اعضا : سید احمد رکنی حسینی ۸۱۰۱۰۰۱۵۴ محمدعلی شاهین فر ۸۱۰۱۰۰۱۶۹ امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

سوال اول :

این کتابخانه ها توابع و سیستم کال های پایه در xv6 را درخود دارند

ulib.o: در این کتابخانه توابعی I/O مثل read , write , open , close وجود دارد که در سطح یوزر اجرا میشوند.همچنین توابعی مثل fork , kill , exit , signal هم وجود دارند.

usys.o: توابع در این قسمت برای ارتباط بین سطح یوزر با کرنل برای انجام سیستم کال استفاده میشود . توابع این فایل به عنوان کنترل کننده سیستم کال ها در نظر گرفته میشوند

Printf.o: توابعی برای نوشتن در کنسول ارائه میدهد که توسط یوزر قابل دسترسی و اجرا هستند.

umalloc.o: توابع مربوط به تخصیص حافظه در این بخش وجود دارد و امکان مدیریت حافظه را ارائه میدهد. برخی توابع آن malloc , free , realloc هستند.

سوال دوم :

Linux has some API we can use for access include system calls, pseudo-files, and libc functions. for example, every ubuntu installation requires 224 system calls, 204 ioctl,fcnttl and prctl codes and hundreds of pseudo files.

system Library APIs: programmers program to more user friendly API in libc and other library. libc exporting an API like POSIX as well as the primary way application developers interact with the OS kernel

pseudo files: in the /proc, /dev and /sys. these are called pseudo-file systems because they are not backed by disk, but rather export the contents of kernel data structures to an application or administrator as if they were stored in a file

سوال سوم :

in the XV6 operating system, the DPL_USER access level refers to user mode, which is a lower privilege level compared to kernel mode. When a trap occurs in user mode, the processor switches to kernel mode to handle the trap. However, the XV6 kernel does not allow traps to be enabled while in user mode.

Enabling traps in user could potentially lead to security vulnerabilities or instability in the system. By restricting trap handling to kernel mode, XV6 ensures that critical operations and privileged instructions are only executed by the kernel, preventing unauthorized access or misuse.

So, to maintain the system's security and reliability, XV6 restricts the ability to enable traps in DPL_USER access level.

سوال چهارم :

رجیستر های ss (stack segment) و esp (stack pointer) زمانی به استک پوش میشوند که تغییر سطح دسترسی از یوزر به کرنل داشته باشیم. این مقادیر برای این است که قبلی حالت استک یوزر حفظ شود و بعدا بتوانیم آنرا بازیابی کنیم.

اگر تغییر دسترسی نداشته باشیم نیازی به پوش کردن این رجیستر ها به استک نیست زیرا وضعیت استک ثابت میماند.

سوال پنجم :

functions accesses the parameters with some pointer math. they access the trap frame struct of the process, containing the user-space registers of the system call that save parameters starting at the esp register. it adds for to skip empty word on the stack.

two checks occur during a call to code argptr. first, the user stack pointer is checked during the fetching of the argument, then the argument, itself a user pointer, is checked.

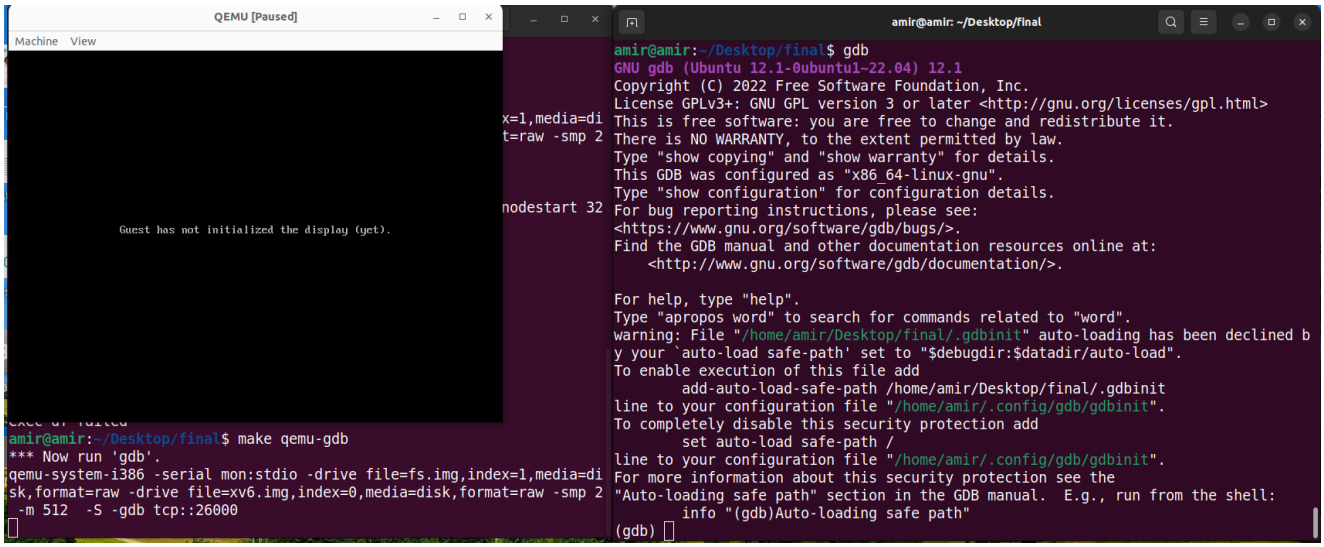
بررسی اجرای فراخوانی سیستمی در سطح کرنل توسط GDB

