



به نام خدا



دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس آزمایشگاه سیستم عامل

پروژه دوم

دانیال سعیدی(810198571)

سروش صادقیان(810898048)

محمد قره حسنلو(810198461)

نام و نام خانوادگی

28/1/1401

تاریخ ارسال گزارش

ریپو گیتهاب

آخرین Commit ID

فهرست گزارش سوالات

2.....	سوال 1
3.....	سوال 2
3.....	سوال 3
3.....	سوال 4
3.....	سوال 5
4.....	بخش 0: ارسال آرگومان های فراخوانی های سیستمی
9.....	بخش 1: پیاده سازی فراخوانی سیستمی شمارش فراخوانی های سیستمی ...
16.....	بخش 2: پیاده سازی فراخوانی سیستمی پرمصرف ترین پردازه
22.....	بخش 3: پیاده سازی فراخوانی سیستمی صبر برای پردازه های دیگر
30.....	خروجی برای همه فراخوانی های سیستمی جدید

سوال 1

ULIB از چهار فایل printf، ulib و umalloc تشکیل شده است.
ulib: از چند تابع تشکیل شده است که در بعضی از آنها فراخوانی سیستمی داریم:
stat: در این تابع از open استفاده شده برای باز کردن فایل و یک خروجی fd
است و از close برای بستن فایل باز شده با file descriptor از open خروجی است.
gets: از read برای خواندن استفاده شده و چون پارامتر اول آن صفر است، از
همان fd که داخلش است، می خواند
printf: از سه تابع تشکیل شده است که در تابع puts، puts کال write استفاده
شده است که در آن کاراکتر c را در fd ورودی مینویسد.
Umalloc: در تابع sbrk از سیستم کال morecore به منظور تغییر دادن data segment
استفاده میشود.

سوال 2

علاوه بر فراخوانی سیستمی راه های دیگری برای دسترسی به هسته وجود دارد؛ مانند exception که در لینوکس دسترسی به هسته داریم تا هسته بتواند خطای از بین ببرد. راه دیگر بر مبنای سوکت است که برنامه هایی که در قسمت برای کاربر نوشته میشود، میتوانند به هسته دسترسی داشته باشند و اطلاعات را به هسته بفرستند و یا دریافت کنند. همچنین برای sys/ و proc/ ... نیاز به دسترسی هسته داریم که محتوای داخل خود هسته را به برنامه در سطح کاربر میدهیم.

سوال 3

نمیتوان فعال نمود. با فعال کردن بقیه trap ها با سطح DPL_USER که یک سطح دسترسی کاربر است، هر دستوری همزمان میتوانست به kernel دسترسی پیدا کند که مشکل امنیتی ایجاد میشد(چون ممکن است حداقل یکی از این دسترسی ها دارای باگ باشند و در kernel تاثیر بگذارد).

سوال 4

ss و esp وقتی استک بین دو سطح موجود kernel و user تغییر میکند، عوض میشوند و وقتی این push کردن انجام میشود که تغییر دسترسی انجام شود، چون از استک پشته قبلی

نمیتوان در سطحی که در آن قرار گرفتیم، استفاده کنیم. اگر تغییر سطح رخ ندهد، نیازی به push نیست؛ چون از همان استک قبلی داریم استفاده میکنیم.

سوال 5

n شماره آرگومان، pp آدرس پوینتر و size اندازه آرگومان، ورودی های آرگومان هستند که مقداری که آرگومان دارد را در pp میریزد.

آدرس یک عدد که با ip مشخص شده را برای گرفتن آرگومانی که به عدد n دادیم، میدهیم.

:argstr آرگومان با توجه به شماره آرگومان که ورودی هست، در pp که یک ورودی است ریخته میشود و آن را برمیگرداند.

در صورت مشخص نشدن بازه، ممکن بود به آدرس اشتباه و در نتیجه مقدار اشتباه دسترسی پیدا میکردیم که دلیل مشکل امنیتی نیز همین موضوع است.

بخش 0: ارسال آرگومان های فراخوانی های سیستمی

خروجی:

```
rm      2 12 15260
sh      2 13 27892
stressfs 2 14 16168
usertests 2 15 67276
wc      2 16 17036
zombie  2 17 14848
find_next_prime 2 18 15372
console 3 19 0
$ find_next_prime_number 10
Kernel: sys_find_next_prime_number() called for number 10
The next prime number is: 11
$ find_next_prime_number 90
Kernel: sys_find_next_prime_number() called for number 90
The next prime number is: 97
$ find_next_prime_number 78
Kernel: sys_find_next_prime_number() called for number 78
The next prime number is: 79
$ find_next_prime_number 99
Kernel: sys_find_next_prime_number() called for number 99
The next prime number is: 101
$ find_next_prime_number 880
Kernel: sys_find_next_prime_number() called for number 880
The next prime number is: 881
$ _
```

مراحل اضافه کردن فراخوانی سیستمی مربوطه:

```
syscall.h x syscall.c x sysproc.c x find_next_prime_number_interface.c x
111 [SYS_exit]    sys_exit,
112 [SYS_wait]    sys_wait,
113 [SYS_pipe]    sys_pipe,
114 [SYS_read]    sys_read,
115 [SYS_kill]    sys_kill,
116 [SYS_exec]    sys_exec,
117 [SYS_fstat]   sys_fstat,
118 [SYS_chdir]   sys_chdir,
119 [SYS_dup]     sys_dup,
120 [SYS_getpid]  sys_getpid,
121 [SYS_sbrk]    sys_sbrk,
122 [SYS_sleep]   sys_sleep,
123 [SYS_uptime]  sys_uptime,
124 [SYS_open]    sys_open,
125 [SYS_write]   sys_write,
126 [SYS_mknod]   sys_mknod,
127 [SYS_unlink]  sys_unlink,
128 [SYS_link]   sys_link,
129 [SYS_mkdir]  sys_mkdir,
130 [SYS_close]  sys_close,
131 [SYS_find_next_prime_number] sys_find_next_prime_number,
```

```
syscall.h x syscall.c x sysproc.c x find_next_prime_number_interface.c x
Visual layout of bidirectional text can depend on the base direction (View | Bidi Text Base Dir)
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); //my change
106 extern int sys_find_next_prime_number(void); //my change
```

در این قسمت حواسمن باید به این نکته باشد که مقدار رجیستر قبل و بعد از فراخوانی سیستمی تغییر نکند که برای این کار به صورت زیر عمل میکنیم:

```
76     release(&tickslock);
77     return 0;
78 }
79
80 // return how many clock tick interrupts have occurred
81 // since start.
82 int
83 sys_uptime(void)
84 {
85     uint xticks;
86
87     acquire(&tickslock);
88     xticks = ticks;
89     release(&tickslock);
90     return xticks;
91 }
92
93 // my change:
94
95 int
96 sys_find_next_prime_number(void)
97 {
98     int number = myproc() ->tf->ebx; //register after eax
99     sprintf("Kernel: sys_find_next_prime_number() called for number %d\n", number);
100    return find_next_prime_number(number);
101 }
102
```

```
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 find_next_prime_number(void);
```

The screenshot shows a code editor window for a GoLand IDE. The file is named `usys.s`. The code lists various system calls using the `SYSSCALL` macro. The `SYSCALL(find_next_prime_number)` macro is highlighted in blue, indicating it is being expanded or is the current selection.

```
11     SYSCALL(exit)
12     SYSCALL(wait)
13     SYSCALL(pipe)
14     SYSCALL(read)
15     SYSCALL(write)
16     SYSCALL(close)
17     SYSCALL(kill)
18     SYSCALL(exec)
19     SYSCALL(open)
20     SYSCALL(mknod)
21     SYSCALL(unlink)
22     SYSCALL(fstat)
23     SYSCALL(link)
24     SYSCALL(mkdir)
25     SYSCALL(chdir)
26     SYSCALL(dup)
27     SYSCALL(getpid)
28     SYSCALL(sbrk)
29     SYSCALL(sleep)
30     SYSCALL(uptime)
31     SYSCALL(find_next_prime_number)
```

پیاده سازی برنامه:

The screenshot shows a code editor window with multiple tabs. The active tab is `proc.c`. The code defines a function `int find_next_prime_number(int n)`. It starts with a base case for `n <= 1`, returning 2. Then it enters a loop where it increments a variable `prime` and checks if it is prime using a helper function `isPrime`. Once a prime is found, it returns that value.

```
549     return 0;
550
551     return 1;
552 }
553
554 int find_next_prime_number(int n)
555 {
556     // Base case
557     if (n <= 1)
558         return 2;
559
560     int prime = n;
561     int found = 0;
562
563     // Loop continuously until isPrime returns
564     // 1 for a number greater than n
565     while (!found) {
566         prime++;
567
568         if (isPrime(prime))
569             found = 1;
570     }
571
572     return prime;
573 }
```

```
syscall.h × defs.h × syscall.c × proc.c × sysproc.c × Makefile ×
 98 // pipe.c
 99 int pipealloc(struct file**, struct file**);
100 void pipeclose(struct pipe*, int);
101 int piperead(struct pipe*, char*, int);
102 int pipewrite(struct pipe*, char*, int);
103
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 find_next_prime_number(int);
124
125
```

```

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_find_next_prime_number 22
24

