

به نام خدا

گزارش پروژه دوم آزمایشگاه سیستم عامل

پاییز ۱۴۰۱

گروه 3: علی هدائی(810199513)، پویا صادقی(810199447)، علی عطااللهی(810199461)

پاسخ به سوالات تشریحی ★

1) کتابخانه های استفاده شده در 6xv (قاعداً سطح کاربر و موجود در ULIB) را از نظر فراخوانی های سیستمی بررسی کنید:

در Makefile، داریم : ULIB = ulib.o usys.o printf.o umalloc.o. به بررسی هرکدام از این فایل ها بصورت مجزا میپردازیم:

○ Ulib : در این فایل، تعدادی توابع برای کار با آرایه ها وجود دارد (بیشتر عملیات ها مربوط به char* ها میباشند، اما تابع memmove با void* کار میکند). سه تابع از جوامع موجود در این فایل، از فراخوانی های سیستمی استفاده میکنند:

I. Memset: از فراخوانی stosb استفاده میکند که با کمک دستور اسمبلی، با تکرار حلقه به اندازه cnt دفعه، داده موجود در رجیستر a (data) را در آدرس حافظه موجود در رجیستر D (addr) مینویسد و سپس مقدار رجیستر D را یک واحد افزایش میدهد.

II. Gets: در یک حلقه که تا حداکثر دفعات تعیین شده مجاز به تکرار میباشد، در هر دور، از فراخوانی سیستمی read استفاده میکند تا به max یا پایان رشته ورودی یا انتهای خط فعلی برسد و هربار، یک کاراکتر میخواند. شاید دیگر نکته قابل ذکر در اینجا، ست بودن fd به ورودی استاندارد باشد.

III. Stat: در ابتدا با فراخوانی سیستمی open، file descriptor مربوط به فایل را دریافت میکند، سپس با فراخوانی سیستمی fstat، استراکچر stat را تکمیل میکند و در نهایت نیز، با فراخوانی سیستمی close، فایل را میبندد

○ Usys: 21 فراخوانی سیستمی پیشفرض موجود را مشخص کرده (این تعداد در ادامه که به فراخوانی ها اضافه میکنیم، دچار تغییر خواهد شد و در اینجا، منظور حالت پایه 6xv میباشد).

○ Printf: توابع موجود در این فایل، وظیفه چاپ یا وظایف مرتبط و کمک به چاپ را برعهده دارند. تابع putc مستقیماً از فراخوانی سیستمی write استفاده میکند و در هر فراخوانی، فقط یک کاراکتر در fd مربوطه مینویسد. تابع printint مقادیر عددی را چاپ میکند و میتواند عدد صحیح ورودی

را با توجه به مبنای داده شده تفسیر و چاپ کند. Printf نیز که wrapper سرآیند این فایل است، از دو تابع فوق بهره میبرد.

○ Umallocc: شامل تعدادی توابع کتابخانه ای به منظور اختصاص دادن حافظه یا آزاد کردن آن میباشد. فقط تابع morecore میباشد که از فراخوانی سیستمی استفاده میکند. هدف آن افزایش حافظه است که به کمک فراخوانی سیستمی sbrk، اندازه data segment را تغییر میدهد.

(2) انواع روش های فراخوانی های سیستمی در لینوکس را اختصار توضیح دهید.

- Interrupt : دو نوع سخت افزاری و نرم افزاری دارند که نوع سخت افزاری از طریق سخت افزار ها مثل دیوایس های I/O اتفاق میفتد و نوع نرم افزاری آن سیستم کال ها و اکسپشن ها هستند.
- Exception : هنگامی که خطا های محاسباتی مانند تقسیم بر صفر رخ میدهد به مود کرنل رفته تا خطا را رفع کند و سپس به مود یوزر باز میگردد.
- Pseudo-file system : لینوکس یکسری api ها را از طریق pseudo-file system ها را ارسال میکنند (/proc /dev /sys/). آن ها pseudo-file system نامیده میشوند چون توسط دیسک بازگردانده نمیشوند بلکه محتویات ساختمان داده های کرنل را طوری که انگار روی فایل ذخیره شده اند را برای اپلیکیشن ها ارسال میکنند.

(3) آیا باقی تله ها را نمیتوان با دسترسی **DPL_USER** فعال نمود؟ چرا؟

خیر؛ چراکه این امر میتواند اشکالات امنیتی به وجود بیاورد. حال اینکه سطح دسترسی **DPL_USER**، سطح کاربر است و امکان فعال کردن سایر تله ها را ندارد. اگر چنین امکانی وجود داشت، امکان دسترسی به هسته توسط برنامه (کاربر) ایجاد میشد که میتواند امنیت سیستم را دچار مخاطره کند. 6xv به منظور جلوگیری از این مورد، پس از تغییر پردازش از مود کاربر به مود هسته، آرگومان های آنرا چک و تایید میکند.

(4) در صورت تغییر سطح دسترسی ، **ss** و **esp** روی پشته **push** می شود. در غیر این صورت نمی شود. چرا ؟

دو پشته داریم. یکی برای سطح یوزر و یکی برای سطح کرنل داریم. زمانی که تغییر سطح دسترسی داشته باشیم، دیگر امکان دسترسی به استک قبلی نداریم و نمیتوانیم از آن استفاده کنیم. پس به همین دلیل باید `ss` و `esp` را `push` کنیم تا زمان بازگشت از سطح دسترسی دیگر از آنها استفاده کنیم و اطلاعات را بازیابی کنیم.

پس وقتی که تغییر سطح دسترسی نداریم، نیازی به `push` کردن `ss` و `esp` نیست چون به پشته دسترسی داریم.

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.

