به نام خدا

اعضای گروه : فرزان رحمانی ۹۹۵۲۱۲۷۱

گلبرگ سپهرآرا ۹۹۵۲۱۳۳٤

همان طور که میدانید موضوع پروژه اضافه کردن یک سیستم کال جدید به سیستم عامل xv٦ میباشد.
ابتدا با تعدادی سوال مواجه میشویم و نیاز به مطالعه و سرچ برای یافتین جواب سوال های خود داریم.

سوال هایی از قبیل :

۱. سیستم عامل xv٦ 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 :

```
syscall.c
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 :

```
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 :

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

در usys.s :

```
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(proc_dump)
```

در user.h :

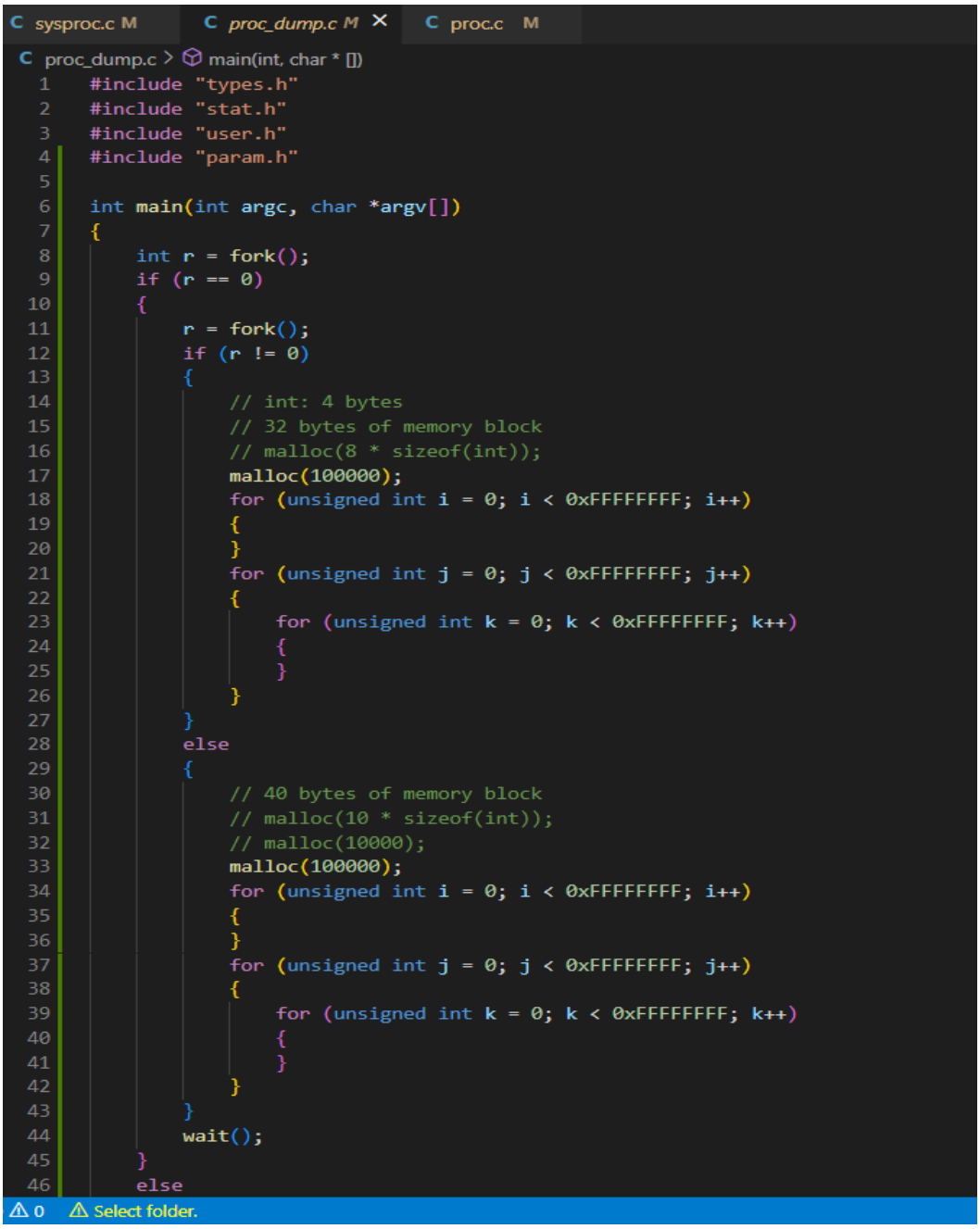
```
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int proc_dump(int , void*);
```