```

syscall.c × find_next_prime_number_interface.c × getpid_gdb.asm × getpid_gdb.c ×

*.c files are supported by CLion

Visual layout of bidirectional text can depend on the base direction (View | Bidi Text Base Direction)

```

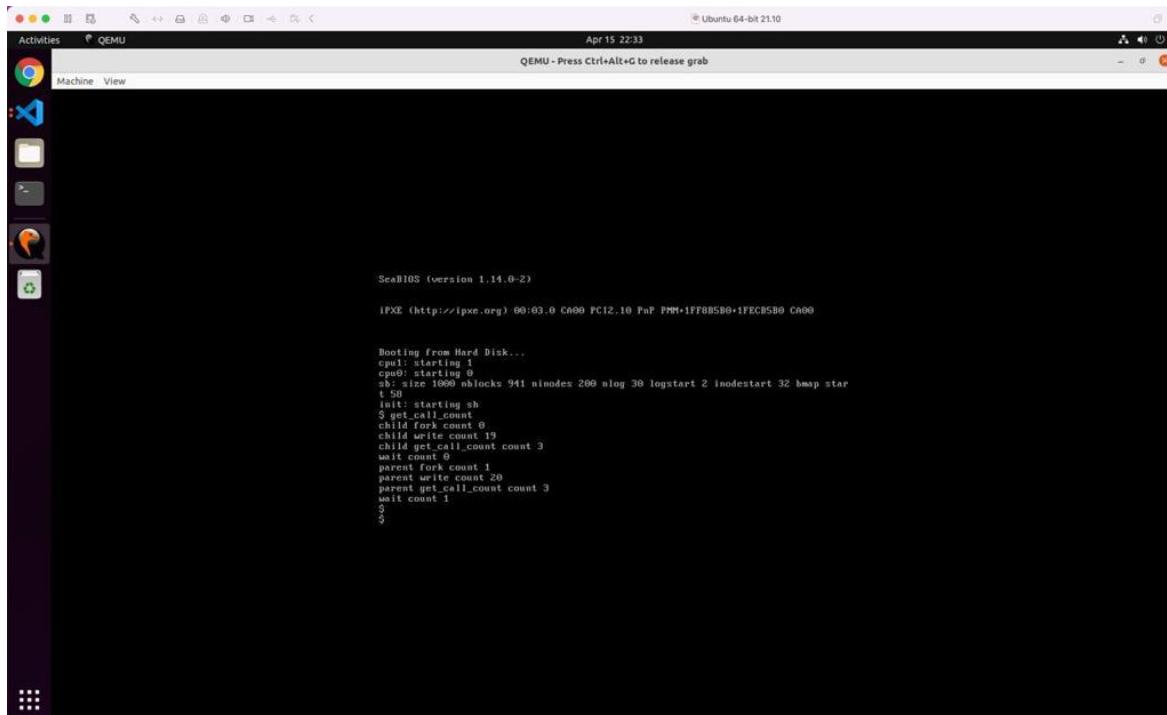
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5 int main(int argc, char *argv[])
6 {
7
8     int saved_ebx, number = atoi(argv[1]);
9     // we can add assembly instructions using asm
10    asm volatile(
11        "movl %%ebx, %0;" : "=r" (saved_ebx)
12        "movl %1, %%ebx;" : "r"(number));
13
14    printf(1, "The next prime number is: %d\n", find_next_prime_number());
15    asm("movl %0, %%ebx" : : "r"(saved_ebx));
16    exit();
17
18 }
19
20

```

بخش 1: پیاده سازی فرآخوانی سیستمی شمارش فرآخوانی های سیستمی

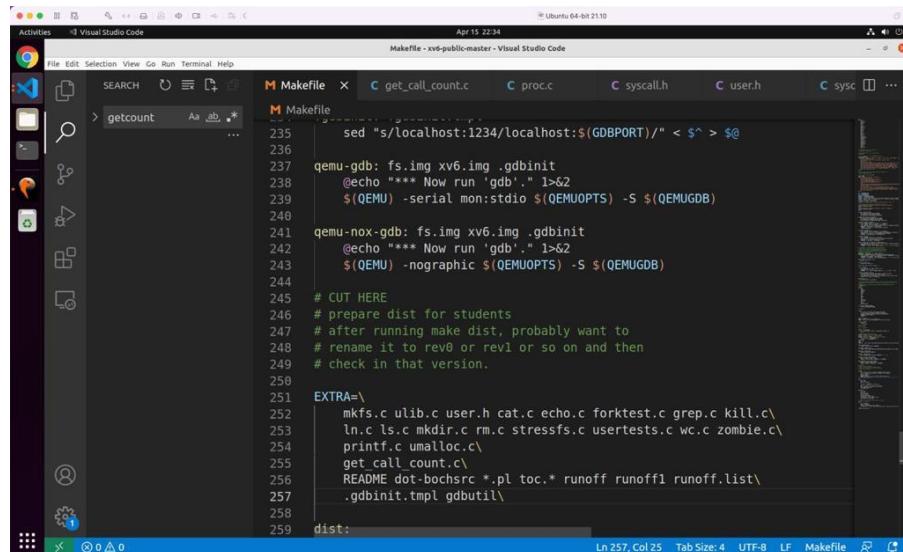
مراحل به صورت زیر انجام شده است:

خروجی:



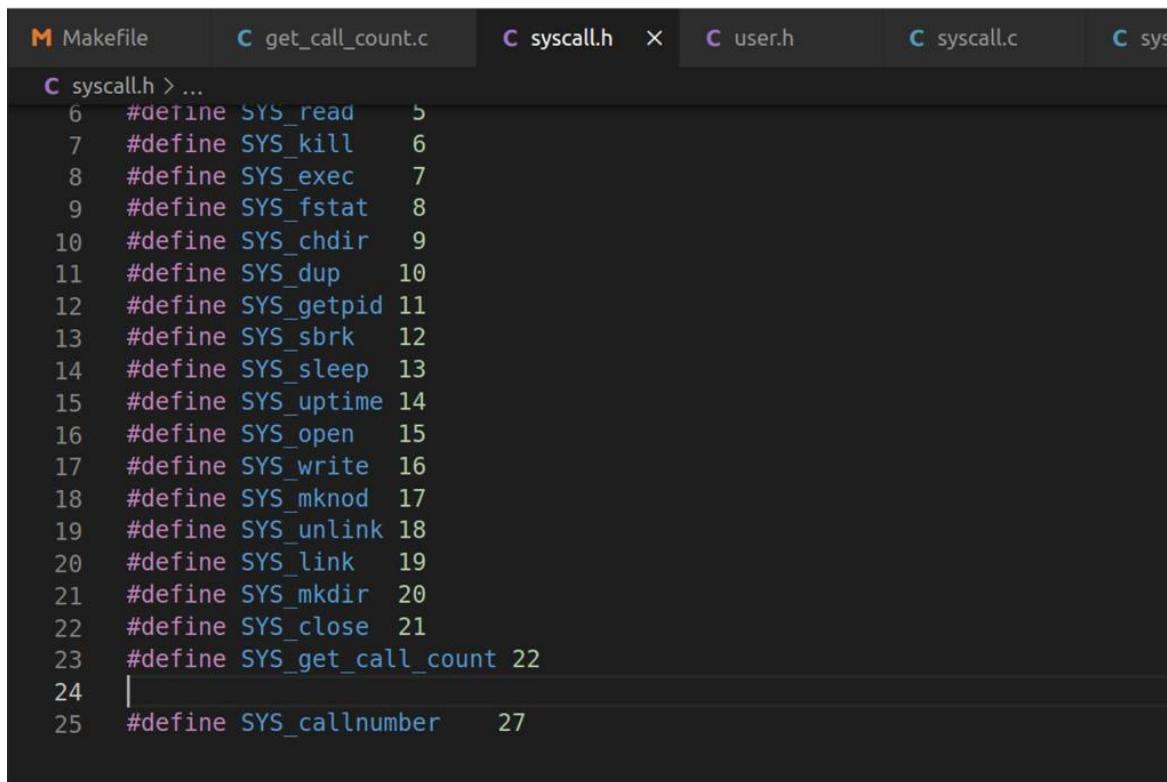
در makefile تغییرات را به صورت زیر اعمال میکنیم:

```
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         _get_call_count\  
185
```



```
Apr 15 22:34
Makefile - xv6-public-master - Visual Studio Code
Activities > Visual Studio Code
File Edit Selection View Go Run Terminal Help
SEARCH ... Makefile x get_call_count.c proc.c syscall.h user.h sysc ...
... Makefile
235 sed "s/localhost:1234/localhost:$(_GDBPORT)/* < $^ > $@"
236
237 qemu-gdb: fs.img xv6.img .gdbinit
238 @echo "*** Now run 'gdb'." 1>&2
239 $(QEMU) -serial mon:stdio $(QEMUOPTS) -S $(QEMUGDB)
240
241 qemu-nox-gdb: fs.img xv6.img .gdbinit
242 @echo "*** Now run 'gdb'." 1>&2
243 $(QEMU) -nographic $(QEMUOPTS) -S $(QEMUGDB)
244
245 # CUT HERE
246 # prepare dist for students
247 # after running make dist, probably want to
248 # rename it to rev0 or rev1 or so on and then
249 # check in that version.
250
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\
255 get_call_count.c\
256 README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
257 .gdbinit.tmpl gdbutil\
258
259 dist:
```

