به نام خدا

سیستم عامل

دکتر انتظاری

پروژه اول

علی عطاریان - ۹۹۵۲۱۴۵۱

امیرحسین میرمحمدی – ۹۹۵۲۲۱۷۶

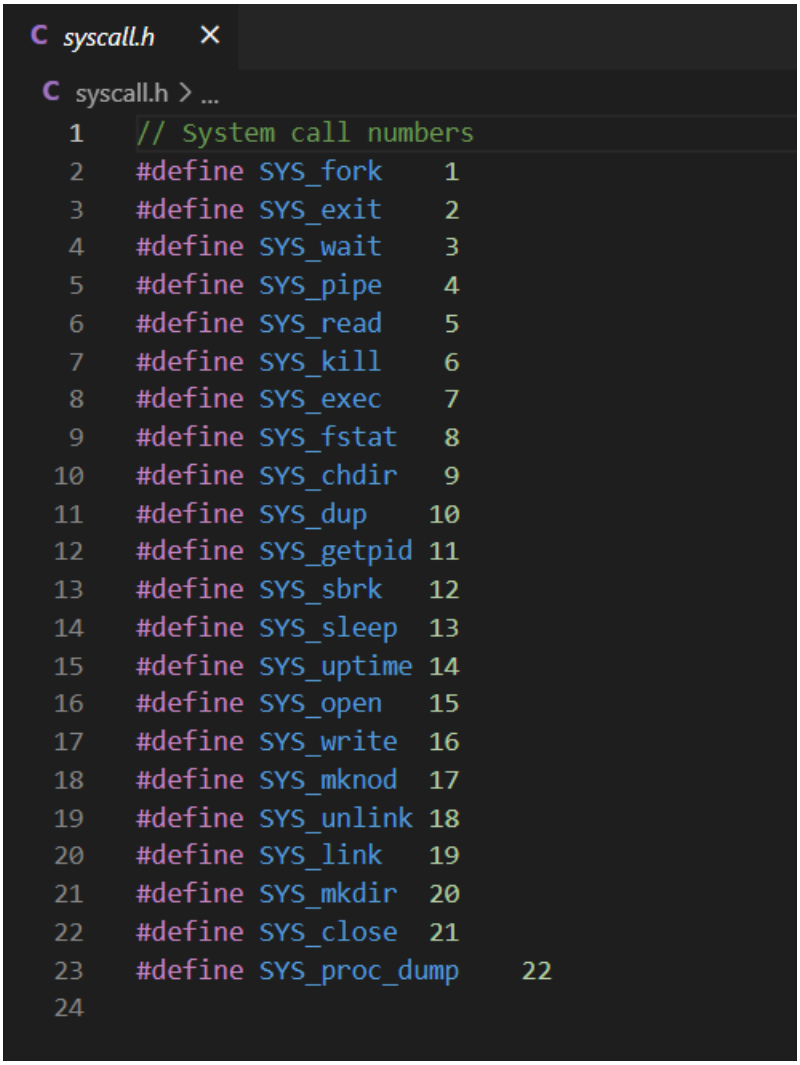
نصب سیستم عامل xv6

طبق آنچه در داک پروژه آمده است سیستم عامل xv6 را نصب می‌کنیم به طوری که نهایتا کنسول این سیستم عامل قابل مشاهده باشد.

