



دانشکده مهندسی کامپیوتر  
سیستم های عامل  
پاییز 1401  
فاز اول

نام استاد : دکتر رضا انتظاری

پوریا رحیمی  
99521289  
فرناز خوش دوست آزاد  
99521253

افزودن یک System Call به معماری XV6

## What are system calls


A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS

حال برای اضافه کردن یک سیستم کال به اسم `proc_dump` در ابتدا طبق داک پروژه ، ریپازیتوری را کلون کرده و بعد از اینکه کلون کردن با موفقیت انجام شد یک سری فایل داریم که باید برای اضافه کردن سیستم کال آن ها را ویرایش کنیم. اسامی فایل هایی که باید ویرایش شوند عبارتند از :

`syscall.h , syscall.c , sysproc.c , usys.S , user.h , proc.c , defs.h , types.h`


در ابتدا از این فایل شروع می کنیم ، در این فایل به هر سیستم کال یک عدد assigne شده است. در این فایل 21 سیستم کال از قبل تعریف شده است ، ما نیز خط 22 را برای سیستم کال خودمان قرار می دهیم و سیستم کالی به اسم SYS\_proc\_dump را در فایل ایجاد می کنیم :

```
// System call numbers
#define SYS_fork      1
#define SYS_exit      2
#define SYS_wait      3
#define SYS_pipe      4
#define SYS_read       5
#define SYS_kill       6
#define SYS_exec       7
#define SYS_fstat      8
#define SYS_chdir      9
#define SYS_dup       10
#define SYS_getpid    11
#define SYS_sbrk      12
#define SYS_sleep     13
#define SYS_uptime    14
#define SYS_open      15
#define SYS_write     16
#define SYS_mknod     17
#define SYS_unlink    18
#define SYS_link      19
#define SYS_mkdir     20
#define SYS_close     21
#define SYS_proc_dump 22
```




سپس در این فایل باید پوینتری را به سیستم کالمان اضافه کنیم که این را در فایل `syscall.c` اضافه می کنیم. این فایل محتوی آرایه ای از پوینتر های تابع است که از `index` ها به عنوان پوینتر برای فراخوانی سیستمی استفاده می کند که در مکان های مختلف تعریف شده اند و باید توجه داشت که این ها در سطح `kernel` می باشند که در عکس زیر می بینیم :

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_proc_dump] sys_proc_dump
};
```



همچنین این بدان معناست که وقتی فراخوانی سیستم با شماره تماس سیستمی 22 انجام می شود ، تابعی که با اشاره گر تابع sys\_proc\_dump نشان داده شده است فراخوانی می شود. بنابراین ما آن را در اینجا درست مانند 21 تماس سیستمی دیگر پیاده سازی می کنیم. در عکس زیر آن را مشاهده می کنید:

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern proc_info* sys_proc_dump(void);
```



## : sysproc.c


سپس زمانی که سیستم کال خود را تعریف کردیم حال باید آن را به فایل sysproc.c اضافه کرد که به صورت زیر آن را به این فایل اضافه می کنیم :

```
proc_info *  
sys_proc_dump()  
{  
    return SortProcesses();  
}
```

## : usys.s

این فایل حاوی رابطی برای دسترسی برنامه کاربری ما به test.c می باشد تا به سیستم کال ما دسترسی داشته باشد برای همین مانند 21 سیستم کالی که از قبل در این فایل تعریف شده اند سیستم کالی دیگر به اسم proc\_dump اضافه می کنیم :


```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(proc_dump)
```



: user.h

حال در این فایل باید proc\_dump ای که تعریف کردیم را پیاده سازی کنیم. این تابع به ما proc\_info های sort شده را بر می گرداند :

```
// ulib.c
int stat(const char *, struct stat *);
char *strcpy(char *, const char *);
void *memmove(void *, const void *, int);
char *strchr(const char *, char c);
int strcmp(const char *, const char *);
void printf(int, const char *, ...);
char *gets(char *, int max);
uint strlen(const char *);
void *memset(void *, int, uint);
void *malloc(uint);
void free(void *);
int atoi(const char *);
proc_info *proc_dump(void);
```




