پروژه دوم آزمایشگاه سیستم عامل

سید محمد امین اطیابی (810198559)
شایان شاه محمدی (810198531)
مرتضی نوری (810198481)

پاسخ سوالات :

1 - متغییر ULIB از چند فایل تشکیل شده است :

- ulib : سه تابع زیر از فراخوانی های سیستمی استفاده کرده اند :
  - gets : در این تابع از سیستم کال read در یک حلقه استفاده شده است تا ورودی را بخواند .
  - stat : در این تابع به کمک دو سیستم کال open و close یک فایل را باز و بسته میکنیم . همچنین به کمک سیستم کال fstat و دادن file desciptor به آن ، اطلاعات مربوط به فایل را درخواست می کنیم .
- printf : تنها یک تابع در این فایل از سیستم کال استفاده کرده است :
  - putc : در این تابع از سیستم کال write جهت چاپ کردن کاراکتر استفاده شده است .
- umalloc : در این فایل نیز تنها یک تابع از سیستم کال استفاده کرده است :
  - morecore : در این فایل از سیستم کال sbrk استفاده شده است که فضای پردازه را به اندازه n بایت افزایش می دهد .

2 - به کمک روش های زیر نیز میتوان با هسته ارتباط برقرار کرد :

- Exception : در صورتی که خطایی رخ دهد به هسته رفتیم و پس از رفع شده ایراد دوباره به سطح کاربر باز خواهیم گشت .
- Socket Based : در این روش برنامه سطح کاربر میتواند به سوکت گوش داده تا اطلاعات دریافت کند .
- Pseudo File System : شبیه فایل سیستم ها به طور واقعی حاوی فایل نیست ، بلکه درگاه برای اپلیکیشن ها جهت استفاده از فایل ها هستند به طوری که گویا واقعا دارای فایلی بوده اند . از آنجایی که این شبیه فایل سیستم ها نیاز به دسترسی سطح هسته دارند ، یکی از راه های ارتباط با هسته ، استفاده از این روش است .

3 - خیر ، چرا که سطح دسترسی DPL_USER سطح کاربر است و لذا فعال کردن باقی tarp ها با این سطح دسترسی می تواند به کاربر اجازه دسترسی به هسته را می دهد و می تواند مشکلات امنیتی به همراه داشته باشد .

4 - ما دو استک داریم ، یکی در سطح کاربر و یکی در سطح هسته . با رفتن به سطح هسته ، دسترسی ما به استک قبلی قطع می شود پس باید ss و esp را روی استک push کنیم که در زمان برگشت از سطح دسترسی بتوانیم اطلاعات را بازیابی کنیم . اما وقتی تغییر سطح دسترسی نداریم ، نیازی به push کردن ss و esp نیست چرا که همچنان دسترسی برقرار است .

5 - توابع دسترسی به پارامترهای سیستم کال به شرح زیر هستند :

- argint : آرگومان 32 بیتی nام سیستم کال را باز می گرداند .
- argstr : آرگومان nام سیستم کال را به صورت یک string pointer بر می گرداند .
- argptr : پوینتری به بلوکی از حافظه که حاوی مقدار آرگومان nام سیستم کال است را بر میگرداند .
- argfd : این تابع file desciptor و ساختار فایل متناظر به آرگومان nام سیستم کال را بر می گرداند .

6 - همانطور که قبلا ذکر شده بود ، در استک همراه با ss و esp ، مقداری با عنوان return address نیز push می شود . وقتی عملیات سطح هسته تمام می شود ، به کمک این آدرس بازگشت دقیقا به محلی بر می گردیم که برنامه از آنجا متوقف شده بود .

