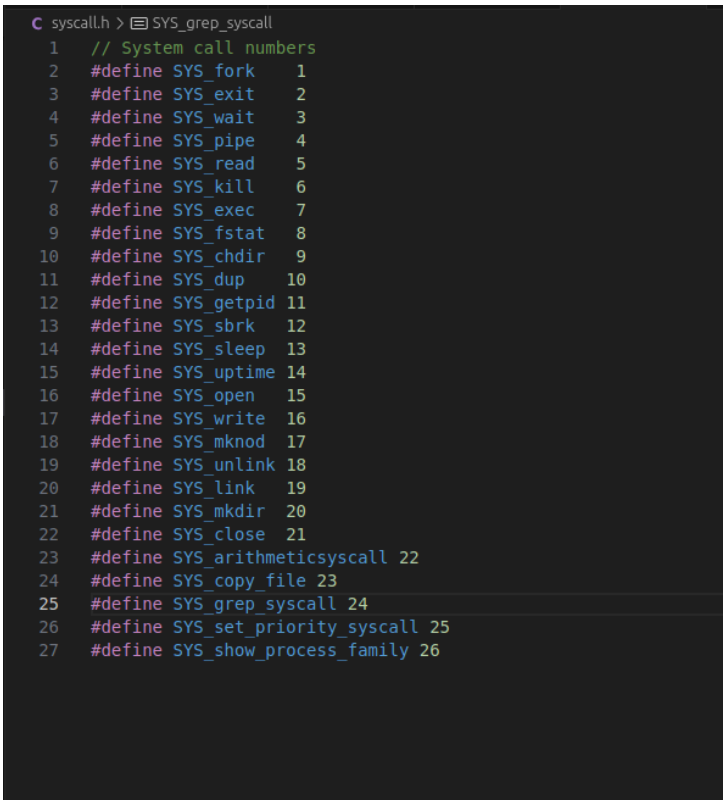
1. **اجرای دستور down:** کاربر دستور down را وارد کرده است.
2. **دریافت خطا:** GDB پاسخ داده است: Bottom (innermost) frame selected; you cannot go down. (لایه‌ی پایینی انتخاب شده است؛ شما نمی‌توانید پایین‌تر بروید).

دلیل خطا:

دلیل این خطا این است که دستور down تلاش می‌کند به لایه‌ای جدیدتر از لایه‌ی فعلی برود. اما برنامه‌ی شما در لایه‌ی صفر (داخلی‌ترین لایه) متوقف شده است. هیچ لایه‌ای "پایین‌تر" یا "جدیدتر" از لایه‌ی صفر وجود ندارد که بتوان به آن رفت. به همین دلیل GDB به درستی اعلام می‌کند که شما در انتهای پشته هستید و نمی‌توانید پایین‌تر بروید.

(دستور مخالف down، دستور up است که به لایه‌ی "بالاتر" (یعنی تابع فراخواننده) می‌رود).

با توجه به اینکه رجیستر eax شماره سیستم کال را ذخیره می کند اما چون قبل از سیستم کال فراخوانی های دیگری انجام می شود این مقدار فرق می کند



فایل syscall.h

در این فایل هدر، یک شماره‌ی شناسایی یکتا (مثلاً 22) برای فراخوانی سیستمی جدید تحت نام SYS_arithmeticsyscall تعریف شد. این شماره‌ی ثابت به هسته اجازه می‌دهد تا هنگام وقوع تله (trap)، این فراخوانی سیستمی را از سایر فراخوانی‌ها تشخیص دهد.

```
106  extern int sys_arithmeticsyscall(void);
107  extern int sys_copy_file(void);
108  extern int sys_grep_syscall(void);
109  extern int sys_set_priority_syscall(void);
110  extern int sys_show_process_family(void);
```

```
134  [SYS_arithmeticsyscall] sys_arithmeticsyscall,
135  [SYS_copy_file] sys_copy_file,
136  [SYS_grep_syscall] sys_grep_syscall,
137  [SYS_set_priority_syscall] sys_set_priority_syscall,
138  [SYS_show_process_family] sys_show_process_family,
139  };
```

فایل syscall.c

در این فایل، دو تغییر اعمال شد تا هسته از تابع جدید آگاه شود:

1. تابع پیاده‌سازی شده در sysproc.c با استفاده از extern int sys_arithmeticsyscall(void); به این فایل معرفی شد.
2. نام تابع به آرایه‌ی syscalls[] اضافه شد. این کار، شماره‌ی SYS_arithmeticsyscall (تعریف شده در syscall.h) را به تابع پیاده‌سازی شده‌ی آن متصل (map) می‌کند.

```
215  int
216  sys_arithmeticsyscall(void)
217  {
218    struct proc *curproc = myproc();
219    int a, b, result;
220    a = curproc->tf->ebx;
221    b = curproc->tf->ecx;
222    result = (a + b) * (a - b);
223    cprintf("Calc: (%d+%d)*(%d-%d)=%d\n", a, b, a, b, result);
224    return result;
225  }
226
```

فایل sysproc.c

منطق اصلی فراخوانی سیستمی در این فایل پیاده‌سازی شد. تابع sys_arithmeticsyscall به انتهای این فایل اضافه گردید که وظایf زیر را انجام می‌دهد:

- آرگومان‌ها را نه از پشته، بلکه مستقیماً از رجیسترهای EBX و ECX (که در قاب تله یا curproc->tf ذخیره شده بودند) می‌خواند، دقیقاً همانطور که در صورت پروژه خواسته شده بود.
- محاسبه‌ی $(a-b)*(a+b)$ را انجام می‌دهد.
- مقادیر ورودی و نتیجه‌ی نهایی را با استفاده از cprintf در کنسول هسته چاپ می‌کند [1].
- مقدار نتیجه‌ی محاسبه شده را به فضای کاربر بازمی‌گرداند.

```
25  int uptime(void);
26  int arithmeticsyscall(int a, int b);
```

فایل user.h

نمونه اولیه (prototype) تابع C، یعنی int arithmeticsyscall(int a, int b);، به این فایل هدر اضافه شد. این کار به برنامه‌های سطح کاربر (مانند فایل تست) اجازه می‌دهد تا این تابع را به صورت قانونی فراخوانی کرده و کامپایلر نوع آرگومان‌های آن را بشناسد.

```
.globl arithmeticsyscall
arithmeticsyscall:
  pushl %ebx
  movl 8(%esp), %ebx
  movl 12(%esp), %ecx
  movl $SYS_arithmeticsyscall, %eax
  int $T_SYSCALL
  popl %ebx
  ret
```

فایل usys.S

از آنجایی که این فراخوانی سیستمی به روشی غیر استاندارد (خواندن از EBX و ECX) نیاز داشت، یک پوشاننده‌ی (wrapper) اسمبلی سفارشی در این فایل نوشته شد. این کد:

1. ابتدا مقدار رجیستر EBX را (به دلیل callee-saved بودن) در پشته push می‌کند تا مقدار آن حفظ شود.
2. آرگومان‌های a و b را از پشته‌ی C می‌خواند.
3. آن‌ها را در رجیسترهای EBX و ECX کپی می‌کند.
4. شماره‌ی فراخوانی سیستمی (SYS_arithmeticsyscall) را در EAX قرار داده و با int تله را اجرا می‌کند.
5. در انتها، EBX را از پشته pop کرده و با ret به برنامه‌ی C بازمی‌گردد.

arithmetictest.c > main(void)

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

int
main(void)
{

  int result1 = arithmeticsyscall(5, 3);


  exit();
}
```

فایل arithmetictest.c (فایل جدید)

این فایل تست جدید در سطح کاربر ایجاد شد. تابع main آن، فراخوانی سیستمی arithmeticsyscall() را با مقادیر نمونه (مثلاً ۵ و ۳) صدا می‌زند تا هم صحت مقدار بازگشتی و هم چاپ شدن خروجی cprintf در کنسول هسته را بررسی و تأیید کند.

```
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _getpidtest\
185     _arithmetictest\
186     _copy_file\
187     _grep_test\
188     _priority_test\
189     _familytest\
```

فایل Makefile

نام فایل تست جدید (_arithmetictest) به لیست UPROGS در Makefile اضافه شد. این کار تضمین می‌کند که برنامه‌ی تست ما کامپایل شده و به عنوان یک برنامه‌ی قابل اجرا در فایل سیستم xv6 (fs.img) قرار گیرد.

```
SeaBIOS (version 1.16.3-debian-1.16.3-2)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFCB050+1EF0B050 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ arithmetictest
Calc: (5+3)*(5-3)=16
$ _
```

```c
// 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_arithmeticsyscall 22
#define SYS_copy_file 23
#define SYS_grep_syscall 24
#define SYS_set_priority_syscall 25
#define SYS_show_process_family 26
```

۱. فایل syscall.h

توضیح:

یک شماره‌ی شناسایی یکتا (مثلاً 23) برای فراخوانی سیستمی جدید تحت نام SYS_copyfile تعریف شد. این شماره به هسته اجازه می‌دهد تا این فراخوانی سیستمی را از سایرین تشخیص دهد.

```
106   extern int sys_arithmeticsyscall(void);
107   extern int sys_copy_file(void);
108   extern int sys_grep_syscall(void);
109   extern int sys_set_priority_syscall(void);
110   extern int sys_show_process_family(void);
```

```
134   [SYS_arithmeticsyscall] sys_arithmeticsyscall,
135   [SYS_copy_file] sys_copy_file,
136   [SYS_grep_syscall] sys_grep_syscall,
137   [SYS_set_priority_syscall] sys_set_priority_syscall,
138   [SYS_show_process_family] sys_show_process_family,
139   };
```

۲. فایل syscall.c

توضیح:

در این فایل، تابع sys_copyfile (که در sysfile.c پیاده‌سازی شده) با استفاده از extern معرفی شد. سپس، نام تابع به آرایه‌ی syscalls[] اضافه شد تا شماره‌ی SYS_copyfile را به پیاده‌سازی آن متصل کند.

```
25  int uptime(void);
26  int arithmeticsyscall(int a, int b);
27  int copy_file(char*, char*);
28  int grep_syscall(const char *keyword, const char *filename, char *user_buffer, int buffer_s
```

۳. فایل user.h

توضیح:

نمونه اولیه (prototype) تابع C، یعنی int copyfile(const char*, const char*);، به این فایل هدر اضافه شد. این کار به برنامه‌های سطح کاربر (مانند فایل تست) اجازه می‌دهد تا این تابع را به صورت قانونی فراخوانی کنند.