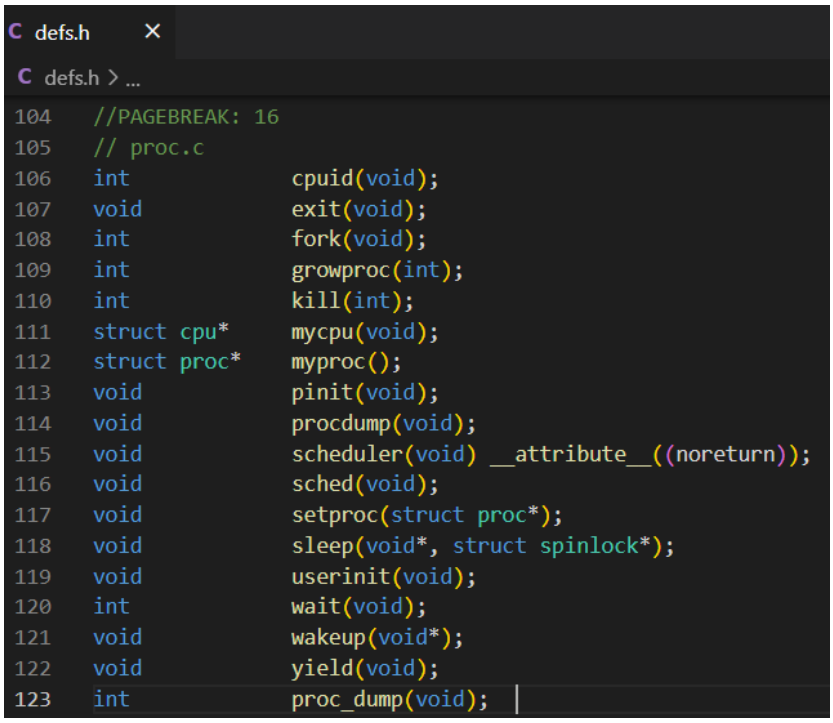
اضافه کردن سیستم کال proc_dump

بر اساس این ویدیوی آموزشی مراحل ساخت سیستم کال موردنظر را مطابق ذیل انجام می‌دهیم.

ابتدا نام سیستم کال موردنظر را با پیشوند SYS_ در فایل هدر syscall.h اضافه می‌کنیم که در خط ۲۳ کد زیر مشاهده می‌کنید.



سپس تعریف این تابع را به فایل هدر defs.h مطابق خط ۱۲۳ تصویر زیر اضافه میکنیم.

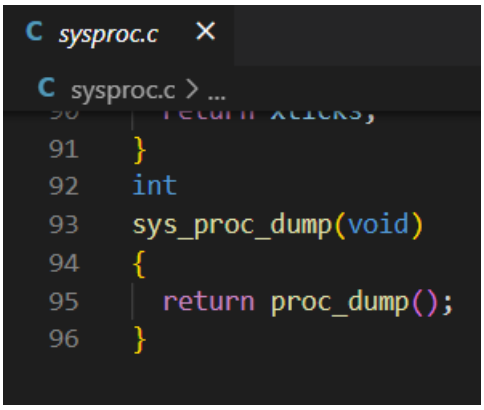


```
104   //PAGEBREAK: 16
105   // proc.c
106   int             cpuid(void);
107   void            exit(void);
108   int             fork(void);
109   int             growproc(int);
110   int             kill(int);
111   struct cpu*     mycpu(void);
112   struct proc*    myproc();
113   void            pinit(void);
114   void            procdump(void);
115   void            scheduler(void) __attribute__((noreturn));
116   void            sched(void);
117   void            setproc(struct proc*);
118   void            sleep(void*, struct spinlock*);
119   void            userinit(void);
120   int             wait(void);
121   void            wakeup(void*);
122   void            yield(void);
123   int             proc_dump(void);
```

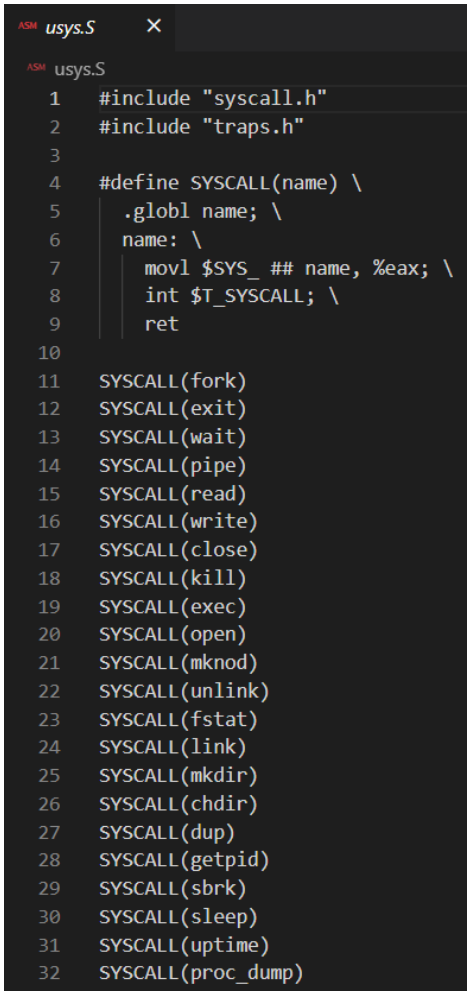
نیاز است تعریف تابع به فایل هدر user.h نیز اضافه شود.

```
1   struct stat;
2   struct rtcdate;
3
4   // system calls
5   int fork(void);
6   int exit(void) __attribute__((noreturn));
7   int wait(void);
8   int pipe(int*);
9   int write(int, const void*, int);
10  int read(int, void*, int);
11  int close(int);
12  int kill(int);
13  int exec(char*, char**);
14  int open(const char*, int);
15  int mknod(const char*, short, short);
16  int unlink(const char*);
17  int fstat(int fd, struct stat*);
18  int link(const char*, const char*);
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(void);
```