: defs.h

در فایل defs.h باید در بخشی که مربوط به proc.c می شود ، signature تابع SortProcesses را تعریف می کنیم  
بعد از اینکه تابع را تعریف کردیم بعدش نوبت به پیاده سازی تابع SortProcesses می رسد :



```
// PAGEBREAK: 16
// proc.c
int cpuid(void);
void exit(void);
int fork(void);
int growproc(int);
int kill(int);
struct cpu *mycpu(void);
struct proc *myproc();
void pinit(void);
void procdump(void);
void scheduler(void) __attribute__((noreturn));
void sched(void);
void setproc(struct proc *);
void sleep(void *, struct spinlock *);
void userinit(void);
int wait(void);
void wakeup(void *);
void yield(void);
proc_info *SortProcesses(void);
```



: proc.c

حال نوبت به پیاده سازی تابع SortProcesses می باشد ، در این تابع ما ابتدا روی process ها که در یک ptable وجود دارد قبل از اینکه حلقه بزنی باید آن ها را قفل کرد که کار دیگه ای روی آن انجام نشود تا زمانی که process table را release کنیم پس به منظور این کار همون اول روی آن ها acquire می کنیم تا آن ها را lock کنیم تا از این اتفاق جلوگیری کنیم سپس بعد از این کار باید یک حلقه ایجاد کنیم به ازای تمام process ها و یه if می داریم که اگر state ما در حالت RUNNING یا RUNNABLE بود در آرایه process ها ذخیره شود.

سپس چون دیگه با ptable کاری نداریم می توانیم آن را release کنیم که عملیات های دیگری که می خواستند انجام بشنود ولی بخاطر lock که زده بودیم نمی تونستند انجام بشنود ، حال بتوانند انجام بشنود. سپس با یک bubblesort معمولی process ها را sort می کنیم. سپس مقایسه انجام می دهد که اگر memsize یک process بزرگتر از process دیگری بود آن را به پایین منتقل کند تا sort ما از کوچک به بزرگ از بالا به پایین مرتب شود و در آخر نیز process های sort شده را پرینت می کند و آن ها را برمی گردانیم. فقط باید این نکته را توجه داشت که ما در سطح kernel می باشیم و printf در سطح kernel ناشناخته می باشد برای همین باید از cprintf استفاده کنیم :

```
proc_info *SortProcesses()
{
    int size = 0;
    struct proc *p;
    int len = 100;
    proc_info result[len];
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if (size > len)
            break;
        if (p->state != RUNNING && p->state != RUNNABLE)
            continue;
        proc_info pro; pro.pid = p->pid;
        pro.memsize = p->sz; result[size] = pro;
        size += 1;
    }
    len = size;
    release(&ptable.lock);
    Bubblesort(result, len);
    for (int i = 0; i < len; i++){
        if (result[i].pid != 0)
            cprintf("pid : %d => memsize : %d\n", result[i].pid, result[i].memsize);
            cprintf("_____ \n");
    }
    return result;
}
```

```

void pswap(proc_info *p1, proc_info *p2){
    proc_info tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}

void Bubblesort(proc_info *p, int len){
    for (int i = 0; i < len; i++){
        for (int j = i + 1; j < len; j++){
            if (p[i].memsize > p[j].memsize){
                pswap(&p[i], &p[j]);
            }
        }
    }
}

```

: types.h

اینجا در این فایل ما type سیستم کال خود را مطابق با آن چیزی که در داک گفته شده است پیاده سازی می کنیم و در فایل types.h قرار می دهیم :

```

struct ProcessInfo
{
    int pid;
    int memsize;
};
typedef struct ProcessInfo proc_info;

```

: test.c

زمانی که به طور کامل سیستم کال خود را بر روی معماری XV6 پیاده سازی کردیم ، نوبت تست کردن آن سیستم کال می باشد. در ابتدا کاربر تعداد فرآیند هایی را که می خواهد به آرایه اضافه کند به عنوان آرگومان به این برنامه وارد می کند. برخی از فرآیندها را فورک می کند و مقدار حافظه متفاوتی را روی هر یک از آن ها ذخیره می کند. و قبل از پایان این فرآیند ، سیستم proc\_dump را فراخوانی می کند تا آرایه proc\_info را پر کند. سپس این آرایه را چاپ می کند. (که البته با استفاده از سیستم کال ما سورت شده می باشد) البته که ما باید برای اینکه از فرآیند های zombie جلوگیری کنیم آن ها را بکشیم و منتظر فرزند آوری باشیم که این اتفاق رخ ندهد.