چهار تابع به این منظور وجود دارند (توجه شود مقدار بازگشتی -1 به معنای ارور میباشد):

- `argint`: به آن میگوییم چندمین آرگومان فراخوانی سیستمی را میخواهیم، مقدار آنرا در آدرس متغیر `int` داده شده (`*int`) قرار میدهد.
- `argptr`: به آن اعلام میکنیم چندمین آرگومان را میخواهیم و سائز آنرا میدهیم، در صورت معتبر بودن اطلاعات، -1 به عنوان ارور برمیگرداند، در غیر این صورت، محتوای آرگومان را بصورت اشاره گر آرایه ای از کاراکترها (اشاره گر به رشته، `**char`) به ما برمیگرداند.
- `argstr`: به آن میگوییم چندمین آرگومان و این تابع پس از بررسی صحت، آنرا در یک اشاره گر به `*char` قرار میدهد (رشته باید با 0 به اتمام برسد یا `nul-terminate` باشد).
- `argfd`: این تابع که برخلاف توابع قبلی در `sysfile` قرار دارد، میتواند `file-descriptor` و ساختار فایل (`struct file`) متناظر با `n`مین آرگومان فراخوانی سیستمی را در اختیار ما قرار دهد.

اگر از بازه معتبر پردازش تجاوز کنیم، از اطلاعات اشتباه و غیرمرتبط با پردازش در ادامه پردازش استفاده خواهیم کرد که اجرای برنامه را دچار مشکل میکند. مثلاً در صورت عدم انجام این چک، ممکن است به یک نام فایل نامعتبر یا غیر آنچه مورد نظر ما می بود، برسیم؛ بطور کلی، امکان ایجاد اشکال در آرگومان های داده شده به `fileread()` به وجود میآید. همچنین میتوان به عنوان سائز، مقدار بزرگی را به `argptr` داد که نه تنها از فضای بافر رد شود، بلکه از محدوده ی `process` نیز خارج شود.

یک برنامه سطح کاربر برای این منظور نوشته ایم. قسمت هایی که در کنسول مینویسند را کامنت میکنیم که فراخوانی های سیستمی زائد نداشته باشیم (فقط دستور printf پاک نشده بود که البته در نتیجه آزمایش تأثیری ندارد). تصویری از اجرای برنامه و فراخوانی و پاسخ دهی آن در **xv6** (در انتها، شرح دستورات gdb استفاده شده آورده میشود):

```
$ pouya@pouya-LP5:~/OS-LAB/os-lab-project2/xv6$ make qemu-gdb
*** Now run 'gdb'.
qemu-system-1386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
Group #3 (TAYAMA) Members:
1- Pouya Sadeghi
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ test_getpid
testing STSCALL getpid()...
getpid() passed!
pid: 3
$
```

اولین بار که به بریک پوینت میرسم، بصورت زیر میباشد که مقدار 5 را مشاهده میکنم. با بررسی در `syscall.h` متوجه میشویم که فراخوانی سیستمی مرتبط با این عدد، `sys_read` میباشد. این مسئله، مورد انتظار ما بود؛ چراکه باید ابتدا کنسول خوانده و دستور داده شده (اجرای برنامه نوشته شده برای تست) توسط سیستم خوانده و پردازش شود.

```

syscall.c
126 [SYS_link] sys_link,
127 [SYS_mkdir] sys_mkdir,
128 [SYS_close] sys_close,
129 ];
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *curproc = myproc();
136
137     num = curproc->tf->eax;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         curproc->tf->eax = syscalls[num]();
140     } else {
141         cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143         curproc->tf->eax = -1;
144     }
145 }
146
147
148
149
150
151

remote Thread 1.1 In: syscall
(gdb) c
Continuing.
=> 0x801052f4 <syscall+20>: lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 2, syscall () at syscall.c:138
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x0010632d in trap (tf=0x8dffe4b4) at trap.c:43
#2  0x001060cf in alltraps () at trapasm.S:20
#3  0x8dffe4b4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) info register eax
eax                0x5                5
(gdb)

```

با ادامه ی فرایند (دستور c) چندین مرتبه دیگر نیز در همین نقطه و با مقدار 5 متوقف میشویم (تا زمانی که دستور بطور کامل از کنسول خوانده شود). سپس خروجی زیر را مشاهده کردیم:

```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6
(gdb) i r eax
eax                0x5                5
(gdb) c
Continuing.
=> 0x801053e4 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:140
140      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) i r eax
eax                0x1                1
(gdb)
```

که مربوط به فراخوانی fork میباشد. در ادامه نیز فراخوانی های wait و sbrk را داریم:

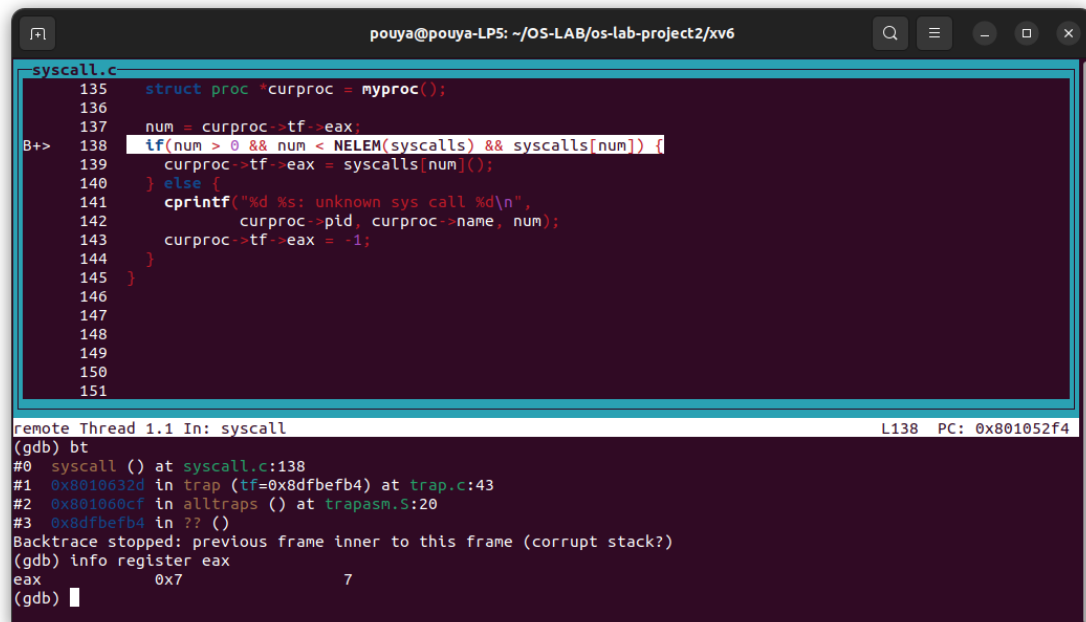
```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6
(gdb) i r eax
eax                0x1                1
(gdb) c
Continuing.
=> 0x801053e4 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:140
140      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) i r eax
eax                0x3                3
(gdb)
```

```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6
(gdb) i r eax
eax                0x3                3
(gdb) c
Continuing.
=> 0x801053e4 <syscall+20>:    lea    -0x1(%eax),%edx

Thread 1 hit Breakpoint 1, syscall () at syscall.c:140
140      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) i r eax
eax                0xc                12
(gdb)
```

حال پس از فراخوانی SYS_sbrk (که شرایط برای اجرای برنامه ما آماده شده)، فراخوانی SYS_exec را میبینیم. در این مرحله، سیستم عامل میخواهد برنامه ی تست ما را اجرا کند.



The screenshot shows a debugger window titled "pouya@pouya-LPS: ~/OS-LAB/os-lab-project2/xv6". The main pane displays the source code of "syscall.c" with line numbers 135 to 151. Line 138 is highlighted, showing a conditional statement: `if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {`. The left margin shows "B+>". The bottom pane shows the "remote Thread 1.1 In: syscall" with a backtrace (bt) and register information. The backtrace lists four frames: #0 at syscall.c:138, #1 in trap.c:43, #2 in trapasm.S:20, and #3 in an unknown location. The register "eax" is shown with the value 0x7.

```
syscall.c
135 struct proc *curproc = myproc();
136
137 num = curproc->tf->eax;
138 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140 } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144 }
145 }
146
147
148
149
150
151

remote Thread 1.1 In: syscall L138 PC: 0x801052f4
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x8010632d in trap (tf=0x8dfbefb4) at trap.c:43
#2  0x801060cf in alltraps () at trapasm.S:20
#3  0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) info register eax
eax                0x7                7
(gdb)
```

پس از این، پردازش ما شروع به کار میکند. سپس فراخوانی سیستمی مربوط به SYS_getpid را میبینیم (توجه شود تمامی دستورات write، کامنت شده بودند).

```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6

syscall.c
135 struct proc *curproc = myproc();
136
137 num = curproc->tf->eax;
B+> 138 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140 } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144 }
145 }
146
147
148
149
150
151
152
153

remote Thread 1.1 In: syscall L138 PC: 0x801052f4

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x8010632d in trap (tf=0x8dfbefb4) at trap.c:43
#2  0x801060cf in alltraps () at trapasm.S:20
#3  0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) info register eax
eax             0xb             11
(gdb)
```

که ما در اینجا فراخوانی مورد نظر را استفاده کردیم و پاسخ را به ما برگردانده است. در ادامه نیز بدلیل ستور printf، فراخوانی سیستمی SYS_write (به دفعات متعدد) را مشاهده میکنیم و در آخر با فراخوانی SYS_exit، از برنامه خارج میشویم.

```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6

syscall.c
135 struct proc *curproc = myproc();
136
137 num = curproc->tf->eax;
B+> 138 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num]();
140 } else {
141     cprintf("%d %s: unknown sys call %d\n",
142             curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144 }
145 }
146
147
148
149
150
151
152
153

remote Thread 1.1 In: syscall L138 PC: 0x801052f4

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb) bt
#0  syscall () at syscall.c:138
#1  0x8010632d in trap (tf=0x8dfbefb4) at trap.c:43
#2  0x801060cf in alltraps () at trapasm.S:20
#3  0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) info register eax
eax             0x10             16
(gdb)
```

```
pouya@pouya-LPS: ~/OS-LAB/os-lab-project2/xv6
syscall.c
135 struct proc *curproc = myproc();
136
137 num = curproc->tf->eax;
138 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     curproc->tf->eax = syscalls[num];
140 } else {
141     cprintf("%d %s: unknown sys call %d\n",
142         curproc->pid, curproc->name, num);
143     curproc->tf->eax = -1;
144 }
145 }
146
147
148
149
150
151
152
153

remote Thread 1.1 In: syscall L138 PC: 0x801052f4
(gdb) ir eax
Undefined command: "ir". Try "help".
(gdb) bt
#0 syscall () at syscall.c:138
#1 0x8010632d in trap (tf=0x8dfbefb4) at trap.c:43
#2 0x801060cf in alltraps () at trapasm.S:20
#3 0x8dfbefb4 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) i r eax
eax                0x2                2
(gdb)
```

به واسطه این آزمایش، متوجه شدیم که برای اجرای یک برنامه، از فراخوانی های سیستمی متعددی استفاده میشود حتی اگر فقط یکی از آنها بطور واضح قابل مشاهده باشد.

