گزارش‌کار پروژه دوم آزمایشگاه سیستم عامل

گروه ۲۷

ارشیا عطایی نائینی | ۸۱۰۱۰۰۲۵۲

فاطمه کرمی محمدی | ۸۱۰۱۰۰۲۵۶

امیر پارسا موبد | ۸۱۰۱۰۰۲۷۱

آدرس مخزن گیتهاب: https://github.com/phatemek/OS-lab-project-2

شناسه آخرین کامیت: e6c6aea86646ac7abc0d1f15f47118d6c77bf105

سوالات تشریحی:

۱. کتابخانه‌های (قاعدتاً سطح کاربر، منظور فایل‌های تشکیل‌دهنده متغیر ULIB در Makefile است) استفاده شده در xv۶ را از منظر استفاده از فراخوانی‌های سیستمی و علت این استفاده بررسی نمایید.

فایل‌های مورد بررسی: ulib.c, usys.s, printf.c, umalloc.c

- ulib.c: توابع این فایل برای کار با آرایه (*char) ها ساخته شده‌اند. در این فایل در توابع gets, stat, memset از فراخوانی‌های سیستمی استفاده شده است که آنها را بررسی می‌کنیم:
gets: در این تابع برای خواندن محتوای stdin از سیستم کال read استفاده شده‌است.
stat: در این تابع ابتدا با استفاده از سیستم کال open، file descriptor فایل مورد نظر گرفته می‌شود، سپس با استفاده از سیستم کال fstat داده‌های مورد نیاز برای ساخت استراکچر stat از فایل خوانده می‌شود و در نهایت با استفاده از سیستم کال close فایل مورد نظر بسته می‌شود.
memset: برای ذخیره داده رجیستر a در بخشی از حافظه که آدرس آن در رجیستر d مشخص شده است از سیستم کال stosb استفاده می‌شود.
- usys.s: در ابتدای این فایل یک ماکرو تعریف شده‌است که در ادامه به ازای هر یک از ۲۱ سیستم کال موجود در حالت پایه xv6 از این ماکرو استفاده می‌شود. در این ماکرو دستورات لازم برای یک سیستم کال مشخص شده است.
- printf.c: در این فایل توابع printf و printint تعریف شده‌اند که این دو تابع putc را صدا می‌زنند. putc برای چاپ کردن یک کاراکتر در fd مورد نظر از سیستم کال write استفاده می‌کند.
- umalloc.c: در این فایل تابع malloc تعریف شده‌است که این تابع برای تخصیص حافظه بیشتر به یک پردازه از سیستم کال sbrk استفاده می‌کند که این سیستم کال اندازه data segment را تغییر می‌دهد.

۲. انواع روش‌های دسترسی سطح کاربر به هسته در لینوکس را به اختصار توضیح دهید.

دسترسی سطح کاربر به هسته با استفاده از interrupt ها انجام می‌شود. Interrupt ها دو نوع نرم افزاری (trap) و سخت افزاری دارند که در زیر توضیح داده می‌شود:

- Interrupt های سخت افزاری: از طریق سخت افزارها مانند I/O اتفاق می‌افتند. برای مثال حرکت موس یا فشردن یک کلید کیبورد از این نوع interrupt ها هستند. این interrupt ها به صورت asynchronous اجرا می‌شوند.
- Interrupt های نرم افزاری: به صورت synchronous اجرا می‌شوند و انواع system call و exeption و signal دارند:

  System call: همان فراخوانی‌های سیستمی هستند.

  Exeption: درصورت بروز استثناهایی مانند تقسیم بر ۰ یا دسترسی غیرمجاز به حافظه دسترسی به مود kernel رفته، exeption را handle می‌کند و به مود user برمی‌گردد.

  Signal: این نوع interrupt در لینوکس انواع مختلفی دارد که پرکاربردترین آنها sigint, sigkill, sigterm هستند.

همچنین در لینوکس برای دسترسی به api ساختارهای داده هسته از تعدادی pseudo-file-system مانند /proc, /dev, /sys استفاده می‌شود که استفاده از آنها نیازمند دسترسی به هسته است. نام گذاری آنها به این دلیل است که آنها محتویات ساختارهای داده را به صورتی که انگار روی یک فایل ذخیره شده‌اند برای اپلیکیشن‌ها می‌فرستند.

