پروژه 2 آزمایشگاه سیستم‌عامل

گروه 15 **اعضای گروه:** نسا عباسی مقدم-آوا میرمحمدمهدی-سپهر آزردار

1) کتابخانه‌های (قاعدتاً سطح کاربر، منظور فایل‌های تشکیل دهنده‌ی متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانیهای سیستمی و علت این استفاده بررسی نمایید.

فایل‌های تشکیل‌دهنده‌ی متغیرULIB شامل ulib.o, usys.o, printf.o, umalloc.o است.

در ulib.c تعدادی تابع مانند strcpy, strcmp, strlen, strchr, atoi برای کار کردن با رشته‌هایی از کاراکتر نوشته شده‌اندو در آنها از فراخوانی‌های سیستمی استفاده نشده است. در تابع memmove نیز که برای ریختن محتوای یک متغیر در دیگری است از فراخوانی سیستمی استفاده نشده است. در تابع memset از فراخوانی سیستمی stosb استفاده شده است؛ در این تابع، حافظه‌ی مورد نظر را با مقداری که می‌خواهیم پر می‌کنیم. در تابع gets از فراخوانی سیستمی read استفاده شده است و در یک حلقه و در هر مرحله یک ورودی از روی STDIN خوانده می‌شود. در تابع stat ابتدا با استفاده از فراخوانی سیستمی open، یک فایل باز می‌شود و سپس با استفاده از فراخوانی سیستمی fstat اطلاعات آن فایل را بدست می‌آوریم و در انتها با استفاده از فراخوانی سیستمی close آن فایل را می‌بندیم.

در printf.c توابع putc, printint, printf وجود دارند که از میان آنها تنها در تابع putc فراخوانی سیستمی به کار رفته است؛ فراخوانی write یک کاراکتر و فایل دیسکریپتور مربوط به آن را گرفته و آن را چاپ می‌کند.

در umalloc.c توابع malloc و free تعریف شده‌اند که کار تخصیص و آزاد کردن حافظه را به عهده دارند و در آنها از فراخوانی‌های سیستمی استفاده نشده است؛ در تابع morecore در این فایل، از فراخوانی سیستمی sbrk استفاده شده است؛ این فراخوانی سیستمی، حافظه‌ی فیزیکی را تخصیص می‌دهد و آن را به آدرس مجازی پردازه، map می‌کند.

در usys.S که به زبان اسمبلی نوشته شده است، تمامی سیستم کال‌ها به صورت SYSCALL(name_of_the_system_call) نوشته شده‌اند و با فراخوانی هریک از این سیستم‌کال‌ها به #define که در خط 4 این فایل قرار دارد مراجعه می‌شود؛ در اینجا نام سیستم کال به صورت گلوبال نوشته می‌شود و همچنین شماره‌ی آیدی این سیستم کال در رجیستر eax نوشته می‌شود؛ $T_SYSCALL نیز برابر با 64 است زیرا شماره تله فراخوانی سیستمی 64 است و برنامه جهت فراخوانی سیستمی دستور int 64 را فراخوانی می‌کند؛ در واقع وقتی software interrupt اتفاق می‌افتد، این اعمال انجام می‌شوند.