🤪:gdb شرح دستورات

- bt : برای مشاهده استک فریم توابع تا به نقطه فعلی میباشد
- up : رفتن به فریم بالاتر (تابعی که در آن تابع فعلی فراخوانی شده)
- down : رفتن به فریم پایین تر (یا تابع داخلی تر)
- c : ادامه ی برنامه (تا رسیدن به نقطه توقف دیگر)
- info register eax : مشاهده ی محتوای ثبات مورد نظر (در اینجا eax)

🐱 ارسال آرگومان های فراخوانی سیستمی

فراخوانی سیستمی شماره 22 را به این فراخوانی ([sys_find_largest_prime_factor](#)) اختصاص میدهیم. این موضوع در فایل syscall.h نمود دارد. در syscall.c این تابع را، به لیست توابع فراخوانی های سیستمی اضافه میکنیم. تعریف آنرا در sysproc.c می آوریم. در user.h، تابع سطح بالاتر [find_largest_prime_factor](#) را اضافه میکنیم و نحوه ی صدا کردن آنرا تعیین میکنیم. نام این تابع را در فایل usys.S اضافه میکنیم و در defs.h نیز forward-declaration را انجام میدهیم (موارد فوق در مرج دیکوئست ها م کامیت های برنچ main مشخص اند).

1 xv6/syscall.h

↑

@@ -20,3 +20,4 @@

20 20 #define SYS_link 19

21 21 #define SYS_mkdir 20

22 22 #define SYS_close 21

23 + #define SYS_find_largest_prime_factor 22

2 xv6/syscall.c

↑

@@ -103,6 +103,7 @@ extern int sys_unlink(void);

103 103 extern int sys_wait(void);

104 104 extern int sys_write(void);

105 105 extern int sys_uptime(void);

106 + extern int sys_find_largest_prime_factor(void);

106 107 static int (*syscalls[])(void) = {

107 108 [SYS_fork] sys_fork,

108 109

↕

@@ -126,6 +127,7 @@ static int (*syscalls[])(void) = {

126 127 [SYS_link] sys_link,

127 128 [SYS_mkdir] sys_mkdir,

128 129 [SYS_close] sys_close,

130 + [SYS_find_largest_prime_factor] sys_find_largest_prime_factor,

129 131 };

10 xv6/sysproc.c

```
.... @@ -89,3 +89,13 @@ sys_uptime(void)
```

```
89      89      release(&tickslock);
```

```
90     90     return xticks;
```

91 91 }

92

```
93 + // SYSCALL to find the largest prime factor of a number
```

```
94 + int
```

```
95 + sys_find_largest_prime_factor(void)
```

 $96 + \{$

```
97 + int number = myproc()->tf->ebx;
```

```
98 + cprintf("Kernel: sys_find_largest_prime_factor(%d) is called\n", number);
```

```
99 +   cprintf("      now calling find_largest_prime_factor(%d)\n", number);
```

```
100 + return find_largest_prime_factor(number);
```

101 + }

1 xv6/user.h

```
.... @ -23,6 +23,7 @@ int getpid(void);
```

```
23      23      char* sbrk(int);
```

```
24      24      int sleep(int);
```

```
25      25      int uptime(void);
```

```
26 + int find_largest_prime_factor(void);
```

26 27

✓ ↕ 1 ■■■■ xv6/usys.S 📄

```
.... @@ -29,3 +29,4 @@ SYSCALL(getpid)
```

```
29      29      SYSCALL(sbrk)
```

```
30      30      SYSCALL(sleep)
```

```
31      31      SYSCALL(uptime)
```

```
32 + SYSCALL(find_largest_prime_factor)
```


1

 xv6/defs.h


```
.... @@ -120,6 +120,7 @@ void userinit(void)
```

```
120      120      int      wait(void);
```

```
121     121     void     wakeup(void*);
```

```
122     122     void     yield(void);
```

```
123 + int find_largest_prime_factor(int);
```

122 124

```
32 xv6/proc.c
@@ -532,3 +532,35 @@ procdump(void)
532 532     cprintf("\n");
533 533 }
534 534 }
535 +
536 + // find the largest prime factor of a number
537 + int
538 + find_largest_prime_factor(int n)
539 + {
540 +     int maxPrime = -1;
541 +
542 +     while (n % 2 == 0) {
543 +         maxPrime = 2;
544 +         n = n / 2;
545 +     }
546 +     while (n % 3 == 0) {
547 +         maxPrime = 3;
548 +         n = n / 3;
549 +     }
550 +
551 +     for (int i = 5; i <= n; i += 6) {
552 +         while (n % i == 0) {
553 +             maxPrime = i;
554 +             n = n / i;
555 +         }
556 +         while (n % (i + 2) == 0) {
557 +             maxPrime = i + 2;
558 +             n = n / (i + 2);
559 +         }
560 +     }
561 +
562 +     if (n > 4)
563 +         maxPrime = n;
564 +
565 +     return maxPrime;
566 + }
```