ابتدا برنامه سطح کاربر زیر را ایجاد میکنیم که **process id** را چاپ میکند

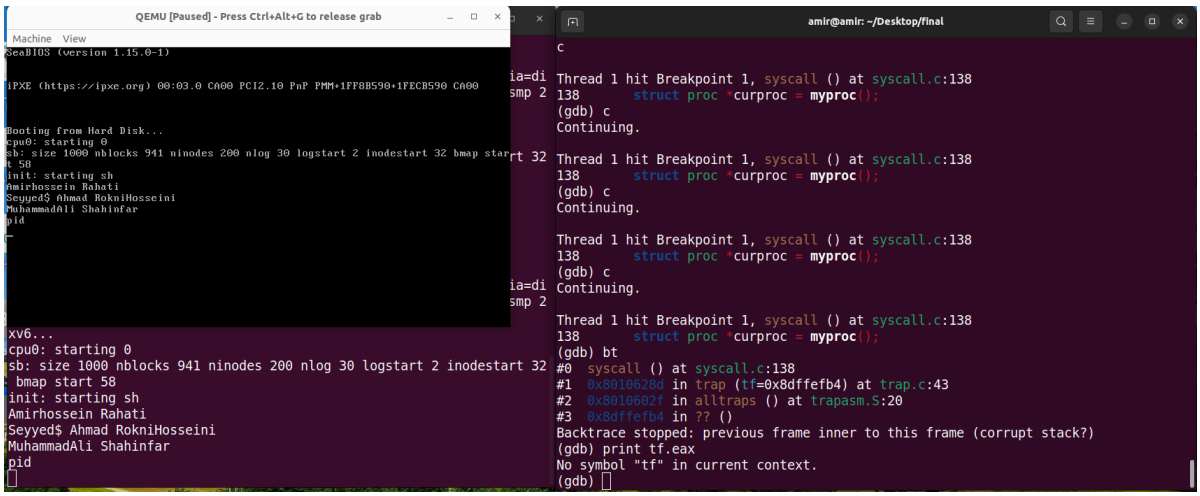
```c
#include "types.h"
#include "user.h"

int main(void)
{
  int my_pid = getpid();
  printf(1, "process ID: %d\n", my_pid);
  exit();
}
```

```c
// Layout of the trap frame
// hardware and by trapasm.S
struct trapframe {
  // registers as pushed by
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;
```

سپس **xv6** را با کامند **make qemu-gdb** کامپایل و اجرا میکنیم. در ترمینالی دیگر **gdb** را به **qemu** متصل میکنیم و سطح دسترسی **gdb** باید کرنل تعریف شود.



سپس روی **syscall** با دستور **break syscall** توقف قرار میدهیم و با دستور **continue** هردفعه از آن عبور میکنیم . زمانی که دستور **pid** را اجرا میکنیم **breakpoint** هایی که قرار دادیم هیت میشوند . در این بین با دستور **bt** میتوان لیست **backtrace** ها را در استک فراخوانی دید . از دستور **up** استفاده میکنیم تا به محل قرار گیری آبجکت **tf** در استک فراخوانی برویم . با دستور **print tf.eax** محتوای رجیستر **eax** نمایش داده میشود که در بین تمام **breakpoint** هایی که فعال میشوند یکی از آن ها مقدار **pid** ای است که برای ما نمایش داده میشود. بقیه مربوط به **process** های سطح کرنل هستند.



```
amir@amir: ~/Desktop/final
syscall.c
132  };
133
134  void
135  syscall(void)
136  {
137    int num;
B+>  138    struct proc *curproc = myproc();
139
140    num = curproc->tf->eax;
141    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
142      curproc->tf->eax = syscalls[num]();
143    } else {
144      cprintf("%d %s: unknown sys call %d\n",
145              curproc->pid, curproc->name, num);
remote Thread 1.1 In: syscall                    L138  PC: 0x80105230
Type "apropos -v word" for full documentation of commands related to "
word".
Command name abbreviations are allowed if unambiguous.
(gdb) c
Continuing.

Thread 1 hit Breakpoint 1, syscall () at syscall.c:138
(gdb)
```