```
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(copy_file)
```

۴. فایل usys.S

توضیح:

پوشاننده‌ی (wrapper) اسمبلی برای تابع copyfile با استفاده از ماکروی استاندارد SYSCALL(copyfile) اضافه شد. این ماکرو به طور خودکار کد لازم برای انتقال از سطح کاربر به هسته (با استفاده از SYS_copyfile) را تولید می‌کند.

```
447  int sys_copy_file(void)
448  {
449      char *path_source;
450      char *path_destination;
451      struct file *source_file;
452      struct file *destination_file;
453      struct inode *source_ip;
454      struct inode *destination_ip;
455      if (argstr(0, &path_source) < 0 || argstr(1, &path_destination) < 0)
456          return -1;
457
458      begin_op();
459      source_ip = namei(path_source);
460      destination_ip = namei(path_destination);
461      if (source_ip == 0 || destination_ip != 0 || (source_ip->type != T_FILE) || (source_ip == destination_ip))
462      {
463          end_op();
464          return -1;
465      }
466      ilock(source_ip);
467      destination_ip = create(path_destination, T_FILE, 0, 0);
468
469      if ((source_file = filealloc()) == 0 || (destination_file = filealloc()) == 0)
470      {
471          iunlockput(destination_ip);
472          iunlockput(source_ip);
473          end_op();
474          return -1;
475      }
476      iunlock(destination_ip);
477      iunlock(source_ip);
478      end_op();
479      source_file->type = FD_INODE;
480      source_file->ip = source_ip;
481      source_file->off = 0;
482      source_file->readable = 1;
```

```
483
484          destination_file->type = FD_INODE;
485          destination_file->ip = destination_ip;
486          destination_file->off = 0;
487          destination_file->writable = 1;
488          int r = copyfile(source_file, destination_file);
489          fileclose(source_file);
490          fileclose(destination_file);
491          return r;
492      }
```

۵. فایل sysfile.c

توضیح: تابع پوشاننده‌ی فراخوانی سیستمی، int sys_copy_file(void)، در انتهای این فایل اضافه شد. وظیفه‌ی این تابع، خواندن دو آرگومان مسیر (مبدا و مقصد) از پشته‌ی کاربر (با argstr) و سپس فراخوانی تابع کمکی copyfile (در file.c) برای انجام کار اصلی است.

```
28  // file.c
29  struct file*    filealloc(void);
30  void            fileclose(struct file*);
31  struct file*    filedup(struct file*);
32  void            fileinit(void);
33  int             fileread(struct file*, char*, int n);
34  int             filestat(struct file*, struct stat*);
35  int             filewrite(struct file*, char*, int n);
36  int             copyfile(struct file *source, struct file *destination);
37
```

۶. فایل defs.h

توضیح:

یک تابع جدید به این فایل هدر سراسری اضافه شد تا توابع عمومی شده‌ی ما در سراسر هسته قابل شناسایی باشند:

1. int copyfile(struct file*, struct file*); (برای تابع کمکی جدیدی که در file.c نوشتیم).

```
160 int copyfile(struct file *source, struct file *destination)
161 {
162     int r1, r2;
163     if (source->readable == 0 || destination->writable == 0)
164         return -1;
165     if (source->type == FD_INODE && destination->type == FD_INODE)
166     {
167         begin_op();
168         ilock(source->ip);
169         ilock(destination->ip);
170
171         char buf[256];
172         while ((r1 = readi(source->ip, buf, source->off, 256)) > 0)
173         {
174             source->off += r1;
175             if ((r2 = writei(destination->ip, buf, destination->off, r1)) > 0)
176                 destination->off += r2;
177             if (r2 < 0)
178                 break;
179             if (r2 != r1)
180                 panic("short filewrite");
181         }
182         iunlock(source->ip);
183         iunlock(destination->ip);
184         end_op();
185         return destination->off == source->off ? 0 : -1;
186     }
187     panic("copyfile");
188 }
189
```

۷. فایل file.c

توضیح:

منطق اصلی و سنگین عملیات کپی در این فایل پیاده‌سازی شد. تابع int copyfile(struct file *f_src, struct file *f_des) به انتهای این فایل اضافه شد. این تابع یک حلقه‌ی while را اجرا می‌کند که به طور مکرر داده‌ها را از فایل مبدا (با fileread) می‌خواند و در فایل مقصد (با filewrite) می‌نویسد.