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
```

2) فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روش‌ها را در لینوکس به اختصار توضیح دهید. میتوانید از مرجع [3] کمک بگیرید.

-**pseudo file systems :** فایل سیستمی که فایل‌های حقیقی ندارد و ورودی‌های مجازی دارد که خود سیستم فایل در آن نقطه ایجاد می‌کند، pseudo file system نام دارد؛ این نوع فایل سیستم، اطلاعاتی را درباره سیستم فعلی درحال اجرا نگه می‌دارد. برای مثال sys/ یک نمایش از device های فیزیکی در ماشین نشان می‌دهد و proc/ اطلاعات زیادی درباره کنترل ست فعلی نگه می‌دارد. از آنجایی که برخی از این pseudo file system ها کارهایی مانند system call ها انجام می‌دهند، نیاز به دسترسی به کرنل دارند.

-**traps and exceptions**: در این حالت دسترسی به کرنل انجام می‌شود تا تله‌ها هندل شوند یا exception برطرف شود و سپس به user mode برمی‌گردیم؛ از انواع exception ها می‌توان به تقسیم بر 0 یا overflow اشاره کرد.

-**interrupts**: برای مثال اگر از زمانی که برای انجام یک پردازه در نظر گرفته شده است، بگذرد، اگر در یوزر مود باشیم نیاز است که به کرنل مود برویم و پردازه‌ی بعدی که آماده‌ی اجرا است را اجرا کنیم(برای اینکار مجددا باید به یوزر مود برویم.)

3) آیا باقی تله‌ها را نمیتوان با دسترسی USER_DPL فعال نمود؟ چرا؟

خیر؛ xv6 به پردازه‌ها این اجازه را نمی‌دهد که interrupt های دیگری را با int فعال کند و در این صورت با یک general protection exception مواجه می‌شوند که به وکتور 13 می‌روند. در واقع سطح دسترسی USER_DPL، دسترسی کاربر است و اگر باقی تله‌ها با همین سطح دسترسی فعال شوند، به علت احتمال وجود باگ یا هرنوع سواستفاده در سطح کاربر، امنیت سیستم در معرض خطر قرار می‌گیرد. وقتی protection level از سطح کاربر به سطح کرنل تغییر می‌کند، کرنل نباید از پشته‌ی مربوط به کاربر استفاده کند چون ممکن است معتبر نباشد؛ پردازه‌ی کاربر ممکن است حاوی خطایی باشد که باعث شود رجیستر esp کاربر شامل آدرسی شود که متعلق به حافظه‌ی پردازه‌ی کاربر نباشد. Xv6 سخت افزار x86 را طوری برنامه‌ریزی می‌کند که هنگام وقوع تله، پشته را سوییچ کند؛ این کار به این صورت انجام می‌شود که یک دیسکریپتور task segment ایجاد می‌کند که از طریق آن، سخت افزار یک stack segment selector و یک مقدار جدید برای esp% لود می‌کند.

4) در صورت تغییر سطح دسترسی، ss و esp روی پشته push می‌شوند. در غیر اینصورت push نمیشود. چرا؟

برای دسنرسی سطح کاربر از پشته کاربر و برای دسترسی سطح کرنل از پشته کرنل استفاده می‌شود. ss رجیستری است که بلوکی از حافظه که برای پشته استفاده می‌شود را مشخص می‌کند و esp رجیستر اشاره‌گر به پشته است و به جایگاه دقیق stack segment که در همه‌ی مواقع در بالای پشته است اشاره می‌کند؛ پس در زمان تغییر سطح دسترسی باید این دو روی پشته push شوند تا هنگام برگشت به سطح دسترسی اولیه اطلاعات آنها از دست نرود و وقتی سطح دسترسی تغییر نکرده است به پشته‌ی آن، دسترسی داریم پس نیازی به push کردن این دو مقدار نیست.

وقتی یک trap اتفاق می‌افتد، اگر پردازنده درحال اجرا روی مود کاربر بود، esp و ss را از دیسکریپتورtask segment لود می‌کند و ss و esp قدیمی را روی پشته‌ی جدید push می‌کند. اگر پردازنده درحال اجرا روی مود کرنل بود، هیچ یک از این اتفاقات

نمی‌افتد و پردازنده %eflags, %cs, %eip را push می‌کند و برای برخی از تله‌ها یک error word نیز push می‌کند و سپس eip و cs را از ورودی IDT مربوطه، لودو می‌کند.

5) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در () argptr بازه آدرسها بررسی میگردند؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟ در صورت عدم بررسی بازه‌ها در این تابع، مثالی بزنید که در آن، فراخوانی سیستمی ()sys_read اجرای سیستم را با مشکل رو به رو سازد.

توابع argint, argstr, argptr، nامین آرگومان system call (اینتیجر، پوینتر یا استرینگ) را بازیابی می‌کنند.

argint از رجیستر %esp استفاده می‌کند تا مکان nامین آرگومان را قرار دهد. %esp به آدرس بازگشتی سیستم کال stub اشاره می‌کند. (توابع سیستم کال stub، یک interface سطح بالا برای تابعی که کار اصلی آن تولید software interrupt (trap) است فراهم می‌کند). آرگومان‌ها بالای آن در %esp + 4 قرار دارند؛ یعنی آرگومان nام در %esp + 4 + 4*n قرار دارد. argint، fetchint را فراخوانی می‌کند تا مقدار موجود در آن آدرس را از user memory بخواند و در *ip بنویسد. fetchint می‌تواند به سادگی آدرس را به پوینتر cast کند چون یوزر و کرنل از یک page table استفاده می‌کنند ولی کرنل باید verify کند که پوینتر به واسطه کاربر، درواقع یک پوینتر در بخش یوزر فضای آدرس‌دهی است. کرنل، سخت افزار page table رابررسی می‌کند تا اطمینان یابد که پردازه نمی‌تواند به حافظه خارج ازحافظه local خصوصی دسترسی یابد. اگر یک برنامه کاربر بخواهد در آدرس p->sz یا بالاتر بنویسد یا از آن بخواند، پردازنده باعث ایجاد یک segmentation trap می‌شود و این تله، پردازه را kill می‌کند پس کرنل باید چک کند که آدرس، پایین p->sz باشد. آرگومان های این تابع، عدد n براش مشخص کردن شماره آرگومان و آدرس یک اینتیجربرای ذخیره‌ی محتوای آرگومان است.

argptr، argint را فراخوانی می‌کند تا آرگومان را به عنوان اینتیجر fetch کند و سپس چک کند که آن اینتیجر به عنوان یوزر پوینتر در بخش یوزر فضای آدرس دهی است یا خیر؛ در واقع در زمان فراخوانی کد argptr، دو فراخوانی انجام می‌شود: چک کردن پوینتر پشته‌ی یوزر درحین fetch کردن آرگومان و چک کردن آرگومان (که خودش پوینتر یوزر است). آرگومان های این تابع، عدد n براش مشخص کردن شماره آرگومان و آدرس یک پوینتر برای ذخیره‌ی محتوای آرگومان و اندازه‌ی آن چیزی است که خوانده می‌شود.

argstr چک می‌کند که پوینتری که به عنوان nامین آرگومان گرفته است به یک رشته‌ی NUL-terminated اشاره کند و رشته‌ی کامل در زیر قسمت پایانی بخش یوزر در فضای آدرس‌دهی باشد. آرگومان های این تابع، عدد n براش مشخص کردن شماره آرگومان و آدرس یک *char برای ذخیره‌ی مقدار خوانده شده است.

درصورتی که آدرس‌ها چک نشوند ممکن است به آدرس خارج از محدوده‌ی آن پردازه دسترسی پیدا کنیم و این موضوع باعث ایجاد مشکلات امنیتی شده و همچنین ممکن است به اطلاعات اشتباه دسترسی پیدا کنیم و یا در اطلاعات برنامه‌ی دیگری تغییرات به وجود بیاوریم و اجرای آن برنامه نیز دچار اختلال شود. برای مثال در فراخوانی تابع read به شکل read(0, buf, 10)، این تابع به اندازه‌ی 10 کاراکتر از stdin می‌خواند و در buf می‌ریزد؛ برای اینکار در ابتدای وقوع سیستم کال به تابع sys_read (پیاده‌سازی

این تابع در فایل sysfile.c قرار دارد.) مراجعه می‌شود و در آنجا سه شرط معتبر بودن فایل دسکریپتور و مقدار بازگشتی تابع argint(2, &n) و همچنین مقدار بازگشتی argptr(1, &p, n) چک می‌شود. (در مورد چگونگی کارکرد این دو تابع در بالا توضیح داده شده است و پارامتر سایز آنها در این مثال عدد 10 است) اگر مقدار n از سایز حافظه‌ای که متعلق به آن پردازه است فراتر رود و در بافر overflow رخ می‌دهد و اگر بازه‌ها در argptr بررسی شود پردازه نمی‌تواند از حافظه‌ای که به آن تخصیص یافته جلوتر برود ولی اگر این اتفاق نیفتد، علاوه بر ایجاد مشکل در پردازه‌ی درحال اجرا، پردازه‌ای که در حافظه‌ی متغلق به آن وارد شدیم نیز دچار مشکل می‌شد.

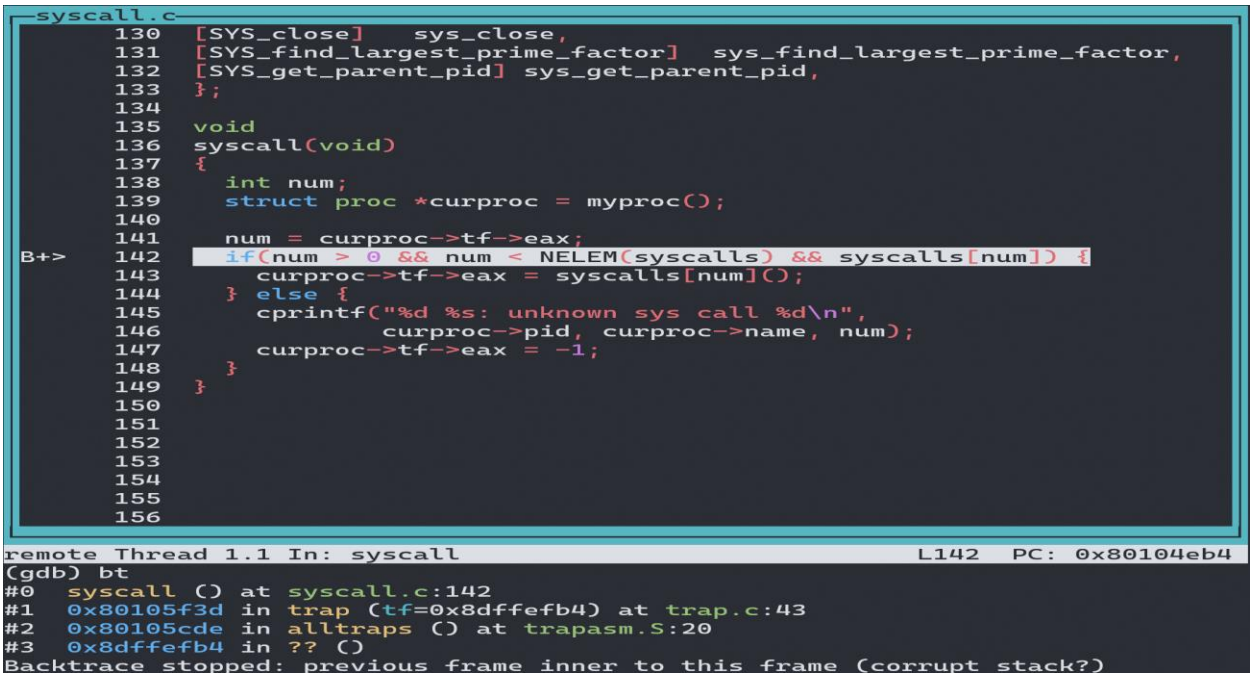
بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb:

برنامه‌ی سطح کاربر نوشته شده در فایل get_pid.c که برای اضافه کردن آن به xv6، آن را در قسمت UPROGS در Makefile اضافه کردیم:

