آزمایشگاه سیستم عامل

گزارش پروژه دوم

- **آرین باستانی** - 810100088
- **مهدیار هرندی** - 810199596
- **محمدرضا نعمتی** - 810100226

- https://github.com/AryanBastani/OS-Lab2
- Last commit hash: e0d73606d453df78415619394ced8f7a2896fd13

1.فایل های تشکیل دهنده ULIB عبارت اند از umalloc.o, printf.o, usys.o, ulib.o.

در ulib.c، در تابع memset از فراخوانی سیستمی stosb استفاده شده که در این تابع حافظه مورد نظر را با مقدار دلخواه پر می کنیم.همچنین در تابع stat از فراخوانی سیستمی open استفاده شده که با آن یک فایل باز شده و سپس با فراخوانی سیستمی fstat اطلاعات فایل باز شده را می گیریم و بعد از آن با فراخوانی سیستمی close آن را می بندیم. در تابع gets نیز از فراخوانی سیستمی read استفاده شده که یک خط از stdin را می خواند.

در umalloc.c، در تابع morecore از فراخوانی سیستمی sbrk استفاده شده که حافظه فیزیکی را تخصیص داده و فضای پردازه را افزایش می دهد.

در printf.c در تابع putc فراخوانی سیستمی write به کار رفته که یک کاراکتر و فایل دسکریپتور آن را گرفته و چاپ می کند.

در usys.S تمام فراخوانی های سیستمی به صورت SYSCALL(name) نوشته شده است که ابتدا سیستم کال به رجیستر eax ریخته می شود و سپس دستور int 64 فراخوانی می شود. این کار زمانی اتفاق می افتد که software interrupt رخ دهد.

2.

- رخ دادن interrupt ها میتواند باعث دسترسی سطح کاربر به هسته شود. Interrupt ها میتوانند نرم افزاری (که trap نامیده میشوند) یا سخت افزار باشند. (مانند کلیک کردن ماوس)
- از انواع trap ها میتوان به exception اشاره کرد که تقسیم بر 0 یا overflow مثال های آن هستند.
- سیستم‌های فایل مجازی یا Pseudo file systems در لینوکس، به جای فایل‌های واقعی، ورودی‌های مجازی را دارند که توسط خود سیستم فایل در لحظه ساخته می‌شوند. این ورودی‌ها در حافظه RAM وجود دارند و بنابراین در ریبوت‌ها حفظ نمی‌شوند. از این سیستم فایل ها میتوان به /dev اشاره کرد که نمایانگر دستگاه ها است. یا /sys که اطلاعاتی در مورد سخت افزار سیستم و درایورهای مرتبط را فراهم میکند. این سیستم فایل ها به دلیل اینکه از برخی از سیستم کال ها استفاده میکنند، به سطح هسته دسترسی پیدا میکنند.

3. خیر.

سیستم عامل xv6 به پراسس های یوزر اجازه فعال کردن trap دیگری را نمیدهد و در این صورت به آنها یک general protection exception میدهد. دلیل این کار این است که سطح دسترسی DPL_USER در سطح یوزر است و در صورتی که یوزر بتواند بقیه trap ها را هم با همین سطح فعال کند، امنیت سیستم ممکن است به خطر بیفتد چون به راحتی یوزر میتواند به سطح هسته دسترسی پیدا کند.

4. در کل در سیستم دو نوع استک user stack و kernel stack داریم که با توجه سطح دسترسی کنونی در لحظه، از یکی از آنها استفاده میشود. همچنین رجیستر ss یا stack segment بلاکی از موری که برای استک استفاده میشود را ذخیره و رجیستر esp یا stack pointer یک پوینتر به محل دقیق stack segment که دقیقا در بالای استک قرار دارد ذخیره میکند. پس در کل این دو رجیستر مشخصات استک را ذخیره میکنند. زمانی

که سطح دسترسی تغییر میکند نیاز است که استک مورد نیاز در آن سطح را در همان stateی که از قبل بود استفاده کنیم. به همین دلیل در زمان عوض شدن سطح دسترسی، آنها را ذخیره میکند ولی زمانی که سطح دسترسی تغییر نمیکند، نیازی نیست هر دفعه آنها را ذخیره کنیم.