برای اضافه کردن فراخوانی سیستمی مورد نظر، قدم های زیر را برمیداریم:



```
Makefile x get_call_count.c syscall.h x user.h syscall.c sys ...
C syscall.h > ...
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_get_call_count 22
24 |
25 #define SYS_callnumber 27
```

برنامه برای تست فراخوانی سیستمی اضافه شده:

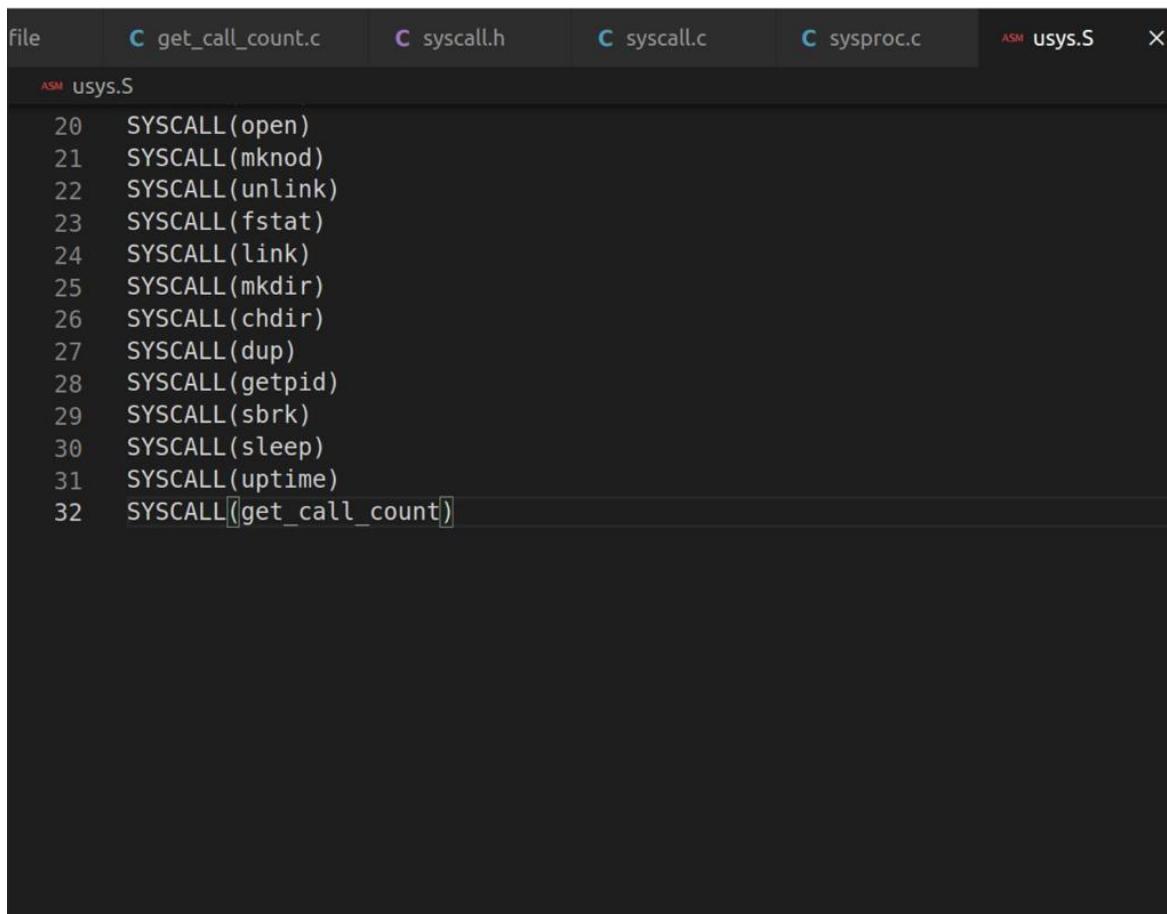
```
M Makefile      C get_call_count.c X  C proc.c      C syscall.h      C user.h      i ▶ v  
C get_call_count.c > ⚡ main(int, char * [] )  
1  #include "types.h"  
2  #include "user.h"  
3  #include "syscall.h"  
4  
5  int  
6  main(int argc, char *argv[] )  
7  {  
8  |  if (fork() == 0) {  
9  |  |  printf(1, "child fork count %d\n", get_call_count(SYS_fork));  
10 |  |  printf(1, "child write count %d\n", get_call_count(SYS_write));  
11 |  |  printf(1, "child get_call_count count %d\n", get_call_count(SYS_get_call)  
12 |  } else {  
13 |  |  wait();  
14 |  |  printf(1, "parent fork count %d\n", get_call_count(SYS_fork));  
15 |  |  printf(1, "parent write count %d\n", get_call_count(SYS_write));  
16 |  |  printf(1, "parent get_call_count count %d\n", get_call_count(SYS_get_call)  
17 |  |  }  
18 |  |  printf(1, "wait count %d\n", get_call_count(SYS_wait));  
19 |  |  exit();  
20 | }  
21
```

```
M Makefile      C get_call_count.c      C syscall.h      C user.h      X  C syscall.c      C sysi  
C user.h > ⚡ get_call_count(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 get_call_count(int sys_call);
```

```
Makefile      C get_call_count.c    C syscall.h    C syscall.c    X  C sysproc.c    ▶ ▷ ✓
C syscall.c > [o] syscalls
10/
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_get_call_count] sys_get_call_count]
131 ];
132
```

```
Makefile      C get_call_count.c    C syscall.h     C syscall.c   X  C sysproc.c  ▾
```

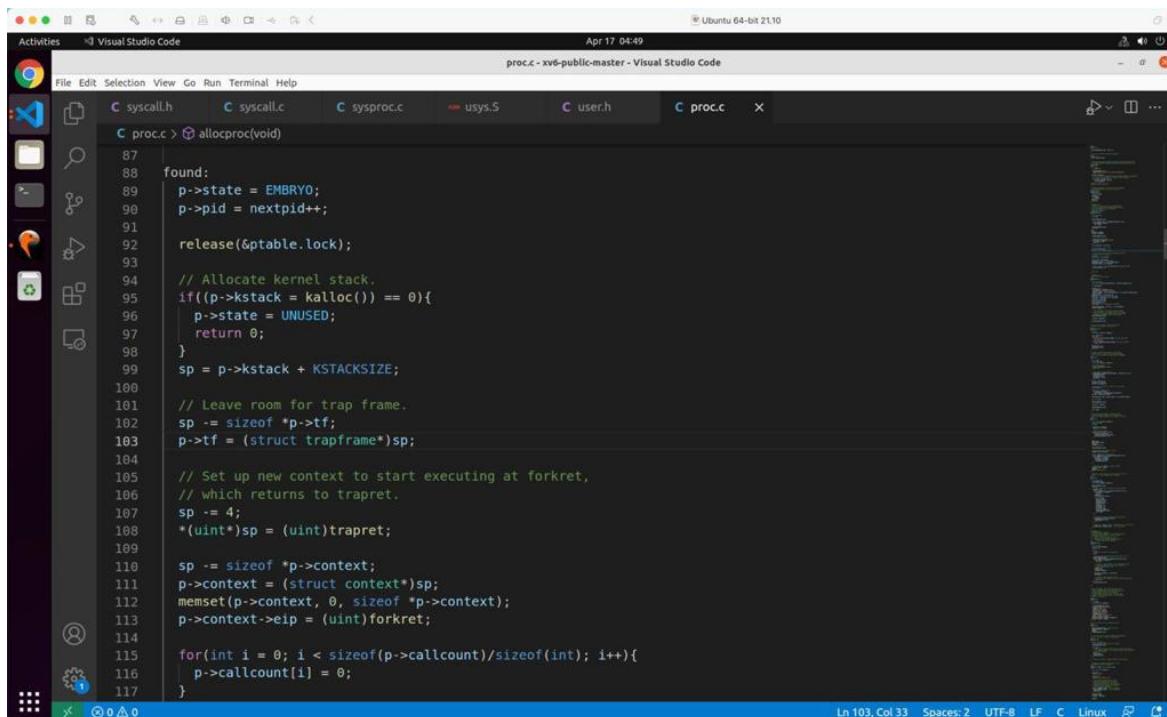
```
C syscall.c > [o] syscalls
130 [SYS_get_call_count] sys_get_call_count
131 ];
132
133 void
134 syscall(void)
135 {
136     int num;
137     struct proc *curproc = myproc();
138
139     num = curproc->tf->eax;
140     curproc->callcount[num - 1] = curproc->callcount[num - 1] + 1;
141     // cprintf("Call Count: %d\n",
142     //           curproc->callcount[num - 1]);
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144         curproc->tf->eax = syscalls[num]();
145     } else {
146         cprintf("%d %s: unknown sys call %d\n",
147                 curproc->pid, curproc->name, num);
148         curproc->tf->eax = -1;
149     }
150 }
151
152 }
```