```
int main(int argc, char *argv[])
{
    int pid = getpid();
    printf(1, "The getpid() value is: %d\n", pid);
    exit();
}
```

با استفاده از دستور bt که به معنای backtrace است ، لیستی از فراخوانی‌های توابع که در حال حاضر در یک thread فعال هستند و در استک اضافه شدهاند، نمایش داده میشود.

با گذاشتن برک پوینت روی خط 142 فایل syscall.c (که در تصویر نشان داده شده است) و اجرای برنامه‌ی سطح کاربر نوشته شده و همچنین اجرای دستور bt نتیجه‌ی زیر حاصل می‌شود:



همان‌طور که می‌دانیم در فایل syscall.h به هر سیستم‌کال یک عدد اختصاص داده شده است و در فایل user.h، declaration سیستم کال‌ها وجود دارد و definition آنها در فایل usys.S به زبان اسمبلی قرار دارد؛ در این فایل شماره سیستم کال در رجیستر eax نوشته می‌شود و دستور int 64 را فراخوانی می‌کند که به دنبال آن به تعریف لیبل vector 64 در فایل vectors.S می‌رود؛ در اینجا مقدار 64 در استک push می‌شود و سپس به alltraps، jump می‌کند که در فایل trapasm.S قرار دارد؛ در اینجا trapfarme ساخته شده و سپس در استک push می‌شود و سپس تابع trap که در فایل trap.c قرار دارد فراخوانی می‌شود. در این تابع، با عدد 64 متوجه می‌شود که یک سیستم‌کال به وجود آمده است و trapfarme ای که از قبل در استک push شده بود، آرگومان این تابع است؛ در این تابع، تابع syscall فراخوانی می‌شود که در آنجا به تابع مربوط به سیستم‌کالی که مربوط به عدد آن عدد (عدد اختصاص یاقته به سیستم کال) است، می‌رویم؛ پس به ترتیب اولین تابعی که فراخوانی شده (jump کردن به تابع مدنظر نیست)، alltraps است و سپس trap و سپس syscall فراخوانی می‌شوند و این توابع به همراه اسم فایلی که در آن قرار دارند به ترتیب با دستور bt نمایش داده شده‌اند.

وکتور64 در فایل vectors.S:

```
318 vector64:
319   pushl $0
320   pushl $64
321   jmp alltraps
```

فایل trapasm.S:

```
 # vectors.S sends all traps here.
.globl alltraps
alltraps:
 # Build trap frame.
 pushl %ds
 pushl %es
 pushl %fs
 pushl %gs
 pushal
```

تابع trap در فایل trap.c:

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

با وارد کردن دستور down که به اندازه‌ی n فریم (در اینجا n برابر با 1 است) به پایین استک می‌رود، خروجی زیر بدست می‌آید:

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

ولی با وارد کردن دستور up که با آن یک فریم به بالای استک و در واقع عقب تر می‌رویم، به تابع trap می‌رویم.

```
trap.c
      31  {
      32    lidt(idt, sizeof(idt));
      33  }
      34
      35  //PAGEBREAK: 41
      36  void
      37  trap(struct trapframe *tf)
      38  {
      39    if(tf->trapno == T_SYSCALL){
      40      if(myproc()->killed)
      41        exit();
      42      myproc()->tf = tf;
  >   43      syscall();
      44      if(myproc()->killed)
      45        exit();
      46      return;
      47    }
      48
      49    switch(tf->trapno){
      50    case T_IRQ0 + IRQ_TIMER:
      51      if(cpuid() == 0){
      52        acquire(&tickslock);
      53        ticks++;
      54        wakeup(&ticks);
      55        release(&tickslock);
      56      }
      57      lapiceoi();
remote Thread 1.1 In: trap                                L43   PC: 0x80105f3d
(gdb) up
#1  0x80105f3d in trap (tf=0x8dffefb4) at trap.c:43
```

در این مرحله با چاپ کردن مقدار رجیستر eax مقدار 5 نمایش داده می‌شود.

