به نام او...

گزارش پروژه دوم آزمایشگاه سیستم عامل

نام اعضای گروه:

سارا توکلی ۸۱۰۱۹۶۶۸۴

زهرا موسوی موحد ۸۱۰۱۹۶۶۲۹

غزل نیسی مینایی ۸۱۰۱۹۶۶۸۳

فراخوانی سیستمی

---

۱) کتابخانه های (قاعدتاً سطح کاربر، منظور فایلهای تشکیل دهنده متغیر ULIB در Makefile است) استفاده شده در xv6 را از منظر استفاده از فراخوانی های سیستمی و علت این استفاده بررسی نمایید.

در c.sh که interprator command را بوجود می آورد تابع runcmd وظیفه اجرای دستورات را دارد که براساس دستور ورودی از سیستم کال های متفاوتی استفاده می کند. اگر دستور از نوع EXEC باشد لازم است قطعه کد برنامه فعلی با برنامه ورود نظر برای اجرا جابه جا شود که برای این کار از سیستم کال exec استفاده می شود. اگر دستور از نوع REDIR باشد لازم است دایرکتوری کنونی که به صورت یک فایل نگه داری می شود بسته شود و دایرکتوری مورد نظر باز شود که از سیستم کال های close و open استفاده می شود. اگر

دستور از نوع LIST باشد لازم است که برنامه ها با تقدم و تاخر انجام شوند که برای این کار از سیستم کال های fork و wait استفاده می شود. که fork یک پروسه جدید می سازد و wait باعث ایجاد تاخر لازم می شود و در پروسه پدر برای فرزند صبر می کنیم. اگر دستور از نوع PIPE باشد لازم است که یک پروسه انجام شود سپس خروجی آن به پروسه دیگر انتقال یابد که مانند قبل تاخر و تقدم داریم و کار با فایل که از سیستم کال های fork و wait و close و dup استفاده می شود. برای اطمینان از قابل باز بودن توصیف گر فایل ها از open استفاده می شود. در c.sh برای صدا زدن runcmd ورودی خوانده شده یک پروسه برای اجرای runcmd اضافه می شود با fork و برای اتمام آن از exit استفاده می شود و برای صبر برای اتمام پروسه فرزند در پروسه پدر از wait استفاده می شود. در c.init که اولین برنامه ای است که طرف کاربر اجرا می شود. با استفاده از سیستم کال های mknode و dup و open توصیفگر های فایل استاندارد برای ارور و ورودی و خروجی باز می شود سپس با fork و exec تابع runcmd اجرا می شودتا کاربر بتواند با os در تعامل باشد.

2) آیا باقی تله ها را نمی توان با سطح دسترسی DPL_USER فعال نمود؟ چرا؟

خیر. دستور int به process level-user ها اجازه صدور signal interrupt های تعریف شده در table descriptor interrupt یا IDP را می دهد. بنابراین، مقدار دهی اولیه IDT باید با دقت صورت گیرد تا جلوی interrupt یا exception های غیر مجاز از سمت کاربر گرفته شود. این مهم با تنظیم کردن مقدار DPL مربوط به descriptor gate یک interrupt یا exception به عدد صفر انجام می شود. بنابراین unit control میتواند با مقایسه DPL با مقدار current level privilege یا CPL ،تحرکات غیر قانونی را شناسایی کرده و یک protection general exception صادر نماید. البته در برخی از موارد لازم می شود که یک process level-user بتواند یک exception از پیش تعیین (program شده را صادر کند که این کار با تنظیم کردن DPL به عدد ۳ امکان پذیر خواهد بود).

3) در صورت تغییر سطح دسترسی، ss و esp روی پشته Push میشود. در غیراینصورت Push نمیشود. چرا؟

هنگامی که یک trap به دلایل مختلف مانند فراخوانی یک system call اتفاق می افتد پیش از آن که از user mode به kernel mode برویم نیاز است تا اطلاعاتی مانند esp , ss که مربوط به

stack کاربر است در kernel به صورت یک trap frame ذخیره شوند تا بتوان وضعیت را بعد از آن دوباره بازیابی کرد. اما هنگامی که در kernel mode هستیم اگر interrupt رخ دهد نیاز به تغییر mode نیست و پس این اطلاعات هم ذخیره نمی شوند.