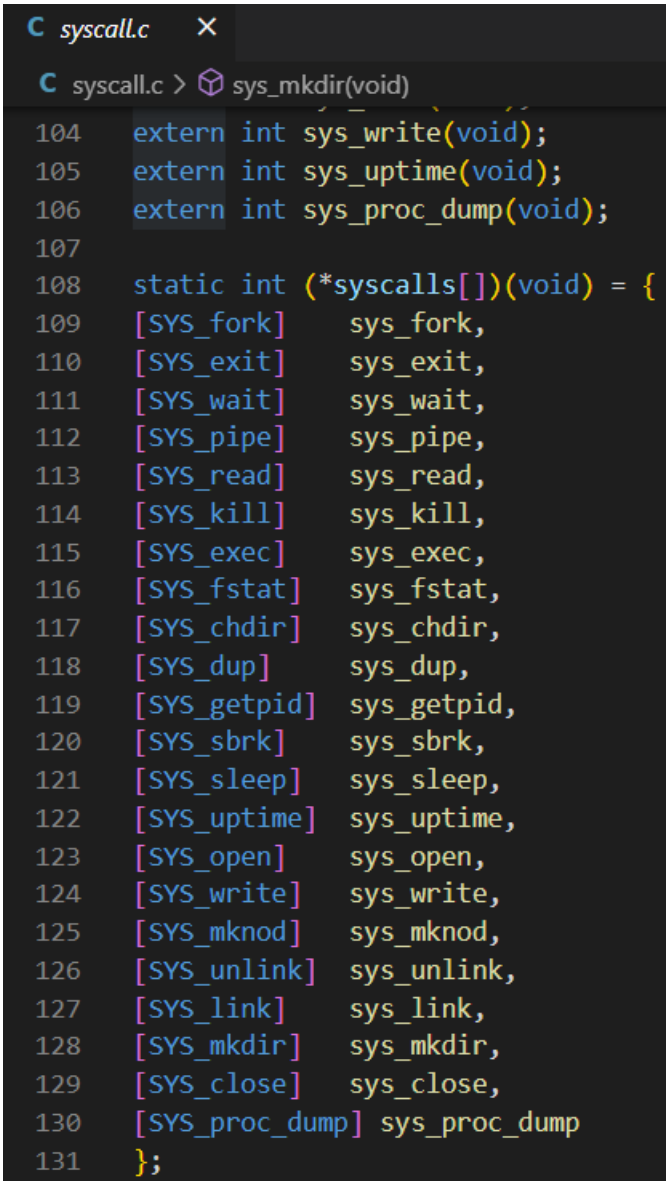
سپس در فایل sysproc.c تابع sys_proc_dump() را تعریف می‌کنیم که در واقع تنها تابع proc_dump() را فراخوانی می‌کند.



همانند فایل‌های پیشین در فایل usys.S نیز تغییر لازم را اعمال می‌کنیم.



برای اینکه سیستم کال را به سیستم عامل اضافه کنیم باید تعریف آن را در فایل syscall.c نیز درج کنیم.



```
C syscall.c
C syscall.c > sys_mkdir(void)
104   extern int sys_write(void);
105   extern int sys_uptime(void);
106   extern int sys_proc_dump(void);
107
108   static int (*syscalls[])(void) = {
109   [SYS_fork]    sys_fork,
110   [SYS_exit]    sys_exit,
111   [SYS_wait]    sys_wait,
112   [SYS_pipe]    sys_pipe,
113   [SYS_read]    sys_read,
114   [SYS_kill]    sys_kill,
115   [SYS_exec]    sys_exec,
116   [SYS_fstat]   sys_fstat,
117   [SYS_chdir]   sys_chdir,
118   [SYS_dup]     sys_dup,
119   [SYS_getpid]  sys_getpid,
120   [SYS_sbrk]    sys_sbrk,
121   [SYS_sleep]   sys_sleep,
122   [SYS_uptime]  sys_uptime,
123   [SYS_open]    sys_open,
124   [SYS_write]   sys_write,
125   [SYS_mknod]   sys_mknod,
126   [SYS_unlink]  sys_unlink,
127   [SYS_link]    sys_link,
128   [SYS_mkdir]   sys_mkdir,
129   [SYS_close]   sys_close,
130   [SYS_proc_dump] sys_proc_dump
131   };
```