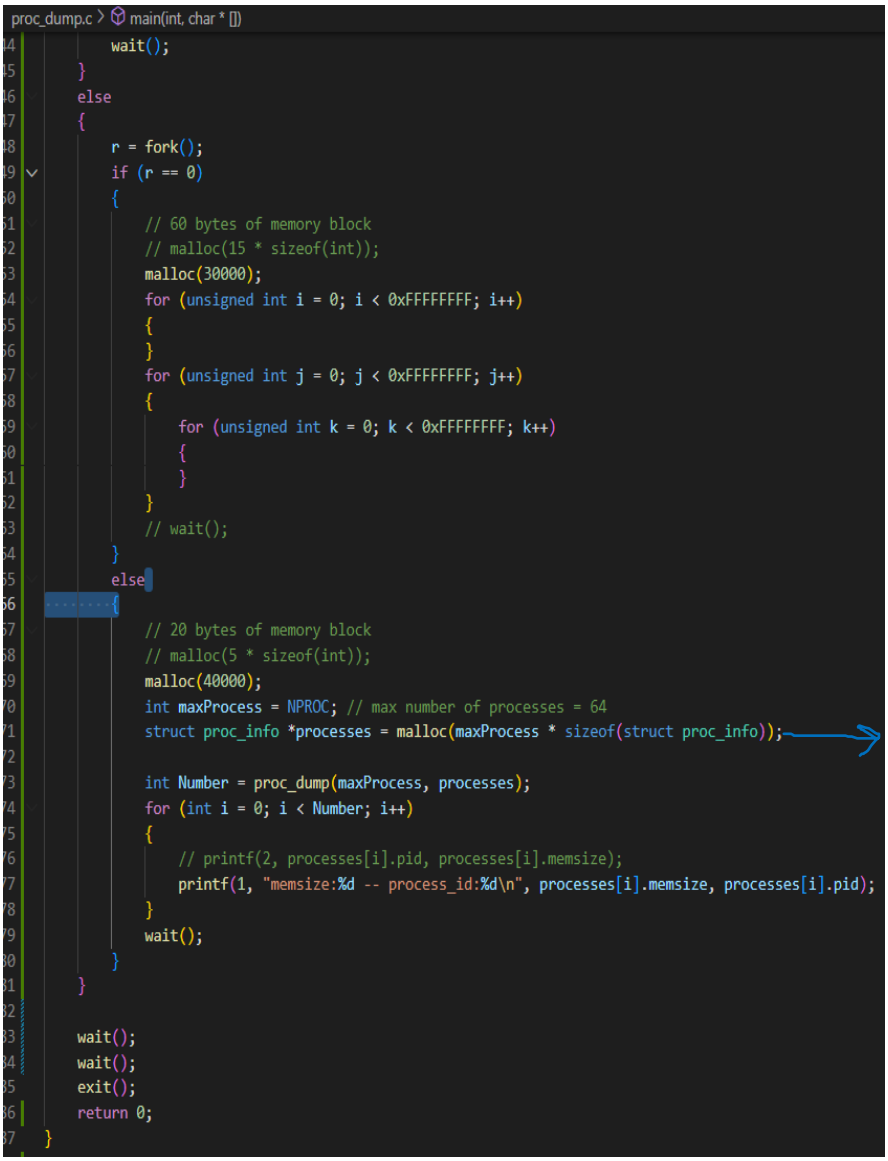
همچنین برای اینکه در ترمینال موقع اجرای فایل ، سیستم کال ما را هم بشناسد در فایل Makefile،فایل Proc_dump.c را در محل مشخص شده در شکل معرفی میکنیم.