4) چرا تغییرات در فراخوانی های سیستمی یک سیستم عامل، به سادگی ممکن نیست؟

فراخوانی های سیستمی باعث میشوند برنامه وارد مود kernel شود که دارای دسترسی های بالاتر و سخت افزاری است. پیاده سازی نادرست سیستم کال ها می تواند مشکلات زیادی به وجود بیاورد و امنیت سیستم را کاهش دهد. همچنین وجود باگ در سیستم کال میتواند باعث توقف سیستم عامل شود. پس در پیاده سازی سیستم کال ها باید با دقت کافی عمل کرد.

همچنین عملیات سیستم کال ها به دلیل ارتباط مستقیم با سخت افزار ممکن است پیچیده باشد.

5) در مورد توابع دسترسی به پارامترهای فراخوانی سیستمی به طور مختصر توضیح دهید. چرا در)(argint (به طور دقیقتر در)(fetchint) بازه آدرسها بررسی میگردد؟ تجاوز از بازه معتبر، چه مشکل امنیتی ایجاد میکند؟

در xv6 آرگومان ها به طور مستقیم به فراخوانی های سیستمی داده نمیشوند بلکه توسط توابعی خوانده شده و در اختیار سیستم کال قرار میگیرند. از این توابع میتوان به argint که برای خواندن و پاس دادن متغیر int استفاده میشود اشاره کرد.

در این تابع n اندیس پارامتر (از صفر) و *ip مقدار خوانده شده. این تابع تابع fetchint را برای دسترسی به محل موردنظر در حافظه فراخوانی میکند. این تابع اول چک میکند که آدرس مورد نظر خارج از محدوده p->sz نباشد چون در این صورت باعث ایجاد یک segment trap میشود که پردازه را kill میکند. خارج شدن از این بازه ممکن است راه دسترسی یک پردازه به فضا یا اطلاعات حساسی را باز کند و راهی برای بدافزارها باشد یا استفاده از مقداری نامعتبر باعث خطا شود.

ارسال آرگومان های فراخوانی های سیستمی

اضافه کردن فراخوانی سیستمی :

برای اضافه کردن فراخوانی سیستمی ابتدا نام آن را به syscall.h اضافه می کنیم و آن را با عددی متناظر می کنیم که شناسه ی آن فراخوانی سیستمی است.

```
#define SYS_count_num_of_digits 22
#define SYS_get_parent_id 23
#define SYS_set    24
#define SYS_set_sleep 25
#define SYS_get_date 26
#define SYS_set_sleep_with_delay 27
#define SYS_get_children 28
```

سپس در syscall.c آن را extern میکنیم.

```
extern int sys_count_num_of_digits(void);
extern int sys_get_parent_id(void);
extern int sys_set(void);
extern int sys_set_sleep(void);
extern int sys_get_date(void);
extern int sys_set_sleep_with_delay(void);
extern int sys_get_children(void);
```

سپس به جدول maping اشاره گر به این تابع را اضافه میکنیم.

```
[SYS_count_num_of_digits]    sys_count_num_of_digits,
[SYS_get_parent_id]          sys_get_parent_id,
[SYS_set]     sys_set,
[SYS_set_sleep] sys_set_sleep,
[SYS_get_date] sys_get_date,
[SYS_set_sleep_with_delay] sys_set_sleep_with_delay,
[SYS_get_children]  sys_get_children
```

سپس بدنه ی تابع را به sysproc.c اضافه می کنیم و در این تابع از توابع تعریف شده در proc.c استفاده می کنیم.

