

به نام خدا

اعضای گروه : فرزانه رحمانی ۹۹۵۲۱۲۷۱

گلبرگ سپهرآرا ۹۹۵۲۱۳۳۴

همان طور که میدانید موضوع پروژه اضافه کردن یک سیستم کال جدید به سیستم عامل xv6 میباشد. ابتدا با تعدادی سوال مواجه میشویم و نیاز به مطالعه و سرچ برای یافتن جواب سوال های خود داریم. سوال هایی از قبیل :

۱. سیستم عامل xv6 چیست ؟

۲. سیستم کال چیست؟

۳. چگونه یک سیستم کال را به سیستم عامل باید اضافه کنیم؟

با فشردن . + ctrl لینک هایی که در یافتن پاسخ سوالات بالا گروهمان استفاده کرده و میتوان گفت مفید اند را میتوان مشاهده کرد.

برای اضافه کردن سیستم کال proc_dump باید فایل های زیر را تغییر دهیم :

How to add system call in xv6 ?

There are already some system calls in xv6 operating system. Here, we see how can we add our own system call in xv6.

For adding system call in xv6,

we need to modify following files :

- 1) syscall.c
- 2) syscall.h
- 3) sysproc.c
- 4) usys.S
- 5) user.h

در syscall.c :

```
83 }
84
85 extern int sys_chdir(void);
86 extern int sys_close(void);
87 extern int sys_dup(void);
88 extern int sys_exec(void);
89 extern int sys_exit(void);
90 extern int sys_fork(void);
91 extern int sys_fstat(void);
92 extern int sys_getpid(void);
93 extern int sys_kill(void);
94 extern int sys_link(void);
95 extern int sys_mkdir(void);
96 extern int sys_mknod(void);
97 extern int sys_open(void);
98 extern int sys_pipe(void);
99 extern int sys_read(void);
100 extern int sys_sbrk(void);
101 extern int sys_sleep(void);
102 extern int sys_unlink(void);
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 extern int sys_proc_dump(void);
```

در syscall.h :

```
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_proc_dump 22
```

در sysproc.c :

```
92
93 int
94 sys_proc_dump(void)
95 {
96     sort_proc();
97     return 0;
98 }
```

در usys.s :

```
23 SYSCALL(fstat)
24 SYSCALL(link)
25 SYSCALL(mkdir)
26 SYSCALL(chdir)
27 SYSCALL(dup)
28 SYSCALL(getpid)
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 SYSCALL(proc_dump) ←
```

در user.h :

```
19 int mkdir(const char*);
20 int chdir(const char*);
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 int proc_dump(int , void*); ←
27
```

همچنین برای اینکه در ترمینال موقع اجرای فایل ، سیستم کال ما را هم بشناسد در فایل Makefile،فایل Proc_dump.c را در محل مشخص شده در شکل معرفی میکنیم.

```
167
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _proc_dump\ ←
185
186 fs.img: mkfs README $(UPROGS)
187     ./mkfs fs.img README $(UPROGS)
188
189 -include *.d
```