```c
#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "stat.h"
int main(int argc, char * argv[]){
    if (argc < 3){
        printf(1, "copy_file: 2 args required\n");
        exit();
    }
    char * src = argv[1];
    char * des = argv[2];
    if (copy_file(src,des) < 0){
        printf(1,"copy_file: cannot copy the file\n");
        exit();
    }
    printf(1,"%s has been copied to %s\n", src, des);
    exit();
}
```

۸. فایل copy_file.c

توضیح:

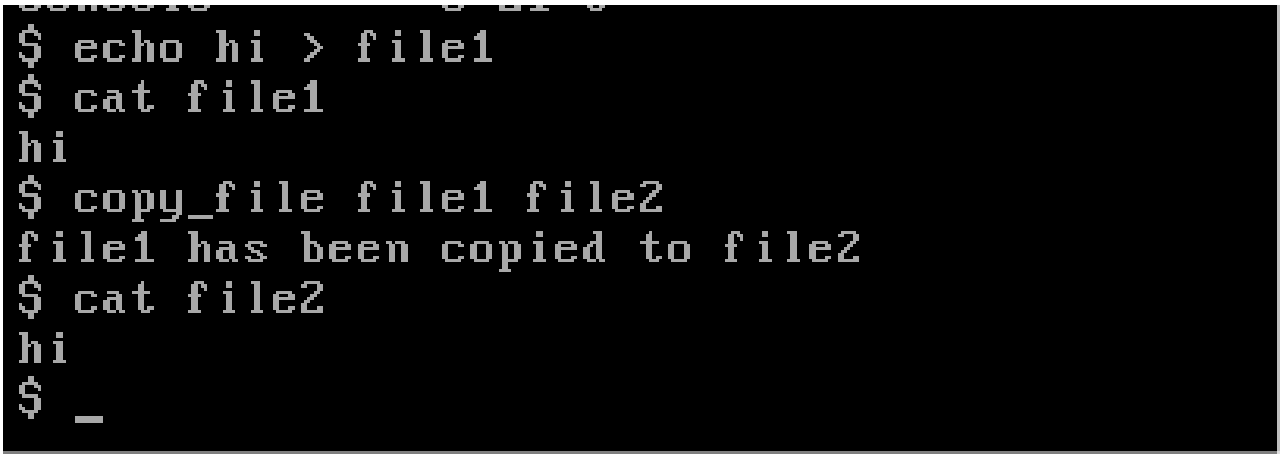
این فایل تست جدید در سطح کاربر ایجاد شد. تابع main آن ابتدا یک فایل مبدا (source.txt) ایجاد می‌کند، در آن می‌نویسد، و سپس فراخوانی سیستمی copyfile() را برای کپی کردن آن در یک فایل مقصد (destination.txt) صدا می‌زند تا صحت عملیات را بررسی کند.

```
168  UPROGS=\
169      _cat\
170      _echo\
171      _forktest\
172      _grep\
173      _init\
174      _kill\
175      _ln\
176      _ls\
177      _mkdir\
178      _rm\
179      _sh\
180      _stressfs\
181      _usertests\
182      _wc\
183      _zombie\
184      _getpidtest\
185      _arithmetictest\
186      _copy_file\
187      _grep_test\
188      _priority_test\
189      _familytest\
```

۹. فایل Makefile

توضیح:

نام فایل تست جدید (_copytest) به لیست UPROGS در Makefile اضافه شد. این کار تضمین می‌کند که برنامه‌ی تست ما کامپایل شده و به عنوان یک برنامه‌ی قابل اجرا در فایل سیستم (fs.img) xv6 قرار گیرد.



گزارش مراحل پیاده‌سازی فراخوانی سیستمی show_process_family

برای افزودن این فراخوانی سیستمی، ما در سه لایه کلیدی کار کردیم: **فضای کاربر (User Space)**، **رابط هسته (Kernel Interface)** و **منطق هسته (Kernel Logic)**.

نکته کلیدی در به‌روزرسانی ما این بود: در ابتدا تمام منطق را در sysproc.c نوشتیم، اما این کار منجر به خطاهای کامپایل شد (چون sysproc.c متغیرهایی مانند ptable را نمی‌شناخت). برای حل این مشکل و پیروی از طراحی استاندارد xv6، ما منطق اصلی را به proc.c (جایی که به آن تعلق داشت) منتقل کردیم و sysproc.c را به عنوان یک «واسط» یا «چسب» ساده نگه داشتیم.

در ادامه، تمام فایل‌هایی که تغییر دادیم و دلیل هر تغییر آمده است:

```
24    #define SYS_copy_file 23
25    #define SYS_grep_syscall 24
26    #define SYS_set_priority_syscall 25
27    #define SYS_show_process_family 26
```

۱. فایل syscall.h (تعریف شماره)

- **تغییر:** خط define SYS_show_process_family 26 به انتهای لیست اضافه شد.
- **دلیل:** ما به یک «شماره شناسایی» (ID) منحصر به فرد برای فراخوانی سیستمی جدید خود نیاز داشتیم. این فایل، منوی رسمی «خدمات» هسته است.

```
106   extern int sys_arithmeticsyscall(void);
107   extern int sys_copy_file(void);
108   extern int sys_grep_syscall(void);
109   extern int sys_set_priority_syscall(void);
110   extern int sys_show_process_family(void);
```

```
34    [SYS_arithmeticsyscall] sys_arithmeticsyscall,
35    [SYS_copy_file] sys_copy_file,
36    [SYS_grep_syscall] sys_grep_syscall,
37    [SYS_set_priority_syscall] sys_set_priority_syscall,
38    [SYS_show_process_family] sys_show_process_family,
39    };
```

۲. فایل syscall.c (اتصال شماره به تابع)

- **تغییرات:**
1. extern int sys_show_process_family(void); در بالای فایل اضافه شد.
2. [SYS_show_process_family] sys_show_process_family, به آرایه‌ی syscalls[] اضافه شد.
- **دلیل:** این فایل «تابلوی برق» هسته است. ما در اینجا رسماً شماره 22 را به نام تابعی (sys_show_process_family) که قرار است آن را مدیریت کند، «سیم‌کشی» کردیم.