Prototype این تابع را به فایل defs.h در بخش توابع proc.c اضافه میکنیم.

```
int             count_num_of_digits(int);
int             get_parent_id(void);
void            set(char*);
void            set_sleep(int);
void            set_sleep_with_delay(int);
void            get_date(void);
int             get_children(int);
```

Prototype فراخوانی سیستمی را به user.h اضافه میکنیم تا برنامه های سطح کاربر بتوانند از این فراخوانی سیستمی استفاده کنند.

```
int count_num_of_digits(int);
int get_parent_id(void);
int set(char*);
int set_sleep(int);
int set_sleep_with_delay(int);
int get_date(void);
int get_children(int);
```

برای از بین بردن پیچیدگی هaای استفاده از system call و ارتباط با سخت افزار یک تابع پوشاننده در usys.S تعریف میکنیم. این توابع این وابستگی هaا را مدیریت میکنند.

```
SYSCALL(count_num_of_digits)
SYSCALL(get_parent_id)
SYSCALL(set)
SYSCALL(set_sleep)
SYSCALL(get_date)
SYSCALL(set_sleep_with_delay)
SYSCALL(get_children)
```

فراخوانی سیستمی SYS_count_num_of_digits(int num)

برای اضافه کردن فراخوانی سیستمی همانند توضیحات آمده در بالا رفتار می کنیم.

در این فراخوانی برای ارسال آرگومان از رجیستر استفاده کردیم. به این صورت که در ابتدا ورودی را در رجیستر esi قرار دادیم و برای جلوگیری از از بین رفتن مقداری قبلی این رجیستر آن را در یک متغیر ذخیره کردیم . سپس شماره فراخوانی سیستمی مورد نظر را در رجیستر eax قرار دادیم و در انتها int 64 را صدا زدیم. و سپس مقدار esi را بازیابی کردیم.

تابع اضافه شده در sysproc.c:

```
int
sys_count_num_of_digits(void)
{
  int number, res;
  asm("movl %%esi,%0":"=r"(number));
  res = count_num_of_digits(number);
  cprintf("count of digits : %d\n", res);
  return count_num_of_digits(number);
}
```

تابع اضافه شده در proc.c:

```
int
count_num_of_digits(int number)
{
  int temp = number;
  int number_of_digits = 0;
  while (temp > 0) {
    temp = temp/10;
    number_of_digits++;
  }
  return number_of_digits;
}
```

برنامه سطح کاربر اضافه شده:

```c
int
main(int argc, char *argv[])
{
  if(argc <= 1){
    printf(1, "cnd: not enough argument\n");
    exit();
  }
  unsigned long temp;
  int number = atoi(argv[1]);

  asm("movl %%esi,%0":"=r"(temp));
  asm("movl %0,%%esi"::"r"(number));
  asm("movl $22,%eax;");
  asm("int $0x40");
  asm("movl %0,%%esi"::"r"(temp));

  number = temp;
  temp = number;
  exit();
}
```

پیاده سازی فراخوانی های سیستمی

۱. اضافه کردن متغیر محیطی PATH

برای اضافه کردن فراخوانی سیستمی مانند توضیحات آمده در بالا رفتار می کنیم.

ابتدا مسیر های مورد نظر کاربر، در برنامه سطح کاربر با دستور set گرفته می شوند. سپس برای اضافه کردن متغیر محیطی، فایل path.h را ساخته و در آرایه دو بعدی PATH، مسیر های وارد شده توسط کاربر را جدا و ذخیره میکنیم.

فایل اضافه شده path.h:

```
1   #ifndef PATH_H
2   #define PATH_H
3
4   #define MAX_PATH 10
5   #define MAX_CHARS 128
6   char PATH[MAX_PATH][MAX_CHARS];
7   int indexPath;
8   #endif
```

تابع اضافه شده در sysproc.c:

```
110  int
111  sys_set(void)
112  {
113    char *newPath;
114    if(argstr(0, &newPath) < 0)
115      return -1;
116
117    set(newPath);
118    return 1;
119  }
```

تابع اضافه شده در proc.c:

```
556  void
557  set(char* newPath)
558  {
559    int i = 0;
560    int j = 0;
561    int k = 0;
562    int pathLen = strlen(newPath);
563    while(pathLen) {
564      if (newPath[i] == ':') {
565        PATH[j][k+1] = '\0';
566        j++;
567        if (j >= 10) {
568          cprintf("Max Paths reached!\n");
569          break;
570        }
571        indexPath = j;
572        i++;
573        k = 0;
574      }
575      PATH[j][k] = newPath[i];
576      k++;
577      pathLen--;
578      i++;
579    }
580    PATH[j+1][k+1] = '\0';
581  }
```

در این تابع، ورودی گرفته شده از کاربر را بر اساس ":" جدا میکنیم و هرکدام را در اولین خانه خالی از PATH ذخیره میکنیم.