```
(gdb) print myproc()->tf->eax
$1 = 5
(gdb)
```

این مقدار برابر با شماره فراخوانی سیستمی ()getpid که برابر با 11 است، برابر نیست؛ دلیل این اتفاق این است که تا پیش از رسیدن به این فراخوانی سیستمی، فراخوانی‌های سیستمی دیگری اجرا می‌شوند که ما با وارد کردن continue و خواندن مجدد مقدار رجیستر eax به آنها پی می‌بریم.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$3 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$4 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$5 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$6 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$7 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$8 = 1
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$9 = 3
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$10 = 12
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$11 = 7
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:142
142         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print myproc()->tf->eax
$12 = 11
```

```
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ get_pid
The getpid() value is: 3
$
```

در تصویر دیده می‌شود که در رجیستر eax قبل از مقدار مدنظر ما (11)، مقادیر 5، 1، 3، 12 و 7 دیده می‌شود که به ترتیب مربوط به فراخوانی‌هاس سیستمی read (برای خواندن دستور تایپ شده در ترمینال)، fork (برای ایجاد پردازه‌ی جدید برای اجرای برنامه سطح کاربر)، wait (اجرا در پردازه پدر برای انتظار پایان یافتن پردازه فرزند)، sbrk (برای تخصیص حافظه به پردازه)، exec(برای اجرای برنامه get_pid ایجاد می‌شود)؛ پس از این مقدار eax برابر با 11 می‌شود که این مقدار شناسه سیستم کال getpid است و بعد از آن سیستم‌کال های دیگری برای چاپ عبارت "The getpid() value is: 3" فراخوانی می‌شوند.

اضافه کردن سیستم کال find_largest_prime_factor():

برای اضافه کردن آیدی این سیستم کال در فایل syscall.h، عدد 22 را به آن اختصاص می‌دهیم (می‌دانیم در xv6، 21 سیستم کال وجود دارد پس تا شماره 21 از قبل رزرو شده است). همچنین در user.h, usys.S, syscall.c و defs.h نام و declaration این تابع را اضافه می‌کنیم. در فایل proc.c، بدنه‌ی تابع int find_largest_prime_factor(int n) را تعریف می‌کنیم که در واقع در آن منطق این برنامه قرار دارد و به ازای گرفتن عدد n، پاسخ را برای ما return می‌کند. در فایل sysproc.c نیز بدنه‌ی تابع int sys_find_largest_prime_factor(void) را تعریف می‌کنیم؛ در این تابع، تابع get_parent_pid() که در کرنل است با مقداری که در رجیستر ebx قرار دارد، فراخوانی می‌شود.

منطق برنامه که در فایل proc.c نوشته می‌شود:

```
int
find_largest_prime_factor(int n)
{
    int div = 2, largest_factor = 1;
    while (n != 0)
    {
        if (n % div != 0)
            div = div + 1;
        else
        {
            largest_factor = n;
            n = n / div;
            if (n == 1)
                break;
        }
    }
    return largest_factor;
}
```

فراخوانی find_largest_prime_factor که با مقدار رجیستر ebx به عنوان آرگومان این تابع انجام می‌شود (در فایل sysproc.c):

```
int
sys_find_largest_prime_factor(void)
{
  int num = myproc()->tf->ebx;
  cprintf("KERNEL sys_find_largest_prime_factor() is called for n = %d\n", num);
  return find_largest_prime_factor(num);
}
```

در برنامه‌ی سطح کاربر نوشته شده (largest_prime_factor.c) در ابتدا معتبر بودن تعداد آرگومان چک می‌شوود و پس از آن مقدار فعلی رجیستر ebx را در متغیری ذخیره کرده و سپس عدد داده شده به عنوان ورودی را در رجیستر ebx می‌ریزیم؛ تابع find_largest_prime_factor() را فراخوانی می‌کنیم و سپس مقدار قبلی رجیستر ebx را که قبلا در متغیری ذخیره کرده بودیم، مجددا در رجیستر ebx می‌ریزیم تا در واقع پس از پایان اجرای این برنامه، مقدار این رجیستر تغییری نکند.

```
int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        if(argc < 2)
            printf(2, "Error: you didn't enter the number!\n");
        else if(argc > 2)
            printf(2, "Error: Too many arguments!\n");
        exit();
    }
    else
    {

        int last_ebx_value;
        int num = atoi(argv[1]);

        asm volatile(
            "movl %%ebx, %0;" // last_ebx_value = ebx
            "movl %1, %%ebx;" // ebx = num
            : "=r" (last_ebx_value)
            : "r"(num)
        );
        printf(1, "USER find_largest_prime_factor() called for n = %d\n" , num);
        int answer = find_largest_prime_factor();
        printf(1, "largest prime factor in number %d is: %d\n" , num , answer);

        asm("movl %0, %%ebx"
            :
            : "r"(last_ebx_value)
        );
    }
    exit();
}
```

برای اضافه کردن این برنامه‌ی سطح کاربر لازم است تا در فایل Makefile، این برنامه را در قسمت EXTRA و UPROGS اضافه کنیم. (تغییرات در commit های github قابل مشاهده است.)

