گزارش پروژهٔ دوم آزمایشگاه سیستم عامل

به تدریس دکتر کارگهی

محمد امین یوسفی

مبینا مهرآذر

متین نبی‌زاده

پاییز 1402

اطلاعات مربوط به گیت‌هاب، در آخرین صفحهٔ گزارش، درج شده است.

مقدمه

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

در makefile سیستم‌عامل xv6، می‌توان متغیر ULIB را یافت به شکل زیر:

```
ULIB = ulib.o usys.o printf.o umalloc.o
```

همان‌طور که واضح است، این متغیر شامل چهار آبجکت فایل است که در ادامه، هر کدام به طور مختصر، بررسی می‌شوند.

- ulib.o

این کتابخانه، شامل توابع پایهٔ c است که توانایی اجرا شدن عملکردهای پایه را به برنامه‌های سطح کاربر، می‌دهد. از جملهٔ توابع موجود در این کتابخانه می‌توان به موارد زیر اشاره کرد:

strcpy	یک رشته را کپی می‌کند.
strcmp	دو رشته را مقایسه می‌کند و اگر یکسان بودند، صفر برمی‌گرداند.
strlen	طول رشته را بازمی‌گرداند.
memset	مقدار اشاره‌گر به حافظه را تغییر می‌دهد.
strchr	اولین دفعهٔ حضور یک حرف در یک رشته را می‌یابد.
gets	از ورودی می‌خواند و در بافر حافظه، ذخیره می‌کند.
stat	اطلاعات مربوط به یک فایل در مسیر مشخص را در یک ساختار مشخص، ذخیره می‌کند.

atoi	یک رشته که ممکن است شامل عددی صحیح باشد را تبدیل به عددی صحیح می‌کند.
memmove	مقداری مشخص از یک بخش از حافظه را به بخش دیگری از حافظه، منتقل می‌کند.

این توابع کمکی هستند و در user.h دیکلر شده اند. فراخوانی سیستمی در دو تابع gets و stat استفاده شده‌است.
تابع gets که یک خط از stdin میخواند از تابع سیستمی read استفاده کرده‌است.
تابع stat با open فایل باز می‌کند، با fstat فیلدهای struct fstat یا همان metadataی فایل مربوطه را می‌گیرد. با close هم فایل مربوطه بسته می‌شود.

- usys.s

این فایل، در اصل حاوی کد اسمبلی‌ای است که یک لایهٔ ارتباطی میان کد c سطح کاربر و فراخوانی‌های سیستمی‌ای که در هسته پیاده‌سازی شده‌اند، فراهم می‌سازد. در این کتابخانه، یک ماکرو[1] به نام SYSCALL تعریف شده است که کد اسمبلی متناظر با هر فراخوانی سیستمی را تولید می‌کند؛ یعنی به آن، ثبّات[2] مشخصی را اختصاص می‌دهد و یک اخلال[3] نرم‌افزاری به هسته ارسال می‌کند که هسته نیز از فراخوانی سیستمی ایجاد شده، مطلع شود.
در ابتدای این فایل ماکرو ای که به ازای هر سیستم کال با جایگزینی اسم سیستم کال در بخش name این ماکرو، اینستراکشن لازم برای یک سیستم کال تولید میشود.

- printf.c

همان‌طور که از نام این فایل برمی‌آید، این کتابخانه شامل توابعی است که چاپ کردن خروجی‌ای مشخص را ممکن می‌سازند. در کتابخانه مربوطه تابع printf تعریف شده است که در user.h دیکلر شده است.
همچنین، توابع کمکی استاتیک putc و printinit که تابع putc در انتهای printinit و printf صدا زده شده است.
بیان این مطلب نیز خالی از لطف نیست که در تابع putc، کد زیر را داریم:

```
write(fd, &c, 1);
```

که کاراکتر ورودی را با تابع سیستمی write با توجه به فایل دیسکریپتور ورودی پرینت می‌کند.

[1] Macro
[2] Register
[3] Interrupt

- umalloc.c

به طور خلاصه، این فایل، شامل ساز و کاری برای تخصیص حافظه است که تعریف تابع malloc را دارد و این تابع در user.h دیکلر شده است. این کتابخانهٔ تخصیص‌دهنده، یک لیست پیوندی از فضاهای خالی حافظه نگهداری می‌کند که در مواقعی که نیاز به تخصیص حافظه است، این لیست پیوندی را بپیماید و فضای مناسب مورد نظر را بیابد. اگر که هیچ فضایی یافت نشد نیز با استفاده از تابع morecore، درخواست حافظهٔ اضافی می‌کند. همچنین، فراخوانی سیستمی sbrk نیز فضای پردازه را در این تابع، افزایش می‌دهد.

2) دقت شود فراخوانی‌های سیستمی تنها روش دسترسی سطح کاربر به هسته نیست. انواع این روشها را در لینوکس به اختصار توضیح دهید.

دسترسی به سطح هسته با اخلال‌هایی که هم جنس نرم‌افزاری (یا همان تله[4]) و هم سخت‌افزاری دارند، صورت می‌گیرد.

نوع سخت‌افزاری از طریق سخت‌افزار رخ می‌دهد؛ همانند یک عملیات I/O. این نوع از اخلال‌ها به شکل غیرهماهنگ[5] هم قابلیت اجرا دارند. نوع نرم‌افزاری نیز توسط برنامهٔ نرم‌افزاری و به شکل هماهنگ[6] پدیدار می‌شوند. از جملهٔ این اخلال‌ها می‌توان به فراخوانی سیستمی‌ای (که در کد برنامه فراخوانی می‌شود)، استثناء (که اگر برنامه دسترسی غیرمجاز به حافظه داشته باشد یا در کد برنامه تقسیم به صفر پدیدار شود رخ می‌دهد)، سیگنال (مانند SIGINT و SIGTERM و SIGKILL) اشاره کرد.
از طرفی دسترسی به هسته در Pseudo-File-Systemهایی نظیر proc/ و یا sys/ یا dev/ نیز انجام می‌شود، بدین شرح که اینها، یک لایه برای داده‌ساختارهای هسته فراهم می‌کنند که برای استفاده از این فایل سیستم ها دسترسی به هسته نیاز می‌شود.
در اصل اینها توسط دیسک بازگردانده نمی‌شوند بلکه محتویات ساختمان داده‌های هسته را به گونه‌ای برای اپلیکیشن‌ها ارسال می‌کنند که گویا روی فایل ذخیره شده‌اند.