```
596  dump_process_family(int pid)
598    struct proc *p;
599    struct proc *iter;
600    int found_child = 0;
601    int found_sibling = 0;
602
603    acquire(&ptable.lock);
604
605    p = 0;
606    for(iter = ptable.proc; iter < &ptable.proc[NPROC]; iter++){
607      if(iter->pid == pid && iter->state != UNUSED){
608        p = iter;
609        break;
610      }
611    }
612
613    if(p == 0){
614      release(&ptable.lock);
615      return -1;
616    }
617
618
619    if(p->parent){
620      cprintf("My id: %d, My parent id: %d\n", p->pid, p->parent->pid);
621    } else {
622      cprintf("My id: %d, I have no parent.\n", p->pid);
623    }
624
625    cprintf("Children of process %d:\n", pid);
626    for(iter = ptable.proc; iter < &ptable.proc[NPROC]; iter++){
627      if(iter->parent == p && iter->state != UNUSED){
628        cprintf("  Child pid: %d\n", iter->pid);
629        found_child = 1;
630      }
631    }
632    if(!found_child){
633      cprintf("  No children found.\n");
634    }
635
636    cprintf("Siblings of process %d:\n", pid);
637    for(iter = ptable.proc; iter < &ptable.proc[NPROC]; iter++){
638      if(p->parent && iter->parent == p->parent && iter != p && iter->state != UNUSED){
639        cprintf("  Sibling pid: %d\n", iter->pid);
640        found_sibling = 1;
641      }
642    }
643    if(!found_sibling){
644      cprintf("  No siblings found.\n");
645    }
646
647    release(&ptable.lock);
648
649    return 0;
650  }
```

۳. فایل proc.c

- **تغییری که انجام دادیم:** ما یک تابع کاملاً جدید در **انتهای** این فایل به نام int dump_process_family(int pid) ایجاد کردیم.
- **دلیل:** این **مهم‌ترین تغییر** ما بود. فایل proc.c «موتور» مدیریت پردازه است و ptable (جدول پردازه‌ها) و struct proc (ساختار پردازه) را به طور کامل می‌شناسد.
- با انتقال تمام منطق اصلی (قفل کردن، جستجو در ptable، و چاپ با cprintf) به این تابع، ما خطاهای «متغیر تعریف نشده» (undefined variable) را که در sysproc.c داشتیم، **برطرف کردیم**.
- این طراحی بسیار تمیزتر است: منطق در جایی قرار می‌گیرد که به داده‌ها دسترسی دارد.

```
120   void            sleep(void*, struct spinlock*);
121   void            userinit(void);
122   int             wait(void);
123   void            wakeup(void*);
124   void            yield(void);
125   int             setpriority(int, int);
126   int             sys_show_process_family(void);
127   int             dump_process_family(int);
```

۴. فایل defs.h (معرفی توابع به هسته)

- **تغییرات:** ما **دو** پیش‌الگو (prototype) به این فایل اضافه کردیم:
1. int sys_show_process_family(void); (برای تابع «چسب» در sysproc.c)
2. int dump_process_family(int); (برای تابع «موتور» در proc.c)
- **دلیل:** این فایل «دفترچه تلفن» کل هسته است. ما باید هر دو تابع جدید را به کل هسته معرفی می‌کردیم تا:
3. syscall.c بتواند sys_show_process_family را پیدا کند.
4. sysproc.c بتواند dump_process_family را صدا بزند.

```
229   int
230   sys_show_process_family(void)
231   {
232     int pid;
233
234     if(argint(0, &pid) < 0)
235       return -1;
236
237     return dump_process_family(pid);
238   }
```

۵. فایل sysproc.c

- **تغییری که انجام دادیم:** تابع sys_show_process_family را که قبلاً نوشته بودیم، **کامل پاک کردیم** و آن را با یک نسخه‌ی بسیار ساده و کوتاه جایگزین کردیم:
دلیل: حالا وظیفه‌ی این فایل فقط انجام کارهای مخصوص فراخوانی سیستمی است: یعنی خواندن آرگومان (argint) از پشته‌ی کاربر. سپس کار اصلی را به تابع «موتور» که در proc.c ساختیم، واگذار می‌کند.

```
15  int mknod(const char*, short, short);
16  int unlink(const char*);
17  int fstat(int fd, struct stat*);
18  int link(const char*, const char*);
19  int mkdir(const char*);
20  int chdir(const char*);
21  int dup(int);
22  int getpid(void);
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int arithmeticsyscall(int a, int b);
27  int copy_file(char*, char*);
28  int grep_syscall(const char *keyword, const char *filename, char *user_buffer, int buffer_size);
29  int show_process_family(int);
30  int set_priority_syscall(int pid, int priority);
31  // ulib.c
```

۶. فایل user.h (معرفی به فضای کاربر)

- **تغییر:** خط int show_process_family(int); به لیست توابع اضافه شد.
- **دلیل:** این «دفترچه تلفن» برنامه‌های **سطح کاربر** است. این به برنامه‌ی familytest.c می‌گوید که تابعی با این نام وجود دارد و می‌تواند آن را صدا بزند.