تصویری از هیت شدن breakpoint ها در حالت layout src که محل فعال شدن breakpoint را نشان میدهد

نتیجه میشود که مقدار process id هر process حین اجرا در رجیستر eax ذخیره میشود.

افزودن چند سیستم کال به کرنل xv6

برای افزودن هر سیستم کال ، باید یک شماره جدید به آن نسبت داد ، فراخوانی های سیستمی را و مکانیزم پاس دادن توابع را تعیین نمود و در آخر توسط برنامه سطح کاربر به آن دسترسی پیدا کرد.

برخی تعییرات و کد های اضافه شده برای دو سیستم کال : get_uncle_count() , get_process_lifetime()

ابتدا توابع و سیستم کال ها در فایل proc.c تعریف و پیاده سازی میشوند.

سپس در فایل syscall.c باید هرکدام از آن ها تعریف و به کرنل شناسانده شوند

```
syscall.c   proc.c  6
proc.c > get_uncle_count(int)
int
get_uncle_count(int pid)
{
  struct proc *p;
  struct proc *grand_parent;
  int count = 0;

  //acquire(&ptable.lock);
  for(p= ptable.proc;p< &ptable.proc[NPROC];p++){
    if(p->pid == pid){
      grand_parent = p->parent->parent;
      for(p = ptable.proc;p<&ptable.proc[NPROC];p++){
        if(p->parent->pid == grand_parent->pid){
          count++;
        }
      }
    }
  }
  return --count;
}
```

```
syscall.c   proc.c  6
proc.c > get_process_lifetime(int)
int
get_process_lifetime(int pid){
  struct proc *p;
  for(p= ptable.proc; p< &ptable.proc[NPROC];p++){
    if(p->pid == pid){
      return ticks - p->ptime;
    }
  }
  return -1;
}
```

در فایل syscall.h و user.h هر سیستم کال را تعریف و میکنیم و به سطح یوزر و کرنل می شناسانیم.

سیستم کال های اضافه شده را به فایل sysproc.c هم اضافه میکنیم. این فایل در واقع محل شناسایی سیستم کال ها و هندل کردن آن ها در سطح یوزر است .

فایل user.h هم نقشی مشابه دارد همچنین سیستم کال ها باید در فایل proc.c هم تعریف شوند که درواقع برای این است که آنها به عنوان یک process تعریف شوند.

نهایتا دو برنامه سطح کاربر برای تست و اجرای سیستم کال ها اضافه میکنیم.

syscall.c 9+ | syscall.h

syscall.h > ...

```c
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_get_uncle_count 22
#define SYS_get_process_lifetime 23
```

syscall.c 9+ | user.h 1

user.h > ...

```c
int get_uncle_count(int);
int get_process_lifetime(int);
```

syscall.c 9+ | proc.c 6

syscall.c > syscalls

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

syscall.c 9+ | sysproc.c

sysproc.c > ...

```c
int
sys_get_uncle_count(void)
{
  int pid;
  if(argint(0, &pid) < 0)
    return -1;
  int result = get_uncle_count(pid);
  return result;
}
```

syscall.c 9+ | sysproc.c

sysproc.c > ...

```c
int
sys_get_process_lifetime(void)
{
  int pid;
  if(argint(0, &pid) < 0)
    return -1;
  int result = get_process_lifetime(pid);
  return result;
}
```

get_process_lifetime_test.c

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

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

int main(int argc,char* argv[]){
    uint pid = fork();
    if(pid==0){
        sleep(1000);
        printf(1,"%d\n",get_process_lifetime(getpid()));
        exit();
    }
    else{
        wait();
        printf(1,"%d\n",get_process_lifetime(getpid()));
        exit();
    }
}
```

get_uncle_count_test.c

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

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"
int main(int argc,char* argv[]){
    uint pid1 = 1,pid2 = 1,pid3 = 1,pid4 = 1;
    pid1 = fork();
    if(pid1==0){
        sleep(100);

    }
    else{
        pid2 = fork();
        if(pid2==0){
            sleep(100);
        }
```