---

[4] Trap
[5] Asynchronous
[6] Synchronous

ساز و کار اجرای فراخوانی سیستمی در xv6

- **بخش سخت‌افزاری و اسمبلی**

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

درست است که استفاده کردن از سطح دسترسی توصیف‌کننده (DPL[7])، ساز و کاری است که به کمک آن، می‌توان دسترسی به گیت‌های به خصوصی از IDT[8] را کنترل کرد، اما علی‌رغم این موضوع، تله‌هایی نیز هستند که به سطح دسترسی عمیق‌تر و مخصوص‌تر، نیاز دارند و با سطح دسترسی DPL_USER، نمی‌توان آن‌ها را فعال کرد. برخی از همین تله‌ها، تنها می‌توانند تحت نظارت سیستم‌عامل و در سطوح دسترسی بالاتر (مانند حلقهٔ 0 که مخصوص هسته است)، فعال شوند. نتیجتا، تلاش برای ایجاد این تله‌ها از سطح دسترسی کاربر، منجر به خطاهایی از جنس نقض شرایط دسترسی می‌شود و امنیت سیستم عامل به خطر می‌افتد. از جملهٔ این تله‌ها، می‌توان به استثناء MCE[9] اشاره کرد. این استثناء، به طور عادی، از سطح کاربر قابل بررسی یا دسترسی نیست زیرا از وجود یک خطا در خود پردازندهٔ مرکزی، خبر می‌دهد. یا برای مثال، استثناء های دوگانه[10] / سه‌گانه[11] (یعنی آن که یک استثناء، پیش از برطرف شدن، باعث به وجود آمدن استثناء(های) دیگری شود) که مولد شرایط بحرانی‌ای هستند، نمی‌توانند ضمن برقرار بودن دسترسی سطح کاربر، رفع و رجوع شوند. ضمن آنکه می‌دانیم تصمیم اینکه ایجاد یا به طور کلی، رفع و رجوع کردن چه تله‌هایی برای چه سطحی از دسترسی مجاز باشند، به طراحی سیستم‌عامل باز می‌گردد و همان طور که گفته شد، در سیستم‌عاملی مانند xv6، درب دسترسی بسیاری از تله‌ها به روی کاربر، گشاده است اما آن دسته از تله‌هایی که به خطاهای بحرانی (یا بسا در سطح سخت‌افزار) مربوط هستند، از حیطهٔ دسترسی کاربر، خارج‌اند.

---

[7] Descriptor Privilege Level
[8] Interrupt Descriptor Table
[9] Machine Check Exception
[10] Double Fault Exception
[11] Triple Fault Exception

4) در صورت تغییر سطح دسترسی، ss و esp روی پشته Push می‌شود اما در غیر این صورت، Push نمی‌شود. چرا؟

همان طور که در قسمت‌های قبل نیز به این موضوع اشاره شده است، با ایجاد یک تله و تغییر سطح دسترسی، از پشتهٔ هسته برای دسترسی به کد و ساختار داده های مورد نیاز هسته استفاده می‌شود. ss و esp نیز ذخیره می‌شوند تا پس از اتمام دسترسی هسته و تغییر دسترسی دوباره به سطح دسترسی اولیه، این مقادیر در پشتهٔ کاربر از دست نروند. اگر تغییر سطح دسترسی‌ای انجام نشود، این مقادیر کماکان موجود می‌مانند و نیازی به ذخیرهٔ مجدد آنها، نخواهد بود.

- **بخش سطح بالا و کنترل‌کنندهٔ زبان سی تله**

5) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در (argptr) بازهٔ آدرس‌ها بررسی می‌گردد؟ تجاوز از بازهٔ 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 روبرو سازد.

می‌دانیم که توابعی که برای دسترسی به آرگومان‌های فراخوانی سیستمی در xv6 تعریف شده‌اند همگی از نوع int هستند. این توابع، در صورت بروز خطا، مقدار 1- را بازمی‌گردانند. این توابع به شرح زیر هستند:

```
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

- در این تابع از فراخوانی تابع دیگری به نام fetchint نیز استفاده شده است. ورودی اول fetchint، طبق فرمول زیر محاسبه شده است:

$$esp + 4 + 4 * n$$

و ورودی دوم تابع همان پوینتر به حافظه مدنظر گرفته شده برای مقدار int که برای به دست آوردن آن این تابع را فرا خوانده‌ایم، است و اگر با خطایی مواجه نشویم، مقدار آرگومان دوم set می‌شود و خروجی ارسال می‌شود.

```
// Fetch the nth word-sized system call argument as a string pointer.
// Check that the pointer is valid and the string is nul-terminated.
// (There is no shared writable memory, so the string can't change
// between this check and being used by the kernel.)
int
argstr(int n, char **pp)
{
  int addr;
  if(argint(n, &addr) < 0)
    return -1;
  return fetchstr(addr, pp);
}
```

در این تابع از argint نیز استفاده شده است که وظیفهٔ مشخص کردن آدرس ابتدای رشته را بر عهده دارد. اگر آدرس در متغیر مربوطه با موفقیت ذخیره نشود، مقدار خطای 1- در شرط چک می‌شود و این تابع هم به اشتباه، خروجی 1- را برمی‌گرداند. در غیر این صورت، با فراخوانی تابع fetchstr و پاس دادن آرگومان‌های مربوطه، عملیات زیر اجرا می‌شود:

```
// Fetch the nul-terminated string at addr from the current process.
// Doesn't actually copy the string - just sets *pp to point at it.
// Returns length of string, not including nul.
int
fetchstr(uint addr, char **pp)
{
  char *s, *ep;
  struct proc *curproc = myproc();

  if(addr >= curproc->sz)
    return -1;
  *pp = (char*)addr;
  ep = (char*)curproc->sz;
  for(s = *pp; s < ep; s++){
    if(*s == 0)
      return s - *pp;
  }
  return -1;
}
```

شرط موجود در این قطعه کد، بررسی می‌کند که آیا آدرس ورودی در حافظه مربوط به پردازه هست یا خیر. اگر نبود، مقدار خروجی 1- را برمی‌گرداند. در غیر این صورت، مقدار آرگومان دوم این تابع را برابر این آدرس قرار داده و سپس با یک ساختار حلقه، از ابتدای اشاره‌گر (یا همان آرگومان دوم) پیمایش می‌کند. اگر در این پیمایش به کاراکتر پوچ[12] (یا همان 0) رسید، طول رشته مدنظر را با یک تفریق برمی‌گرداند. وگرنه مقدار -1 را به نشانه بروز خطا خروجی می‌دهد.

```
// Fetch the nth word-sized system call argument as a pointer
// to a block of memory of size bytes.  Check that the pointer
// lies within the process address space.
int
argptr(int n, char **pp, int size)
{
  int i;
  struct proc *curproc = myproc();

  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->sz)
    return -1;
  *pp = (char*)i;
  return 0;
}
```

فراخوانی تابع argint، مشابه توضیحات قبلی، آدرس پوینتر را دریافت می‌کند. آرگومان سوم argptr یا همان سایز پوینتر هم با تابع argint مقداردهی می‌کند و سپس شرط وجود پوینتر با سایز منحصر به فردش در حافظه پردازه چک می‌شود. اگر وجود نداشت با برگرداندن مقدار -1 بروز خطا را نشان می‌دهد وگرنه آرگومان دوم تابع همان char **pp را مقداردهی می‌کند و با خروجی 0 عملیات موفقیت آمیز را نشان می‌دهد.

به عنوان جمع‌بندی، می‌توان گفت که این توابع دسترسی به پارامترهای فراخوانی سیستمی، در اصل برای آن تعبیه شده‌اند که راهی باشند برای تعامل میان سطح کاربر و سیستم‌عامل.همچنین، وظیفهٔ به درستی انتقال دادن پارامترهای مذکور از فضای کاربری به فضای هسته را نیز بر عهده دارند. به عنوان مثال، در مورد تابع اشاره شده (argptr)، می‌توان گفت که بازهٔ برگردانده شده توسط این تابع، مورد بررسی قرار می‌گیرد تا اطمینان حاصل شود که اشاره‌گرهای تشکیل‌دهندهٔ این بازه، به قسمت‌های درستی از حافظه اشاره می‌کنند و تحت حیطهٔ‌های مجاز حافظه، فعالیت می‌کنند. واضح است که اگر

[12] NULL

این بازهٔ برگردانده شده (از تابع مذکور) از حد مجاز خودش فراتر رود، ممکن است منجر به سرریز بافر شود! در این صورت، مقدار حافظهٔ سرریز شده، ناگزیر به اقسام دیگر حافظه راه می‌یابد که این امر، محافظت (و بسا امنیت) سیستم را مورد مخاطره قرار می‌دهد؛ چه می‌دانیم که بسیاری از حملات احتمالی صورت گرفته به سیستم، بر مبنای تحت تاثیر قرار دادن همین نقاط حساس آن، پی‌ریزی می‌شوند. به عنوان مثال، سناریویی را در نظر بگیرید که تابع argptr، به قسمتی از حافظه اشاره کند که اختیار یا حق مدیریت آن را ندارد و در همین زمان، فراخوانی سیستمی sys_read نیز صدا زده شود. در این صورت، احتمال آن می‌رود که دادهٔ خوانده شده (یا حتی بعدها، دادهٔ نوشته شده!) در حافظه، برآمده از مدیریت نادرستی بوده باشد.

بررسی گام‌های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb

در ابتدا، کدی ساده برای اجرا شدن فراخوانی سیستمی getpid در سطح کاربر می‌نویسیم که پیاده‌سازی آن به صورت زیر است. سپس آن را به makefile (در کنار بقیه برنامه‌های سطح کاربر)، می‌افزاییم:

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

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

حال دو ترمینال در دایرکتوری کد های سیستم‌عامل باز می‌کنیم و در یکی با اجرای دستور زیر:

```
make qemu-gdb
```

سیستم عامل را روی حالت gdb بالا می‌آوریم.
در ترمینال دیگر، با اجرای دستور زیر:

```
 gdb kernel
```

هستهٔ gdb را لود کرده و سپس ریموت را وارد می‌کنیم. در ادامه دستور continue را زده تا سیستم عامل Boot شود و بعد از اتمام عملیات، یک نقطهٔ توقف[13] در خط 145 (که مربوط به شرط موجود در فایل syscall.c می‌باشد)، با دستور زیر قرار می‌دهیم:

```
break syscall.c:145
```

و باز هم continue را وارد می‌کنیم.

[13] Breakpoint

در اصل، هدفمان این است که با اجرای برنامه gdb_test دیباگر در خط حاوی نقطهٔ توقف، متوقف شود.در ادامه، با اجرای دستور bt، در صفحه ترمینال می‌بینیم:

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) bt
#0  syscall () at syscall.c:145
#1  0x000000c8 in ?? ()
#2  0x80106471 in trap (tf=0x1010101) at trap.c:43
#3  0x0000003a in ?? ()
#4  0x801061bc in alltraps () at trapasm.S:20
```

همان طور که واضح است، این وضعیت پشتهٔ فراخوانی[14] در لحظهٔ کنونی اجرای برنامه می‌باشد.
از آنجایی که توابع صدا زده شده در طول برنامه یک فریم پشتهٔ مختص به خودشان دریافت می‌کنند و این فریم پشته حاوی اطلاعاتی (نظیر آدرس بازگشت، متغیرهای محلی و ...) است، هر خط این خروجی دریافت شده نماینده یک فریم پشته می‌باشد که از درونی‌ترین فریمی که در آن قرار داریم شروع می‌شود.
در ادامه، مقادیر نشان داده شده توسط bt به طور مختصر، مورد بررسی قرار می‌گیرند:
الف) فریم alltraps: قبل از فراخوانی تابع trap (که در trap.c قرار دارد)، با ساختن فریم مربوط به trap، آن را داخل پشته قرار می‌دهد و سپس، تابع trap فراخوانی می‌شود.

