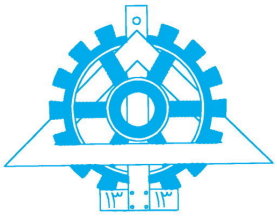


پروژه آزمایشگاه شماره ۲

دانشکده مهندسی برق و کامپیوتر

سیستم عامل - پاییز ۱۳۹۹

استاد : **دکتر کارگهی**

گروه ۱۷

اعضای گروه: دانشور امراللهی، علیرضا توکلی، امین ستایش

---

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

متغیر ULIB چند فایل تشکیل‌دهنده دارد که هر کدام را به طور مجزا توضیح خواهیم داد:

- ulib: تعدادی از توابع این کتابخانه به منظور کار با آرایه‌ی کاراکترها نوشته شده‌اند. این توابع فرخوانی‌های سیستمی ندارند. اما سه تابع gets، memset و stat وجود دارند که در آن‌ها از فراخوانی سیستمی استفاده شده است.
  - memset برای پر کردن حافظه‌ای با مقادیر دلخواه استفاده می‌شود. gets تابعی است که از read استفاده می‌کند تا از ورودی بخواند. stat نیز
  - در تابع gets یک حلقه اجرا می‌شود و ورودی گرفته می‌شود. در هر مرحله از انجام این کار از فراخوانی سیستمی read استفاده شده است.
  - در تابع stat ابتدا از فراخوانی سیستمی open استفاده شده است که برای باز کردن یک فایل از این فراخوانی سیستمی استفاده شده است. در بخش بعد به کمک فراخوانی سیستمی fstat اطلاعات فایل مرتبط با file descriptor درخواستی داده می‌شود. در انتها به کمک فراخوانی سیستمی close، فایل بسته می‌شود.
- printf: تعدادی از توابع این کتابخانه به منظور توابع کمکی برای چاپ در حالت‌های مختلف نوشته شده‌اند. تنها تابعی که در این بخش از فراخوانی سیستمی استفاده می‌کند، تابع putc است.
  - در تابع putc از فراخوانی سیستمی write به جهت چاپ کردن استفاده شده است. که برای آن fd موردنظر جهت چاپ کردن و کاراکتر موردنظر انتخاب شده است.
- umalloc: تعدادی از توابع این کتابخانه به منظور اختصاص دادن حافظه استفاده شده‌اند. در این کتابخانه تابع morecore از فراخوانی سیستمی استفاده می‌کند.

- تابع morecore به جهت افزایش حافظه استفاده می‌شود. در آن از فراخوانی سیستمی sbrk استفاده شده است. این فراخوانی سیستمی اندازه data segment را تغییر می‌دهد.

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

- Pseudo-file systems: در pseudo-file systemها (مثل proc/ و dev/ و sys/) دسترسی کرنل نیاز است. این pseudo-file systemها مثل filesystemها صراحتا file به معنای فایلی که زمان ساخت، حجم و ... ندارند و یک سری entry مجازی هستند که می‌توانند محتوای داده‌ساختارهای درون کرنل را به یک اپلیکیشن یا ادمین تحویل دهند به طوری که انگار آن محتوا روی یک فایل ذخیره شده بودند.
- Socket based: در این حالت برنامه‌های سطح کاربر می‌توانند بر روی socket گوش بدهند و اطلاعات را دریافت کنند.
- Exceptions: در این حالت هم دسترسی به kernel انجام میشود تا خطا رفع شود و سپس دوباره به سطح کاربر بازگردانده می‌شویم.

۳. آیا باقی تله‌ها را نمی‌توان با دسترسی DPL_USER فعال نمود؟ چرا؟

خیر. اگر یک پردازه بخواهد یه interrupt دیگری را فعال کند، xv6 با آن این اجازه رو نمی‌دهد و با یک protection exception مواجه می‌شوند که به vector شماره ۱۳ هدایت می‌شوند. علت این امر این است که ممکن است در برنامه سطح کاربر باگی وجود داشته باشد، یا کاربر سوءاستفاده کند و امنیت سیستم به خطر بیفتد.