The screenshot shows a terminal window with the file `usys.S` open. The assembly code lists various system calls:

```
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(get_call_count)
```

خط 115 و 116 به `proc.c` برای مقداردهی اولیه اضافه شده است.



The screenshot shows the `proc.c` file in Visual Studio Code. The code has been modified to include the following lines at the bottom:

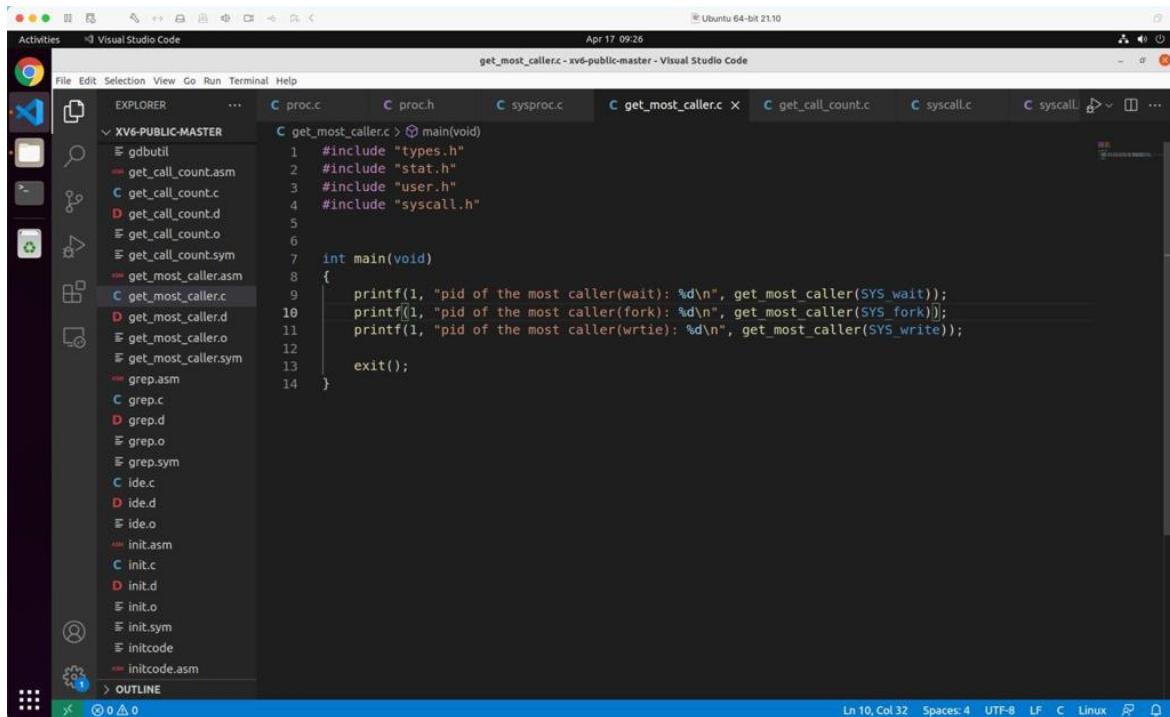
```
110    sp -= sizeof *p->context;
111    p->context = (struct context*)sp;
112    memset(p->context, 0, sizeof *p->context);
113    p->context->eip = (uint)forkret;
114
115    for(int i = 0; i < sizeof(p->callcount)/sizeof(int); i++){
116        p->callcount[i] = 0;
117    }
```

```
Makefile      C get_call_count.c      C syscall.h      C syscall.c      C sysproc.c  X  ▾ ▷ ▾
C sysproc.c > ...
1 #include "types.h"
2 #include "x86.h"
3 #include "defs.h"
4 #include "date.h"
5 #include "param.h"
6 #include "memlayout.h"
7 #include "mmu.h"
8 #include "proc.h"
9 |
10 int sys_get_call_count(void){
11     int n;
12     if(argint(0, &n) < 0)
13     |    return -1;
14     return(myproc()->callcount[n-1]);
15 }
16
17 int
18 sys_fork(void)
19 {
20     return fork();
21 }
22
23 int
24 sys_exit(void)
25 {
```

بخش 2: پیاده سازی فراخوانی سیستمی پر مصرف ترین پردازه

توجیه: یک چون اومده چند تا پراسس دیگه درست کرده خودش واسه همین fork بیشترین دفعات توسط خود پراسس ۱ انجام شده

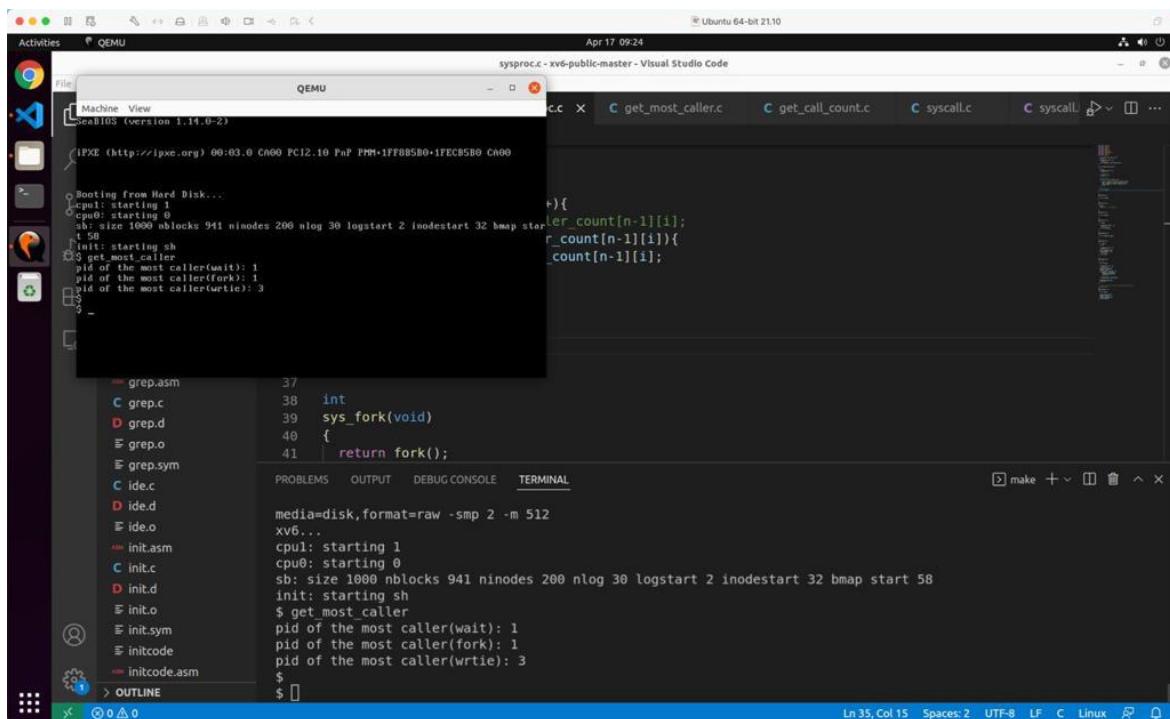
در عکس زیر برنامه تست نوشته شده است:



```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "syscall.h"

int main(void)
{
    printf(1, "pid of the most caller(wait): %d\n", get_most_caller(SYS_wait));
    printf(1, "pid of the most caller(fork): %d\n", get_most_caller(SYS_fork));
    printf(1, "pid of the most caller(wrtie): %d\n", get_most_caller(SYS_write));
    exit();
}
```