ب) فریم trap: تابع trap که در فریم قبل سخن از فراخوانی آن در میان بود، ضمن مشخص کردن نوع فراخوانی سیستمی، trapframe مربوط به پردازهٔ فعلی را برابر trapframe ای که در پشته وارد شده بود قرار داده و تابع syscall را صدا می‌کند.

ج) در ابتدای فایل syscall.c یک آرایه وجود دارد که توابع مربوط به فراخوانی‌های سیستمی را به شماره تعریف‌شدهٔ آنها متصل می‌کند. تابع syscall شماره فراخوانی سیستمی را از ثبّات eax در trapframe پردازهٔ فعلی خوانده و خروجی را همان‌جا ذخیره می‌کند.
با دستور up و down می‌توانیم روی فریم‌های پشته حرکت کنیم. در اینجا، چون در پایین‌ترین (و در واقع داخلی‌ترین) فریم پشته قرار داریم، نمی‌توانیم عمیق‌تر بشویم و نتیجتا با ایراد دستور down، با خطا مواجه می‌شویم.

[14] Call Stack

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

حال مقدار num که در eax قرار دارد را نمایش می‌دهیم:

```
(gdb) print num
$1 = 5
(gdb)
```

این شمارهٔ 5 متعلق به فراخوانی سیستمی read می‌باشد اما شمارهٔ فراخوانی سیستمی getpid برابر با 11 است. این فراخوانی‌های سیستمی read برای خواندن دستور هستند. ما به طور متوالی continue و print num را وارد می‌کنیم اما تنها تغییرات شمارهٔ فراخوانی سیستمی را در اینجا نمایش می‌دهیم.

همان‌طور که گفته شد، باید تعدادی فراخوانی سیستمی read انجام شود. این تعداد، برابر با 8 است چرا که دستور gdb_test دارای 8 کاراکتر می‌باشد.

```
Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$5 = 5
(gdb) continue
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$6 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$7 = 5
(gdb) continue
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$8 = 5
(gdb) continue
Continuing.
```

پس از خوانده شدن کامل دستور، فراخوانی سیستمی fork (با شمارهٔ 1) فراخوانی می‌شود. این فراخوانی سیستمی، پردازه‌ای جدید برای اجرای برنامهٔ سطح کاربر ایجاد می‌کند.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$9 = 1
```

سپس فراخوانی سیستمی wait (با شمارهٔ 3) نمایش داده می‌شود. کاربرد این فراخوانی سیستمی در ادامهٔ پروژه توضیح داده شده است اما کاربرد کلی آن، این است که برای اجرای پردازهٔ فرزند منتظر بماند.

```
Thread 1 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$10 = 3
```

در ادامه، فراخوانی سیستمی sbrk (با شمارهٔ 12) که برای تخصیص حافظه به پردازهٔ ایجاد شده استفاده می‌شود، فراخوانی می‌شود.

```
Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$11 = 12
```

سپس فراخوانی سیستمی exec (با شمارهٔ 7) فراخوانی می‌شود که پردازهٔ ایجاد شده را اجرا می‌کند.

```
Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$12 = 7
```

در ادامه، فراخوانی سیستمی getpid (با شمارهٔ 11) فراخوانی می‌شود که در راستای برنامه سطح کاربر می‌باشد.

```
Thread 2 hit Breakpoint 1, syscall () at syscall.c:145
145       if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
(gdb) print num
$13 = 11
```

سپس چند فراخوانی سیستمی write (با شمارهٔ 16) برای نوشتن خروجی نمایش داده می‌شود. در ادامه نیز آنقدر continue را وارد می‌کنیم تا خروجی به طور کامل نمایش داده شود.

```
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ gdb_test
PID: 3
$
```

ارسال آرگومان های فراخوانی‌های سیستمی

- **Find digital root**

در ابتدا در فایل syscall.h، شماره فراخوانی سیستمی 22 را به این فراخوانی سیستمی تخصیص می‌دهیم.

```
#define SYS_find_digital_root 22
```

سپس در فایل user.h، تابع مدنظر را declare می‌کنیم.

```
int find_digital_root(void);
```

تابع به این علت void تعریف شده که ما آرگومان را نه از طریق استک، بلکه از طریق رجیسترها منتقل می‌کنیم.

در فایل usys.S نیز ماکرو SYSCALL را با آرگومان find_digital_root اضافه می‌کنیم.

```
SYSCALL(find_digital_root)
```

برای پیاده سازی تابع در سطح کرنل، در فایل syscall.c، آن را extern می‌کنیم و سپس در آرایه syscalls، تابع را با عدد تخصیص یافته شده به این فراخوانی سیستمی، map می‌کنیم.

```
extern int sys_find_digital_root(void);
```

```
[SYS_find_digital_root] sys_find_digital_root,
```

برای پیاده‌سازی این برنامه در سطح کرنل و یوزر، یک فایل به نام sysextra ایجاد کردیم که پیاده‌سازی دو تابع digital_root و sys_find_digital_root در آن وجود دارد. توجه شود که تابع اول، محاسبات انجام شده در این برنامه پیاده‌سازی شده است.