```
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(copy_file)
33  SYSCALL(grep_syscall)
34  SYSCALL(set_priority_syscall)
35  SYSCALL(show_process_family)
36
```

۷. فایل usys.S (ساخت رابط اسمبلی)

- **تغییر:** خط SYSCALL(show_process_family) به انتهای فایل اضافه شد.
- **دلیل:** این ماکرو، کد اسمبلی لازم برای اجرای واقعی فراخوانی سیستمی را به صورت خودکار تولید می‌کند (یعنی قرار دادن شماره 22 در رجیستر eax و اجرای دستور int 64).

familytest.c > main(void)

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

int
main(void)
{
  int pid_child1, pid_child2;
  int my_pid = getpid();

  printf(1, "--- Family Test Program Starting ---\n");
  printf(1, "Parent PID is: %d\n", my_pid);

  pid_child1 = fork();

  if(pid_child1 == 0){
    sleep(100);
    exit();
  }

  pid_child2 = fork();

  if(pid_child2 == 0){
    sleep(100);
    exit();
  }

  sleep(10);

  printf(1, "\n=== 1. Testing Parent (PID %d) ===\n", my_pid);
  show_process_family(my_pid);

  printf(1, "\n=== 2. Testing Child 1 (PID %d) ===\n", pid_child1);
  show_process_family(pid_child1);

  printf(1, "\n=== 3. Testing Child 2 (PID %d) ===\n", pid_child2);
  show_process_family(pid_child2);

  printf(1, "\n=== 4. Testing Non-existent PID (999) ===\n");
  int result = show_process_family(999);
  printf(1, "Result for PID 999: %d (Expected: -1)\n", result);

  wait();
  wait();

  printf(1, "\n--- Family Test Program Finished ---\n");
  exit();
}
```

۸. فایل familytest.c (ایجاد تست)

- **تغییر:** این فایل جدید را ایجاد کردیم.
- **دلیل:** برای اینکه مطمئن شویم تمام این سیم‌کشی‌ها از فضای کاربر تا هسته و بازگشت آن به درستی کار می‌کند و منطق ما نتایج درستی را برمی‌گرداند.

```
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _getpidtest\
185     _arithmetictest\
186     _copy_file\
187     _grep_test\
188     _priority_test\
189     _familytest\
```

۹. فایل Makefile (تنظیمات کامپایل)

- **تغییرات:**

1. _familytest\ به لیست UPROGS اضافه شد.

- **دلیل:**

2. برای اینکه make برنامه‌ی familytest.c را کامپایل کرده و در فایل سیستمِ xv6 قرار دهد.

:grep

۱. فایل‌های اصلاح‌شده

برای افزودن فراخوانی سیستمی، فایل‌های زیر تغییر کردند:

- syscall.h: افزودن شماره سیستم کال (SYS_grep_syscall).

```
24  #define SYS_copy_file 23
25  #define SYS_grep_syscall 24
26  #define SYS_set_priority_syscall 25
27  #define SYS_show_process_family 26
```

- syscall.c: افزودن تابع sys_grep_syscall به آرایه syscalls.

```
106 extern int sys_arithmeticsyscall(void);
107 extern int sys_copy_file(void);
108 extern int sys_grep_syscall(void);
109 extern int sys_set_priority_syscall(void);
110 extern int sys_show_process_family(void);
```

```
134 [SYS_arithmeticsyscall] sys_arithmeticsyscall,
135 [SYS_copy_file] sys_copy_file,
136 [SYS_grep_syscall] sys_grep_syscall,
137 [SYS_set_priority_syscall] sys_set_priority_syscall,
138 [SYS_show_process_family] sys_show_process_family,
139 };
```

- user.h: اعلان تابع برای استفاده در فضای کاربر.

```
15  int mknod(const char*, short, short);
16  int unlink(const char*);
17  int fstat(int fd, struct stat*);
18  int link(const char*, const char*);
19  int mkdir(const char*);
20  int chdir(const char*);
21  int dup(int);
22  int getpid(void);
23  char* sbrk(int);
24  int sleep(int);
25  int uptime(void);
26  int arithmeticsyscall(int a, int b);
27  int copy_file(char*, char*);
28  int grep_syscall(const char *keyword, const char *filename, char *user_buffer, int buffer_size);
29  int show_process_family(int);
30  int set_priority_syscall(int pid, int priority);
31  // ulib.c
```

- usys.s: افزودن نام تابع برای ساخت usys.S.

```
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(copy_file)
33  SYSCALL(grep_syscall)
34  SYSCALL(set_priority_syscall)
35  SYSCALL(show_process_family)
36
```

- defs.h: اعلان تابع برای استفاده داخلی کرنل و افزودن تابع کمکی k_strstr.

```
153  int             strncmp(const char*, const char*, uint);
154  char*           strncpy(char*, const char*, int);
155  char*           k_strstr(const char *, const char *);
```

- string.c: پیاده‌سازی تابع k_strstr برای جستجوی رشته در کرنل.

```c
char *
k_strstr(const char *line_buf, const char *keyword)
{
    const char *l, *k;
    if (!*keyword)
        return (char *)line_buf;

    for (; *line_buf; line_buf++)
    {
        l = line_buf;
        k = keyword;
        while (*l && *k && *l == *k)
        {
            l++;
            k++;
        }
        if (!*k)
            return (char *)line_buf;
    }
    return 0;
}
```

۲. منطق اصلی در sysproc.c

تابع sys_grep_syscall کارهای زیر را به ترتیب انجام می‌دهد:

- دریافت آرگومان‌ها: دریافت filename، keyword و user_buffer از فضای کاربر با استفاده از argstr و argptr.
- تخصیص حافظه: تخصیص یک صفحه حافظه در کرنل با kalloc.
- باز کردن فایل: باز کردن فایل در سطح کرنل با namei (برای گرفتن inode) و قفل کردن آن با ilock.
- خواندن و جستجو:
  - خواندن فایل به صورت کاراکتر به کاراکتر با readi.
  - ذخیره کاراکترها در بافر کرنل (line_buf).
  - با رسیدن به \n (انتهای خط)، جستجوی keyword با k_strstr.
  - اگر کلمه پیدا شد، حلقه متوقف می‌شود.
  - اگر پیدا نشد، line_pos (اندیس بافر) برای خط بعدی صفر می‌شود.
- بازگرداندن نتیجه:
  - در صورت موفقیت: کپی کردن خط پیدا شده (line_buf) به بافر کاربر (user_buffer) با copyout و بازگرداندن طول خط.