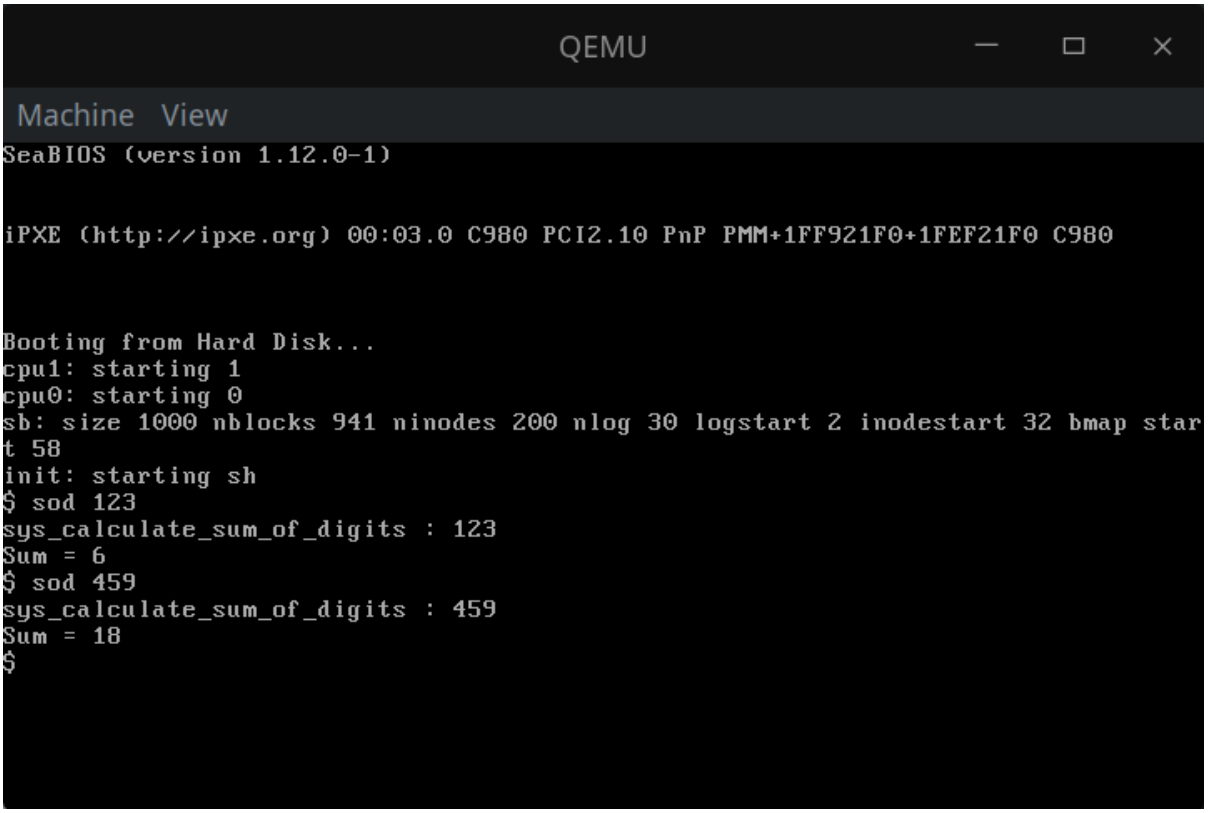
7 - می دانیم وقتی یک فایل در حافظه پاک می شود بدین معنا نیست که خانه های حافظه اشغال شده به صفر یا مقدار دیگر تغییر وضعیت می دهند ، بلکه داده های جدید اجازه نوشته شدن روی این داده ها را دارند . اگر فایل ما حجیم بوده و بلوک های داده آن متوالی نباشند ، احتمال اینکه داده های دیگری روی داده هایی که ما نیاز به بازیابی داریم نوشته شوند بیشتر می شود و با کمی تغییر در داده های ما ممکن است بازیابی دیگر ممکن نباشید . برای همین دلیل است به چند بخشی بودن داده های ما در بخش های مختلف حافظه ، امکان تغییر آن را بالاتر برده و بازیابی را با مشکل مواجه می کند .