```c
static int digital_root(int n)
{
    if (n < 0)
        return -1;

    while (n > 9)
    {
        int sum = 0;

        while (n != 0)
        {
            sum += n % 10;
            n /= 10;
        }

        n = sum;
    }
    return n;
}

int sys_find_digital_root(void)
{
    return digital_root(myproc()->tf->ebx);
}
```

حال ما ابزار لازم برای اجرای این برنامه را داریم اما باید در یک تابع دیگر، از این ابزار ها استفاده کنیم تا بتوانیم برنامه را اجرا کنیم. این فایل که find_digital_root.c نام دارد، شامل دو تابع می‌شود. در تابع main، ورودی‌ها را از کاربر می‌گیریم و در صورت نیاز، خطاها نمایش می‌دهیم. در صورتی که خطایی وجود نداشت، تابع digital_root_syscall فراخوانی می‌شود. بخشی از این تابع که به زبان اسمبلی نوشته شده است، مقدار رجیستر ebx که آرگومان دستور(n) قرار است در آن نوشته شود را در جایی ذخیره می‌کند تا پس از اجرای دستور، ebx همان مقدار قبلی خود را داشته باشد و این مقدار از دست نرود.

```c
#define NEGATIVE_INPUT_ERROR "Invalid input. Input must be positive!\n"
#define FEW_ARGUMANT_ERROR "Less input. One integer input is needed!\n"

int digital_root_syscall(int n) {
    int prev_ebx;
    asm volatile(
        "movl %%ebx, %0\n\t"
        "movl %1, %%ebx"
        : "=r"(prev_ebx)
        : "r"(n)
    );
    int result = find_digital_root();
    asm volatile(
        "movl %0, %%ebx"
        :: "r"(prev_ebx)
    );
    return result;
}

```

```c
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf(2, FEW_ARGUMANT_ERROR);
        exit();
    }
    int n;
    if (strlen(argv[1]) >= 2 && argv[1][0] == '-')
    {
        argv[1][0] = '0';
        n = atoi(argv[1]);
        n = -n;
    }
    else
        n = atoi(argv[1]);

    if (n < 0)
    {
        printf(2, NEGATIVE_INPUT_ERROR);
    }
    else
    {
        int result = digital_root_syscall(n);
        printf(1, "Digital root is: %d\n", result);
    }
    exit();
}
```

پس از اضافه کردن همه بخش‌های بالا، نوبت به اضافه کردن فایل‌های جدید به makefile می‌رسد. برای این کار باید sysextra.o را به آرایه OBJS و find_digital_root_ را به آرایه برنامه های سطح کاربر یا همان UPROGS اضافه کنیم.

```
169  UPROGS=\
170      _cat\
171      _echo\
172      _forktest\
173      _grep\
174      _init\
175      _kill\
176      _ln\
177      _ls\
178      _mkdir\
179      _rm\
180      _sh\
181      _stressfs\
182      _usertests\
183      _wc\
184      _zombie\
185      _strdiff\
186      _find_digital_root\
187      _get_uncle_count_test\
188      _get_process_lifetime_test\
```

حال برنامه را اجرا می‌کنیم و هر 3 حالت خروجی ممکن را نمایش می‌دهیم.

```
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
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ find_digital_root
Less input. One integer input is needed!
$ find_digital_root -10
Invalid input. Input must be positive!
$ find_digital_root 23
Digital root is: 5
$ _
```

پیاده‌سازی فراخوانی‌های سیستمی

برای اضافه کردن یک فراخوانی سیستمی به xv6، باید چندین فایل را تغییر دهیم که عبارتند از:

1. syscall.h
2. syscall.c
3. sysfile.c
4. usys.S
5. user.h

• پیاده‌سازی فراخوانی سیستمی کپی کردن فایل

برای پیاده سازی فراخوانی سیستمی copy_file (که اسامی دو فایل مبدأ و مقصد را به ترتیب می‌گیرد)، ابتدا از فایل syscall.h شروع می‌کنیم. پیش از هر چیز، بایستی خط زیر را به تعریف‌های موجود اضافه کرد:

```
#define SYS_copy_file 23
```

با این کار شماره فراخوانی 23 به این فراخوانی تخصیص داده شده است.
حال باید یک اشاره‌گر به این فراخوانی سیستمی، اضافه کنیم که در فایل syscall.c تعریف می‌شود. عنصر زیر را به آرایهٔ شامل اشاره‌گرهایی به توابع سیستم کال ها اضافه می‌کنیم:

```
[SYS_copy_file]   sys_copy_file
```

در اصل زمانی که سیستم کالی با شماره 23 فراخوانی شود، تابعی که این اشاره‌گر به تابع[15] مربوطه در این آرایه اشاره می‌کند فراخوانی میشود.
در فرایند پیاده سازی این تابع و تعریف این فراخوانی سیستمی در سطح هسته، پیش‌تعریف[16] این تابع را در همین فایل به شرح زیر وارد می‌کنیم:

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

[15] Function Pointer
[16] Prototype

که در کنار پیش‌تعریف‌های مربوط به باقی فراخوانی‌های سیستمی، قرار می‌گیرد. حال تعریف کد فراخوانی سیستمی پیاده‌سازی شده را در فایل sysfile.c وارد می‌کنیم.

```c
int sys_copy_file(void)
{
  char *src, *dst;
  if (argstr(0, &src) < 0 || argstr(1, &dst) < 0)
  {
    cprintf("copy_file: couldn't get file names.\n");
    return -1;
  }

  begin_op();
  struct inode *srcIp = namei(src);
  struct inode *dstIp = namei(dst);

  if (srcIp == 0)
  {
    end_op();
    cprintf("copy_file: src file doesn't exist.\n");
    return -1;
  }
  if (srcIp == dstIp)
  {
    cprintf("copy_file: src and dst file are same and a file cannot be copied in itself.\n");
    end_op();
    return -1;
  }
  if (dstIp != 0)
  {
    cprintf("copy_file: dst file already exists.\n");
    end_op();
    return -1;
  }

  dstIp = create(dst, T_FILE, 0, 0);
  if (dstIp == 0)
  {
    end_op();
    return -1;
  }

  struct file *dstFd;
  if ((dstFd = filealloc()) == 0)
  {
    iunlockput(dstIp);
    end_op();
    return -1;
  }