- در صورت شکست: بازگرداندن مقدار -1.

- آزادسازی: آزاد کردن حافظه‌ی کرنل با kfree در هر دو حالت.

```
94  int sys_grep_syscall(void)
95  {
96      char *keyword;
97      char *filename;
98      char *user_buffer;
99      int buffer_size;
100
101     char *line_buf;
102     struct inode *ip;
103     int offset = 0;
104     int line_pos = 0;
105     int found = 0;
106     int n;
107     char c;
108
109     if (argstr(0, &keyword) < 0 ||
110         argstr(1, &filename) < 0 ||
111         argint(3, &buffer_size) < 0 ||
112         argptr(2, &user_buffer, buffer_size) < 0)
113     {
114         return -1;
115     }
116
117     if ((line_buf = kalloc()) == 0)
118     {
119         return -1;
120     }
121
122     begin_op();
123
124     if ((ip = namei(filename)) == 0)
125     {
126         end_op();
127         kfree(line_buf);
128         return -1;
129     }
130     ilock(ip);
```

```
132     while ((n = readi(ip, &c, offset, 1)) == 1)
135
136         if (c == '\n')
137         {
138             line_buf[line_pos] = '\0';
139             if (k_strstr(line_buf, keyword))
140             {
141                 found = 1;
142                 break;
143             }
144             line_pos = 0;
145         }
146         else
147         {
148             if (line_pos < PGSIZE - 1)
149             {
150                 line_buf[line_pos++] = c;
151             }
152         }
153     }
154
155     if (!found && line_pos > 0)
156     {
157         line_buf[line_pos] = '\0';
158         if (k_strstr(line_buf, keyword))
159         {
160             found = 1;
161         }
162     }
163
164     iunlockput(ip);
165     end_op();
```

```
    if (found)
    {
        int len_to_copy = line_pos;
        if (len_to_copy >= buffer_size)
        {
            len_to_copy = buffer_size - 1;
        }

        if (copyout(myproc()->pgdir, user_buffer, line_buf, len_to_copy) < 0)
        {
            kfree(line_buf);
            return -1;
        }

        char null_term = '\0';
        if (copyout(myproc()->pgdir, user_buffer + len_to_copy, &null_term, 1) < 0)
        {
            kfree(line_buf);
            return -1;
        }

        kfree(line_buf);
        return len_to_copy;
    }

    kfree(line_buf);
    return -1;
}
```

۳. برنامه تست

فایل grep_test.c به عنوان یک برنامه سطح کاربر نوشته شد تا این فراخوانی سیستمی جدید را با دریافت کلمه کلیدی و نام فایل از خط فرمان، اجرا کرده و نتیجه را چاپ کند.

```
$ grep_test efforts README
grep_test: our efforts to the RISC-V version
$ grep_test fff README
grep_test: Keyword not found or file error.
```

```
NOTE: we have stopped maintaining the x86 version of xv6, and switched
our efforts to the RISC-V version
(https://github.com/mit-pdos/xv6-riscv.git)
```

:priority

۱. فایلهای اصلاح شده

- syscall.h: افزودن شماره سیستم کال SYS_set_priority_syscall

```
24  #define SYS_copy_file 23
25  #define SYS_grep_syscall 24
26  #define SYS_set_priority_syscall 25
27  #define SYS_show_process_family 26
```

- syscall.c: افزودن تابع sys_set_priority_syscall به آرایه syscalls

```
106  extern int sys_arithmeticsyscall(void);
107  extern int sys_copy_file(void);
108  extern int sys_grep_syscall(void);
109  extern int sys_set_priority_syscall(void);
110  extern int sys_show_process_family(void);
```

```
134  [SYS_arithmeticsyscall] sys_arithmeticsyscall,
135  [SYS_copy_file] sys_copy_file,
136  [SYS_grep_syscall] sys_grep_syscall,
137  [SYS_set_priority_syscall] sys_set_priority_syscall,
138  [SYS_show_process_family] sys_show_process_family,
139  };
```

- user.h: اعلان تابع set_priority_syscall برای استفاده در فضای کاربر

```
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int arithmeticsyscall(int a, int b);
int copy_file(char*, char*);
int grep_syscall(const char *keyword, const char *filename, char *user_buffer, int buffer_size);
int show_process_family(int);
int set_priority_syscall(int pid, int priority);
// ulib.c
```

- usys.s: افزودن نام تابع set_priority_syscall برای ساخت خودکار usys.S

```
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(copy_file)
SYSCALL(grep_syscall)
SYSCALL(set_priority_syscall)
SYSCALL(show_process_family)
```

- defs.h: اعلان تابع کمکی setpriority برای استفاده داخلی کرنل

```
void            yield(void);
int             setpriority(int, int);
int             sys_show_process_family(void);
int             dump_process_family(int);
```

- proc.h: افزودن فیلد int priority به ساختار struct proc

```c
// 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)
  int priority;
};
```