اضافه کردن چند فراخوانی سیستمی

calculate_sum_of_digits()

در ابتدا نیاز داریم تا امضای این سیستم کال و شماره اختصاصی آن را به کد هسته اضافه کنیم . مانند بقیه سیستم کال ها امضا و توابع مورد نیاز این سیستم کال نیز نوشته می شود . امضای این سیستم کال در فایل های ، defs.h user.h ، syscall.c ، syscall.h و usys.S نوشته شده است .همچنین پیاده سازی تابع جمع زننده ارقام در فایل proc.c انجام شده است .

برای آزمایش درستی از سیستم کال ، فایلی به نام sod.c ساخته و جهت اجرا داخل makefile اضافه شد . در این فایل به جهت ارسال آرگومان توسط رجیستر ها ، ابتدا مقدار فعلی رجیستر در متغییری ذخیره و پس از اجرا سیستم کال به مقدار قبلی بازگردانده شد .



get_file_sectores()

همانند بقیه سیستم کال ها، نیاز است که شماره و امضای این سیستم کال را به کد هسته اضافه کنیم و رابط کاربری آن را در سطح برنامه کاربر تعریف کنیم. این تغییرات در فایل های user.h, syscall.h, syscall.c, usys.S, sysfile.c اعمال می شوند.

sysfile.c > sys_get_file_sectors(void)

```c
int
sys_get_file_sectors(void)
{
  int fd;
  int *sectors;
  struct file *f;
  if(argfd(0, &fd, &f) < 0 || argptr(1, (void*)&sectors, sizeof(*sectors) * 13) < 0)
    return -1;
  cprintf("sys_get_file_sectores(%d, sectors)\n", fd);
  for(int i = 0; i < 13; i++)
    sectors[i] = f->ip->addrs[i];
  return 0;
}
```

user.h > ...

```c
int calculate_sum_of_digits(void);
int get_parent_pid(void);
int get_file_sectors(int fd, int sectors[]);
```

get_sectors.c > main(int, char * [])

```c

int
main(int argc, char *argv[])
{
    if(argc < 2)
    {
        printf(1, "Please enter the file path.\n");
        exit();
    }
    int fd = open(argv[1], 0);
    if(fd < 0)
    {
        printf(1, "File not found.\n");
        exit();
    }
    int sectors[13];
    if(get_file_sectors(fd, sectors) < 0)
    {
        printf(1, "Error while running get file sectors system call.\n");
        exit();
    }
    for(int i = 0; i < 13; i++)
        printf(1, "block %d in sector %d\n", i + 1, sectors[i]);
    exit();
}
```

```
$ get_sectors 400byte.txt
sys_get_file_sectores(3, sectors)
block 1 in sector 65
block 2 in sector 0
block 3 in sector 0
block 4 in sector 0
block 5 in sector 0
block 6 in sector 0
block 7 in sector 0
block 8 in sector 0
block 9 in sector 0
block 10 in sector 0
block 11 in sector 0
block 12 in sector 0
block 13 in sector 0
$ _
```

```
$ get_sectors 5kbyte.txt
sys_get_file_sectores(3, sectors)
block 1 in sector 67
block 2 in sector 68
block 3 in sector 69
block 4 in sector 70
block 5 in sector 71
block 6 in sector 72
block 7 in sector 73
block 8 in sector 74
block 9 in sector 75
block 10 in sector 76
block 11 in sector 0
block 12 in sector 0
block 13 in sector 0
$
```

```
$ get_sectors 10kbyte.txt
sys_get_file_sectores(3, sectors)
block 1 in sector 77
block 2 in sector 78
block 3 in sector 79
block 4 in sector 80
block 5 in sector 81
block 6 in sector 82
block 7 in sector 83
block 8 in sector 84
block 9 in sector 85
block 10 in sector 86
block 11 in sector 87
block 12 in sector 88
block 13 in sector 89
$
```