یعنی سطح دسترسی DPL_USER سطح کاربر است و اگر میخواستیم باقی تله‌ها هم با همین سطح دسترسی فعال کنیم به راحتی میشد که به kernel دسترسی داشت و امنیت سیستم به خطر میفتاد.

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

به طور کلی دو پشته یکی برای سطح کاربر و دیگری برای kernel موجود است. هنگامی که میخواهیم دسترسی را تغییر بدهیم، مثلا از سطح کاربر به kernel برویم، دیگر نمیتوانیم از پشته قبلی استفاده کنیم.

پس باید ss و esp روی پشته push بشوند تا بتوان دوباره پس از بازگشت از سطح دسترسی دیگر از آنها استفاده کرد و اطلاعات از دست نروند.

از طرفی وقتی تغییر سطح دسترسی نداشته باشیم، نیازی به push کردن ss و esp نیست. چون به همان پشته هنوز دسترسی داریم.

۵. 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.

سه تابع را در این بخش می‌توان نام برد، argint ،argstr و argptr که هر کدام را به اختصار توضیح میدهیم.

- argint: برای آن شماره آرگومان مشخص می‌کنیم و آدرس یک int را برای گرفتن متغیر به آن میدهیم.
- argstr: دو پارامتر میگیرد اولی شماره آرگومان است و دومی آدرس یک متغیر از نوع char* است که در آن آرگومان ریخته می‌شود.
- argptr: شماره آرگومان، آدرس یک پوینتر و اندازه چیزی که خوانده می‌شود را میگیرد و در آدرس پوینتر محتوای آرگومان ریخته می‌شود.

در صورتی که آدرس چک نشود ممکن است خارج از محدوده معتبر باشده و به اطلاعات اشتباه دسترسی پیدا کنیم و مشکل در پردازش به وجود بیاید.

۶. فراخوانی‌های سیستمی، همگام و وقفه‌ها غیرهمگام با اجرای پردازه‌ها رخ می‌دهند. چگونه می‌توان همگام/غیرهمگام بودن این ورودی‌ها به هسته را نشان داد؟ (راهنمایی: منظور با کمک قاب تله است.)

متغیر type در struct gatedesc چندین مقدار می‌تواند بگیرد اما در xv6 این متغیر به دو مقدار STS_TG32 برای حالت تله و STS_IG32 برای حالت سیستم‌کال استفاده می‌شود. در کد از define مشخص شده در mmu.h که SETGATE نام دارد، استفاده می‌شود که برای ساخت Gate استفاده می‌شود و ورودی دوم آن تله بودن یا نبودن را مشخص می‌کند.

سیستم‌کال‌های اضافه شده:

سیستم‌کال calculate_biggest_perfect_square():

برای اضافه کردن این سیستم‌کال به هسته، نیازمند آن هستیم که امضای تابع، شماره‌ی اختصاص داده شده به سیستم‌کال و بدنه‌ی سیستم‌کال را اضافه کنیم. برای دو کار اول در user.h، syscall.h syscall.c، defs.h و usys.S تغییراتی که در کامیت‌ها مشخص است را انجام می‌دهیم. برای بدنه‌ی سیستم‌کال نیز در sysproc.c تابعی مانند بقیه‌ی توابع سیستم‌کال‌های نوشته شده را می‌نویسیم به طوری که ثبات ebx را که ثبات پس از eax است می‌خواند و تابع calculate_biggest_perfect_square را که در proc.c است، با مقدار این ثبات صدا می‌زند.

```
xv6-public/sysproc.c

@@ -89,3 +89,11 @@ sys_uptime(void)
89      release(&tickslock);
90      return xticks;
91    }
92  +
93  + int
94  + sys_calculate_biggest_perfect_square(void)
95  + {
96  +   int number = myproc()->tf->ebx; //register after eax
97  +   cprintf("Kernel: sys_calculate_biggest_perfect_square() called for number %d\n", number);
98  +   return calculate_biggest_perfect_square(number);
99  + }
```

اما در proc.c ما باید جواب را پیدا کنیم. برای این کار تا وقتی که توان دوی متغیرمان کمتر از عدد است، مقدارش را زیاد کرده و سپس مقدار خواسته شده را بر می‌گردانیم.

```c
int calculate_biggest_perfect_square(int n)
{
    int ans = 1;
    while (ans * ans < n)
      ans++;
    ans--;

    return ans * ans;
}
```

حال برای برنامه‌ی سطح کاربر نیاز است تا سیستم‌کال امتحان شود. برای این کار قطعه کد biggest_perfect_square.c نوشته شده است که در آن پس از چک کردن تعداد ورودی‌ها، ثبات قبلی را ذخیره کرده، مقدار جدید را در ثبات قدیمی نوشته، تابع را صدا زده و در انتها مقدار اولیه‌ی ثبات را به آن برمی‌گرداند.