تغییر فایل exec.c:

```
11  int
12  exec(char *path, char **argv)
13  {
14    char *s, *last;
15    int i, off;
16    uint argc, sz, sp, ustack[3+MAXARG+1];
17    struct elfhdr elf;
18    struct inode *ip;
19    struct proghdr ph;
20    pde_t *pgdir, *oldpgdir;
21    struct proc *curproc = myproc();
22    char pathBuff[MAX_CHARS+MAX_CHARS];
23
24    begin_op();
25
26    if((ip = namei(path)) == 0){
27      int flag = 0;
28      for(int i = 0; i < indexPath; i++) {
29        memset(pathBuff, '\0', 2*MAX_CHARS);
30        int j, k;
31        for(j = 0; j < strlen(PATH[i]); j++) {
32          pathBuff[j] = PATH[i][j];
33        }
34        if(pathBuff[0] != '/') {
35          pathBuff[j] = '/';
36          j++;
```

```
26    if((ip = namei(path)) == 0){
27      int flag = 0;
28      for(int i = 0; i < indexPath; i++) {
29        memset(pathBuff, '\0', 2*MAX_CHARS);
30        int j, k;
31        for(j = 0; j < strlen(PATH[i]); j++) {
32          pathBuff[j] = PATH[i][j];
33        }
34        if(pathBuff[0] != '/') {
35          pathBuff[j] = '/';
36          j++;
37        }
38        for(k = 0; k < strlen(path); k++) {
39          pathBuff[k+j] = path[k];
40        }
41        ip = namei(pathBuff);
42        if (ip != 0) {
43          flag = 1;
44          break;
45        }
46      }
47      if(flag == 0) {
48        end_op();
49        cprintf("exec: fail\n");
50        return -1;
51      }
52    }
```

ابتدا flag را تعریف میکنیم و برابر 0 می گذاریم و در صورت وجود دستور مورد نظر در یکی از مسیر های موجود در PATH، آن را 1 می کنیم.

به ازای هر یک از مسیر های موجود در PATH، مسیر و دستور وارد شده توسط کاربر را در pathBuff ذخیره می کنیم و تابع namei را با ورودی pathBuff صدا می زنیم. اگر خروجی آن مخالف 0 باشد، flag را 1 کرده و از حلقه خارج میشویم. (یعنی دستور وارد شده در

این مسیر تعریف شده و وجود دارد و دستور اجرا میشود.) اگر حلقه تمام شود و flag صفر بماند، end_op را صدا می زنیم.

برنامه سطح کاربر اضافه شده:

```
1   #include "types.h"
2   #include "stat.h"
3   #include "user.h"
4   #include "path.h"
5
6   char buf[16];
7
8   int
9   main(int argc, char *argv[])
10  {
11    if(argc <= 2){
12      printf(1, "set: not enough argument\n");
13      exit();
14    }
15    set(argv[2]);
16    exit();
17  }
18
```

این برنامه با دستور زیر اجرا میشود:

Set <paths> (seperated by ‘:’)

2. خواباندن پردازه

برای اضافه کردن فراخوانی سیستمی مثل قبل عمل میکنیم.

برای گرفتن تاریخ در سطح کاربر سیستم کال get_date را پیاده سازی میکنیم.

تابع اضافه شده در sysproc.c :

```
int
sys_get_date(void)
{
  systemTime time;
  cmostime(&time);
  int res = 1;
  res = res*100 + time.hour;
  res = res*100 + time.minute;
  res = res*100 + time.second;
  return res;
}
```

این تابع برای راحتی کار ساعت و دقیقه و ثانیه را با فرمت:

1hhmmss

به صورت int بازمیگرداند.

از این سیستم کال برای تست set_sleep استفاده کردیم.