5. برای دسترسی به پارامترهای system call ها از سه تابع argint, argstr, argptr استفاده میشود.

- **تابع argint:** از این تابع برای دسترسی به پارامتر های intی که به سیستم کال داده شده است استفاده میشود. در زمان فراخوان یک سیستم کال، پارامتر ها با ترتیب خاصی در استک قرار میگیرند. تابع argint دو ورودی میگیرد. ورودی اول n است که نشان دهنده پارامتر nام است. ورودی دوم پوینتر به یک integer است. در واقع تابع argint مقدار پارامتر nام را در integerی که ورودی دوم به آن اشاره میکند، قرار میدهد. این تابع ابتدا آدرس آرگومان -nام ورودی در حافظه را محاسبه میکند. میدانیم رجیستر esp به تاپ استک اشاره میکند. سپس با فرمول $ptr = esp + 4 + 4*n$ محل پارامتر را پیدا میکند. اولین +4 بخاطر این است که esp به محل آخرین مقدار قرار داده شده در استک اشاره میکند و مقادیر بعدی در خانه های بعدی قرار دارند. همچنین عدد 4 به دلیل این است که خود int در واقع 4 بایت دارد.

- **تابع argstr:** از این تابع برای دسترسی به پارامتر های stringی که به سیستم کال داده شده است استفاده میشود. تابع argstr دو ورودی میگیرد. ورودی اول n است که نشان دهنده پارامتر nام است. ورودی دوم پوینتر به یک character است. در واقع تابع argstr پوینتر به اولین کارکتر استرینگ را در آرگومان دوم قرار میدهد. این تابع از ابتدای رشته شروع میکنه و در صورتی که NULL terminator برسد طول رشته را ریترن میکند و در صورتی که به انتهای حافظه پراسس برسد -1 را به نشانه ارور ریترن میکند.

- **تابع argptr:** این تابع برای دسترسی به پارامترهای پوینتری ای است که به سیستم کال داده شده است. مثلا some_syscall(&my_struct). این تابع ابتدا به کمک تابع argint آدرس پوینتر مورد نظر را دریافت میکند. سپس، آرگومان سوم که سایز پوینتر است را نیز به کمک تابع argint دریافت میکند و بررسی میکند که پوینتر با سایز داده شده در حافظه پردازه قرار داشته باشد. سپس در صورتی که همه چیز اوکی بود، مقدار دوم ورودی را که پوینتر به یک متغیر است را مقداردهی میکند.

نه تنها در argptr بلکه تمامی توابع گفته شده بررسی می شود که محل مموری ای که در حال بررسی است در حافظه پراسس قرار داشته باشد و از آن خارج نشده باشد. چون در غیر این صورت با دادن ورودی هایی با مقادیر زیاد یا کم (منفی) میتوان به خارج فضای مموری ای که برای پراسس اختصاص داده شده است دسترسی داشت و این باگ امنیتی بسیار بزرگی است و همچنین میتواند برای پراسس های دیگر مشکل ایجاد کند.

```
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

این کد مربوط به sys_read که در واقع پیاده سازی سیستم کال read است.

```
read(int fd, void* buffer, int max);
```

میتوان دید در آن ابتدا ارور هندلینگ های مربوط به فایل دسکریپتور داده شده و همچنین argptr و argint انجام می شود و سپس تابع fileread صدا زده می شود. با تابع argptr بررسی می شود که آیا کل فضای آدرس دهی از ابتدای پوینتر به buffer تا انتهای آن به طول max در حافظه پراسس قرار میگیرد یا نه. بدون این ارور هندلینگ میتوانیم به read مقدار max بسیار بزرگ و یا فایل بزرگ بدهیم. در این صورت، هنگام

خواندن از فایل و نوشتن در buffer, سیستم عامل از حافظه این پراسس خارج میشود و در در حافظه پراسس دیگری بنویسد که این باگ و اشکال امنیتی بسیار مهمی است. همچنین در صورتی که max از طول بافر بیشتر باشد، حتی در صورتی که به بیرون از حافظه پراسس نرسیم، همچنان buffer میتواند overflow کند.

بررسی گام های اجرای سیستم کال در سطح کرنل توسط gdb

برنامه get_pid که pid پراسس فعلی را به ما می دهد در makefile به قسمت UPROGS اضافه می کنیم .

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

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

حال یک breakpoint در خط 131 فایل syscall.c قرار می دهیم. پس از اجرای سیستم عامل با اجرای برنامه getpid اجرای برنامه در این خط متوقف میشود و سپس با اجرای دستور bt به این خروجی دست می یابیم:

```
(gdb) b syscall.c:131
Breakpoint 1 at 0x80105160: file syscall.c, line 133.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:135
135         struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:135
#1  0x801061ad in trap (tf=0x8dffffb4) at trap.c:43
#2  0x80105f4f in alltraps () at trapasm.S:20
```

در فایل syscall.h به هر سیستم کال یک عدد اختصاص می یابد و declaration آن ها در user.h و definition آن ها در usys.S قرار دارد. این فایل دستور int 64 را فراخوانی می کند و لیبل vector 64 در vector.S تعریف می شود که پس از push شدن 64 به alltraps در trapasm.S می رویم که در این بخش trapframe ساخته و در استک push می شود که در ادامه trap در فایل trap.c فراخوانی میشود که در اینجا تابع syscall فراخوانی می شود که به تابع مربوط به سیستم کال آن عدد می رویم.

با دستور down یک فریم در استک به پایین می رویم:

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

به دلیل اینکه در درونی ترین فریم هستیم، دستور اجرا نمی شود.

حال بریک پوینت قبلی را حذف و در خط 138 بریک پوینت قرار میدهیم که مقدار eax را به دست آوریم. میتوان مشاهده کرد که مقدار آن با get_pid برابر نمی باشد که این موضوع به دلیل آن است که قبل از get_pid فراخوانی های دیگری انجام شده است که با چند بار continue میتوانیم به آن پی ببریم.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138         if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$1 = 7
```

چون شماره سیستم کال getpid، عدد 11 است، تا جایی که به 11 برسیم continue میکنیم و اجازه میدهیم به پراسس برنامه ما برسد. البته برای پرینت شدن مقدار در صفحه، باید چندین بار دیگه هم continue زده شود تا پرینت انجام شود. پراسس آیدی برنامه ما 3 است که به درستی نمایش داده شده.

```
init: starting sh
Group #3:
1. Mohammad Reza Nemati
2. Aryan Bastani
3. Mahdiar Harandi
$ get_pid
crocess ID: 3
$
```

```
(gdb) print num
$5 = 5
(gdb) c
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$6 = 5
(gdb) c
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$7 = 1
(gdb) c
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$8 = 3
(gdb) c
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$9 = 12
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$10 = 7
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
138       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$11 = 11
(gdb)
```

ارسال آرگومان های فراخوانی های سیستمی

ابتدا باید همانند سیستم کال های دیگر، این سیستم کال را در فایل های مختلف تعریف کنیم. برای اینکار ابتدا به سیستم کال های دیفاین شده در فایل syscall.h این سیستم کال را اضافه کرده و شماره ی بعدی را به آن اختصاص میدهیم.

```
20      #define SYS_link   19
21      #define SYS_mkdir  20
22      #define SYS_close  21
23    + #define SYS_find_digital_root 22
```

حالا در فایل syscall.c این تابع را به لیست تابع ها اضافه میکنیم:

```
103     extern int sys_wait(void);
104     extern int sys_write(void);
105     extern int sys_uptime(void);
106   + extern int sys_find_digital_root(void);
107
108     static int (*syscalls[])(void) = {
109     [SYS_fork]    sys_fork,
```

```
127     [SYS_link]    sys_link,
128     [SYS_mkdir]   sys_mkdir,
129     [SYS_close]   sys_close,
130   + [SYS_find_digital_root] sys_find_digital_root,
131     };
```

و سپس تعریف این تابع(بصورت بدون آرگومان) را در فایل جدید sysothers.c میاوریم (چرا که این تابع به فایل و sysproc ربطی ندارد بنابراین در فایل دیگری آن را مینویسیم).