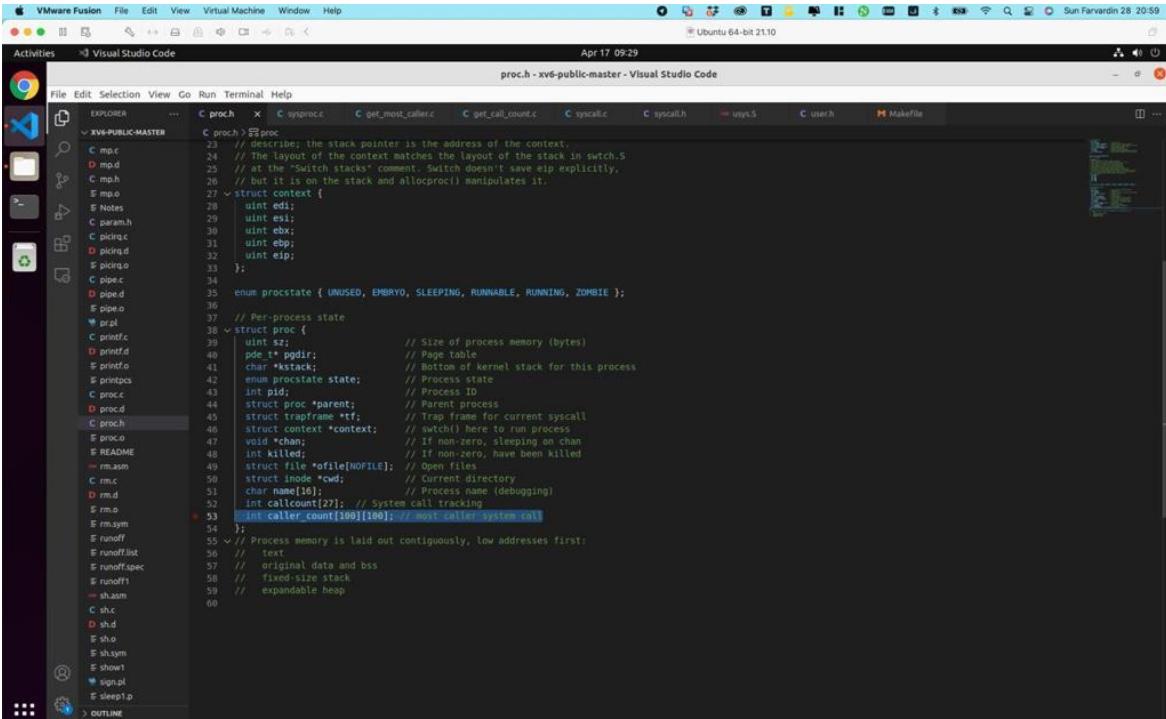
خروجی:



```
IPXE (http://ipxe.org) 00:03.0 CN60 PC12.10 FnP PMM+1FF885B0+1FECB5B0 CN60
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 blocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ get_most_caller
pid of the most caller(wait): 1
pid of the most caller(fork): 1
pid of the most caller(wrtie): 3
```

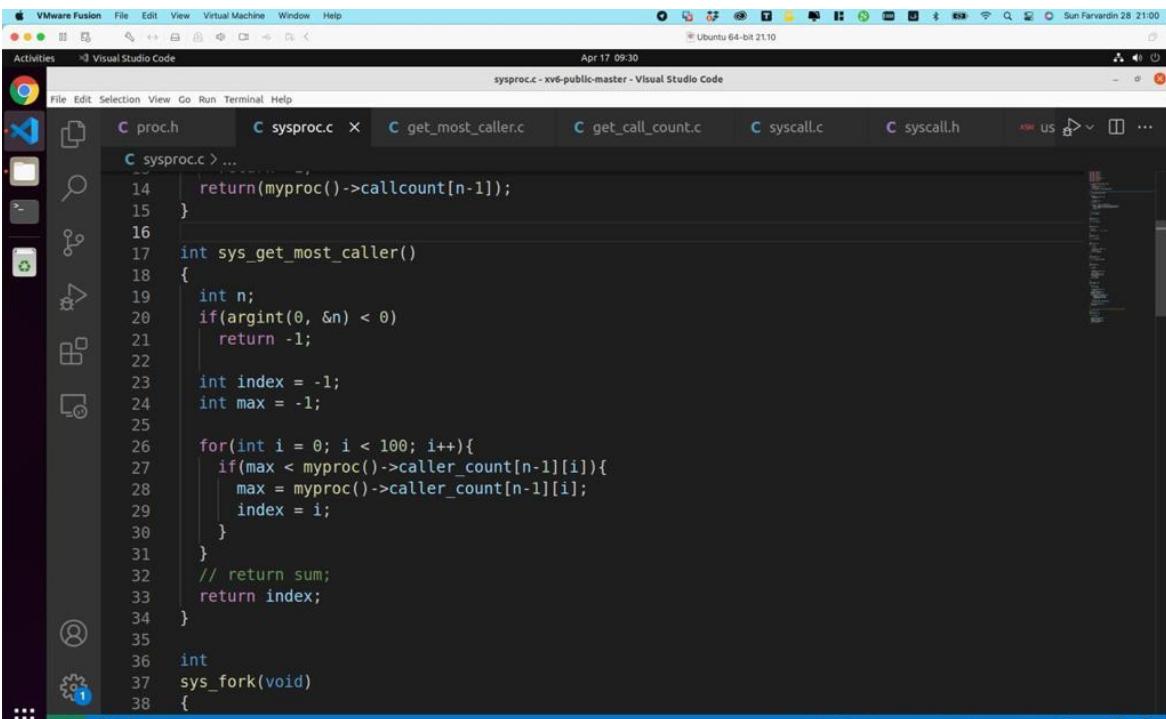
```
37 int
38     sys_fork(void)
39 {
40     return fork();
41 }
```

برای اضافه کردن فراخوانی سیستمی این قسمت به صورت زیر عمل میکنیم:



```
proc.h // proc
23 // describe; the stack pointer is the address of the context,
24 // The layout of the context matches the layout of the stack in swtch.S
25 // At the "switch stacks" comment. Switch doesn't save eip explicitly,
26 // but caller's is on the stack and allocproc() manipulates it.
27 struct context {
28     uint edi;
29     uint esi;
30     uint ebx;
31     uint ebp;
32     uint eip;
33 };
34
35 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37 // Per-process state
38 struct proc {
39     uint sz;           // Size of process memory (bytes)
40     pde_t *pdir;      // Page table
41     char *stack;      // Bottom of kernel stack for this process
42     enum procstate state; // Process state
43     int pid;          // Process ID
44     struct proc *parent; // Trap frame for current syscall
45     struct context *context; // swtch() here to run process
46     void *chan;        // If non-zero, sleeping on chan
47     int killed;        // If non-zero, have been killed
48     struct file *ofile[NFILE]; // Open files
49     struct inode *cwd; // Current directory
50     char name[16];   // Process name (debugging)
51     int callcount[27]; // System call tracking
52     int caller_count[100][100]; // most_caller system call
53 };
54
55 // Process memory is laid out contiguously, low addresses first:
56 // Text
57 // original data and bss
58 // fixed-size stack
59 // expandable heap
60
```

شیوه پیاده سازی:



```
sysproc.c > ...
14     return(myproc()->callcount[n-1]);
15 }
16
17 int sys_get_most_caller()
18 {
19     int n;
20     if(argint(0, &n) < 0)
21         return -1;
22
23     int index = -1;
24     int max = -1;
25
26     for(int i = 0; i < 100; i++){
27         if(max < myproc()->caller_count[n-1][i]){
28             max = myproc()->caller_count[n-1][i];
29             index = i;
30         }
31     }
32     // return sum;
33     return index;
34 }
35
36 int
37 sys_fork(void)
38 {
```

VMware Fusion File Edit View Virtual Machine Window Help

Activities -> Visual Studio Code

File Edit Selection View Go Run Terminal Help

C proc.h C sysproc.c C get_most_caller.c C syscall.c X C syscall.h usys.S user.h

```
C syscall.c > syscall(void)
114 [SYS_read] sys_read,
115 [SYS_kill] sys_kill,
116 [SYS_exec] sys_exec,
117 [SYS_fstat] sys_fstat,
118 [SYS_chdir] sys_chdir,
119 [SYS_dup] sys_dup,
120 [SYS_getpid] sys_getpid,
121 [SYS_sbrk] sys_sbrk,
122 [SYS_sleep] sys_sleep,
123 [SYS_uptime] sys_uptime,
124 [SYS_open] sys_open,
125 [SYS_write] sys_write,
126 [SYS_mknod] sys_mknod,
127 [SYS_unlink] sys_unlink,
128 [SYS_link] sys_link,
129 [SYS_mkdir] sys_mkdir,
130 [SYS_close] sys_close,
131 [SYS_get_call_count] sys_get_call_count,
132 [SYS_get_most_caller] sys_get_most_caller
133 };
134
135 int caller_count[100][100] = {0};
136 void
137 syscall(void)
138 {
139     int num;
```