۳. آیا همه تله‌ها را نمی‌توان با سطح دسترسی USER_DPL فعال نمود؟ چرا؟

خیر. سطح دسترسی USER_DPL همان سطح دسترسی کاربر است. اگر کاربر امکان فعال کردن هرگونه تله ای را داشته باشد امنیت سیستم به خطر می‌افتد چرا که ممکن است در برنامه کاربر مشکلی وجود داشته باشد یا به هر دلیلی برنامه او باعث ایجاد مشکل در سیستم شود. برای جلوگیری از این اتفاق سیستم عامل xv6 پس از تغییر به مود kernel آرگومان های مربوط را چک می‌کند و درصورت فعال سازی تله‌های خاص با سطح دسترسی USER_DPL به کاربر protection exeption خواهد داد.

۴. در صورت تغییر دسترسی، ss و esp روی پشته Push می‌شود. در غیر این‌صورت Push نمی‌شود. چرا؟

دو پشته یکی برای کاربر و یکی برای هسته داریم. وقتی که یک تله فعال می‌شود و تغییر در سطح دسترسی داریم برای اینکه بتوانیم به پشته دیگر دسترسی داشته باشیم باید ss و esp که به پشته فعلی اشاره دارند را ذخیره کنیم. پس از اتمام رسیدگی به تله، مقادیر قدیمی این دو رجیستر بازیابی می‌شوند و برنامه از همان جای قبلی ادامه خواهد یافت. اما در صورتی که تغییری در سطح دسترسی نداشته باشیم، از آنجا که همچنان با همان پشته قبلی کار داریم، نیازی به ذخیره این دو رجیستر نخواهیم داشت.

۵. در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی مختصر توضیح دهید. چرا در argptr بازه آدرس‌ها بررسی می‌گردند؟ تجاوز از بازه معتبر، چه مشکل امنیتی‌ای 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.

در مجموع چهارتابع برای دسترسی به پارامتر های فراخوانی سیستمی داریم که در ادامه نامبرده و به اختصار توضیح داده شده اند. هرکدام از این توابع در صورت ارور 1- بر می‌گردانند. یکی از این حالات تجاوز از بازه معتبر پردازه است. زیرا در صورت تجاوز از بازه معتبر مقدار نادرست و نامرتبط از پردازه برگردانده می‌شود که می‌تواند اجرای ادامه برنامه را دچار مشکل کند. به طور مثال در فراخوانی sys_read() تابع fileread(struct file *f, char *addr, int n) فراخوانده می‌شود که در صورت عدم چک کردن این بازه‌ها می‌تواند آرگومان های نامعتبری به این تابع داده شود که فایلی نامعتبر باشد یا آنچه که می‌خواستیم نباشد. در ضمن این دسترسی به فایل های ناخواسته می‌تواند شکاف امنیتی را در پیش داشته باشد.

این چهار تابع عبارتند از :

- int argint(int n, int *ip) : nامین آرگیومنت ۳۲بیتی سیستم کال را بر می‌گرداند.
- int argptr(int n, char **pp, int size) : nامین کلمه به اندازه size را به صورت یک اشاره‌گر به بلوکی از حافظه به تعداد size بایت بر می‌گرداند.
- int argstr(int n, char **pp) : اشاره‌گری به رشته ای با شروع از nامین بایت که باید به null ختم شود.
- int argfd(int n, int *pfd, struct file **pf) : nامین ۳۲بیت آرایه را به عنوان یک file descriptor بر می‌گرداند.

بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

کاربرد دستور bt(backtrace): این دستور برای نمایش call stack برنامه استفاده می‌شود. وقتی یک تابع در برنامه صدا زده می‌شود متغیرهای محلی و آدرس بازگشت و ... خود را در یک stack frame ذخیره می‌کند. دستور bt، stack frame ها را به ترتیب از درونی ترین frame نمایش می‌دهد.