در این فایل باید یک تابع دارای آرگومان و یک تابع بدون آرگومان تعریف کنیم که جواب تابع آرگومان دار را ریترن میکند.

دقت شود که برای پیدا کردن دیجیتال روت یک عدد، اثبات میشود که اگر عدد به 9 تقسیم پذیر بود، 9 جواب میشود و در غیر اینصورت جواب با باقی مانده ی این عدد به 9 برابر میشود.

```
+ #include "types.h"
+ #include "x86.h"
+ #include "defs.h"
+ #include "date.h"
+ #include "param.h"
+ #include "memlayout.h"
+ #include "mmu.h"
+ #include "proc.h"
+
+ int find_digital_root(int n)
+ {
+     if(n % 9 == 0)
+         return(9);
+
+     return(n % 9);
+ }
+
+ int sys_find_digital_root(void)
+ {
+     return(find_digital_root(myproc()->tf->ebx));
+ }
```

و چون فایل جدید همانند **sysproc** و **sysfile** ایجاد شده است، باید **sysothers.o** را به **OBJS** اضافه کنیم.

```
27          uart.o\
28          vectors.o\
29          vm.o\
30    +     sysothers.o\
```

سپس باید این سیستم کال را به **user.h** و **usys.S** نیز اضافه کنیم.

```
24    int sleep(int);
25    int uptime(void);
26
27  + // new system calls
28  + int find_digital_root(void);
29  +
30    // ulib.c
31    int stat(const char*, struct stat*);
32    char* strcpy(char*, const char*);
```

```
29    SYSCALL(sbrk)
30    SYSCALL(sleep)
31    SYSCALL(uptime)
32  + SYSCALL(find_digital_root)
```

حالا برای تست کردن این سیستم کال، یک برنامه ی سطح کاربر مینویسیم. در این برنامه ابتدا محتوای ebx را با استفاده از سینتکس asm volatile در یک متغیر ذخیره کرده و عدد ورودی را در ebx میریزیم تا تابع آن را دریافت کند. حالا تابع را صدا زده و پس از تمام شدن کار تابع find_digital_root مقدار قبلی درون ebx را به آن برمیگردانیم و سپس digital_root بدست آمده را پرینت میکنیم.

```
+ #include "types.h"
+ #include "user.h"
+ #include "fcntl.h"
+
+ void test_root(int n)
+ {
+     int ebx_content, dig_root;
+
+     asm volatile(
+         "movl %%ebx, %0;"
+         "movl %1, %%ebx;"
+         : "=r"(ebx_content)
+         : "r"(n)
+     );
+
+     dig_root = find_digital_root();
+
+     asm volatile(
+         "movl %0, %%ebx;"
+         : : "r"(ebx_content)
+     );
+
+     printf(1, "%d\n", dig_root);
+ }
+
+ int main(int argc, char* argv[])
+ {
+     test_root(atoi(argv[1]));
+     exit();
+ }
```

از آنجایی که فایل **find_digital_root.c** را به فایلهای برای ران شدن اضافه کردیم، آن را به انتهای **UPROGS** نیز اضافه میکنیم.

```
183          _wc\
184          _zombie\
185          _get_pid\
186   +      _find_digital_root\
187          _get_uncles_count\
```

خروجی های برنامه:

```
init: starting sh
$ find_digital_root 123456
3
$ find_digital_root 465151
4
$ find_digital_root 90125
8
$ _
```