برای پیاده سازی فراخوانی سیستمی set_sleep از دو روش مختلف استفاده کردیم:

۱) استفاده از ticks :

set_sleep_with_delay()

تابع اضافه شده در sysproc.c :

```
int
sys_set_sleep_with_delay(void)
{
  int n;
  if(argint(0, &n) < 0)
    return -1;
  n = n*100;

  uint ticks0;
  ticks0 = ticks;
  while (1) {
    sti();
    if (ticks - ticks0 >= n) {
      break;
    }
  }
  return 0;
}
```

از آنجا که ورودی n به ثانیه است باید اول تعداد کلاک های خواسته شده را حساب کنیم. این عدد حدودا ۱۰۰ کلاک است.

متغیر ticks0 مقدار ticks در لحظه فراخوانی را ذخیره میکند و حلقه بی نهایت تا رسیدن اختلاف ticks و ticks0 به مقدار خواسته شده پردازه را متوقف میکند.

با فعال کردن sti از interrupt ها و درنتیجه تغییر ticks آگاه میشویم.

۲) استفاده از cmostime :

set_sleep()

تابع اضافه شده در sysproc.c :

```
int
sys_set_sleep(void)
{
  int n, h, m, s;
  if(argint(0, &n) < 0)
    return -1;
  h = n / 3600;
  n = n - h * 3600;
  m = n / 60;
  n = n - m * 60;
  s = n;

  systemTime time1, time2;
  cmostime(&time1);
  sti();
  while (1) {
    cmostime(&time2);
    if(time2.hour == time1.hour + h &&
       time2.minute == time1.minute + m &&
       time2.second == time1.second + s) {
      break;
    }
  }

  return 0;
}
```

در این روش محاسبه میکنیم که آیا زمان سیستم به زمان دلخواه ما رسیده است یا خیر.

اختلاف ساعت و دقیقه و ثانیه را در هر لحظه با حالت اول (لحظه کال کردن) را چک میکنیم و در صورت رسیدن به عدد ورودی حلقه را قطع میکنیم.

برنامه سطح کاربر برای تست:

```c
int
diff(int time1, int time2)
{
  int res = 0;
  int temp1, temp2;
  temp1 = time1%100;
  time1 /= 100;
  temp2 = time2%100;
  time2 /= 100;
  res = temp2 - temp1;

  temp1 = time1%100;
  time1 /= 100;
  temp2 = time2%100;
  time2 /= 100;
  res += (temp2 - temp1) * 60;

  temp1 = time1%100;
  time1 /= 100;
  temp2 = time2%100;
  time2 /= 100;
  res += (temp2 - temp1) * 3600;

  return res;
}

int
main(int argc, char *argv[])
{
  if(argc <= 2) {
    printf(1, "set_sleep: not enough argument\n");
    exit();
  }
  int option = atoi(argv[1]);
  int delay = atoi(argv[2]);
  int firstTime, secTime;
  firstTime = get_date();
  if (option == 1)
    set_sleep(delay);
  else
    set_sleep_with_delay(delay);

  secTime = get_date();
  printf(1, "diff = %d\n", diff(firstTime, secTime));

  exit();
}
```

این برنامه با دستور زیر اجرا میشود:

Set_sleep <option> <seconds>

ورودی option اگر ۱ باشد sleep از روش دوم و درغیر این صورت از روش اول اجرا میشود.

ورودی دوم مدت زمان sleep به ثانیه را نشان میدهد.

با استفاده از فراخوانی سیستمی get_date زمان قبل و بعد از فراخوانی های فوق ذخیره میشود و با تابع diff اختلاف این دو زمان به ثانیه محاسبه میشود و در خروجی چاپ میگردد.

دلیل اختلاف در فراخوانی به روش ticks :

در فراخوانی با روش ticks مقدار کمی اختلاف با ساعت سیستم مشاهده میشود که یکی از دلایل آن نادقیق بودن عدد ۱۰۰ یعنی تعداد تیک ها در ثانیه است. از طرفی سیستم کال ها (get_date) و دستورات قبل و بعد سربار به سیستم عامل تحمیل میکنند که محاسبه نشده است و این سربار رو عدد نهایی تاثیرگزار است.