2 selections (76 characters selected) Spaces: 2 UTF-8 LF C Linux

VMware Fusion File Edit View Virtual Machine Window Help

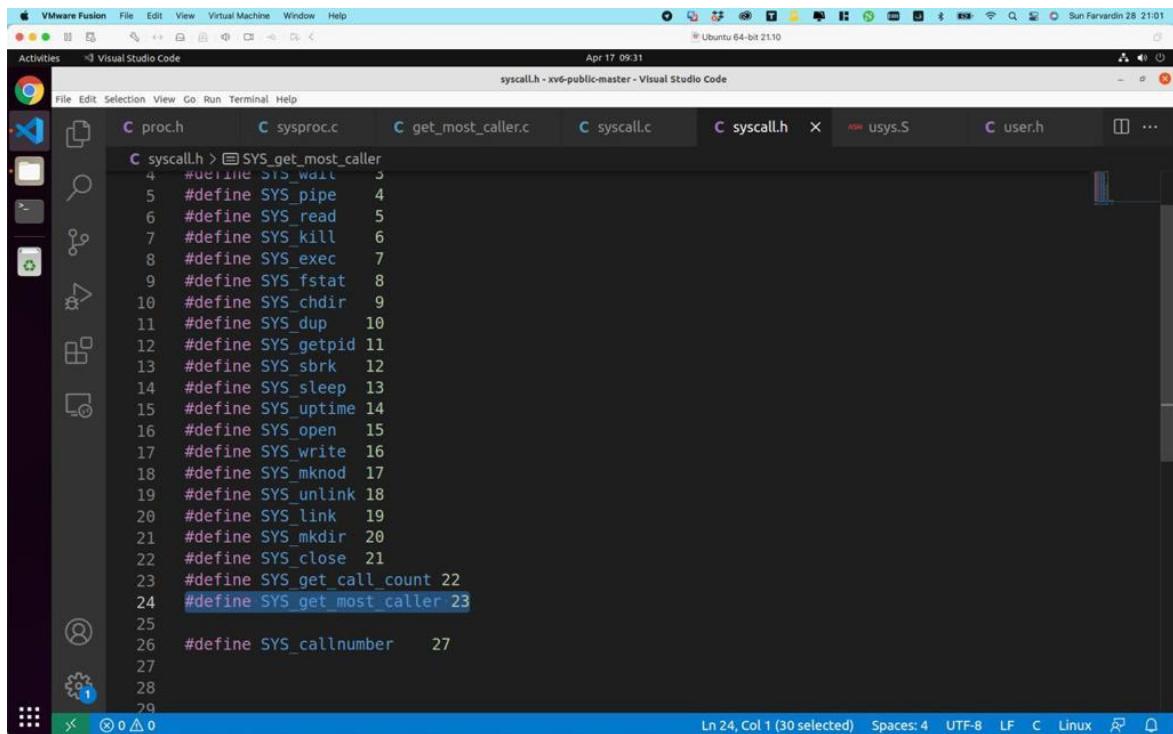
Activities -> Visual Studio Code

File Edit Selection View Go Run Terminal Help

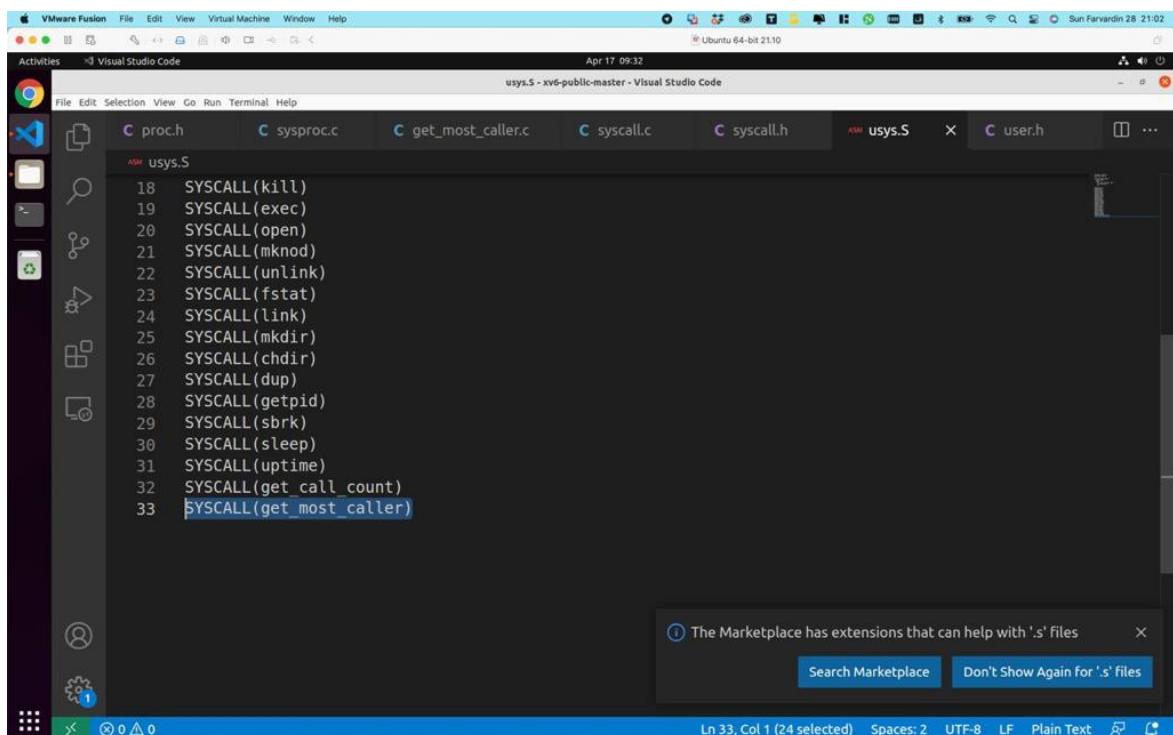
C proc.h C sysproc.c C get_most_caller.c C syscall.c C syscall.h usys.S user.h

```
C syscall.c > syscall(void)
135 int caller_count[100][100] = {0};
136 void
137 syscall(void)
138 {
139     int num;
140     struct proc *curproc = myproc();
141
142     num = curproc->tf->eax;
143     curproc->callcount[num - 1] = curproc->callcount[num - 1] + 1;
144
145     caller_count[num - 1][curproc->pid] = caller_count[num - 1][curproc->pid] + 1;
146
147     for(int i = 0; i < 100; i++){
148         for (int j = 0; j < 100; j++){
149             curproc->caller_count[i][j] = caller_count[i][j];
150         }
151     }
152
153
154 //     printf("Call Count: %d\n",
155 //            curproc->callcount[num - 1]);
156     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
157
158         curproc->tf->eax = syscalls[num]();
159     } else {
160 }
```

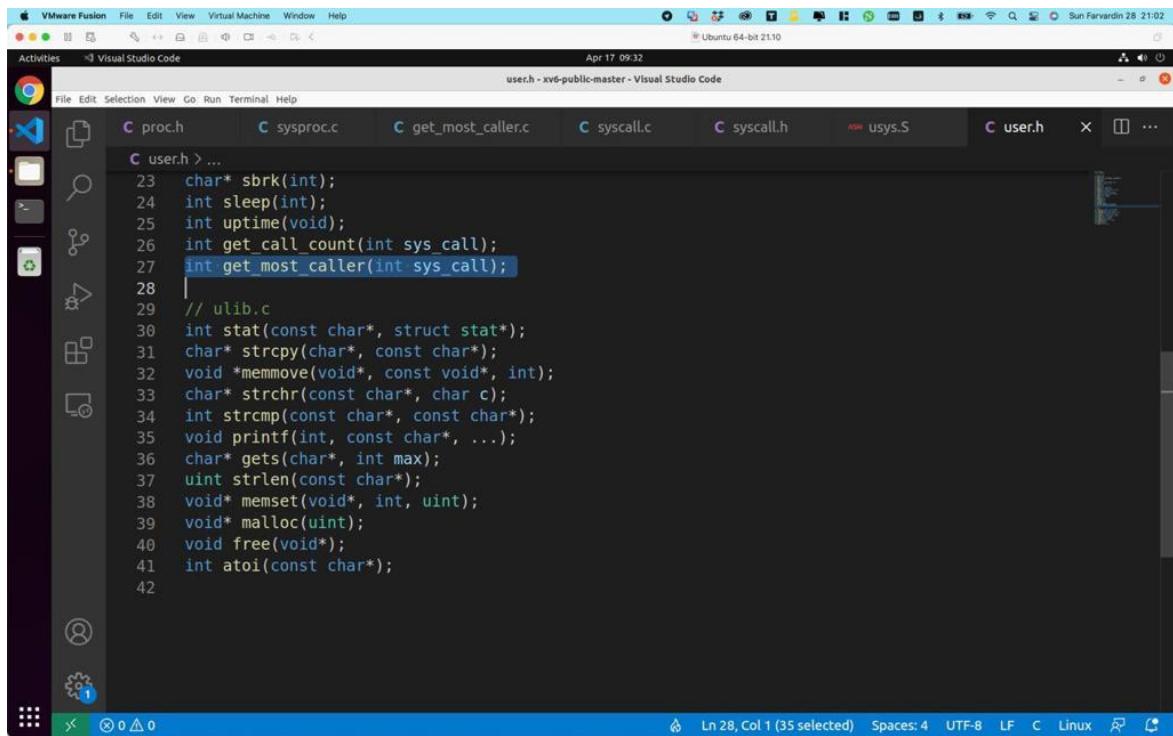
Ln 145, Col 1 (218 selected) Spaces: 2 UTF-8 LF C Linux



```
#define SYS_WAIT 5
#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_GET_CALL_COUNT 22
#define SYS_GET_CALLER 23
#define SYS_CALLNUMBER 27
```



```
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(get_call_count)
SYSCALL(get_most_caller)
```