خروجی دستور bt پس از گذاشتن break point در خط ۱۴۲ فایل syscall.c و اجرای برنامه نوشته شده به این شکل است:

```
Thread 2 hit Breakpoint 1, syscall () at syscall.c:146
146       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) bt
#0  syscall () at syscall.c:146
#1  0x80104055 in mycpu () at proc.c:45
#2  0x8dffefb4 in ?? ()
#3  0x801065e1 in trap (tf=0x1010101) at trap.c:43
#4  0x80104da5 in popcli () at spinlock.c:117
#5  popcli () at spinlock.c:117
#6  0x01010101 in ?? ()
#7  0x00000000 in ?? ()
(gdb)
```

برای تعریف و اجرای یک سیستم کالا ابتدا یک عدد به آن سیستم کالا اختصاص می‌یابد، declaration آن سیستم کالا در فایل user.h نوشته شده و در usys.s، definition آن سیستم کالا به زبان اسمبلی تعریف شده. در این فایل شماره system call در رجیستر eax نوشته می‌شود، دستور int 64 فراخوانی می‌شود و وارد بخش تعریف vector64 در فایل vectors.s می‌شود. مقدار 64 در استک push می‌شود و سپس بخش alltraps در فایل trapasm.s، trap frame مربوطه را می‌سازد و آن را در استک push می‌کند و تابع trap را صدا می‌زند. این تابع در مواجهه با عدد 64 متوجه وجود سیستم کالا شده و trap frame که در استک push شده بود را به عنوان trap frame پردازه فعلی قرار می‌دهد و تابع syscall را صدا می‌کند. این تابع، تابع مربوط به آن سیستم کالا را صدا می‌زند و سپس خروجی آن را در trap frame پردازه فعلی ذخیره می‌کند.

خروجی دستور bt که در تصویر آمده است همان داده‌های درون call stack را نمایش می‌دهد که توضیح چگونگی آنها در بالا آمده است.

کارکرد دستور down: این دستور به اندازه n فریم به پایین استک می‌رود. در این قسمت n برابر ۱ بوده و ما در داخلی ترین فریم استک قرار داریم بنابراین با صدا کردن دستور down به این ارور می‌خوریم:

```
#7  0x00000000 in ?? ()
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb)
```

اما اگر دستور up صدا زده شود و یک فریم به بالای استک برویم خروجی به این شکل خواهد بود:

```
(gdb) up
#1  0x80105f3d in trap (tf=0x8dffefb4) at trap.c:43
```

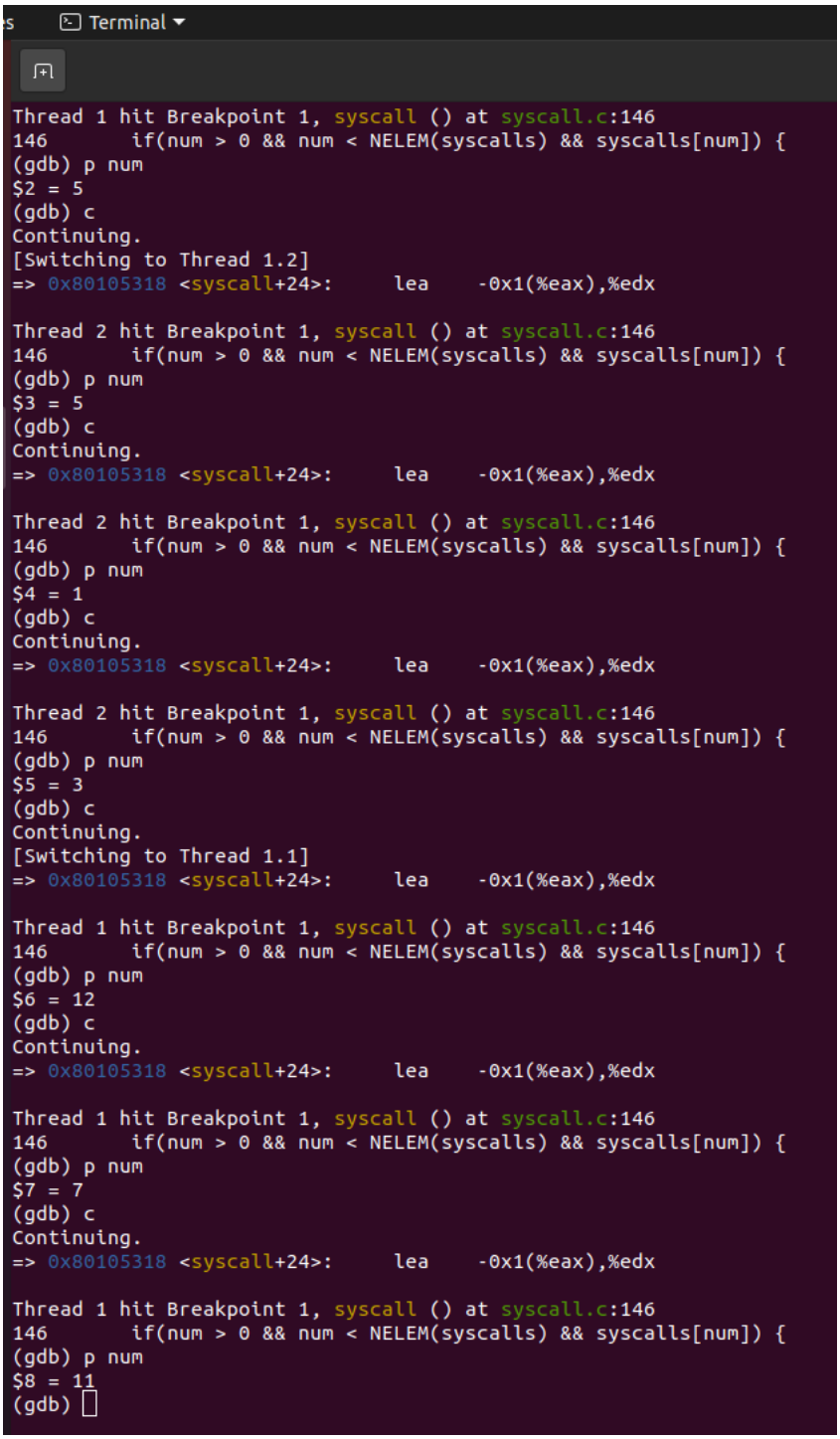
خروجی نشان دهنده این است که به تابع trap رفتیم. با چاپ کردن مقدار رجیستر eax خروجی به این شکل خواهد بود:

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

مقدار این رجیستر برابر ۵ است اما شماره فراخوانی سیستمی getpid() برابر با ۱۱ بود. دلیل این تفاوت این است که پیش از رسیدن به فراخوانی سیستمی getpid() فراخوانی‌های سیستمی دیگری اجرا می‌شوند (مانند سیستم کال read برای خواندن stdin). با چند بار اجرای دستور continue و خواندن دوباره مقدار رجیستر eax خروجی به این صورت خواهد بود:

- سیستم کال شماره ۵ (read) اجرا شده تا دستور وارد شده را از stdin بخواند.
- سیستم کال شماره ۱ (fork) اجرا شده تا یک process جدید برای اجرای برنامه سطح کاربر ایجاد کند.
- سیستم کال شماره ۱۲ (sbrk) اجرا شده تا به process ایجاد شده حافظه اختصاص دهد.
- سیستم کال شماره ۷ (exec) اجرا شده تا برنامه pid را در process ایجاد شده اجرا کند.
- سیستم کال شماره ۳ (wait) اجرا شده تا process پدر برای پایان یافتن اجرای process فرزند (pid) منتظر بماند.
- سیستم کال شماره ۱۱ (getpid) همان سیستم کال مربوط به برنامه سطح کاربر است.

پس از انجام این مراحل مقدار eax برابر با ۱۱ خواهد بود (شناسه سیستم کال pid). پس از این سیستم کال های دیگری برای چاپ خروجی برنامه صدا می‌شوند.



خروجی برنامه به این شکل خواهد بود:

```
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ pid
Process ID: 3
```

ارسال آرگومانهای فراخوانی‌های سیستمی

اضافه کردن سیستم کال find_digit_root:

در xv6 در حالت پایه ۲۱ سیستم کال وجود دارد که اعداد ۱ تا ۲۱ به هرکدام از آنها اختصاص داده شده‌است. پس برای ساخت سیستم کال جدید باید در فایل syscall.h عدد ۲۲ به آن اختصاص داده شود. پس از این در فایل‌های syscall.c, usys.s, user.h, defs.h، declaration این تابع اضافه می‌شود. definition این تابع در فایل proc.c نوشته می‌شود که به این شکل خواهد بود:

```
int
find_digit_root(int n){
  if (n <= 0) return -1;
  int x = 0;
  while (n > 9){
    x = n;
    n = 0;
    while (x){
      n += x % 10;
      x /= 10;
    }
  }
  return n;
}
```

در ادامه نیز در تابع sysproc.c بدنه فراخوانی سیستمی آن را به صورت زیر تعریف می‌کنیم:

```
int
sys_find_digit_root(void){
  cprintf("kernel is running sys_find_digit_root\n");
  return find_digit_root(myproc()->tf->ebx);
}
```

که فراخوانی این تابع با استفاده رجیستر ebx به عنوان آرگومان ورودی انجام می‌شود.