3. گرفتن پردازههای فرزند یک پردازه و امتیازی

برای اضافه کردن فراخوانی سیستمی همانند توضیحات آمده در بالا رفتار می کنیم.

Get_parent_id

برای پیاده سازی این فراخوانی سیستمی به ازای پردازه ی فعلی با استفاده از struct proc آیدی پدر آن را به دست آوردیم.

تابع اضافه شده در sysproc.c:

```
int
sys_get_parent_id(void)
{
  return get_parent_id();
}
```

تابع اضافه شده در proc.c:

```
int
get_parent_id()
{
  struct proc *p = myproc();
  return p->parent->pid;
}
```

برنامه سطح کاربر اضافه شده:

```c
void
par_pid()
{
  printf(1, "current pid %d\n", getpid());
  int pid1 = fork();
  if (pid1 == 0) {
      printf(1, "result is: %d\n", get_parent_id());
  }
  else {
      wait();
  }
}

int
main(int argc, char *argv[])
{
  par_pid();
  exit();
}
```

Get_children

برای پیاده سازی این فراخوانی سیستمی که با گرفتن یک آیدی بچه ها و نوه ها و ... آن پردازه را نشان میدهد ابتدا یک تابع کمکی اضافه کردیم که به ازای هر پردازه با پیمایش ptable بچه های آن را به صورت یک عدد بر میگرداند . سپس برای به دست آوردن نوه ها به ازای هر بچه بچه ها ی آن را با استفاده از همان تابع به دست آوردیم و برای به دست آوردن نتیجه ها به ازای هر نوه این کار را کردیم و به همین ترتیب تا زمانی که دیگر بچه ای وجود نداشته باشد.

تابع اضافه شده در sysproc.c:

```
int
sys_get_children(void)
{
  int pid;

  if(argint(0, &pid) < 0)
    return -1;
  return get_children(pid);
}
```

توابع اضافه شده در proc.c:

```
int
get_process_children(int pid)
{
  struct proc *p;
  int children =0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if (p->parent->pid == pid) {
      children = children*10 +p->pid;
    }
  }

  release(&ptable.lock);
  return children;
}
```

```c
int num_of_degits(int num)
{
  int digits = 0;
  while(num > 0)
  {
    num = num/10;
    digits++;
  }
  return digits;
}

int pow(int ten, int n)
{
  for(int i=0;i<n-1;i++)
  {
    ten = ten * ten;
  }
  return ten;
}
```

```
int
get_children(int pid)
{
  int cur_pid;
  int num_of_cur_children;
  int cur_children;
  int all_children = get_process_children(pid);
  int num_of_all_children = num_of_degits(all_children);
  int final = all_children;
  int i=0;

  if(num_of_all_children == 1 && all_children == 0)
  {
      return 0;
  }
  while(i <= num_of_all_children)
  {
    if(all_children < 10){
      cur_pid = all_children;
    }
    else
    {
      cur_pid = all_children / pow(10,(num_of_all_children-1));
    }

    cur_children = get_process_children(cur_pid);
    num_of_cur_children = num_of_degits(cur_children);

    if(cur_children != 0)
    {
      final = final* pow(10,num_of_cur_children) + cur_children;
      all_children = all_children* pow(10,num_of_cur_children) + cur_children;
       num_of_all_children = num_of_all_children + num_of_cur_children;
      i = i+1;

    }
    else
    {
      i = i+1;
      num_of_cur_children = 0;
    }
    all_children = all_children % pow(10,num_of_all_children-1);
    num_of_all_children = num_of_all_children  - 1;
  }
  return final;
}
```

برنامه سطح کاربر اضافه شده:

```
int
main(int argc, char *argv[])
{
  int res;
  if(fork() == 0)
  {
    if(fork()!=0)
    {
      wait();
    }
    else
    {
     sleep(100);
    }

  }
  else {
    if(fork()!=0){
      wait();
    }
    else
    {
      sleep(500);
    }
    wait();
  }
  if(getpid()==6){
    res = get_children(2);
    printf(1, "children: %d\n", res);
  }
  exit();
}
```