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

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

-include *.d
```

user level test program

Proc_dump.c : این در واقع تابع تستمان است . در مین این فایل چندین فورک میزنیم اگر چایلد 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 .




proc_dump.c > main(int, char * [])
wait();
}
else
{
r = fork();
if (r == 0)
{
// 60 bytes of memory block
// malloc(15 * sizeof(int));
malloc(30000);
for (unsigned int i = 0; i < 0xFFFFFFFF; i++)
{
}
for (unsigned int j = 0; j < 0xFFFFFFFF; j++)
{
for (unsigned int k = 0; k < 0xFFFFFFFF; k++)
{
}
}
// wait();
}
else
{
// 20 bytes of memory block
// malloc(5 * sizeof(int));
malloc(40000);
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(2, processes[i].pid, processes[i].memsize);
printf(1, "memsize:%d -- process_id:%d\n", processes[i].memsize, processes[i].pid);
}
wait();
}
}
wait();
wait();
exit();
return 0;
}
result
array

در این فایل همچنین یک مدل دیگر تست نوشته شده است که در آن از ورودی تعداد پراسس ها و سایزشان 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 سایز داده شده در ورودی ، ممووری 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>۰) ، در آن صورت پرنتمان باید تابع proc_dump را صدا کند.

```c
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 پراسس از نظرmemsize یک اندازه بودند ، بر اساس pid از کوچک به بزرگ مرتب میکنیم.