نتیجه‌ی اجرای برنامه با عدد 276 در زیر آمده است:

```
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ largest_prime_factor 276
USER: find_largest_prime_factor() is called for n = 276
KERNEL: sys_find_largest_prime_factor() is called for n = 276
largest prime factor in number 276 is: 23
$
```

اضافه کردن سیستم کال change_file_size():

برای اضافه کردن این سیستم کال declaration تابع سطح کاربر آن را در user.h اضافه می‌کنیم و همچنین در syscall.c نیز declaration تابع سطح کرنل آن را اضافه می‌کنیم؛ در syscall.h به این سیستم کال، شناسه‌ی 24 را اختصاص می‌دهیم و در usys.S نیز SYSCALL(change_file_size) را اضافه می‌کنیم.

در sysfile.c تابع sys_change_file_size را پیاده‌سازی می‌کنیم. (چون در این سیستم‌کال با فایل کار می‌کنیم، آن را در sysfile.c پیاده‌سازی می‌کنیم و نه در sysproc.c). در این تابع ابتدا آرگومان ها را با استفاده از دو تابع argstr و argint، دو پارامتر سایز دلخواه و آدرس فایل را از استک می‌خوانیم و سپس با کمک definition تابع sys_open، تکه کدی می‌نویسیم تا فایل مورد نظر را باز کند؛ در حالتی که سایز دلخواه وارد شده بزرگتر از سایز فایل بود، از انتهای فایل (offset = ip->size) به تعداد اختلاف سایز دلخواه و سایز اصلی فایل، به کمک تابع writei ، کاراکتر نال اضافه می‌کنیم و در حالتی که سایز دلخواه وارد شده کوچکتر از سایز فایل بود، سایز inode ساخته شده (ip->size) را برابر با سایز دلخواه قرار می‌دهیم و تابع iupdate را با inode متعلق به فایل مورد نظر، فراخوانی می‌کنیم. تصویر این تابع در صفحه‌ی بعد آورده شده است:

```
int
sys_change_file_size(void)
{
  char *path;
  int length;
  struct file *f;
  struct inode *ip;
  cprintf("KERNEL: change_file_size() is called\n");
  if(argstr(0, &path) < 0 || argint(1, &length) < 0 )
    return -1;

  begin_op();

  ip = create(path, T_FILE, 0, 0);
  if(ip == 0)
  {
    end_op();
    return -1;
  }

  if((f = filealloc()) == 0 )
  {
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = 1;
  f->writable = 1;
  int inode_size = ip->size;
  int r;
  uint offset;
  begin_op();
  ilock(f->ip);
  if (inode_size < length)
  {
    offset = ip->size;
    char null = '\0';
    for(int i=0; i < length - inode_size; i++)
      if ((r = writei(f->ip, &null, offset, 1)) > 0)
        offset += r;
  }
  else
  {
    ip->size = length;
    iupdate(ip);
  }
  iunlock(f->ip);
  end_op();
  return 0;
}
```

برای این سیستم کال برنامه‌ی سطح کاربر زیر در فایل change_file_size.c نوشته شده است؛ همچنین برای اضافه کردن این برنامه به Makefile، آن را در قسمت UPROGS اضافه می‌کنیم.

```
int main(int argc, char *argv[])
{
    if(argc != 3)
    {
        printf(1, "error: numbers of arguments are uncorrect!\n");
        exit();
    }

    printf(1, "USER: change_file_size() is called\n");
    change_file_size(argv[1], atoi(argv[2]));
    exit();
}
```

در تصویر زیر نمونه‌ای از اجرای این برنامه دیده می‌شود که در آن سایز فایل توسط دستور ls [file_name] قایل مشاهده است.