test_find_largest_prime_factor را بمنظور تست کردن فراخوانی سیستمی ایجاد شده میسازیم و همانند آزمایش اول، آنرا در دسترس کاربر قرار میدهیم. نکته قابل توجه این است که برای تغییر نکردن محتوای رجیستر استفاده شده در این فراخوانی، در فایل تست، مقدار آنرا در یک متغیر در سطح حافظه نگه داشته، سپس مقدار مورد نظر را در آن قرار میدهیم، فراخوانی سیستمی را انجام میدهیم و در آخر، مقدار قبلی رجیستر استفاده شده را در آن مینویسیم.

```

35  xv6/test_find_largest_prime_factor.c
...  ...  @@ -0,0 +1,35 @@

1  + //
2  + // Created by pouya on 11/11/22.
3  + //
4  + #include "types.h"
5  + #include "fcntl.h"
6  + #include "user.h"
7  +
8  + // simple program to test find_largest_prime_factor() system call
9  + int main(int argc, char *argv[]) {
10 +     write(1, "testing find_largest_prime_factor system call...\n", 49);
11 +     if (argc != 2) {
12 +         printf(2, "Error in syntax; please call like:\n>> test_bpf <number>\n");
13 +         exit();
14 +     }
15 +     int n = atoi(argv[1]), prev_ebx;
16 +     asm volatile(
17 +         "movl %%ebx, %0;"
18 +         "movl %1, %%ebx;"
19 +         : "=r" (prev_ebx)
20 +         : "r"(n)
21 +         );
22 +     printf(1, "calling find_largest_prime_factor(%d)...\n", n);
23 +     int result = find_largest_prime_factor();
24 +     asm volatile(
25 +         "movl %0, %%ebx;"
26 +         : : "r"(prev_ebx)
27 +         );
28 +     if (result == -1) {
29 +         write(1, "find_largest_prime_factor () failed!\n", 37);
30 +         write(1, "please check i you entered an integer bigger than 1\n", 52);
31 +         exit();
32 +     }
33 +     printf(1, "find_largest_prime_factor(%d) = %d\n", n, result);
34 +     exit();
35 + }
```

```

4  xv6/Makefile
...  ...  @@ -186,6 +186,8 @@ UPROGS=\
186 186     _zombie\
187 187     _prime_numbers\
188 188     _test_getpid\
189 189 +     _test_find_largest_prime_factor\
190 190 +
189 191
190 192     fs.img: mkfs README $(UPROGS)
191 193     ./mkfs fs.img README $(UPROGS)
...  ...  @@ -255,7 +257,7 @@ qemu-nox-gdb: fs.img xv6.img .gdbinit
255 257     EXTRA=\
256 258     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
257 259     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
258 258 -     printf.c umalloc.c\ prime_numbers.c\ test_getpid.c\
260 260 +     printf.c umalloc.c\ prime_numbers.c\ test_getpid.c\ test_find_largest_prime_factor.c\
259 261     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
```

نمونه ای از اجرای تست فراخوانی:

```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 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
Group #3 (TAYAMA) Members:
1- Pouya Sadeghi
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ test_find_largest_prime_factor 276
testing find_largest_prime_factor system call...
calling find_largest_prime_factor(276)...
Kernel: sys_find_largest_prime_factor(276) is called
      now calling find_largest_prime_factor(276)
find_largest_prime_factor(276) = 23
$ _
```

اضافه کردن سیستم کال  : `get_callers`

```

xv6 > C defs.h > ...
106 int      cpuid(void);
107 void     exit(void);
108 int      fork(void);
109 int      growproc(int);
110 int      kill(int);
111 struct cpu* mycpu(void);
112 struct proc* myproc();
113 void     pinit(void);
114 void     procdump(void);
115 void     scheduler(void) __attribute__((noreturn));
116 void     sched(void);
117 void     setproc(struct proc*);
118 void     sleep(void*, struct spinlock*);
119 void     userinit(void);
120 int      wait(void);
121 void     wakeup(void*);
122 void     yield(void);
123 int      find_largest_prime_factor(int);
124 void     push_pid_in_stack(int,int);
125 void     get_callers(int);
126

```

در اینجا تابع `push_pid_in_stack` و `get_callers` را به منظور استفاده کردن تعریف آنها در فایل های دیگر در `defs.h` قرار داده ایم.

```

xv6 > C proc.h > ...
56 // original data and bss
57 // fixed-size stack
58 // expandable heap
59
60 #define NUM_OF_SYSCALLS 30
61 #define MAX_PID_NUM_SAVED 1000
62 #define MAX_PID_TRACED 1000
63 #define MAX_PID_OUTPUT_IN_ONE_LINE 11
64

```

در اینجا تعدادی ثابت برای آنکه تعداد سیستم کال ها و pid هایی که trace می شوند (حداکثر) مشخص شود.

```

xv6 > C syscall.c > [0] syscalls
94  extern int sys_link(void);
95  extern int sys_mkdir(void);
96  extern int sys_mknod(void);
97  extern int sys_open(void);
98  extern int sys_pipe(void);
99  extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_find_largest_prime_factor(void);
107 extern int sys_get_callers(void);
108 extern int sys_change_file_size(void);

xv6 > C syscall.c > [0] syscalls
125 [SYS_open]      sys_open,
126 [SYS_write]     sys_write,
127 [SYS_mknod]     sys_mknod,
128 [SYS_unlink]    sys_unlink,
129 [SYS_link]      sys_link,
130 [SYS_mkdir]     sys_mkdir,
131 [SYS_close]     sys_close,
132 [SYS_find_largest_prime_factor] sys_fi
133 [SYS_get_callers] sys_get_callers,
134

```

در این دو قسمت سیستم کال مربوط get_callers را قرار داده ایم تا بتوانیم در آن از تابع get_callers بهره ببریم.

```

xv6 > C syscall.c > ...
141 struct proc *curproc = myproc();
142
143 num = curproc->tf->eax;
144 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
145     int pid = curproc->pid;
146     push_pid_in_stack(pid, num);
147     curproc->tf->eax = syscalls[num]();
148 } else {
149     cprintf("%d %s: unknown sys call %d\n",
150           curproc->pid, curproc->name, num);
151     curproc->tf->eax = -1;
152 }
153 }
154

```

در اینجا تابع push_pi_in_stack را صدا می‌زنیم تا هر بتوانیم یک هیستوری از pid هایی که سیستم کال ها را صدا می‌زنند داشته باشیم تا تحلیل فراخوانی سیستم کال ها میسر بشود.

```

xv6 > C syscall.h > ...
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_find_largest_prime_factor 22
24 #define SYS_get_callers 23
25