user level test program

Proc_dump.c : این در واقع تابع تستمان است . در مین این فایل چندین فورک میزنیم اگر چاپلند بود با استفاده از فور با رنج بالا ، به جای sleep ، کمی زمان صرف میکنیم تا پروسس تمام نشود و هنگام سورت کردن وجود داشته باشد. داخل فورک ها ، مقادیری متفاوت از حافظه را با دستور malloc میگیریم تا بتوانیم تست کنیم .

```
C sysproc.c M C proc_dump.c M X C proc.c M
C proc_dump.c > main(int, char * [])
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "param.h"
5
6 int main(int argc, char *argv[])
7 {
8     int r = fork();
9     if (r == 0)
10     {
11         r = fork();
12         if (r != 0)
13         {
14             // int: 4 bytes
15             // 32 bytes of memory block
16             // malloc(8 * sizeof(int));
17             malloc(100000);
18             for (unsigned int i = 0; i < 0xFFFFFFFF; i++)
19             {
20             }
21             for (unsigned int j = 0; j < 0xFFFFFFFF; j++)
22             {
23                 for (unsigned int k = 0; k < 0xFFFFFFFF; k++)
24                 {
25                 }
26             }
27         }
28         else
29         {
30             // 40 bytes of memory block
31             // malloc(10 * sizeof(int));
32             // malloc(10000);
33             malloc(100000);
34             for (unsigned int i = 0; i < 0xFFFFFFFF; i++)
35             {
36             }
37             for (unsigned int j = 0; j < 0xFFFFFFFF; j++)
38             {
39                 for (unsigned int k = 0; k < 0xFFFFFFFF; k++)
40                 {
41                 }
42             }
43         }
44         wait();
45     }
46     else
```

```

proc_dump.c > main(int, char *[])
34     wait();
35 }
36 else
37 {
38     r = fork();
39     if (r == 0)
40     {
41         // 60 bytes of memory block
42         // malloc(15 * sizeof(int));
43         malloc(30000);
44         for (unsigned int i = 0; i < 0xFFFFFFFF; i++)
45         {
46         }
47         for (unsigned int j = 0; j < 0xFFFFFFFF; j++)
48         {
49             for (unsigned int k = 0; k < 0xFFFFFFFF; k++)
50             {
51             }
52         }
53         // wait();
54     }
55     else
56     {
57         // 20 bytes of memory block
58         // malloc(5 * sizeof(int));
59         malloc(40000);
60         int maxProcess = NPROC; // max number of processes = 64
61         struct proc_info *processes = malloc(maxProcess * sizeof(struct proc_info));
62
63         int Number = proc_dump(maxProcess, processes);
64         for (int i = 0; i < Number; i++)
65         {
66             // printf(2, processes[i].pid, processes[i].memsize);
67             printf(1, "memsize:%d -- process_id:%d\n", processes[i].memsize, processes[i].pid);
68         }
69         wait();
70     }
71 }
72
73 wait();
74 wait();
75 exit();
76 return 0;
77 }

```

result array

در این فایل همچنین یک مدل دیگر تست نوشته شده است که در آن از ورودی تعداد پراسس ها و سائیزشان را گرفته و با توجه به آن فور و malloc مان را مینویسیم. در هر حلقه فور یکبار عمل فورک را انجام داده و اگر چایلد باشد با توجه به سائیز داده شده در ورودی ، مموری allocate میکنیم. دلیل وجود while(true) این است که چایلد سریع تمام نشود ، اگر از فور استفاده کنیم باز هم به این مشکل بر خواهیم خورد. اگر چایلد نبود ، (r>0) ، در آن صورت پرنتمان باید تابع proc_dump را صدا کند.

```
int main(int argc, char *argv[])
{
    /// proc_dump 5 10000 25000 25000 50000 68000 (5 processes add to current)
    for (int i = 0; i < argc; i++)
    {
        int pid = fork();
        if (pid == 0)
        {
            malloc(atoi(argv[i]));

            while (1)
            {
                // long tmp = 0;
                // for (unsigned int j = 0; j < 0xFFFFFFFF; j++)
                // {
                //     for (unsigned int k = 0; k < 0xFFFFFFFF; k++)
                //     {
                //         tmp += j * k;
                //     }
                // }
                // }
                exit();
            }
        }
    }

    int maxProcess = NPROC; // max number of processes = 64
    struct proc_info *processes = malloc(maxProcess * sizeof(struct proc_info));

    int Number = proc_dump(maxProcess, processes);
    for (int i = 0; i < Number; i++)
    {
        // printf(1, "memsize:%d -- process_id:%d\n", processes[i].memsize, processes[i].pid);
        char str1[50];
        char str2[50];
        itoa(str1, processes[i].memsize);
        itoa(str2, processes[i].pid);
        printf(1, "memsize:");
        printf(1, str1);
        printf(1, " -- process_id:");
        printf(1, str2);
        printf(1, "\n");
    }

    // for (int i = 0; i < argc; i++)
    // {
    //     wait();
    // }

    // for (int i = 0; i < argc; i++)
    // {
    //     kill(pids[i]);
    // }

    exit();
}
```