- proc.c: پیاده‌سازی تابع کمکی setpriority، مقداردهی اولیه اولویت (به 1) در allocproc و بازنویسی منطق scheduler

```c
570  int setpriority(int pid, int priority)
571  {
572      struct proc *p;
573      int found = 0;
574
575      if (priority < 0 || priority > 2)
576          return -1;
577
578      acquire(&ptable.lock);
579      for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
580      {
581          if (p->pid == pid)
582          {
583              p->priority = priority;
584              found = 1;
585              break;
586          }
587      }
588      release(&ptable.lock);
589
590      return (found ? 0 : -1);
591  }
```

- sysproc.c: پیاده‌سازی تابع sys_set_priority_syscall

۲. منطق اصلی (در sysproc.c و proc.c)

تابع sys_set_priority_syscall کارهای زیر را به ترتیب انجام می‌دهد:

- دریافت آرگومان‌ها: دریافت pid و priority از فضای کاربر با استفاده از argint (مشابه)
- فراخوانی تابع کمکی: فراخوانی تابع setpriority(pid, priority) که در proc.c قرار دارد.
- اجرای منطق در proc.c:
  - تابع setpriority معتبر بودن priority (باید 0، 1 یا 2 باشد) را بررسی می‌کند.
  - ptable را قفل کرده، پردازه با pid مورد نظر را پیدا می‌کند و فیلد priority آن را به‌روز می‌کند.
- تغییر زمان‌بند: منطق تابع scheduler (در proc.c) بازنویسی شد تا پردازه‌های RUNNABLE را بر اساس اولویت (ابتدا 0، سپس 1، و در آخر 2) انتخاب کند.

```
196  int sys_set_priority_syscall(void)
197  {
198      int pid;
199      int priority;
200
201      if (argint(0, &pid) < 0)
202          return -1;
203      if (argint(1, &priority) < 0)
204          return -1;
205
206      return setpriority(pid, priority);
207  }
208
209
```

۳. برنامه تست

- فایل priority_test.c به عنوان یک برنامه سطح کاربر نوشته شد.
- این برنامه دو فرزند با کار محاسباتی سنگین ایجاد می‌کند. سپس با استفاده از set_priority_syscall به یکی اولویت بالا (0) و به دیگری اولویت پایین (2) تخصیص می‌دهد تا نشان دهد فرزندی که اولویت بالا دارد، زودتر تمام می‌شود (مشابه).

priority_test.c > main(void)

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

void cpu_intensive_task(int pid)
{
    long i;
    for (i = 0; i < 2000000000; i++)
    {
        if (i % 200000000 == 0)
        {
            printf(1, "PID %d: Working...\n", pid);
        }
    }
}

int main(void)
{
    int pid1, pid2;
    int mypid = getpid();

    pid1 = fork();
    if (pid1 == 0)
    {
        int child1_pid = getpid();
        printf(1, "Child 1 (PID %d) created. Will run CPU task.\n", child1_pid);
        cpu_intensive_task(child1_pid);
        printf(1, "* Child 1 (PID %d - LOW PRIORITY) finished. *\n", child1_pid);
        exit();
    }

    pid2 = fork();
    if (pid2 == 0)
    {
        int child2_pid = getpid();
        printf(1, "Child 2 (PID %d) created. Will run CPU task.\n", child2_pid);
        cpu_intensive_task(child2_pid);
        printf(1, "* Child 2 (PID %d - HIGH PRIORITY) finished. *\n", child2_pid);
        exit();
    }
    sleep(200);

    printf(1, "Parent (PID %d): Setting priorities...\n", mypid);
    printf(1, "Parent: Priority for Child 1 (PID %d) set to LOW (2).\n", pid1);
    printf(1, "Parent: Priority for Child 2 (PID %d) set to HIGH (0).\n", pid2);

    set_priority_syscall(pid1, 2);
```

```
31
32      pid2 = fork();
33      if (pid2 == 0)
34      {
35          int child2_pid = getpid();
36          printf(1, "Child 2 (PID %d) created. Will run CPU task.\n", child2_pid);
37          cpu_intensive_task(child2_pid);
38          printf(1, "* Child 2 (PID %d - HIGH PRIORITY) finished. *\n", child2_pid);
39          exit();
40      }
41      sleep(200);
42
43      printf(1, "Parent (PID %d): Setting priorities...\n", mypid);
44      printf(1, "Parent: Priority for Child 1 (PID %d) set to LOW (2).\n", pid1);
45      printf(1, "Parent: Priority for Child 2 (PID %d) set to HIGH (0).\n", pid2);
46
47      set_priority_syscall(pid1, 2);
48      set_priority_syscall(pid2, 0);
49
50      wait();
51      wait();
52
53      printf(1, "Parent: Both children finished. Test complete.\n");
54      exit();
55  }
```

```
PID 6: Working...
PID 6: Working...
Parent (PID 5): Setting priorities...
Parent: Priority for Child 1 (PID 6) set to LOW (2).
Parent: Priority for Child 2 (PID 7) set to HIGH (0).
Child 2 (PID 7) created. Will run CPU task.
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
PID 7: Working...
* Child 2 (PID 7 - HIGH PRIORITY) finished. *
PID 6: Working...
PID 6: Working...
PID 6: Working...
PID 6: Working...
PID 6: Working...
PID 6: Working...
PID 6: Working...
PID 6: Working...
* Child 1 (PID 6 - LOW PRIORITY) finished. *
Parent: Both children finished. Test complete.
```