  iunlock(dstIp);
```

```
506   iunlock(dstIp);
507
508   struct file *srcFd;
509   if ((srcFd = filealloc()) == 0)
510   {
511     end_op();
512     return -1;
513   }
514
515   srcFd->type = FD_INODE;
516   srcFd->ip = srcIp;
517   srcFd->off = 0;
518   srcFd->readable = 1;
519   srcFd->writable = 0;
520
521   dstFd->type = FD_INODE;
522   dstFd->ip = dstIp;
523   dstFd->off = 0;
524   dstFd->readable = 0;
525   dstFd->writable = 1;
526
527   char buffer[512] = {'\0'};
528   int read_bit_count;
529
530   while ((read_bit_count = fileread(srcFd, buffer, 512)) > 0)
531   {
532     if (filewrite(dstFd, buffer, read_bit_count) != read_bit_count)
533     {
534       break; // this means there was error in writing in dst file
535     }
536   }
537
538   fileclose(srcFd);
539   fileclose(dstFd);
540
541   end_op();
542
543   int return_condition;
544   return_condition = (read_bit_count == 0) ? 0 : -1;
545   return return_condition;
546 }
```

در فایل usys.S ماکرو SYSCALL را با آرگومان copy_file اضافه کرده،

```
SYSCALL(copy_file)
```

و شناسهٔ فراخوانی سیستمی را در user.h وارد می‌کنیم:

```
int copy_file(const char *src, const char *dest);
```

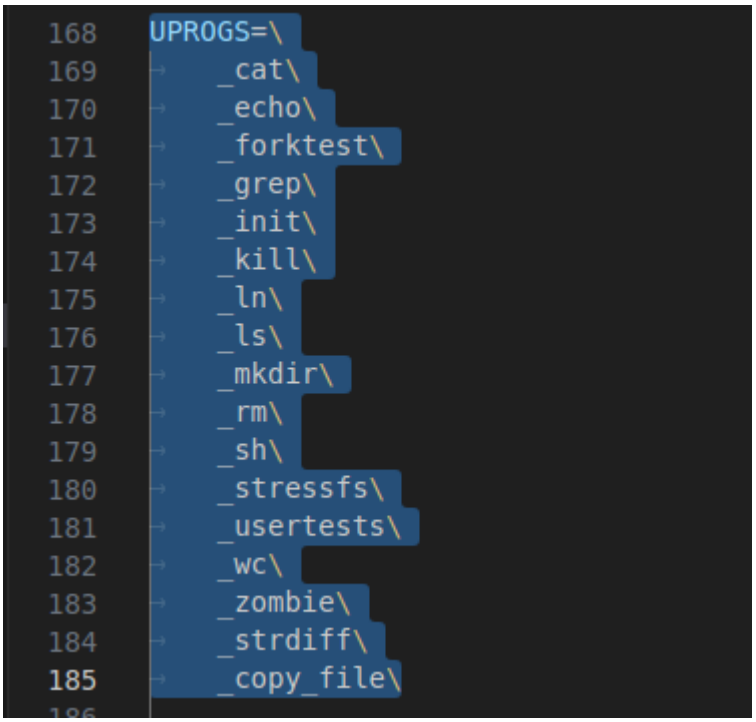
حال، برنامه‌ای در سطح کاربر و به شرح زیر، می‌نویسیم:

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

int main(int argc, char *argv[])
{

  if (argc != 3)
  {
    printf(2, "copy_file takes two arguments to run.\n");
    exit();
  }

  int copy_result = copy_file(argv[1], argv[2]);
  if (copy_result < 0)
  {
    printf(2, "copy_file failed...\n");
  }
  else
  {
    printf(1, "copy_file done!\n");
  }

  exit();
}
```

همچنین، تست برنامه نیز بایستی به شرح زیر انجام شود:

```
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
init: starting sh
Group 10:
1. Matin Nabizadeh
2. Mobina Mehrazar
3. Mohammad Amin Yousefi
$ echo hello world! > file1.txt
$ cat file1.txt
hello world!
$ copy_file file1.txt file2.txt
copy_file done!
$ cat file2.txt
hello world!
$
```

لازم به ذکر است که برنامهٔ سطح کاربر از قبل به این، طریق وارد Makefile شده‌است:



• پیاده‌سازی فراخوانی سیستمی تعداد uncle های پردازه

برای پیاده سازی فراخوانی سیستمی get_uncle_count (که process_id پردازه‌ای که قصد شمردن عموهای آن را داریم را می‌گیرد)، ابتدا از فایل syscall.h شروع می‌کنیم. پیش از هر چیز، بایستی خط زیر را به تعریف‌های موجود اضافه کرد:

```
#define SYS_get_uncle_count 24
```

حال باید یک اشاره‌گر به این فراخوانی سیستمی، اضافه کنیم که در فایل syscall.c تعریف می‌شود. عنصر زیر را به آرایهٔ شامل اشاره‌گرهایی به توابع سیستم کال ها اضافه می‌کنیم:

```
[SYS_get_uncle_count]   sys_get_uncle_count
```

در اصل، زمانی که سیستم کالی با شمارهٔ 24 فراخوانی می‌شود، تابعی که این اشاره‌گر به تابع مربوطه در این آرایه اشاره می‌کند است که فراخوانی می‌شود!