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

int main(int argc, char *argv[]){
     if(argc < 2){
            printf(2, "You must enter exactly 1 number!\n");
            exit();
     }
    else
    {
            // We will use ebx register for storing input number
            int saved_ebx, number = atoi(argv[1]);
            //
            asm volatile(
                    "movl %%ebx, %0;" // saved_ebx = ebx
                    "movl %1, %%ebx;" // ebx = number
                    : "=r" (saved_ebx)
                    : "r"(number)
            );
            printf(1, "User: calculate_biggest_perfect_square() called for number: %d\n" , number);
            printf(1, "Biggest perfect square lower than %d is: %d\n" , number , calculate_biggest_perfect_square());
            asm("movl %0, %%ebx" : : "r"(saved_ebx)); // ebx = saved_ebx -> restore
            exit();
    }

    exit();
}
```

حال در انتها نیز برای آن که این برنامه به درستی کامپایل شود، نیاز است مانند فاز اول، Makefile را تغییر دهیم و این برنامه را به آن اضافه کنیم. جزئیات آن در کامیت مربوطه نیز وجود دارد.

نمونه‌ای از خروجی نیز به صورت زیر است.

```
$ biggest_perfect_square 200
User: calculate_biggest_perfect_square() called for number: 200
Kernel: sys_calculate_biggest_perfect_square() called for number 200
Biggest perfect square lower than 200 is: 196
$ _
```

سیستم‌کال set_sleep():

مراحل توضیح داده شده برای هر نوشتن هر سیستم‌کال را مانند مورد قبلی تکرار می‌کنیم. با این تفاوت که برای گرفتن مقدار زمان sleep باید از argint استفاده کنیم که این تابع برای خواندن آرگومان استفاده می‌شود.

```
void set_sleep(int n)
{
  uint ticks0;
  ticks0 = ticks;

  while(ticks - ticks0 < n * 100)
      sti();


}
```

تابع cli() اینتراپت‌ها را disable می‌کند و تابع sti() اینتراپت‌ها را enable می‌کند. البته کرنل xv6 به طور مستقیم با این توابع کار نمی‌کند و از pushcli و popcli استفاده می‌کند.

همان‌طور که گفته شده بود از ticks استفاده کردیم و برای ثبت زمان نیز سیستم‌کال دیگری به نام set_date داریم که struct rtcdate را برای ما با تابع cmostime مقداردهی می‌کند.

```
void
sys_set_date(void)
{
  struct rtcdate *r;
  if(argptr(0, (void*)&r, sizeof(*r)) < 0)
    cprintf("Kernel: sys_set_date() has a problem.\n");

  cmostime(r);
}
```

تمامی سیستم‌کال‌ها به شکلی که در قسمت اول گفته شد نوشته شده‌اند و در هر قسمت این قسمت‌ها را تکرار نمی‌کنیم. پس از آن نیز برنامه‌ی سطح کاربر نوشته می‌شود که با نام proc_sleep نوشته شده است. زمان فعلی را ثبت کرده سیستم‌کال را صدا می‌زند و سپس اختلاف زمان را نیز محاسبه می‌کند. خروجی به صورت زیر خواهد بود.