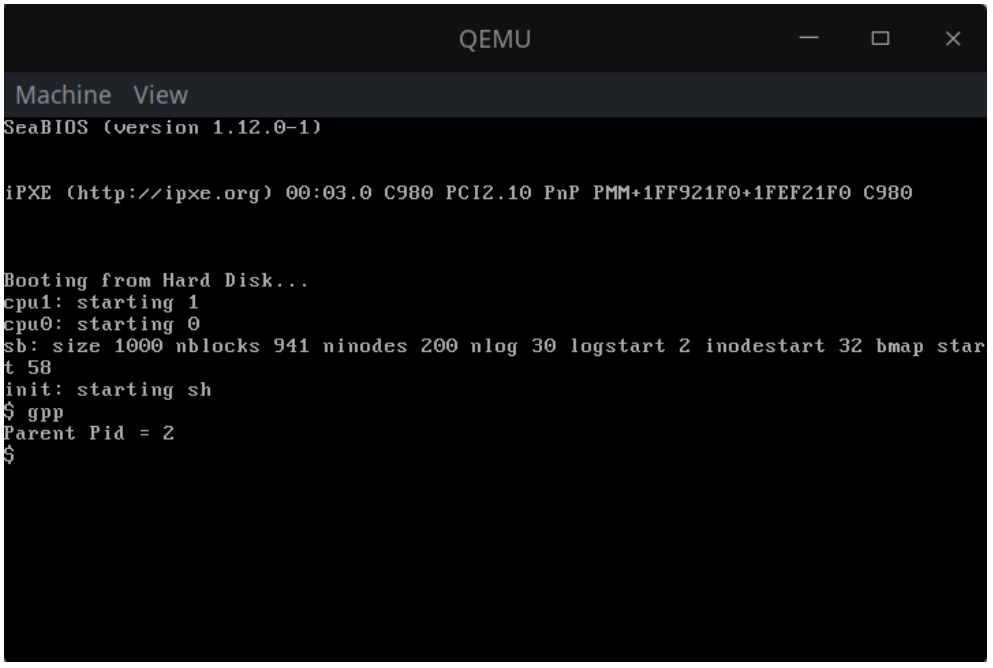
همانطور که در تصاویر خروجی مشاهده می کنیم، بلاک های دیتا در سکتورهای متوالی از حافظه نوشته می شوند، نحوه پر شدن سکتور ها به این صورت است ابتدا بلاک ها در ۱۲ سکتور به صورت direct (در آرایه addrs[0:11] در ساختار فراداده inode ذکر شده است) ساماندهی می شوند،اگر حجم فایل بیشتر از ۱۲ بلاک بود در این صورت سکتور بعدی که به صورت indirect تعریف شده است، حاوی آدرس شروع ۱۲۸ سکتور متوالی است که بقیه بلاک ها بتوانند ذخیره شوند. بدین ترتیب حجم فایل ها در xv6 ، حداکثر ۱۴۰ سکتور یا ۱۴۰ * ۵۱۲ بایت خواهد بود.

7 - می دانیم وقتی یک فایل در حافظه پاک می شود بدین معنا نیست که خانه های حافظه اشغال شده به صفر یا مقدار دیگر تغییر وضعیت می دهند ، بلکه داده های جدید اجازه نوشته شدن روی این داده ها را دارند . با توجه به توضیحات بالا اگر داده ها به صورت متوالی در حافظه ذخیره نشوند، رهگیری آنها سخت می شود. (به اضافه اینکه سربار اضافه ای نیز به فایل سیستم وارد می شود) در واقع ما نمی توانیم تشخیص دهیم که مثلا سکتور قبلی یا بعدی سکتور فعلی مربوط به چه فایلی است.

get_parent_id()

مانند سیستم کال های قبلی ، امضا این سیستم کال نیز به کد های هسته سیستم عامل اضافه شد . پیاده سازی این سیستم کال به صورت صدا کردن تابع myproc() جهت گرفتن اطلاعات پردازه فعلی ، رفتن به متغییر parent و خواندن متغییر pid آن بوده است .