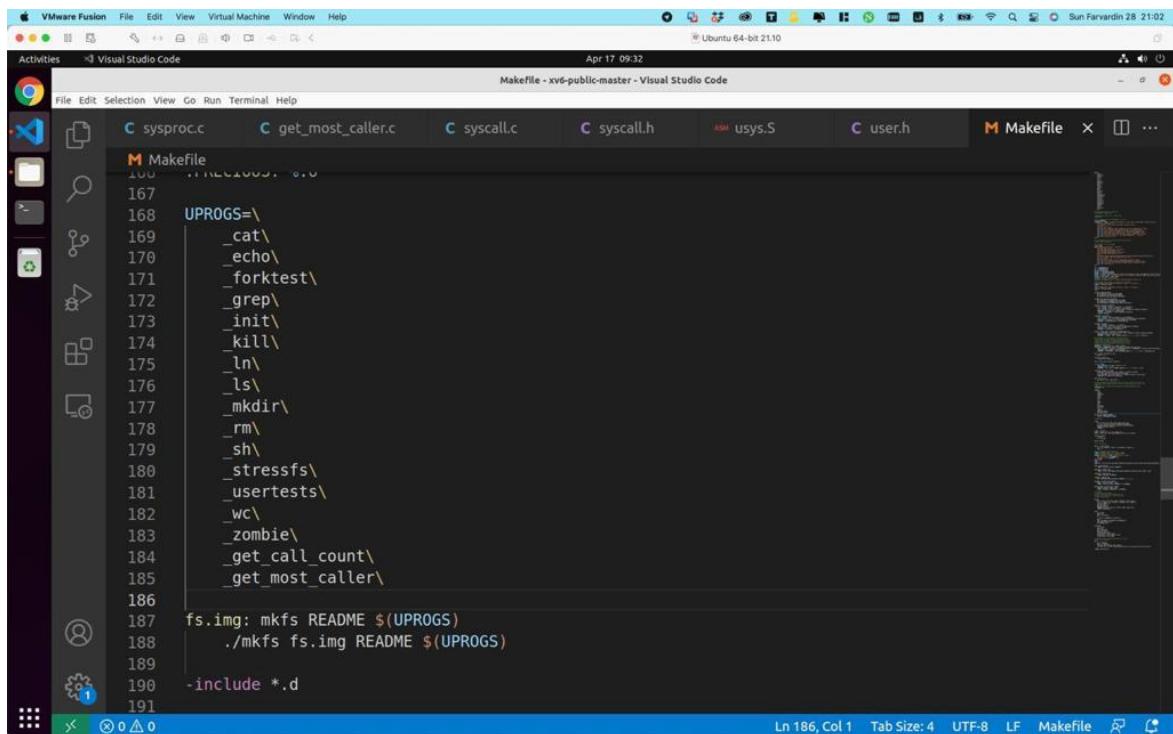


The screenshot shows the Visual Studio Code interface on an Ubuntu 64-bit 21.10 system. The title bar indicates the file is 'usec.h - xv6-public-master - Visual Studio Code'. The code editor displays the 'user.h' header file, which includes declarations for system calls like sbrk, sleep, uptime, and various string manipulation functions. The status bar at the bottom shows 'Ln 28, Col 1 (35 selected) Spaces: 4 UTF-8 LF C Linux'.

```
char* sbrk(int);
int sleep(int);
int uptime(void);
int get_call_count(int sys_call);
int get_most_caller(int sys_call);

// 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*);
```

تغییرات makefile برای فراخوانی سیستمی این قسمت به صورت زیر است:



The screenshot shows the Visual Studio Code interface on an Ubuntu 64-bit 21.10 system. The title bar indicates the file is 'Makefile - xv6-public-master - Visual Studio Code'. The code editor displays the 'Makefile' script, which defines the variable 'UPROGS' to include several user programs: cat, echo, forktest, grep, init, kill, ln, ls, mkdir, rm, sh, stressfs, usertests, wc, and zombie. It also includes targets for creating an image file ('fs.img') using 'mkfs' and running the image ('./mkfs fs.img'). The status bar at the bottom shows 'Ln 186, Col 1 Tab Size: 4 UTF-8 LF Makefile'.

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

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

#include *.d
```

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

بخش 3: پیاده سازی فراخوانی سیستمی صبر برای پردازهای دیگر

مراحل به صورت زیر انجام شده است:

The screenshot shows a code editor with four tabs at the top: 'syscall.h M X', 'syscall.c M', 'sysproc.c M', and 'syscall.h > SYS_wait_for_process'. The current view is under the 'syscall.h > SYS_wait_for_process' tab. The code in the editor is as follows:

```
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_wait_for_process 29
```

```
C syscall.h M C syscall.c M X C sysproc.c M C defs.h M
C syscall.c > ⚭ syscalls
103     extern int sys_wait_for_process(void);
104     extern int sys_write(void);
105     extern int sys_uptime(void);
106     extern int sys_wait_for_process(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_wait_for_process] sys_wait_for_process,
131     ];

```

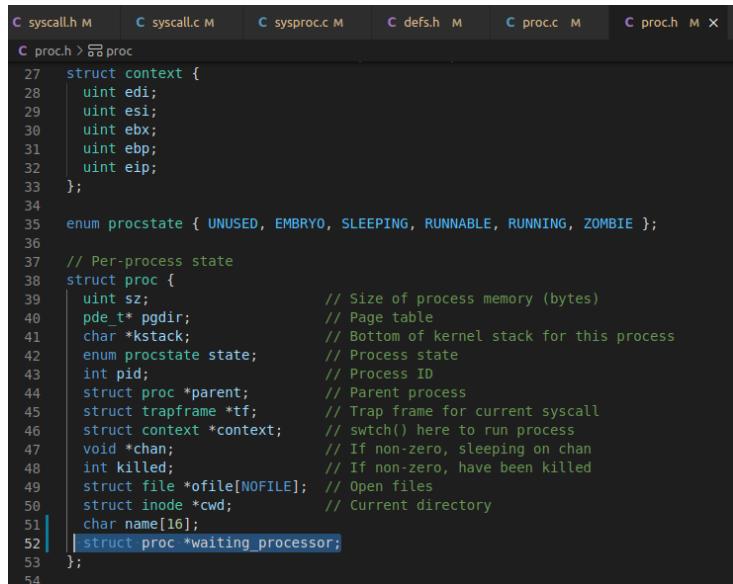
```
C syscall.h M C syscall.c M X C sysproc.c M C
C sysproc.c > ⚭ sys_wait_for_process(void)
82     ...
83     sys_uptime(void)
84     {
85         uint xticks;
86
87         acquire(&tickslock);
88         xticks = ticks;
89         release(&tickslock);
90         return xticks;
91     }
92
93     int
94     sys_wait_for_process(void)
95     {
96         int pid;
97         if(argint(0, &pid)<0){
98             return -1;
99         }
100        else{
101            return wait_for_process(pid);
102        }
103    }

```

```
C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C proc.h M ASM usys.S M X C us
ASM usys.S
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(wait_for_process)
33
```