تعریف تابع سیستم کال

حال خود تابع را بایست در فایل proc.c تعریف کنیم تا مشخص کنیم چه کاری انجام می‌دهد.

```c
C proc.c X
C proc.c > ...
                    swap(&arr[j], &arr[j+1]);}
552
553   }
554   int
555   proc_dump()
556   {
557     struct proc *p;
558     sti();
559     acquire(&ptable.lock);
560     struct proc_info return_process[NPROC];
561     int idx=0;
562     for(p=ptable.proc; p< &ptable.proc[NPROC];p++) {
563       if (p->state == RUNNABLE || p->state == RUNNING){
564         return_process[idx].pid = p->pid;
565         return_process[idx].memsize = p->sz;
566         idx++;
567       }
568     }
569     bubbleSort(return_process,idx);
570     for(int i=0; i<idx; i++){
571       cprintf("pid: %d ---------- memsize: %d\n",return_process[i].pid,return_process[i].memsize)
572     }
573     release(&ptable.lock);
574     return 0;
575   }
```

ابتدا در خط ۵۵۷ یک پراسس می‌سازیم که بعدا بتوانیم به وسیله آن از پراسس های ptable استفاده کنیم. خط ۵۵۸ به این معناست که اجازه interrupt دادن به پراسسور می‌دهیم. سپس قفل ptable را می‌گیریم که به دیگر پراسس ها اجازه ندهیم در این بازه به ptable دسترسی داشته باشند. باید بدانیم ptable در واقع آرایه‌ای از پراسس هاست. سپس با یک for در این آرایه می‌گردیم تا پراسس های runnable یا running را پیدا کنیم. pid و مموری سایز های مربوط به این پراسس ها را میگیریم و در آرایه‌ای از استراکت proc_info (که مطابق خواسته داک پروژه در فایل proc.h تعریف شده است) قرار می‌دهیم. در نهایت این آرایه را طبق pid و اگر مساوی بود، طبق مموری سایز مرتب می‌کنیم. در آخر قفل ptable را آزاد می‌کنیم.

proc.c

proc.c > ptable

```
534   }
535   void swap(struct proc_info* xp, struct proc_info* yp)
536   {
537       struct proc_info temp = *xp;
538       *xp = *yp;
539       *yp = temp;
540   }
541
542   void bubbleSort(struct proc_info arr[], int n)
543   {
544       int i, j;
545       for (i = 0; i < n - 1; i++)
546           for (j = 0; j < n - i - 1; j++){
547               if (arr[j].memsize > arr[j + 1].memsize)
548                   swap(&arr[j], &arr[j + 1]);
549               if (arr[j].memsize == arr[j+1].memsize)
550                 if (arr[j].pid > arr[j+1].pid)
551                   swap(&arr[j], &arr[j+1]);}
552
553   }
```

تست سیستم کال ایجاد شده

برای تست کردن proc_dump() یک فایل test_proc_dump.c ایجاد می‌کنیم که در کنسول سیستم عامل نیز باید نام همین فایل سی را در کنسول تایپ کنیم تا برنامه را اجرا کند.

