

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 نهایتاً 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.

```
C syscall.h X
C syscall.h > ...
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
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
24
```

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

```
C defs.h x
C defs.h > ...
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.

```
C user.h x
C user.h > stat
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.

```
C sysproc.c X
C sysproc.c > ...
90 |     return xticks;
91 | }
92 | int
93 | sys_proc_dump(void)
94 | {
95 |     return proc_dump();
96 | }
```

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

```
ASM usys.S X
ASM usys.S
1 | #include "syscall.h"
2 | #include "traps.h"
3 |
4 | #define SYSCALL(name) \
5 |     .globl name; \
6 |     name: \
7 |         movl $SYS_ ## name, %eax; \
8 |         int $T_SYSCALL; \
9 |         ret
10 |
11 | SYSCALL(fork)
12 | SYSCALL(exit)
13 | SYSCALL(wait)
14 | SYSCALL(pipe)
15 | SYSCALL(read)
16 | SYSCALL(write)
17 | SYSCALL(close)
18 | SYSCALL(kill)
19 | SYSCALL(exec)
20 | SYSCALL(open)
21 | SYSCALL(mknod)
22 | SYSCALL(unlink)
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)
```

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

```
C syscall.c X
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 proc.c x
C proc.c > ...
551 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.

C proc.c X

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.

```
C test_proc_dump.c U X
C test_proc_dump.c > main(int, char * [])
1  #include "types.h"
2  #include "stat.h"
3  #include "user.h"
4  #include "fcntl.h"
5
6  int
7  main(int argc, char* argv[])
8  {
9      // printf(1, "hello world");
10     // printf(5, "what are u doing\n");
11     // printf(1, "org\n");
12     int pid[2];
13     pid[0] = fork();
14     if (pid[0]>0){
15         int* arr;
16         arr = (int*) malloc(500 * sizeof(int));
17         arr[5] = 9;
18     }
19     else{
20         while(1){}
21     }
22     pid[1] = fork();
23     if (pid[1]>0){
24         int* arr1;
25         arr1 = (int*) malloc(10000 * sizeof(int));
26         arr1[5] = 9;
27     }
28     else{
29         int* arr3;
30         arr3 = (int*) malloc(30000 * sizeof(int));
31         arr3[5] = 9;
32         while(1){}
33     }
34     pid[2] = fork();
35     if (pid[2]>0){
36         int* arr2;
37         arr2 = (int*) malloc(50000 * sizeof(int));
38         arr2[5] = 9;
39         sleep(50);
40         proc_dump();
41         exit();
42     }
43     else{
44         while(1){}
45     }
46 }
```


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. راه‌های 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

```
QEMU
Machine View
SeaBIOS (version 1.15.0-1)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00

Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test_proc_dump
pid: 4 ----- memsize: 12288
pid: 6 ----- memsize: 85064
pid: 5 ----- memsize: 165064
pid: 3 ----- memsize: 285072
$
```

پراسس شماره ۳ همان پراسس والد است که تا آخر می ماند و پس از ۳ بار گرفتن مموری بیشترین سائز را دارد. پراسس ۵ فرزندی است که از دومین فورک تولید می شود، که یکبار از قبل مموری گرفته و یک بار پس از تولد مموری می گیرد. پراسس ۶ فرزندی تولید شده از سومین فورک است که دو بار قبل از تولد حافظه ملوک کرده است. در نهایت پراسس ۴ فرزند اولین فورک است که حافظه ای ملوک نکرده است.