۱.پیاده سازی فراخوانی سیستمی کپی کردن فایل

ابتدا همانند قسمت قبل، این سیستم کال را به فایل های syscall.h ,usys.S , user.h و syscall.c اضافه میکنیم:

```
int get_uncle_count(int);
int get_process_lifetime(int);
+ int copy_file(const char* src, const char* dst);
```

```
extern int sys_get_process_lifetime(void);
+ extern int sys_copy_file(void);
```

```
#define SYS_get_uncle_count 23
#define SYS_get_process_lifetime 24
+ #define SYS_copy_file 25
```

```
[SYS_get_uncle_count] sys_get_uncle_count,
[SYS_get_process_lifetime] sys_get_process_lifetime,
+ [SYS_copy_file] sys_copy_file,
};
```

حالا با استفاده از نوشتن دو تابع is_valid_file و check_exists چک میکنیم که به ترتیب، اولا فایل مبدا وجود داشته باشد و از فرمت فایل باشد و دوما فایل مقصد از قبل وجود نداشته باشد:

```
+ int is_valid_file(char* file)
  {
+   struct inode *ip = namei(file);
+
+   if(ip == 0 || ip->type != T_FILE)
+   {
+     cprintf("your src_file is invalid");
+     return(INVALID);
+   }
+
+   return(VALID);
+ }
```

```
+
+ int check_exists(char* file)
+ {
+   struct inode *ip = namei(file);
+
+   if(ip != 0)
+   {
+     cprintf("dst_file already exists");
+     return(ALREADY_EXISTS);
+   }
+
+   return(NOT_FOUND);
+ }
```

حالا برای پیاده سازی تابع sys_copy_file ، در ابتدا برای اسم های ورودی داده شده ارور های احتمالی را چک میکنیم و سپس با nami , create در وهله ی بعد inode این ورودیها را ساخته و سپس فایلهای مبدا و مقصد را با استفاده از آنها تعریف میکنیم:

```
int
sys_copy_file(void)
{
  char *src, *dst;
  if(argstr(0, &src) < 0 || argstr(1, &dst) < 0)
    return(-1);
  if(!is_valid_file(src) || check_exists(dst))
    return(-1);

  struct file *src_file, *dst_file;
  begin_op();
  src_file = filealloc();
  dst_file = filealloc();

  src_file->ip = namei(src);
  src_file->type = FD_INODE;
  src_file->off = 0;
  src_file->readable = 1;
  src_file->writable = 0;

  dst_file->ip =  create(dst, T_FILE, 0, 0);
  iunlock(dst_file->ip);
  dst_file->type = FD_INODE;
  dst_file->off = 0;
  dst_file->readable = 1;
  dst_file->writable = 1;
```

و سپس با استفاده از یک بافر، تا جایی که فایل مبدا محتوا دارد از آن خوانده و در فایل مبدا میریزیم. ارور های احتمالی هنگام خواندن یا نوشتن را نیز هندل کرده و در نهایت فایل ها را میبندیم.

```
474     char* buf = kalloc();
475     memset(buf, 0, BSIZE);
476     int n;
477     while ((n = fileread(src_file, buf, sizeof(buf))) > 0)
478     {
479         if (filewrite(dst_file, buf, n) != n)
480         {
481             cprintf("copy_file: error in filewrite\n");
482             fileclose(src_file);
483             fileclose(dst_file);
484             end_op();
485             return -1;
486         }
487     }
488     if (n < 0)
489     {
490         cprintf("copy_file: error in fileread\n");
491         fileclose(src_file);
492         fileclose(dst_file);
493         end_op();
494         return -1;
495     }
496
497     fileclose(src_file);
498     fileclose(dst_file);
499     end_op();
500     return(0);
501 }
```

حالا فایل copy_file.c را برای تست اضافه میکنیم که در آن تابع copy_file را صدا میکنیم و البته ارور های احتمالی ورودی را نیز بررسی میکنیم:

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

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        printf(2, "Usage: copy_file source destination\n");
        exit();
    }

    if (copy_file(argv[1], argv[2]) < 0)
        printf(2, "Error: Copy failed\n");
    else
        printf(1, "Copy successful\n");

    exit();
}
```

این فایل را به makefile نیز اضافه میکنیم:

```
_get_uncles_count\
_process_lifetime\
+ _copy_file\
```

خروجی سیستم کال برای فایل README :