test_proc_dump.c U

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

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

int
main(int argc, char* argv[])
{
    // printf(1, "hello world");
    // printf(5,"what are u doing\n");
    // printf(1,"org\n");
    int pid[2];
    pid[0] = fork();
    if (pid[0]>0){
        int* arr;
        arr = (int*) malloc(500 * sizeof(int));
        arr[5] = 9;
    }
    else{
        while(1){}
    }
    pid[1] = fork();
    if (pid[1]>0){
        int* arr1;
        arr1 = (int*) malloc(10000 * sizeof(int));
        arr1[5] = 9;
    }
    else{
        int* arr3;
        arr3 = (int*) malloc(30000 * sizeof(int));
        arr3[5] = 9;
        while(1){}
    }
    pid[2] = fork();
    if (pid[2]>0){
        int* arr2;
        arr2 = (int*) malloc(50000 * sizeof(int));
        arr2[5] = 9;
        sleep(50);
        proc_dump();
        exit();
    }
    else{
        while(1){}
    }
}
```

با ۳ فورک متوالی ۴ پراسس می‌سازیم. دلیل اینکه ۴ پراسس ساخته می‌شود این است که هر دفعه پراسس فرزند را در یک لوپ بی‌نهایت مشغول می‌کنیم. اگر از لوپ بی‌نهایت استفاده نکنیم به مشکل پراسس زامبی می‌خوریم. راه های حل دیگر مانند wait() را امتحان کردیم اما فقط while(1) مشکل را برطرف می‌کند. دلیل استفاده از sleep(50) این است که مطمئن شویم پراسس فرزند ملوک را انجام داده است.

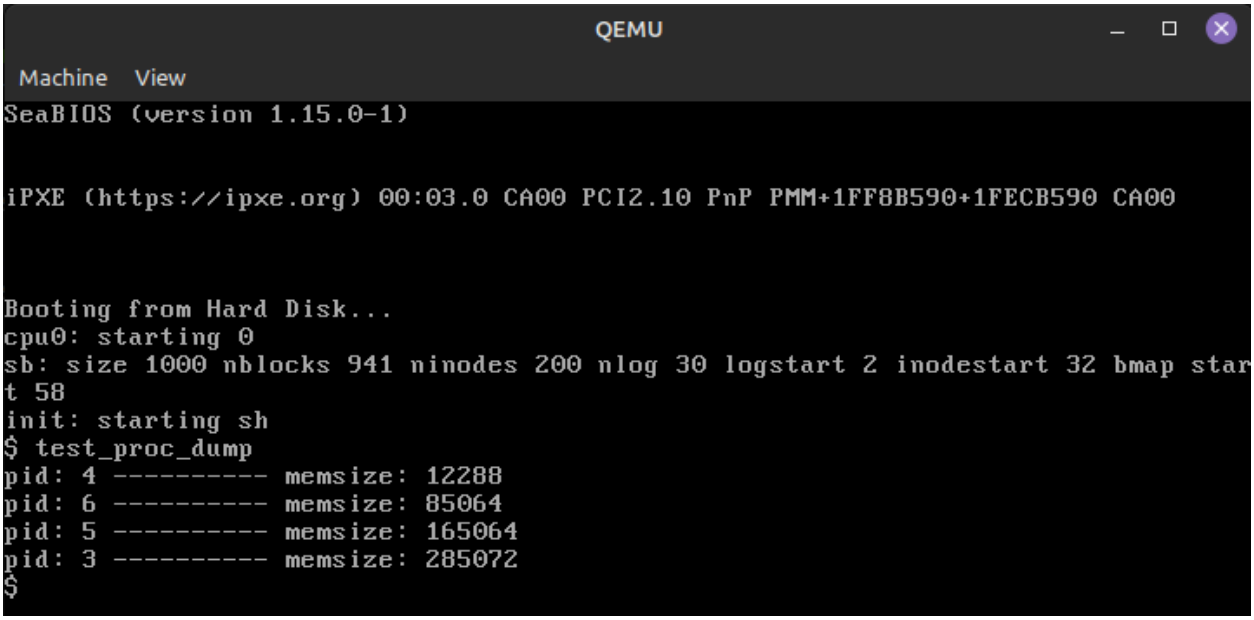
اضافه کردن تست به makefile

برای آنکه فایل سی که ایجاد کردیم در هنگام انجام دستور make ساخته شود، نیاز است تغییراتی در فایل makefile انجام بدهیم.

```
251 EXTRA=\
252     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
253     ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
254     printf.c umalloc.c test_proc_dump.c\
255     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
256     .gdbinit.tmpl gdbutil\
257
```

```
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     _test_proc_dump\
185
```

خروجی نهایی



پراسس شماره ۳ همان پراسس والد است که تا آخر می‌ماند و پس از ۳ بار گرفتن مموری بیشترین سایز را دارد. پراسس ۵ فرزندی است که از دومین فورک تولید می‌شود، که یکبار از قبل مموری گرفته و یک بار پس از تولد مموری می‌گیرد. پراسس ۶ فرزندی تولیدشده از سومین فورک است که دو بار قبل از تولد حافظه ملوک کرده است. در نهایت پراسس ۴ فرزند اولین فورک است که حافظه‌ای ملوک نکرده است.