```

شماره سیستم کال مربوطه را تعریف می‌کنیم.

```

6 > C sysproc.c > ...
99 // SYSCALL to get callers pids
100 void
101 sys_get_callers(void)
102 {
103     int number;
104     argint(0, &number);
105     call_syscall_func_message("get_callers", number);
106     get_callers(number);
107 }

```

در اینجا خود syscall مربوطه را تعریف و تابع مربوط به آن را که get_callers است صدا می‌زنیم

```

xv6 > C test_get_callers.c > main(int, char * [])
1 //
2 // Created by ata on 12/11/22.
3 //
4 #include "types.h"
5 #include "fcntl.h"
6 #include "user.h"
7 #include "syscall.h"
8
9 void call_tests(int sys_call) {
10     printf(1, "calling get_callers for %d\n", sys_call);
11     get_callers(sys_call);
12 }
13
14 // simple program to test get_callers() system call
15 int main(int argc, char *argv[]) {
16     printf(1, "testing get_callers system call\n");
17
18     call_tests(SYS_fork);
19     call_tests(SYS_wait);
20     call_tests(SYS_write);
21
22     exit();
23 }

```


فایل تست `get_callers` که در آن سه سیستم کال `sys_fork` , `sys_wait` , `sys_write` بررسی شده‌اند تا پروسس‌هایی که آنها را صدا زده‌اند شمرده بشوند.

```
xv6 > asm usys.S
29  SYSCALL(sbrk)
30  SYSCALL(sleep)
31  SYSCALL(uptime)
32  SYSCALL(find_largest_prime_factor)
33  SYSCALL(get_callers)
34
```

در این قسمت هم تابع مربوط به `syscall` را قرار می‌دهیم.

```
$ test_fil get_callers
testing get_callers system call
calling get_callers for 1
Kernel: sys_get_callers(1) is called
       now calling get_callers(1)
pid : order that pid called this syscall

1:89 ,2:23
calling get_callers for 3
Kernel: sys_get_callers(3) is called
       now calling get_callers(3)
pid : order that pid called this syscall

1:90 ,2:24
calling get_callers for 16
Kernel: sys_get_callers(16) is called
       now calling get_callers(16)
pid : order that pid called this syscall

1:7 ,1:8 ,1:9 ,1:10 ,1:11 ,1:12 ,1:13 ,1:14 ,1:15 ,1:16 ,1:17 ,
1:18 ,1:19 ,1:20 ,1:21 ,1:22 ,1:23 ,1:24 ,1:25 ,1:26 ,1:27 ,1:28 ,
1:29 ,1:30 ,1:31 ,1:32 ,1:33 ,1:34 ,1:35 ,1:36 ,1:37 ,1:38 ,1:39 ,
1:40 ,1:41 ,1:42 ,1:43 ,1:44 ,1:45 ,1:46 ,1:47 ,1:48 ,1:49 ,1:50 ,
1:51 ,1:52 ,1:53 ,1:54 ,1:55 ,1:56 ,1:57 ,1:58 ,1:59 ,1:60 ,1:61 ,
1:62 ,1:63 ,1:64 ,1:65 ,1:66 ,1:67 ,1:68 ,1:69 ,1:70 ,1:71 ,1:72 ,
1:73 ,1:74 ,1:75 ,1:76 ,1:77 ,1:78 ,1:79 ,1:80 ,1:81 ,1:82 ,1:83 ,
1:84 ,1:85 ,1:86 ,1:87 ,1:88 ,2:4 ,2:5 ,3:3 ,3:4 ,3:5 ,3:6 ,
3:7 ,3:8 ,3:9 ,3:10 ,3:11 ,3:12 ,3:13 ,3:14 ,3:15 ,3:16 ,3:17 ,
3:18 ,3:19 ,3:20 ,3:21 ,3:22 ,3:23 ,3:24 ,3:25 ,3:26 ,3:27 ,3:28 ,
3:29 ,3:30 ,3:31 ,3:32 ,3:33 ,3:34 ,3:35 ,3:36 ,3:37 ,3:38 ,3:39 ,
3:40 ,3:41 ,3:42 ,3:43 ,3:44 ,3:45 ,3:46 ,3:47 ,3:48 ,3:49 ,3:50 ,
3:51 ,3:52 ,3:53 ,3:54 ,3:55 ,3:56 ,3:57 ,3:58 ,3:59 ,3:60 ,3:62 ,
3:63 ,3:64 ,3:65 ,3:66 ,3:67 ,3:68 ,3:69 ,3:70 ,3:71 ,3:72 ,3:73 ,
3:74 ,3:75 ,3:76 ,3:77 ,3:78 ,3:79 ,3:80 ,3:81 ,3:82 ,3:83 ,3:84 ,
3:85 ,3:86 ,3:87 ,3:89 ,3:90 ,3:91 ,3:92 ,3:93 ,3:94 ,3:95 ,3:96 ,
3:97 ,3:98 ,3:99 ,3:100 ,3:101 ,3:102 ,3:103 ,3:104 ,3:105 ,3:106 ,3:107 ,
3:108 ,3:109 ,3:110 ,3:111 ,3:112 ,3:113 ,3:114 ,3:115
$
```

این نتیجه فراخوانی فایل تست سیستم کال `get_callers` می باشد که برای سه سیستم کال ذکر شده در صورت نتیجه بدین صورت می باشد. علاوه بر خواسته سوال که شماره پروسس های مربوطه بوده ، ترتیب صدا زده شدن آن سیستم کال هم چاپ شده تا تحلیل بهتری بتوانیم داشته باشیم.