```
$ proc_sleep 3
User: calling set_sleep for 3 seconds...
Current system time: 36
Current system time: 39
Difference: 3
$
```

همان‌طور که مشاهده می‌کنید، اختلاف زمان دقیقا همان مقدار اصلی شد. دلیل آن می‌تواند عدم دقت زیاد rtcdate باشد که دقیق‌ترین واحد آن ثانیه است. اما دلیلی که ممکن است باعث اختلاف زمان شود، فراخوانی‌های مختلف توابع و به وجود آمدن سربار می‌تواند باشد

سیستم‌کال process_start_time():

ابتدا در فایل proc.h که به تعریف استراکت proc یک پارامتر به اسم creation_time از جنس uint اضافه می‌کنیم تا هنگام ساخته شدن process به آن مقداردهی کنیم.

1 xv6-public/proc.h

```
@@ -49,6 +49,7 @@ struct proc {
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)
+   uint creation_time;
};

// Process memory is laid out contiguously, low addresses first:
```

سپس در فایل proc.h خط زیر را در تابع fork اضافه می‌کنیم که هنگام ساخته شدن زمان سیستم، به کمک متغیر ticks زمان فعلی را در پارامتر creation_time ذخیره کند:

2 xv6-public/proc.c

```
@@ -183,6 +183,7 @@ fork(void)
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();
+   curproc->creation_time = ticks;

    // Allocate process.
    if((np = allocproc()) == 0){
```

برای نوشتن سیستم‌کالی که زمان ساخت یک پردازه را برگرداند مشابه همه مراحل تعدادی تغییرات در فایل‌های user.h و usys.S و syscall.h و syscall.c و defs.h نیاز است که بین همه سیستم‌کال‌ها مشترک است و فرآیند یکسانی دارند که در commitها می‌توانید آن‌ها را ببینید (اضافه کردن شماره متناظر با سیستم‌کال، اضافه کردن امضای سیستم‌کال و ...)
بخش اصلی یعنی پیاده‌سازی تابع این سیستم‌کال به شکل زیر است که در فایل sysproc.c سیستم‌کال را به صورت زیر پیاده‌سازی می‌کنیم:

7 ■■■■■ xv6-public/sysproc.c

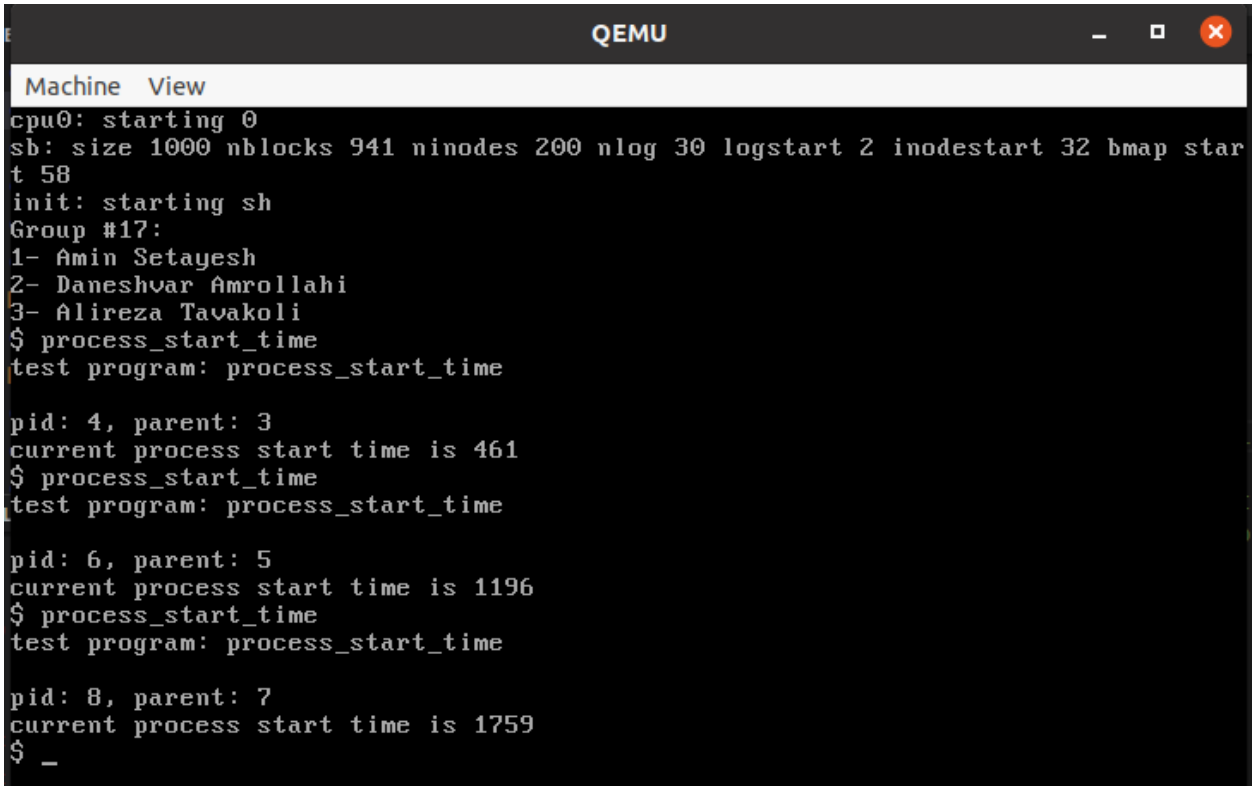
```
@@ -132,4 +132,11 @@ sys_get_descendants(void)
132 132     int pid = myproc()->tf->ebx;
133 133     cprintf("Kernel: sys_get_descendants() called for pid %d\n", pid);
134 134     get_descendants(pid);
135     + }
136     +
137     + int
138     + sys_process_start_time(void)
139     + {
140     +   struct proc *curproc = myproc();
141     +   return (int)(curproc->creation_time);
```

همچنین برای تست برنامه سطح کاربری به شکل زیر نوشتیم و با تغییرات لازم در makefile موفق به اجرای آن در ترمینال xv6 شدیم:

25 xv6-public/process_start_time.c