## : test.c

ابتدا وقتی که یک عدد را از ورودی به عنوان تعداد process ها گرفتیم یک حلقه به همین تعداد ایجاد می کنیم تا بتوانیم به همین اندازه process ایجاد کنیم و در هر iteration از fork استفاده می کنیم تا یک process جدید ایجاد شود. حال زمانی که این process ها را تولید کردیم یک عدد random به عنوان حافظه به آن ها می دهیم مقدار آن حافظه را با set ، memset می کنیم. سپس چون که خواسته ما این است که process ها در حالت RUNNING یا RUNNABLE بماند یک (1)while می گذاریم که شرطش همیشه true می باشد و process ها به اتمام نمی رسند و ما به خواسته خود می رسیم. بعد از اینکه حلقه ما تمام شد جهت اطمینان برای تمامی process ها یک sleep ایجاد می کنیم تا همه process ها با هم آماده باشند. سپس تابع proc\_dump() را که از قبل تعریف کرده بودیم صدا می زنیم تا process ها را sort کند ، سپس process ها را kill می کنیم و سپس با دستور exit() برنامه به پایان می رسد.

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "user.h"
#include "x86.h"


int main(int argc, const char *argv[]){
    int pcount = atoi(argv[1])-1;
    int pids[NPROC];
    for (int i = 0; i < pcount; ++i){
        pids[i] = fork();
        if(!pids[i]){
            int random_num = ((i+1)*123456+(pcount-i)*654321);
            char *data = malloc(sizeof(char) * random_num);
            memset(data, random_num, sizeof(char) * random_num);
            while (1)
                ;}
        }
        sleep(25);
        proc_dump();
        for (int i = 0; i < pcount; ++i){
            kill(pids[i]);
            wait();
        }exit();
    }
}
```

## : Makefile


حال همانطور که در داک پروژه توضیح داده شده است ، نام فایل تست خودم را به UPROGS و EXTRA اضافه کردم تا کاربر بتواند با وارد کردن "test" برنامه را اجرا کند و سپس تست را وارد کند و برنامه اجرا بشود :

```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
_init\
_kill\
_ln\
_ls\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_test\
_zombie\

fs.img: mkfs README $(UPROGS)
./mkfs fs.img README $(UPROGS)
```



```
EXTRA=\
mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c test.c zombie.c\
printf.c umalloc.c\
README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
.gdbinit.tmpl gdbutil\
```





## : output

پس از اینکه تمامی مراحل (تعریف سیستم کال – ایجاد کردن سیستم کال – تعریف تابع سورت و ...) به پایان رسید حال باید برنامه خود را با توجه به تستی که برای آن نوشتیم تست کنیم و یک خروجی از آن بگیریم تا مطمئن شویم که سیستم کال ما به درستی کار می کند و process ها را با توجه به sort ، memsize می کند و در خروجی به ما نشان می دهد :

```
$ test 10
pid : 10  => memsize : 12288
-----
pid : 19  => memsize : 1777728
-----
pid : 18  => memsize : 2308592
-----
pid : 17  => memsize : 2839456
-----
pid : 16  => memsize : 3370320
-----
pid : 15  => memsize : 3901184
-----
pid : 14  => memsize : 4432048
-----
pid : 13  => memsize : 4962912
-----
pid : 12  => memsize : 5493776
-----
pid : 11  => memsize : 6024648
-----
$ _
```

<https://stackoverflow.com/questions/47744641/how-to-initialize-a-globe-struct-in-xv6>

<https://www.geeksforgeeks.org/xv6-operating-system-adding-a-new-system-call/>

<https://www.guru99.com/system-call-operating-system.html>

## : resources



با تشکر از توجه شما