جواب پرسش در رابطه با تحلیل خروجی `fork` :

```
Kernel: sys_get_callers(1) is called
      now calling get_callers(1)
pid : order that pid called this syscall

1:89 ,2:23
calling get_callers for 3
Kernel: sys_get_callers(3) is called
      now calling get_callers(3)
pid : order that pid called this syscall

1:90 ,2:24
```

با توجه به ترتیب صدا زدن سیستم کال ها در توسط پروسس های یک و دو ، متوجه میشویم که هر دو بعد از صدا زدن `sys_fork` ، سیستم کال `sys_wait` را صدا زده اند. در واقع هر بار که فورک توسط پروسس پدر صدا زده می شود، یک کپی از این پروسس ایجاد شده و پروسس فرزند ایجاد می شود. سپس سیستم کال `sys_wait` صدا زده می شود و پروسس وارد ویتینگ می شود و ادامه فرایند توسط پروسس فرزند ادامه پیدا می کند تا فرایند انشعاب تکمیل گردد.

افزافه کردن سیستم کال : `get_parent_id`

ابتدا باید شماره آن را در `syscall.h` اضافه کنیم

```
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 | #define SYS_get_parent_pid 25
24
```

سپس در `syscall.c` آن را به صورت زیر اضافه میکنیم.

```
96 106 extern int sys_find_largest_prime_factor(void);
107 + extern int sys_get_parent_pid(void);
108 +

133 + [SYS_get_parent_pid] sys_get_parent_pid,
131 134 };
```

افزافه کردن در `user.h` :

```
25 int uptime(void);
26 | int get_parent_pid(void);
```

اضافه کردن در sysproc.c:

```
92
93 int
94 sys_get_parent_pid(void)
95 {
96     return get_parent_pid();
97 }
98
```

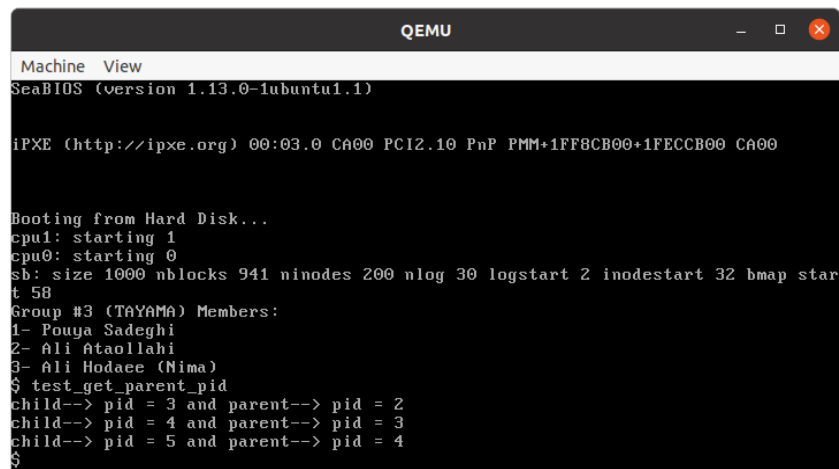
پیاده سازی get_parent_pid در proc.c:

```
536
537 // return parents's pid
538 int
539 get_parent_pid() {
540     struct proc *p = myproc()->parent;
541     return p->pid;
542 }
543
```

در نهایت این سیستم کال را به usys.S هم اضافه کردیم:

```
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(get_parent_pid)
33
```

تست کردن سیستم کال get_parent_pid:



```
QEMU
Machine View
SeaBIOS (version 1.13.0-1ubuntu1.1)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 CA00

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
Group #3 (TAYAMA) Members:
1- Pouya Sadeghi
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ test_get_parent_pid
child--> pid = 3 and parent--> pid = 2
child--> pid = 4 and parent--> pid = 3
child--> pid = 5 and parent--> pid = 4
$
```

برای تست کردن آن برنامه سطح کاربر `test_get_parent_pid.c` را نوشتیم که به صورت زیر است :

```
xv6 > C test_get_parent_pid.c > main(int, char *[])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4
5  int main(int argc, char *argv[])
6  {
7      printf(1, "child--> pid = %d and parent--> pid = %d\n",
8             getpid(), get_parent_pid());
9      int pid, pid1;
10     pid = fork();
11     if (pid < 0)
12     {
13         printf(1, "Error in fork()\n");
14         while (wait() != -1);
15         exit();
16     }
17     else if (pid == 0)
18     {
19         printf(1, "child--> pid = %d and parent--> pid = %d\n",
20                getpid(), get_parent_pid());
21         pid1 = fork();
22         if (pid1 < 0)
23         {
24             printf(1, "Error in fork()\n");
25             while (wait() != -1);
26             exit();
27         }
28         else if (pid1 == 0)
29         {
30             printf(1, "child--> pid = %d and parent--> pid = %d\n",
31                    getpid(), get_parent_pid());
32         }
33     }
34     while (wait() != -1);
35     exit();
36 }
```

سپس برای اضافه کردن آن در Makefile آن را در متغیرهای `UPROGS` و `EXTRA` اضافه کردیم.

```
170
171 UPROGS=\
172     _cat\
173     _echo\
174     _forktest\
175     _grep\
176     _init\
177     _kill\
178     _ln\
179     _ls\
180     _mkdir\
181     _rm\
182     _sh\
183     _stressfs\
184     _usertests\
185     _wc\
186     _zombie\
187     _prime_numbers\
188     _test_getpid\
189     _test_get_parent_pid\
190
```

اضافه کردن سیستم کال : change_file_size :

برای اضافه کردن فایل تست به صورت برنامه سطح کاربر، در Makefile

```
M Makefile
182     _sh\
183     _stressfs\
184     _usertests\
185     _wc\
186     _zombie\
187     _prime_numbers\
188     _test_getpid\
189     _test_find_largest_prime_factor\
190     _test_get_callers\
191     _test_get_parent_pid\
192     _test_change_file_size\
193
```

```
M Makefile
261     mktls.c ulib.c user.h cat.c echo.c forkttest.c grep.c kill
262     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie
263     printf.c umalloc.c prime_numbers.c test_getpid.c test_ge
264     test_find_largest_prime_factor.c test_get_callers.c\
265     test change_file_size.c\
```