```
@@ -0,0 +1,25 @@
+ #include "types.h"
+ #include "stat.h"
+ #include "user.h"
+
+ int main(int argc, char *argv[])
+ {
+
+
+   int pid0;
+
+   printf(1, "test program: process_start_time\n\n");
+   pid0 = fork();
+   if (pid0)
+     printf(1, "pid: %d, parent: %d\n", pid0, getpid());
+
+
+   if(pid0){
+     printf(1, "current process start time is %d\n", process_start_time());
+
+   }
+   while(wait() != -1);
+
+
+   exit();
+ }
```

در ادامه اجرای برنامه سطح کاربر را مشاهده می‌کنید که با ساخت processهای جدید، زمان creation_time آن‌ها نیز بیشتر شده چون زمان گذشته.



سیستم‌کال get_ancestors():

برای بار دیگر در این سیستم‌کال موارد مشابه از قبیل برداشتن مقدار pid از ثبات ebx، اضافه کردن امضای توابع و ... را انجام می‌دهیم. اما در بدنه‌ی اصلی تابع، ابتدا acquire را صدا می‌زنیم که ptable در اختیارمان قرار گیرد و تغییری نکند. سپس proc مربوطه را پیدا کرده و تا وقتی که به پراسس اولیه نرسیده باشیم، parent این پراسس را با خود پراسس جایگزین می‌کنیم. در انتها نیز با تابع release که در انتهای کد آمده است، ptable را به حالت اولیه برمی‌گردانیم.

```
void get_ancestors(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if (p->pid == pid)
        break;
    }

    while (p->pid != 1)
    {
      cprintf("my id: %d, ", p->pid);
      cprintf("my parent id: %d\n ", p->parent->pid);

      p = p->parent;
    }

    //cprintf("my parent id: %d\n ", p->parent->pid);

    release(&ptable.lock);

    //pid = p->parent->pid;

}
```

برنامه‌ی سطح کاربر نیز به این صورت خواهد شد که تعدادی fork می‌زنیم و در انتها pid یکی از ریشه‌ها را در ثبات مورد نظر ریخته و سیستم‌کال را صدا می‌زنیم. پس از این سیستم‌کال با توجه به صورت پروژه به ما خروجی خواهد داد و ثبات به مقدار اولیه‌ی خود برمی‌گردد.