مثال کوچک کردن سایز فایل:

```
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ echo hiiiiiiiiiiiiiiiiiiiiiiiii > test.txt
$ ls test.txt
test.txt       2 24 29
$ change_file_size test.txt 10
USER: change_file_size() is called
KERNEL: change_file_size() is called
$ cat test.txt
hiiiiiiiii$ ls test.txt
test.txt       2 24 10
$
```

مثال بزرگ کردن سایز فایل:

```
$ echo hellooooooooooooooooo > test2.txt
$ ls test2.txt
test2.txt      2 25 40
$ change_file_size test2.txt 60
USER: change_file_size() is called
KERNEL: change_file_size() is called
$ cat test2.txt
hellooooooooooooooooo
$ ls test2.txt
test2.txt      2 25 60
$
```

اضافه کردن سیستم کال ()get_callers:

برای اضافه کردن این سیسستمم کال در فایل syscall.h عدد 25 را به عنوان شناسه‌ی این سیستم کال در نظر می‌گیریم و در فایل syscall.c، user.h و usys.S نیز مانند سیستم کال‌های قبلی تغییراتی به وجود می‌آوریم؛ در فایل sysproc.c یک آرایه‌ی دو بعدی تعریف می‌کنیم تا به ازای هر سیستم کال، شناسه پردازه‌هایی که آن را فراخوانی می‌کنند ذخیره کند (process_history)؛ همچنین در یک آرایه‌ی یک بعدی نیز به ازای هر سیستم کال تعداد پردازه هایی که آن را فراخوانی کردند، ذخیره می‌کنیم. (process_count)؛ در فایل sysproc.c، تابع ()add_process_history را تعریف می‌کنیم که در آن عملیات اضافه شدن هر پردازه به لیست پردازه‌های یک سیستم کال انجام می‌شود؛ همچنین در این تابع سیستم کال ()get_callers را نیز پیاده سازی می‌کنیم.

خروجی به ازای سیستم کال write، اعداد 1 و 2 و 3 است که عدد 1 مربوط به INIT است که در ابتدای سیستم عامل اجرا می‌شود و برای مثال نام اعضای گروه را چاپ می‌کند؛ عدد 2 مربوط به shell است که $ را چاپ می‌کند و عدد 3 مربوط به پردازه ی برنامه‌ی سطح کاربر نوشته شده است (چون بعد از بالا آمدن سیستم عامل این برنامه را اجرا کردیم عدد 3 نمایش داده شده و اگر پس از اجرای پردازه‌های دیگر برنامه‌ی get_callers را اجرا می‌کردیم، این عدد متفاوت بود)؛ تعداد عددهای 1 و 3 چاپ شده زیاد است چون به ازای هر کاراکتری که در کنسول نوشته می‌شود، یک بار سیستم کال write فراخوانی می‌شود.

خروجی به ازای سیستم کال fork، عدد 1 و 2 است؛ در واقع پردازه‌ی INIT که در ابتدای بالا آمدن سیستم عامل اجرا می‌شود یک پردازه را ()fork می‌کند و سپس این پردازه‌ی فرزند، ()exec را اجرا می‌کند تا shell تشکیل شود و سپس پردازه‌ی shell یک پردازه‌ی جدید را ()fork می‌کند و این پردازه که همان پردازه‌ی اجرای برنامه‌ی ما است، ()exec را اجرا می‌کند پس به همین دلیل سیستم کال‌هایی که ()fork را فراخوانی می‌کنند دارای شناسه 1 و 2 هستند.

خروجی به ازای سیستم کال wait نیز عدد 1 و 2 است؛ دلیل این امر این است که در هر بار فراخوانی fork، پردازه‌ی پدر wait را نیز فراخوانی می‌کند.

فراخوانی تابع اضافه کردن پردازه به پردازه‌های یک سیستم کال در فایل syscalll.c آمده است:

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  add_process_history(num, curproc->pid);
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

همچنین پیاده‌سازی سیستم کال get_callers و تابع add_process_history در فایل sysproc.c آمده است:

```
#define PROC_HIST_SIZE 1000
#define SYSCALL_SIZE 25

int process_count[SYSCALL_SIZE] = {0};
int process_history[SYSCALL_SIZE][PROC_HIST_SIZE] = {0};

void add_process_history(int sys_call_id, int pid)
{
  process_history[sys_call_id - 1][process_count[sys_call_id - 1] % PROC_HIST_SIZE] = pid;
  process_count[sys_call_id - 1] += 1;
}

int
sys_get_callers(void)
{
  int sys_call;
  if(argint(0, &sys_call) < 0)
    return -1;
  cprintf("KERNEL: sys_get_callers() is called for system call with id = %d\n", sys_call);
  if(process_count[sys_call - 1] <= PROC_HIST_SIZE)
  {
    for (int i = 0; i < process_count[sys_call - 1]; i++)
    {
      if(i != process_count[sys_call - 1] - 1)
        cprintf("%d,", process_history[sys_call - 1][i]);
      else
        cprintf("%d\n", process_history[sys_call - 1][i]);
    }
  }
  else
  {
    int start = process_count[sys_call - 1] % PROC_HIST_SIZE;
    for (int i = start; i < start + PROC_HIST_SIZE; i++)
    {
      int index = i % PROC_HIST_SIZE;
      if(index  == start - 1)
        cprintf("%d\n", process_history[sys_call - 1][index]);
      else
        cprintf("%d,", process_history[sys_call - 1][index]);
    }
  }
  return 0;
}
```