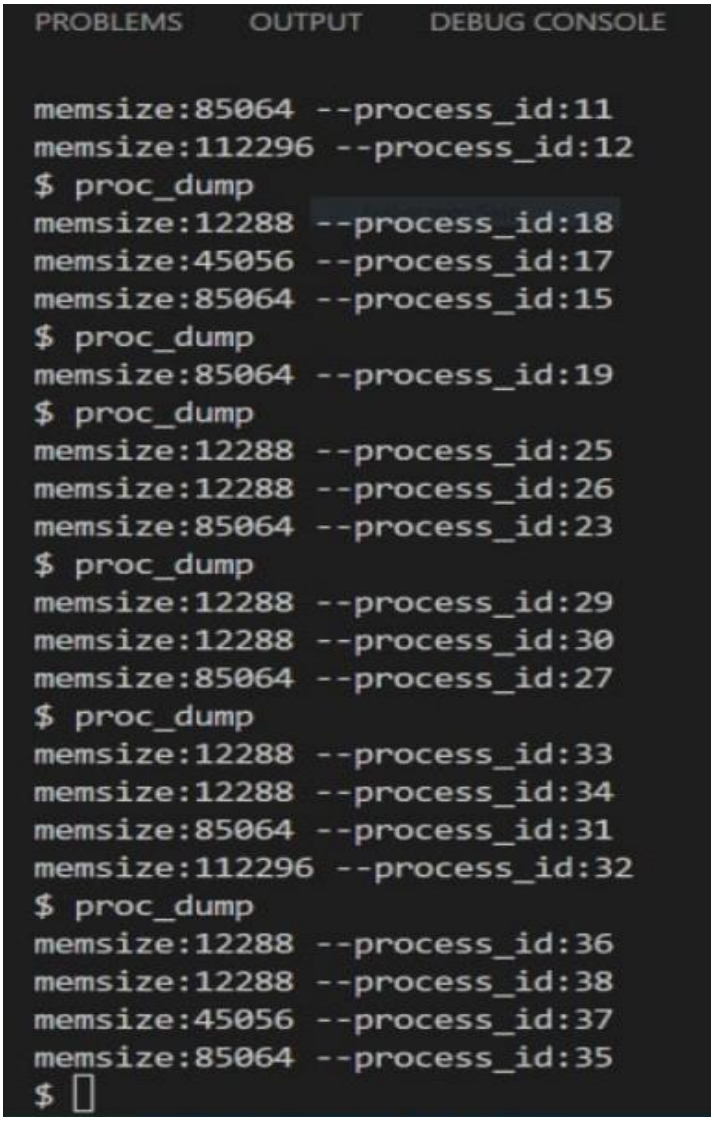
```
541  // struct proc_info*
542  int
543  sort_proc(void)
544  {
545    int max;
546    int count = 0;
547    struct proc *p;
548    struct proc_info *result ;
549
550    argint(0, &max);
551    argptr(1, (char **)&result, max*sizeof(struct proc_info));
552
553    for(p = ptable.proc; p < &ptable.proc[NPROC]; 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 :

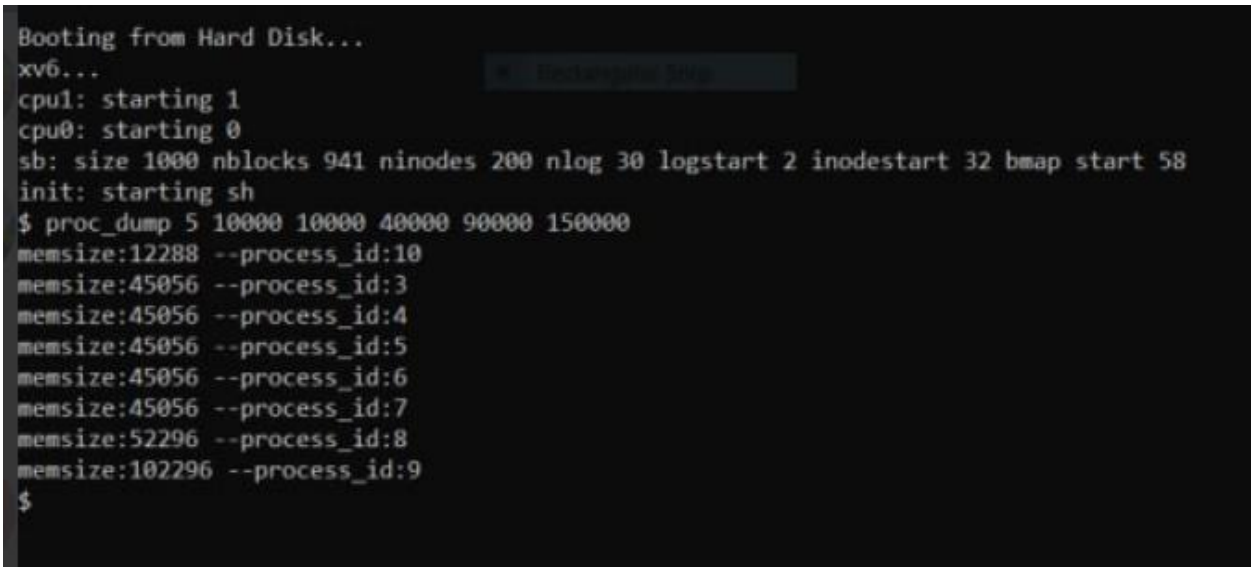
۱. یکی از چالش هایی که با آن رو به رو شدیم این بود که پس از اجرای سیستم کال ، چایلد هایی که مربوط به تولید اعداد میشدند ، تبدیل به زامبی میشدند ، برای حل این مشکل گفتیم برای پراسس پرنت بایدWAIT بذاریم ، یعنی بگوییم اگر R>۰ بود ، باید صبر کند تا کار بچه ها تمام شود.

۲. یکی دیگر از مشکلاتی که داشتیم این بود که چون رنج for هایمان برای ملوک کم بود (اعداد ۳۰۰ – ۵۰۰)، memsize هایی که میداد همه یکی بودند ، برای این منظور محدوده ی فور را بیشتر کردیم.

نمونه ای از خروجی برنامه :



خروجی تست اول



خروجی تست دوم

پایان !