اضافه کردن به لیست سیستم کال ها

```
C syscall.c > [?] syscalls
132     [SYS_mkdir]          sys_mkdir,
133     [SYS_close]          sys_close,
134     [SYS_find_largest_prime_factor] sys_find_largest_prime_fact
135     [SYS_get_callers]     sys_get_callers,
136     [SYS_change_file_size] sys_change_file_size,
137     [SYS_get_parent_pid]  sys_get_parent_pid,
138     ];
```

C syscall.c > [🔍] syscalls

```
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_find_largest_prime_factor(void);
107 extern int sys_get_callers(void);
108 extern int sys_change_file_size(void);
109 extern int sys_get_parent_pid(void);
110
111
```

C syscall.h > ...

```
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_find_largest_prime_factor 22
24 #define SYS_get_callers 23
25 #define SYS_change_file_size 24
26 #define SYS_get_parent_pid 25
27
```

اضافه کردن به لیست فانکشن های یوزر

C user.h > [🔍] printf(int, const char *, ...)

```
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int find_largest_prime_factor(void);
27 int get_callers(int);
28 int change_file_size(const char*, int);
29 int get_parent_pid(void);
30
31
32 // ulib.c
```

اضافه کردن فایل تست

```
C test_change_file_size.c > main(int, char * [])
pouya, 16 hours ago | 1 author (pouya)
1 //
2 // Created by pouya on 11/13/22.
3 //
4 #include "types.h"
5 #include "stat.h"
6 #include "user.h"
7
8 int
9 main(int argc, char *argv[]) {
10     if (argc != 3) {
11         printf(1, "Usage: test_change_file_size <file_name>
12         exit();
13     }
14     change_file_size(argv[1], atoi(argv[2]));
15     exit();
16 }
```

اضافه کردن سیستم کال مربوطه

```
C sysfile.c > sys_change_file_size(void)
446
447 int
448 sys_change_file_size(void){
449     char* path;
450     int length;
451     cprintf("kernel: called sys_change_file_size\n");
452     if(argstr(0, &path) < 0 || argint(1, &length) < 0){
453         return -1;
454     }
455     cprintf("        calling change_file_size(%s, %d)\n", path, length);
456     return change_file_size(path, length);
457 }
458
```

اضافه کردن فانکشن مربوط به سیستم کال

```

C file.c > change_file_size(const char *,int)
185 int change_file_size(const char* path, int length) {
186     if (length < 0) {
187         cprintf("change_file_size: length is negative\n");
188         return -1;
189     }
190
191     if (file_inode_exists(path) == 0) {
192         return -1;
193     }
194
195     begin_op();
196     struct inode *ip;
197     ip = namei(path);
198     ilock(ip);
199
200     if (ip->size < length) {
201         // increase file size
202         int old_size = ip->size;
203         char* buf = kalloc();
204         memset(buf, 0, BSIZE);
205         for (int i = old_size; i < length; i += BSIZE) {
206             int n = length - i;
207             n = n > BSIZE ? BSIZE : n;
208             writei(ip, buf, i, n);
209         }
210         ip->size = length;
211         iupdate(ip);
212     }
213     else if (ip->size > length) {
214         // decrease file size
215         ip->size = length;
216         iupdate(ip);
217     }
218     iunlock(ip);
219     end_op();
220     return 0;
221 }

```

تابع کمکی برای آنکه change_size_file به درستی کار کند که در آن چک میشود فایل وجود داشته باشد (و از نوع فایل نیز باشد)


```

58
59 int file_inode_exists(char *path)
60 {
61     struct inode *ip;
62     ip = namei(path);
63     if(ip == 0){
64         cprintf("file error: file not found\n");
65         return 0;
66     }
67     switch (ip->type) {
68         case T_FILE:
69             return 1;
70         case T_DIR:
71             cprintf("file error: file is a directory\n");
72             return 0;
73         case T_DEV:
74             cprintf("file error: file is a device\n");
75             return 0;
76         default:
77             cprintf("file error: file is of unknown type\n");
78             return 0;
79     }
80 }
81

```

اجرای فایل تست

```

pouya@pouya-LPS: ~/OS-LAB/os-lab-project2/xv6
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00

Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 8
Group #3 (TAYAMA) Members:
1- Pouya Sadeghi
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ echo hello-os > sample.txt
$ test_change_file_size sample.txt 5
kernel: called sys_change_file_size
      calling change_file_size(sample.txt, 5)
$ cat sample.txt
hello$ test_change_file_size sample.txt 23
kernel: called sys_change_file_size
      calling change_file_size(sample.txt, 23)
$ cat sample.txt
hello$

```

توجه شود که با اجرای دستور افزایش سایز آن، سایز فایل افزایش یافته اما در ترمینال، null نمایش داده نشده، پس با کمک ls آنرا به نمایش میگذاریم:

```
pouya@pouya-LP5: ~/OS-LAB/os-lab-project2/xv6
$ cat sample.txt
hello$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat       2 3 15652
echo      2 4 14532
forktest  2 5 8976
grep      2 6 18496
init      2 7 15228
kill      2 8 14616
ln        2 9 14516
ls        2 10 17084
mkdir     2 11 14640
rm        2 12 14624
sh        2 13 28680
stressfs  2 14 15548
usertests 2 15 63052
wc        2 16 16076
zombie    2 17 14200
prime_numbers 2 18 16868
test_getpid 2 19 14544
test_find_larg 2 20 15228
test_get_calle 2 21 14800
test_get_paren 2 22 15108
test_change_fi 2 23 14576
console   3 24 0
text.txt  2 25 12
sample.txt 2 26 23
$
```