در انتها برای آزمودن این سیستم کال فایلی با نام gpp.c ساخته و جهت اجرا به makefile اضافه شد . در این فایل با صدا زدن تابع get_parent_pid شماره پردازه پدر پردازه فعلی را چاپ می کنیم .



set_process_parent()

مانند سیستم کال‌های قبلی، امضای سیستم کال را به هسته اضافه می‌کنیم(فایل‌های user.h، usys.S، syscall.c، defs.h، syscall.h) و سپس به struct proc در فایل proc.h سه متغیر جدید اضافه می‌کنیم، is_tracer برای اینکه مشخص شود پردازه یک tracer هست یا نه، tracer_parent برای دسترسی به پدر tracer که آن را همان پدر پردازنده

اصلی می‌گذاریم و tracer_child برای دسترسی به پردازه‌ای که می‌خواهیم دیباگ کنیم. سپس در sysproc.c سیستم کال را مشخص می‌کنیم و سیستم کال get_parent_pid را تغییر میدهیم تا بتوانیم پدر درست پردازه‌ها را بر اساس اینکه tracer هستند یا نه مشخص کنیم.

سپس تابع اصلی set_process_parent را در فایل proc.c تعریف می‌کنیم به این صورت که روی تمام پردازه‌های موجود در جدول یک حلقه می‌زنیم تا پردازه‌ای که آیدی آن را دادیم پیدا کنیم و سپس متغیرهای دو پردازه را ست می‌کنیم تا پدر پردازه اصلی به پردازه tracer اشاره کند و پدر پردازه tracer به پدر پردازه اصلی اشاره کند و متغیرهای مورد نیاز پردازنده tracer را ست می‌کنیم.

سپس دو فایل tracer.c و traced.c را تعریف می‌کنیم تا پردازه tracer، پردازه traced را دنبال کند.

دستور wait در فایل tracer.c صبر می‌کند تا پروسه فرزند tracer.c تمام شود و سپس پروسه را ببندد.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int is_tracer;
  struct proc *tracer_parent;
  struct proc *traced_process;
};
```

تغییرات proc.h

```c
int sys_get_parent_pid(void)
{
  struct proc *p = myproc()->parent;
  while (p->is_tracer) {
    p = p->tracer_parent;
  }
  return p->pid;
}

void sys_set_process_parent(void)
{
  int pid = myproc()->tf->ebx;
  cprintf("sys_set_process_parent for process %d\n", pid);
  return set_process_parent(pid);
}
```

تغییرات sysproc.c