برنامه find_digit_root.c یک برنامه سطح کاربر برای تست آن است که ابتدا شرط آرگومان‌ها را چک می‌کند و سپس مقدار رجیستر ebx را در متغیری ذخیره می‌کند و مقدار ورودی آرگومان مورد نظر ما را به ورودی ebx می‌دهد و بعد از فراخوانی تابع find_digit_root مقدار قبلی ebx را بازنویسی می‌کند:

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

// simple program to test find_largest_prime_factor() system call
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf(2, "Error in syntax\n");
        exit();
    }
    int n = atoi(argv[1]), prev_ebx;
    asm volatile(
            "movl %%ebx, %0;"
            "movl %1, %%ebx;"
            : "=r" (prev_ebx)
            : "r"(n)
            );
    int result = find_digit_root();
    asm volatile(
            "movl %0, %%ebx;"
            : : "r"(prev_ebx)
            );
    if (result == -1) {
        write(1, "find_digit_root() failed!\n", 26);
        exit();
    }
    printf(1, "%d\n", result);
    exit();
}
```

برای استفاده از این دستور سطح کاربری آن را در Makefile، در قسمت Extra و UPROGS اضافه می‌کنیم.

نمونه‌ای از اجرای این برنامه:

```
init: starting sh
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ find_digit_root 32493
kernel is running sys_find_digit_root
3
$
```

پیاده‌سازی فراخوانی‌های سیستمی

اضافه کردن سیستم کال copy_file:

برای اضافه کردن این سیستم کال ابتدا declaration در سطح کاربری و کرنل به ترتیب در user.h و syscall.c انجام می‌دهیم و به آن شناسه ۲۳ اختصاص می‌دهیم و در آخر در usys.S SYSCALL(copy_file)را اضافه می‌کنیم.

بدنه تابع sys_copy_file را در sysfile.c می‌نویسیم و با کمک argstr آرگومان را می‌خوانیم:

```
int
sys_copy_file(void)
{
  cprintf("kernel is running sys_copy_file\n");
  char *src, *dest;
  if (argstr(0, &src) < 0 || argstr(1, &dest) < 0){
    return -1;
  }
  struct inode *source, *destination;
  begin_op();
  destination = create(dest, T_FILE, 0, 0);
  source = namei(src);
  if (source == 0){
    end_op();
    return -1;
  }
  if (destination){
    ilock(source);
    char buf[512];
    int off = 0;
    while (1){
      int diff = source->size - off;
      if (diff <= 0) break;
      int read_size = (diff > 512)  ? 512:diff;
      if (readi(source, buf, off, read_size) < 0) return -1;
      if (writei(destination, buf, off, read_size) < 0) return -1;
      off += read_size;
    }
    destination->size = source->size;
    iunlock(source);
    iunlock(destination);
    end_op();
  }else{
    end_op();
    return -1;
  }
  return 0;
}
```

حالا برای برنامه سطح کاربر آن یعنی copy_file.c یک برنامه می‌نویسیم که در Makefile در Extra و UPROGS اضافه می‌کنیم.

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

int
main(int argc, char *argv[]) {
    if (argc != 3) {
        printf(1, "incorrect input arguments\n");
        exit();
    }
    if (copy_file(argv[1], argv[2]) < 0){
        printf(1, "incorrect file name\n");
    }
    exit();
}
```

یک نمونه از خروجی:

```
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ echo salam >> arshia
$ cat arshia
salam
$ copy_file arshia mobed
kernel is running sys_copy_file
$ cat mobed
salam
$ _
```

اضافه کردن سیستم کال get_uncle_count:

برای اضافه کردن این سیستم کال ابتدا declaration در سطح کاربری و کرنل به ترتیب در user.h و syscall.c انجام می‌دهیم و به آن شناسه ۲۴ اختصاص می‌دهیم و در آخر در usys.S SYSCALL(get_uncle_count) را اضافه می‌کنیم.