```
+ #include "types.h"
+ #include "stat.h"
+ #include "user.h"
+
+ int main(int argc, char *argv[])
+ {
+
+     int saved_ebx;
+
+   int pid0, pid1, pid2;
+   //pid1, pid2;
+   printf(1, "test program: forking childs\n\n");
+   pid0 = fork();
+   if (pid0)
+     printf(1, "pid: %d, parent: %d\n", pid0, getpid());
+
+   pid1 = fork();
+   if (pid1 && pid0 == 0)
+     printf(1, "pid: %d, parent: %d\n", pid1, getpid());
+
+   pid2 = fork();
+   if (pid2 && pid1 == 0 && pid0 == 0)
+     printf(1, "pid: %d, parent: %d\n", pid2, getpid());
+
+
+
+   if(pid0 == 0 && pid1 == 0 && pid2){
+     printf(1, "User: printing ancestors for pid: %d\n" , pid2);
+
+     asm volatile(
+     "movl %%ebx, %0;" // saved_ebx = ebx
+     "movl %1, %%ebx;" // ebx = number
+     : "=r" (saved_ebx)
+     : "r"(pid2)
+   );
+
+   //printf(1, "Biggest perfect square lower than %d is: %d\n" , number , calculate_biggest_perfect_square());
+     get_ancestors();
+     asm("movl %0, %%ebx" : : "r"(saved_ebx)); // ebx = saved_ebx -> restore
+   }
+   while(wait() != -1);
+
```

نکته‌ی مهم در این کد این است که باید حواسمان باشد تا زمانی که پراسس‌های ایجاد شده تمام نشده باشند، نباید پراسس پدر تمام شود که مشکل‌ساز خواهد شد. خروجی نمونه را در تصویر پایین مشاهده می‌کنید.

```
$ get_ancestors
test program: forking childs

pid: 6, parent: 5
pid: 7, parent: 6
pid: 11, parent: 7
User: printing ancestors for pid: 11
Kernel: sys_get_ancesrots() called for pid 11
my id: 11, my parent id: 7
 my id: 7, my parent id: 6
 my id: 6, my parent id: 5
 my id: 5, my parent id: 2
 my id: 2, my parent id: 1
 $
```

سیستم‌کال get_descendants():

در این سیستم‌کال نیز بسیار شبیه به سیستم‌کال قبلی عمل می‌کنیم با این تفاوت که بدنه‌ی اصلی سیستم‌کال متفاوت خواهد بود.

```
void get_descendants(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->parent->pid == pid)
    {
      cprintf("my id: %d, ", p->pid);
      cprintf("my parent id: %d\n ", pid);

      release(&ptable.lock);
      get_descendants(p->pid);
      acquire(&ptable.lock);
    }
  }
  release(&ptable.lock);
}
```

بسیار شبیه به حالت قبلی عمل می‌کنیم با این تفاوت که همه‌ی پراسس‌ها را مشاهده کرده و در صورتی که پردازه‌ای فرزند پردازه فعلی بود و زودتر از بقیه بود، آن را نیز به صورت بازگشتی صدا می‌زند. acquire و release نیز مانند حالت قبلی خواهد بود.

برنامه‌ی سطح کاربر نیز به همان صورت خواهد بود با این تفاوت که پراسس را از ریشه صدا می‌زنیم.

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

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

    int saved_ebx;

  int pid0, pid1, pid2;
  //pid1, pid2;
  printf(1, "test program: forking childs\n\n");
  pid0 = fork();
  if (pid0)
    printf(1, "pid: %d, parent: %d\n", pid0, getpid());

  pid1 = fork();
  if (pid1 && pid0 == 0)
    printf(1, "pid: %d, parent: %d\n", pid1, getpid());

  pid2 = fork();
  if (pid2 && pid1 == 0 && pid0 == 0)
    printf(1, "pid: %d, parent: %d\n", pid2, getpid());



  if(pid0 == 0 && pid1 == 0 && pid2){
    printf(1, "User: printing children for pid: %d\n" , 1);

    asm volatile(
    "movl %%ebx, %0;" // saved_ebx = ebx
    "movl %1, %%ebx;" // ebx = number
    : "=r" (saved_ebx)
    : "r"(1)
  );

    get_descendants();
    asm("movl %0, %%ebx" : : "r"(saved_ebx)); // ebx = saved_ebx -> restore
  }
  while(wait() != -1);


  exit();
}
```

و خروجی به شکل زیر خواهد شد که همان‌طور که انتظار داشتیم است.