```c
void set_process_parent(int pid)
{
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid == pid)
    {
      break;
    }
  }
  struct proc* myp = myproc();
  myp->is_tracer = 1;
  myp->tracer_parent = p->parent;
  myp->traced_process = p;
  p->parent = myproc();
  cprintf("process %d parent changed to %d\n", p->pid, myp->pid);
}
```

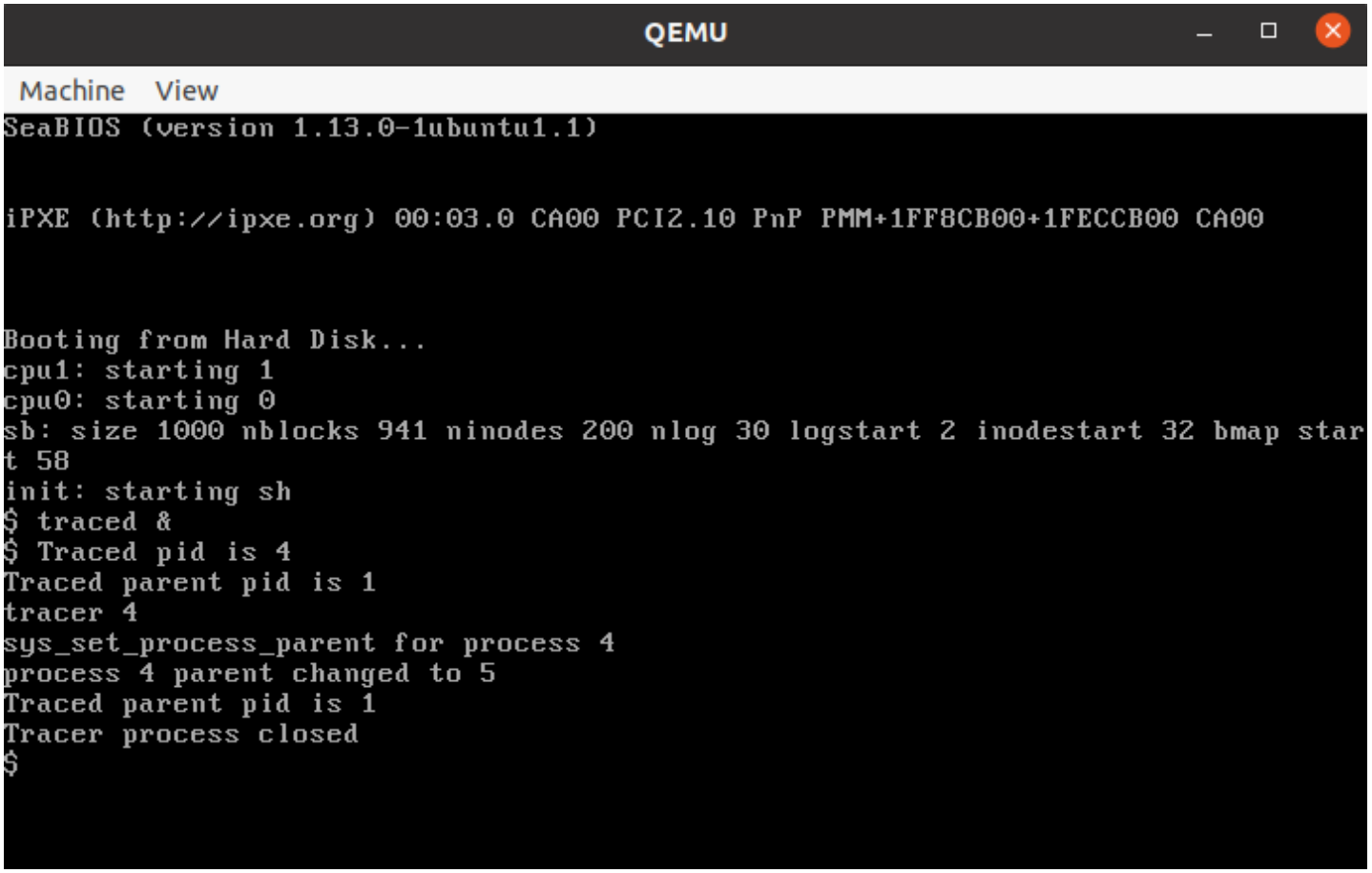
تابع اصلی در proc.c

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

int main(int argc, char *argv[])
{
    if(argc != 1)
    {
        printf(1, "Too many arguments!!!\n");
        exit();
    }

    printf(1, "Traced pid is %d\n" , getpid());
    printf(1, "Traced parent pid is %d\n", get_parent_pid());
    sleep(1000);

    printf(1, "Traced parent pid is %d\n", get_parent_pid());
    exit();
}
```

فایل traced.c

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

int main(int argc, char *argv[])
{
    if(argc != 2)
    {
        printf(1, "Please enter only one argument\n");
        exit();
    }
    int traced_pid = atoi(argv[1]);
    int prev_value = 0;
    asm volatile(
        "movl %%ebx, %0;"
        "movl %1, %%ebx;"
        : "=r" (prev_value)
        : "r"(traced_pid)
    );

    set_process_parent();

    asm("movl %0, %%ebx" : : "r"(prev_value));
    wait();

    printf(2, "Tracer process closed\n");

    exit();
}
```

فایل tracer.c



خروجی برنامه

با توجه به خروجی، آیدی پردازه اصلی ۴ است و آیدی پدر آن ۱ است. سپس پدر آن را به پردازه tracer با آیدی ۵ تغییر می‌دهیم که چون پردازه ۵، tracer است پدر اصلی پردازه ۴ همان ۱ می‌ماند.