در فرایند پیاده سازی این تابع، پیش‌تعریف این تابع را در همین فایل به شرح زیر وارد می‌کنیم:

```
extern int sys_get_uncle_count(void);
```

که در کنار پیش‌تعریف‌های مربوط به باقی فراخوانی‌های سیستمی، قرار می‌گیرد. همچنین در فایل usys.S، ماکرو SYSCALL را با آرگومان نام تابع اضافه می‌کنیم:

```
SYSCALL(get_uncle_count)
```

سپس، در فایل user.h، تابع سطح کاربر را تعریف می‌کنیم:

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

حال کد فراخوانی سیستمی پیاده‌سازی شده را در فایل sysproc.c وارد می‌کنیم.

```
92  int
93  sys_get_uncle_count(void)
94  {
95    int pid;
96    if (argint(0, &pid) < 0)
97      return -1;
98    return uncle_count(pid);
99  }
```

در این تابع ابتدا آرگومان تابع (که ماهیت آن در ابتدا توضیح داده شده است) را از روی پشته بدست آورده و سپس، تابع uncle_count (که در فایل proc.c پیاده‌سازی شده است) را فرا می‌خوانیم. علت پیاده‌سازی این تابع در proc.c این است که ما برای بدست آوردن عموهای پردازهٔ فعلی، به جدول پردازه‌ها (ptable) نیاز داریم که این جدول در فایل proc.c قابل دسترسی می‌باشد. پیش از پیاده‌سازی تابع در فایل proc.c، در فایل proc.h، آن را تعریف می‌کنیم:

```
int uncle_count(int pid);
```

```
539  int uncle_count(int pid)
540  {
541    int p_ind = -1;
542    for (int i = 0; i < NPROC; i++)
543    {
544      if (ptable.proc[i].pid == pid)
545      {
546        p_ind = i;
547        break;
548      }
549    }
550
551    if (p_ind >= 0)
552    {
553      int counter = 0;
554      for (int i = 0; i < NPROC; i++)
555      {
556        if (ptable.proc[p_ind].parent->parent->pid == ptable.proc[i].parent->pid)
557        {
558          if(ptable.proc[i].parent->state != UNUSED)
559            counter++;
560        }
561      }
562
563      if (counter > 0)
564      {
565        return counter - 1;
566      }
567      else
568      {
569        return -1;
570      }
571    }
572    else
573    {
574      return -1;
575    }
576  }
```

در این تابع، ابتدا اندیس مربوط به پردازه‌ای که قصد شمردن عموهای آن را داریم را در جدول ptable می‌یابیم. سپس به ازای هر پردازه‌ای که pid پدرش با pid پدربزرگ پردازه مدنظر ما یکسان بود، متغیر counter را یک واحد افزایش می‌دهیم. در نهایت نیز برای اینکه پدر پردازهٔ فعلی را جزو جواب‌ها محاسبه نکرده باشیم، counter - 1 را به عنوان خروجی، برمی‌گردانیم. سپس، برای تست این فراخوانی سیستمی، برنامه get_uncle_count_test را پیاده‌سازی کرده‌ایم. در این برنامه، سه فرزند برای پردازهٔ اصلی ایجاد کرده‌ایم که هر فرزند پس از ایجاد شدن، sleep می‌شود تا به حالت UNUSED در نیایند و بتوانیم آنها را جزو عموهای نوهٔ پردازهٔ اصلی محاسبه کنیم.

وقتی که یک پردازهٔ فرزند به اتمام می‌رسد، به حالت ZOMBIE در می‌آید؛ برای مثال، هنگامی که دستور exit وارد می‌شود. تابع wait، همه پردازه‌های فرزندش را از حالت ZOMBIE به حالت UNUSED تبدیل می‌کند و همچنین پدر این پردازهٔ فرزند را برابر 0 قرار می‌دهد.

همچنین توجه داشته باشید که تابع getpid، شناسهٔ پردازهٔ فعلی را برمی‌گرداند.

```
36  void second_child()
37  {
38      int pid2 = fork();
39      if (pid2 < 0)
40      {
41          printf(2, "Second child fork failed\n");
42          exit();
43      }
44      else if (pid2 == 0)
45          sleep(50);
46      else
47          third_child();
48      wait();
49  }
50
51  int main(int argc, char *argv[])
52  {
53      int pid1 = fork();
54      if (pid1 < 0)
55      {
56          printf(2, "First child fork failed\n");
57          exit();
58      }
59      else if (pid1 == 0)
60      {
61          sleep(50);
62      }
63      else
64      {
65          second_child();
66      }
67      wait();
68      exit();
69  }
```

```c
void grand_child_for_third()
{
    int childpid = fork();
    if (childpid > 0)
        wait();
    else if (childpid == 0)
    {
        printf(1, "The child for third process has %d uncles\n", get_uncle_count(getpid()));
        exit();
    }
    else
    {
        printf(2, "Grandchild fork failed.\n");
        exit();
    }
    exit();
}

void third_child()
{
    int pid3 = fork();
    if (pid3 < 0)
    {
        printf(2, "Third child fork failed.\n");
        exit();
    }
    else if (pid3 == 0)
        grand_child_for_third();
    else
        wait();
}
```

- نمونه اجرای این دستور:

```
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ get_uncle_count_test
The child for third process has 2 uncles
$ _
```

• پیاده‌سازی فراخوانی سیستمی طول عمر پردازه

برای پیاده سازی فراخوانی سیستمی get_process_lifetime (که process_id پردازه‌ای که قصد محاسبهٔ طول عمر آن را داریم را می‌گیرد)، ابتدا از فایل syscall.h شروع می‌کنیم. پیش از هر چیز، بایستی خط زیر را به تعریف‌های موجود اضافه کرد:

```
#define SYS_get_process_lifetime 25
```

حال باید یک اشاره‌گر به این فراخوانی سیستمی، اضافه کنیم که در فایل syscall.c تعریف می‌شود. عنصر زیر را به آرایهٔ شامل اشاره‌گرهایی به توابع سیستم کال ها اضافه می‌کنیم:

[SYS_get_process_lifetime] get_process_lifetime,

در اصل زمانی که سیستم کالی با شماره 25 فراخوانی شود، تابعی که این اشاره‌گر به تابع مربوطه در این آرایه اشاره می‌کند فراخوانی میشود.
در فرایند پیاده سازی این تابع، پیش‌تعریف این تابع را در همین فایل به شرح زیر وارد می‌کنیم:

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

که در کنار پیش‌تعریف‌های مربوط به باقی فراخوانی‌های سیستمی، قرار می‌گیرد.
همچنین در فایل usys.S، ماکرو SYSCALL را با آرگومان نام تابع اضافه می‌کنیم:

```
SYSCALL(get_process_lifetime)
```

سپس در فایل user.h، تابع سطح کاربر را تعریف می‌کنیم:

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

برای اینکه امکان محاسبه طول عمر یک پردازه را داشته باشیم، باید زمان ایجاد شدن آن را ذخیره کنیم. برای این کار، در داده‌ساختار proc، یک متغیر از نوع uint، با نام generated_time ایجاد کرده‌ایم و در تابع fork، این متغیر را برابر با ticks/100 قرار می‌دهیم. یک وقفهٔ سخت‌افزاری وجود دارد که پس از boot شدن سیستم‌عامل، در هر ثانیه 100 بار فعال می‌شود. متغیر ticks، تعداد فعال شدن این وقفه را در خود ذخیره کرده و نتیجتا، ticks/100 نشان می‌دهد که از boot شدن سیستم چه مقدار زمان گذشته است.

```
41  // Per-process state
42  struct proc {
43    uint sz;                     // Size of process memory (bytes)
44    pde_t* pgdir;                // Page table
45    char *kstack;                // Bottom of kernel stack for this process
46    enum procstate state;        // Process state
47    int pid;                     // Process ID
48    struct proc *parent;         // Parent process
49    struct trapframe *tf;        // Trap frame for current syscall
50    struct context *context;     // swtch() here to run process
51    void *chan;                  // If non-zero, sleeping on chan
52    int killed;                  // If non-zero, have been killed
53    struct file *ofile[NOFILE];  // Open files
54    struct inode *cwd;           // Current directory
55    char name[16];               // Process name (debugging)
56    uint generated_time;         // Added by me.
57  };
```

```
180  int
181  fork(void)
182  {
183    int i, pid;
184    struct proc *np;
185    struct proc *curproc = myproc();
186
187    // Allocate process.
188    if((np = allocproc()) == 0){
189      return -1;
190    }
191
192    // Copy process state from proc.
193    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
194      kfree(np->kstack);
195      np->kstack = 0;
196      np->state = UNUSED;
197      return -1;
198    }
199    np->sz = curproc->sz;
200    np->parent = curproc;
201    *np->tf = *curproc->tf;
202
203    // Added by me
204    np->generated_time = ticks/100;
```

- حال کد فراخوانی سیستمی پیاده‌سازی شده را در فایل sysproc.c وارد می‌کنیم:

```
101  int
102  sys_get_process_lifetime()
103  {
104      int pid;
105    if (argint(0, &pid) < 0)
106      return -1;
107    return process_lifetime(pid);
108  }
```

در این تابع، ابتدا آرگومانش (که ماهیت آن در ابتدا توضیح داده شد) را از روی پشته بدست می‌آوریم. سپس، تابع process_lifetime (که در فایل proc.c پیاده‌سازی شده است) را فرا می‌خوانیم. علت

پیاده‌سازی این تابع در proc.c آن است که ما برای بدست آوردن طول عمر پردازهٔ فعلی، به جدول پردازه‌ها نیاز داریم که این جدول در فایل proc.c قابل دسترسی می‌باشد.

پیش از پیاده‌سازی تابع در فایل proc.c، در فایل proc.h، آن را تعریف می‌کنیم:

```
int process_lifetime (int pid);
```

```
578  int process_lifetime(int pid)
579  {
580    int i=0;
581    for(; i<NPROC; i++)
582    {
583      if(ptable.proc[i].pid == pid)
584        return (ticks/100) - ptable.proc[i].generated_time;
585    }
586    return -1;
587  }
588
```

در این تابع، ابتدا پردازهٔ مدنظر را از جدول پردازه‌ها پیدا کرده و سپس، اختلاف زمانی حال حاضر و زمان ایجاد شدن پردازه را بازمی‌گردانیم.

در نهایت، برای تست این فراخوانی سیستمی، برنامه get_process_lifetime_test را پیاده‌سازی کرده‌ایم. در این برنامه، ابتدا یک پردازه ایجاد می‌کنیم که در واقع حکم پردازهٔ پدر را دارد. سپس، در این پردازه، یک پردازهٔ دیگر ایجاد می‌کنیم که پردازهٔ فرزند باشد و در ادامه، پردازهٔ فرزند را به مدت 5 ثانیه با استفاده از حلقه while متوقف می‌کنیم و پس از این توقف، پردازهٔ فرزند را از بین برده و پردازهٔ پدر را به مدت 5 ثانیه دیگر متوقف می‌کنیم. در نهایت، پردازهٔ فرزند به مدت 5 ثانیه و پردازه پدر به مدت 10 ثانیه متوقف شده‌اند.

```
30  int main(int argc, char* argv[])
31  {
32      int parent_pid = fork();
33      if (parent_pid > 0)
34      {
35          wait();
36      }
37      else if (parent_pid == 0)
38      {
39          child_test();
40          while (1)
41          {
42              if (get_process_lifetime(getpid()) >= 10)
43              {
44                  printf(1, "parent lifetime: %d\n seconds",get_process_lifetime(getpid()));
45                  break;
46              }
47          }
48          exit();
49      }
50      else
51      {
52          printf(2, "Failed to create parent process.\n");
53      }
54      exit();
55  }
```

```c
void child_test()
{
    int child_pid = fork();
    if (child_pid > 0)
    {
        wait();
    }
    else if (child_pid == 0)
    {
        while (1)
        {
            if (get_process_lifetime(getpid()) >= 5)
            {
                printf(1, "child lifetime: %d\n seconds",get_process_lifetime(getpid()));
                kill(child_pid);
                break;
            }
        }
        exit();
    }
    else
    {
        printf(2, "Failed to create child process.\n");
    }
}
```

- نمونه اجرای این برنامه:

```
init: starting sh
Group #10:
1. Mohamad Amin Yousefi
2. Mobina Mehrazar
3. Matin Nabizade
$ get_process_lifetime_test
child lifetime: 5 seconds
parent lifetime: 10 seconds
$ _
```

آخرین هش در ریپازیتوری قبلی:

- Github Hash: c35c8a887b103869c1e775fd79c7287ad712ac81

آخرین هش در ریپازیتوری جدید:

- Github Hash: f90a19b7f306119717754e511c1bec2550755511