سیستم کال proc_dump، تابع sort_proc را صدا میزند که در آن دو بار سورت انجام میشود و هر دو sort از نوع BubbleSort هستند. اولویت اولمان سورت بر اساس memsize به طوری که پروسس ها را از memsize کوچک به بزرگ مرتب میکنیم، اگر دو پروسس از نظر memsize یک اندازه بودند، بر اساس pid از کوچک به بزرگ مرتب میکنیم.

```
541 // struct proc_info*
542 int
543 sort_proc(void)
544 {
545     int max;
546     int count = 0;
547     struct proc *p;
548     struct proc_info *result ;
549
550     argint(0, &max);
551     argptr(1, (char **)&result, max*sizeof(struct proc_info));
552
553     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
554         if(p->state == UNUSED) continue;
555         if(p->state == RUNNABLE || p->state == RUNNING){
556             result[count].pid = p -> pid;
557             result[count].memsize = p -> sz;
558             count ++;
559         }
560     }
561
562     // sort by memsize and pid
563     for(int i = 0; i < count; i++){
564         for(int j = i + 1; j < count; j++){
565             if(result[i].memsize > result[j].memsize){
566                 struct proc_info temp = result[i];
567                 result[i] = result[j];
568                 result[j] = temp;
569             }
570             else if (result[i].memsize == result[j].memsize)
571             {
572                 if(result[i].pid > result[j].pid){
573                     struct proc_info temp = result[i];
574                     result[i] = result[j];
575                     result[j] = temp;
576                 }
577             }
578         }
579     }
580
581     for(int i = 0; i < count; i++){
582         cprintf("memsize:%d --process_id:%d\n", result[i].memsize, result[i].pid);
583     }
584     return count;
585 }
```

چالش ها :

۱. یکی از چالش هایی که با آن رو به رو شدیم این بود که پس از اجرای سیستم کال ، چایلد هایی که مربوط به تولید اعداد میشدند ، تبدیل به زامبی میشدند ، برای حل این مشکل گفتیم برای پراسس پرنت باید WAIT بذاریم ، یعنی بگوییم اگر R>۰ بود ، باید صبر کند تا کار بچه ها تمام شود.
۲. یکی دیگر از مشکلاتی که داشتیم این بود که چون رنج for هایمان برای ملوک کم بود (اعداد ۳۰۰ – ۵۰۰) ، memsize های که میداد همه یکی بودند ، برای این منظور محدوده ی فور را بیشتر کردیم.

نمونه ای از خروجی برنامه :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE

memsize:85064 --process_id:11
memsize:112296 --process_id:12
$ proc_dump
memsize:12288 --process_id:18
memsize:45056 --process_id:17
memsize:85064 --process_id:15
$ proc_dump
memsize:85064 --process_id:19
$ proc_dump
memsize:12288 --process_id:25
memsize:12288 --process_id:26
memsize:85064 --process_id:23
$ proc_dump
memsize:12288 --process_id:29
memsize:12288 --process_id:30
memsize:85064 --process_id:27
$ proc_dump
memsize:12288 --process_id:33
memsize:12288 --process_id:34
memsize:85064 --process_id:31
memsize:112296 --process_id:32
$ proc_dump
memsize:12288 --process_id:36
memsize:12288 --process_id:38
memsize:45056 --process_id:37
memsize:85064 --process_id:35
$ □
```

خروجی تست اول


```
Booting from Hard Disk...
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ proc_dump 5 10000 10000 40000 90000 150000
memsize:12288 --process_id:10
memsize:45056 --process_id:3
memsize:45056 --process_id:4
memsize:45056 --process_id:5
memsize:45056 --process_id:6
memsize:45056 --process_id:7
memsize:52296 --process_id:8
memsize:102296 --process_id:9
$
```

خروجی تست دوم

پایان !