برنامه‌ی سطح کاربر در فایل get_callers.c نوشته شده است که در آن get_callers به ازای pid های 16 (write)، 1 (fork) و 3 (wait) فراخوانی شده است:

```
int main(int argc, char *argv[])
{
    printf(1, "USER: get_callers() is called for write (id = 16)\n");
    get_callers(16);
    printf(1, "USER: get_callers() is called for fork (id = 1)\n");
    get_callers(1);
    printf(1, "USER: get_callers() is called for wait (id = 3)\n");
    get_callers(3);
    exit();
}
```

خروجی دستور get_callers نیز به صورت زیر خواهد بود:

```
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ get_callers
USER: get_callers() is called for write (id = 16)
KERNEL: sys_get_callers() is called for system call with id = 16
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,
3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
USER: get_callers() is called for fork (id = 1)
KERNEL: sys_get_callers() is called for system call with id = 1
1,2
USER: get_callers() is called for wait (id = 3)
KERNEL: sys_get_callers() is called for system call with id = 3
1,2
$
```

اضافه کردن سیستم کال ()get_parent_pid:

برای اضافه کردن این سیستم کال مانند سیستم کال ()find_largest_prime_factor عمل می‌کنیم با این تفاوت که از رجیستر ebx استفاده نمی‌کنیم چون آرگومانی نداریم.

در فایل proc.c، تابع int get_parent_pid(void) را به صورت زیر پیاده‌سازی کردیم؛ این تابع فقط مقدار عددی pid ولد پردازه‌ی کنونی را برمی‌گرداند:

```
int
get_parent_pid(void)
{
    return myproc()->parent->pid;
}
```

در فایل sysproc.c نیز بدنه‌ی تابع int sys_get_parent_pid(void) قرار دارد که در آن تابع ()get_parent_pid که در کرنل قرار دارد، فراخوانی می‌شود:

```
int
sys_get_parent_pid(void)
{
  cprintf("KERNEL: sys_get_parent_pid() is called\n");
  return get_parent_pid();
}
```

در برنامه‌ی سطح کاربر نوشته شده (get_parent_pid.c) در ابتدا با استفاده از تابع fork() یک پردازه‌ی فرزند تشکیل می‌دهیم و سپس برای این پردازه فرزند توسط fork()، فرزندی می‌سازیم. (برای این که این پردازه‌ی جدید به عنوان فرزند پردازه تشکیل شود و نه به عنوان فرزند پردازه‌ی اولیه، نیاز است که توسط شرطی، pid پردازه چک شود و درصورتی که برابر با 0 بود (یعنی خودش پردازه‌ی فرزند بود)، برای آن توسط fork() فرزند جدیدی تشکیل دهیم؛ برای بدست آوردن pid پردازه‌ی پدر نیز در هر مرحله، تابع get_parent_pid() را فراخوانی می‌کنیم. دلیل اضافه کردن wait() این است که وقتی یک پردازه به پایان می‌رسد،resource های آن توسط سیستم عامل deallocate می‌شوند؛ با این حال entry آن در process table باقی می‌ماند تا زمانی که والدش wait() را فراخوانی کند چون process table شامل وضعیت خروج پردازه است.

```c
int main(int argc, char *argv[])
{
    int process_id = getpid();
    printf(1, "USER: get_parent_pid() is called\n");
    int parent_pid = get_parent_pid();
    printf(1, "The first proccess id is: %d, and its parent pid is: %d\n", process_id, parent_pid);

    int pid0 = fork();
    if(!pid0)
    {
        int process_id = getpid();
        printf(1, "USER: get_parent_pid() is called\n");
        int parent_pid = get_parent_pid();
        printf(1, "The second proccess id is: %d, and its parent pid is: %d\n", process_id, parent_pid);

        int pid1 = fork();
        if(!pid1)
        {
            int process_id = getpid();
            printf(1, "USER: get_parent_pid() is called\n");
            int parent_pid = get_parent_pid();
            printf(1, "The third process id is: %d, and its parent pid is: %d\n", process_id, parent_pid);
        }
    }
    wait();
    exit();
}
```

در نهایت با اجرای get_parent_pid داریم:

```
init: starting sh
Group #15:
1- Ava Mirmohammadmahdi
2- Nesa Abassi
3- Sepehr Azardar
$ get_parent_pid
USER: get_parent_pid() is called
KERNEL: sys_get_parent_pid() is called
The first proccess id is: 3, and its parent pid is: 2
USER: get_parent_pid() is called
KERNEL: sys_get_parent_pid() is called
The second proccess id is: 4, and its parent pid is: 3
USER: get_parent_pid() is called
KERNEL: sys_get_parent_pid() is called
The third process id is: 5, and its parent pid is: 4
$
```