پیاده سازی تابع در proc.c:

```
int
get_uncle_count(int pid){
  int koj = -1;
  for (int i = 0; i < NPROC; i++){
    if (ptable.proc[i].pid == pid){
      koj = i;
    }
  }
  if (koj < 0) return -1;
  int cnt = -1;
  for (int i = 0; i < NPROC; i++){
    if (ptable.proc[i].parent == ptable.proc[koj].parent->parent && ptable.proc[i].state != UNUSED) cnt++;
  }
  return cnt;
}
```

پیاده سازی در sysproc.c که با استفاده از argint آرگومان گرفته می‌شود:

```
int
sys_get_uncle_count(void){
  int pid;
  cprintf("kernel is running sys_get_uncle_count\n");
  argint(0, &pid);
  return get_uncle_count(pid);
}
```

در آخر برنامه سطح کاربر get_uncle_cout را مانند قبلی‌ها به Makefile اضافه می‌کنیم:

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

void child4(){
    int forkpid = fork();
    if (forkpid > 0)
        wait();
    else if (forkpid == 0)
        printf(1, "number of process %d uncles: %d\n", getpid(), get_uncle_count(getpid()));
    else
        printf(2, "child4 fork failed.\n");
    exit();
}

void child3(){
    int forkpid3 = fork();
    if (forkpid3 < 0){
        printf(2, "Third child fork failed.\n");
        exit();
    }
    else if (forkpid3 == 0)
        child4();
}

void child2(){
    int forkpid2 = fork();
    if (forkpid2 < 0){
        printf(2, "Second child fork failed\n");
        exit();
    }
    if (forkpid2 == 0)
        sleep(50);
    else
        child3();
    while (wait() != -1);
}

int main(int argc, char *argv[])
{
    int forkpid1 = fork();
    if (forkpid1 < 0){
        printf(2, "First child fork failed\n");
        exit();
    }
    else if (forkpid1 == 0){
        sleep(50);
    }
    else{
        child2();
    }
    exit();
}
```

نمونه‌ای از این برنامه:

```
init: starting sh
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ get_uncle_count
kernel is running sys_get_uncle_count
number of process 7 uncles: 2
$ _
```

اضافه کردن سیستم کال get_process_lifetime:

برای اضافه کردن این سیستم کال ابتدا declaration در سطح کاربری و کرنل به ترتیب در user.h و syscall.c انجام می‌دهیم و به آن شناسه ۲۵ اختصاص می‌دهیم و در آخر در usys.S SYSCALL(get_process_lifetime) را اضافه می‌کنیم.

پیاده سازی تابع در proc.c:

```
int
get_process_lifetime(int pid){
  int koj = -1;
  for (int i = 0; i < NPROC; i++){
    if (ptable.proc[i].pid == pid){
      koj = i;
      break;
    }
  }
  return ticks - ptable.proc[koj].st;
}
```

پیاده سازی آن در sysproc.c که به کمک argint آرگومان آن گرفته می‌شود:

```
int
sys_get_process_lifetime(void){
  cprintf("kernel is running sys_get_process_lifetime\n");
  int pid;
  argint(0, &pid);
  return get_process_lifetime(pid);
}
```

مانند سیستم کالهای قبلی، برنامه سطح کاربر get_process_lifetime را به Makefile اضافه می‌کنیم:

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


int main(int argc, char *argv[])
{
    int forkpid = fork();
    if (forkpid == 0){
        sleep(1000);
        printf(1, "child lifetime:%d\n", get_process_lifetime(getpid()) / 100);
    }else{
        wait();
        sleep(200);
        printf(1, "parent lifetime:%d\n", get_process_lifetime(getpid()) / 100);
    }
    exit();
}
```

نمونه‌ای از این برنامه سطح کاربری:

```
Group #27:
1.Arshia Ataei
2.Fateme karami
3.AmirParsa Mobed
$ get_process_lifetime
kernel is running sys_get_process_lifetime
child lifetime:10
kernel is running sys_get_process_lifetime
parent lifetime:12
$ _
```