```
t 58
init: starting sh
$ copy_file README new_file
Copy successful
$
```

و با زدن دستور ls این فایل جدید نمایش داده خواهد شد:

```
wc             2 16 16060
zombie         2 17 14180
get_pid        2 18 14416
find_digital_r 2 19 14800
get_uncles_cou 2 20 14912
get_process_li 2 21 15008
copy_file      2 22 14596
console        3 23 0
new            2 24 2286
new_file       2 25 2286
$
```

سپس با زدن دستور cat new_file محتوای فایل جدید نمایش داده خواهد شد:

```
Nelson Elhage, Saar Ettinger, Alice Ferrazzi, Nathaniel Filardo, Peter
Froehlich, Yakir Goaron,Shivam Handa, Bryan Henry, Jim Huang, Alexander
Kapshuk, Anders Kaseorg, kehao95, Wolfgang Keller, Eddie Kohler, Austin
Liew, Imbar Marinescu, Yandong Mao, Matan Shabtay, Hitoshi Mitake, Carmi
Merimovich, Mark Morrissey, mtasm, Joel Nider, Greg Price, Ayan Shafqat,
Eldar Sehayek, Yongming Shen, Cam Tenny, tyfkda, Rafael Ubal, Warren
Toomey, Stephen Tu, Pablo Ventura, Xi Wang, Keiichi Watanabe, Nicolas
Wolovick, wxdao, Grant Wu, Jindong Zhang, Icenowy Zheng, and Zou Chang Wei.

The code in the files that constitute xv6 is
Copyright 2006-2018 Frans Kaashoek, Robert Morris, and Russ Cox.

ERROR REPORTS

We don't process error reports (see note on top of this file).

BUILDING AND RUNNING XV6

To build xv6 on an x86 ELF machine (like Linux or FreeBSD), run
"make". On non-x86 or non-ELF machines (like OS X, even on x86), you
will need to install a cross-compiler gcc suite capable of producing
x86 ELF binaries (see https://pdos.csail.mit.edu/6.828/).
Then run "make TOOLPREFIX=i386-jos-elf-". Now install the QEMU PC
simulator and run "make qemu".$
```

۲.پیاده سازی فراخوانی سیستمی تعداد uncleهای پردازه

ابتدا در فایل proc.h متغیری را برای نگهداری تعداد uncleهای پراسس اضافه میکنیم:

```
int uncle_count;
```

در sysproc.c یک تابع جدید برای اجرای سیستم کال اضافه می کنیم:

```
int
sys_get_uncle_count(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;

  return get_uncle_count(pid);
}
```

در user.h، سیستم کال را دیکلر می کنیم:

```
int get_uncle_count(int);
```

پس از آن تعریف این تابع را در usys.S به صورت زیر انجام می دهیم:

SYSCALL(get_uncle_count)

در syscall.h، یک عدد برای سیستم کال تعریف می کنیم:

#define SYS_get_uncle_count 23

سپس دیکلر تابع get_uncle_count را در فایل syscalls.c انجام می دهیم و پوینتر این تابع را به عدد دیفاین شده مپ میکنیم:

extern int sys_get_uncle_count(void);

[SYS_get_uncle_count] sys_get_uncle_count,

در تابع sys_get_uncles_count کافیست ارور احتمالی ورودیها را چک کنیم و get_uncles_count را صدا کنیم:

```
int
sys_get_uncle_count(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;

  return get_uncle_count(pid);
}
```

در get_uncle_count ابتدا تعداد عموها را صفر تعریف کرده و بعد با استفاده از ptable این نود را پیدا کرده و سپس به ازای هر نود دیگری که پدرش با پدربزرگ نود ما یکی بود، تعداد عموها را یکی اضافه میکنیم:

```
int get_uncle_count(int pid) {
  struct proc *curr_p;
  acquire(&ptable.lock);

  for(curr_p = ptable.proc; curr_p < &ptable.proc[NPROC]; curr_p++){
    if(curr_p->pid == pid){

      int uncle_count = 0;
      struct proc *p;
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->parent == (curr_p->parent)->parent && p != (curr_p->parent)) {
          uncle_count++;
        }
      }

      release(&ptable.lock);
      return uncle_count;
    }
  }

  release(&ptable.lock);
  return -1; // If PID not found
}
```

برنامه سطح کاربر را در فایلی جدید به نام get_uncles_count.c می نویسیم که در آن سه فرزند ساخته و

برای آنها یک فرزند دیگر میسازیم و برای یکی از آنها تعداد عموها را پرینت میکنیم:

```c

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

int main(void) {
    int pid, grandchild;
    for (int i = 0; i < 3; i++) {  // Create 3 child processes
        pid = fork();
        if (pid == 0) {  // Child process
            grandchild = fork();
            if (grandchild == 0 && i == 2) {  // Grandchild process
                printf(1,"PID:%d->", getpid());
                printf(1,"%d\n", get_uncle_count(getpid()));
                exit();
            }
            wait();  // Wait for the grandchild to finish
            exit();
        }
    }
    while (wait() != -1);  // Parent waits for all children to finish
    exit();
}
```

با وارد کردن دستور get_uncles_count می توانیم خروجی برنامه را مشاهده کنیم:

```
t 58
init: starting sh
$ get_uncles_count
PID:9->2
$
```

۳.پیاده سازی فراخوانی سیستمی طول عمر پردازه

ابتدا در فایل proc.h, در استراکت proc, متغیر creation_time را اضافه میکنیم:

```
int creation_time;
```

در فایل proc.c در تابع fork زمان تولید پراسس فرزند را به استراکت آن پراسس اضافه میکنیم:

```
np->creation_time = ticks;
```

در فایل proc.c یک تابع جدید برای اجرای سیستم کال اضافه می کنیم که در آن با استفاده از ptable ، پراسس مورد نظر را پیدا کرده و سپس کافیست با ticks، زمان حال را منهای زمان شروع شدن پراسس کنیم:

```
int sys_get_process_lifetime(void) {
  int pid;
  struct proc *p;
  if(argint(0, &pid) < 0)
    return -1;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if(p->pid == pid) {
      release(&ptable.lock);
      return ticks - p->creation_time;
    }
  }
  release(&ptable.lock);
  return -1; // Process not found
}
```

در user.h، سیستم کال را دیکلر می کنیم:

```
int get_process_lifetime(int);
```

پس از آن تعریف این تابع را در usys.S به صورت زیر انجام می دهیم:

```
SYSCALL(get_process_lifetime)
```

در syscall.h، یک عدد برای سیستم کال تعریف می کنیم:

#define SYS_get_process_lifetime 24

سپس دیکلر تابع get_process_lifetime را در فایل syscalls.c انجام می دهیم و پوینتر این تابع را به عدد دیفاین شده مپ میکنیم:

extern int sys_get_process_lifetime(void);

[SYS_get_process_lifetime] sys_get_process_lifetime,

برنامه سطح کاربر را در فایلی جدید به نام process_lifetime.c می نویسیم:

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

int main(void) {
    int pid = fork();

    if (pid < 0) {
        printf(1, "Fork failed\n");
        exit();
    }

    if (pid == 0) {
        // Child process
        printf(1, "Child process created\n");
        sleep(100); // Sleep for 10 seconds
        int child_lifetime = get_process_lifetime(getpid());
        printf(1, "Child process lifetime: %d ticks\n", child_lifetime);
        exit();
    } else {
        // Parent process
        wait(); // Wait for the child to finish
        int parent_lifetime = get_process_lifetime(getpid());
        printf(1, "Parent process lifetime: %d ticks\n", parent_lifetime);
        printf(1, "1 tick = 0.1 seconds\n");
        exit();
    }
}
```

با وارد کردن دستور process_lifetime می توانیم خروجی برنامه را مشاهده کنیم:

```
init: starting sh
$ process_lifetime
Child process created
Child process lifetime: 100 ticks
Parent process lifetime: 101 ticks
1 tick = 0.1 seconds
$ _
```