```
C syscall.h M C syscall.c M C sysproc.c M C defs.h M X C proc.c M C
C defs.h > wait_for_process(int)
98 // pipe.c
99 int pipealloc(struct file**, struct file**);
100 void pipeclose(struct pipe*, int);
101 int piperead(struct pipe*, char*, int);
102 int pipewrite(struct pipe*, char*, int);
103
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 int wait_for_process(int);
122 void wakeup(void*);
123 void yield(void);
```

در استراکچر proc یک متود به نام waiting_processor میشود تا در ادامه از آن استفاده کنیم:

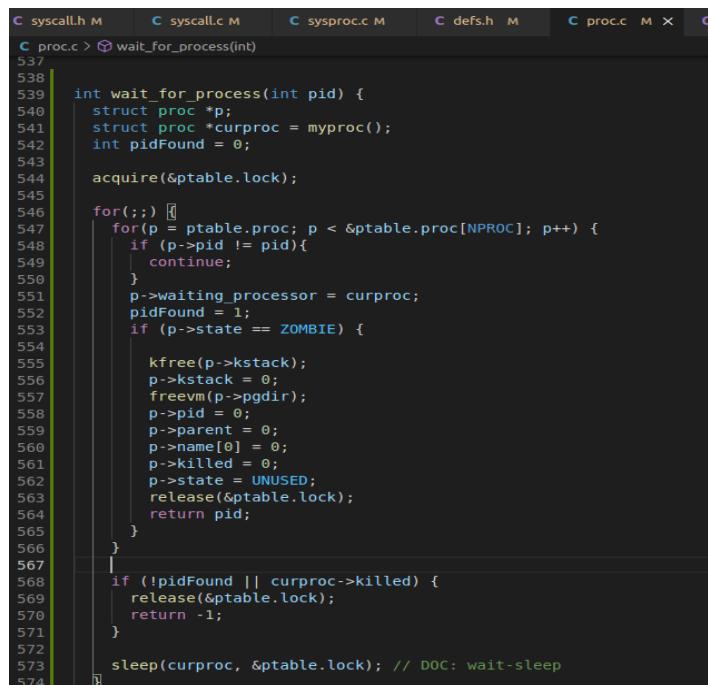


```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C proc.h M X
C proc.h > proc
27 struct context {
28     uint edi;
29     uint esi;
30     uint ebx;
31     uint ebp;
32     uint eip;
33 };
34
35 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
36
37 // Per-process state
38 struct proc {
39     uint sz;                      // Size of process memory (bytes)
40     pde_t* pgdir;                 // Page table
41     char *kstack;                 // Bottom of kernel stack for this process
42     enum procstate state;        // Process state
43     int pid;                      // Process ID
44     struct proc *parent;          // Parent process
45     struct trapframe *tf;         // Trap frame for current syscall
46     struct context *context;      // swtch() here to run process
47     void *chan;                   // If non-zero, sleeping on chan
48     int killed;                  // If non-zero, have been killed
49     struct file *ofile[NFILE];    // Open files
50     struct inode * cwd;          // Current directory
51     char name[16];
52     struct proc *waiting_processor;
53 };
54

```

در این قسمت، فراخوانی سیستمی wait_for_process که از فراخوانی سیستمی wait الگو گرفته شده، با این تفاوت که در حلقه ما به جای parent pid به دنبال ورودی pid میگردیم و آن را return میکنیم. همچنین تفاوت دیگر این است که مقدار curproc هم برابر proc در استراکچر waiting_processor میکنیم:



```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M X C proc.h M X
C proc.c > wait_for_process(int)
537
538     int wait_for_process(int pid) {
539         struct proc *p;
540         struct proc *curproc = myproc();
541         int pidFound = 0;
542
543         acquire(&ptable.lock);
544
545         for(;;) {
546             for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
547                 if (p->pid != pid){
548                     continue;
549                 }
550                 p->waiting_processor = curproc;
551                 pidFound = 1;
552                 if (p->state == ZOMBIE) {
553
554                     kfree(p->kstack);
555                     p->kstack = 0;
556                     freevm(p->pgdir);
557                     p->pid = 0;
558                     p->parent = 0;
559                     p->name[0] = 0;
560                     p->killed = 0;
561                     p->state = UNUSED;
562                     release(&ptable.lock);
563                     return pid;
564                 }
565             }
566         }
567
568         if (!pidFound || curproc->killed) {
569             release(&ptable.lock);
570             return -1;
571         }
572
573         sleep(curproc, &ptable.lock); // DOC: wait-sleep
574

```

در تابع exit یک تغییر کوچک ایجاد میکنیم که متود waiting_processor که در استراکچر proc تعریف کردیم، هم از خواب بلند میکنیم:

```
227 void
228 exit(void)
229 {
230     struct proc *curproc = myproc();
231     struct proc *p;
232     int fd;
233
234     if(curproc == initproc)
235         panic("init exiting");
236
237     // Close all open files.
238     for(fd = 0; fd < NOFILE; fd++){
239         if(curproc->ofile[fd]){
240             fileclose(curproc->ofile[fd]);
241             curproc->ofile[fd] = 0;
242         }
243     }
244
245     begin_op();
246     input(curproc->cwd);
247     end_op();
248     curproc->cwd = 0;
249
250     acquire(&ptable.lock);
251
252     // Parent might be sleeping in wait().
253     wakeup1(curproc->parent);
254     wakeup1(curproc->waiting_processor);
255     curproc->waiting_processor = '\0';
256
257     // Pass abandoned children to init.
```

```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C proc.h M ASM usys.S M C user.h
C user.h > ⚡ wait_for_process(int)
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
9 int wait_for_process(int);
10
11 int pipe(int*);
12 int write(int, const void*, int);
13 int read(int, void*, int);
14 int close(int);
15 int kill(int);
16 int exec(char*, char**);
17 int open(const char*, int);
18 int mknod(const char*, short, short);
19 int unlink(const char*);
20 int fstat(int fd, struct stat*);
21 int link(const char*, const char*);
22 int mkdir(const char*);
23 int chdir(const char*);
24 int dup(int);
25 int getpid(void);
26 char* sbrk(int);
27 int sleep(int);
28 int uptime(void);

```

در makefile را تعريف میکنیم:

```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C wait_for_process.c U M Makefile M X
M Makefile
161
162 # Prevent deletion of intermediate files, e.g. cat.o, after first build, so
163 # that disk image changes after first build are persistent until clean. More
164 # details:
165 # http://www.gnu.org/software/make/manual/html\_node/Chained-Rules.html
166 .PRECIOUS: %.o
167
168 UPROGS=\
169   _sort_string\
170   _cat\
171   _echo\
172   _forktest\
173   _grep\
174   _init\
175   _kill\
176   _ln\
177   _ls\
178   _mkdir\
179   _rm\
180   _sh\
181   _stressfs\
182   _usertests\
183   _wc\
184   _zombie\
185   _wait_for_process\
186

```

```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C wait_for_process.c U
M Makefile
246 # CUT HERE
247 # prepare dist for students
248 # after running make dist, probably want to
249 # rename it to rev0 or rev1 or so on and then
250 # check in that version.
251
252 EXTRA=\
253     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
254     ln.c ls.c sort_string.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
255     printf.c umalloc.c\
256     wait_for_process.c\
257 README dot-bochssrc *.pl toc.* runoff runoff1 runoff.list\
258 .gdbinit tmpl gdbutil\
259

```

تست را به صورت زیر طبق صورت پروژه تعریف میکنیم:

```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C wait_for_process.c U X M N
C wait_for_process.c > ↗ wait_for_process_test(void)
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4 #include "syscall.h"
5
6 void wait_for_process_test(void) {
7     int pid1, pid2;
8     if((pid1 = fork())!=0) // Create a child
9     {
10         // We are in the parent
11         if((pid2 = fork())!=0)
12         {
13             // We are in the parent
14             wait();
15             printf(1, "Second child has finished \n");
16             printf(1, "Parent has been finished\n");
17             exit();
18         }
19         else
20         {
21             // We are in the 2nd child
22             int termpid = wait_for_process(pid1);
23             if(termpid > 0){
24                 printf(1, "First child has finished \n");
25             }
26             exit();
27         }
28     }
29 }
30
31

```

```

C syscall.h M C syscall.c M C sysproc.c M C defs.h M C proc.c M C wait_for_process.c U X M Makefile
C wait_for_process.c > ⚙ wait_for_process_test(void)

30     else
31     {
32         // We are in the 1st child
33         for(int i=0;i<100000000;i++){
34             if(i== 9000){
35                 printf(1, "Wait %d iterations for first child\n", i);
36             }
37             else if(i== 90000){
38                 printf(1, "Wait %d iterations for first child\n", i);
39             }
40             else if(i== 900000){
41                 printf(1, "Wait %d iterations for first child\n", i);
42             }
43             else if(i== 9000000){
44                 printf(1, "Wait %d iterations for first child\n", i);
45             }
46         }
47         exit();
48     }
49 }

50 int
51 main(int argc, char *argv[])
52 {
53     printf(1, "Testing wait_for_process function\n");
54     wait_for_process_test();
55
56     exit();
57 }

```

خروجی:

```

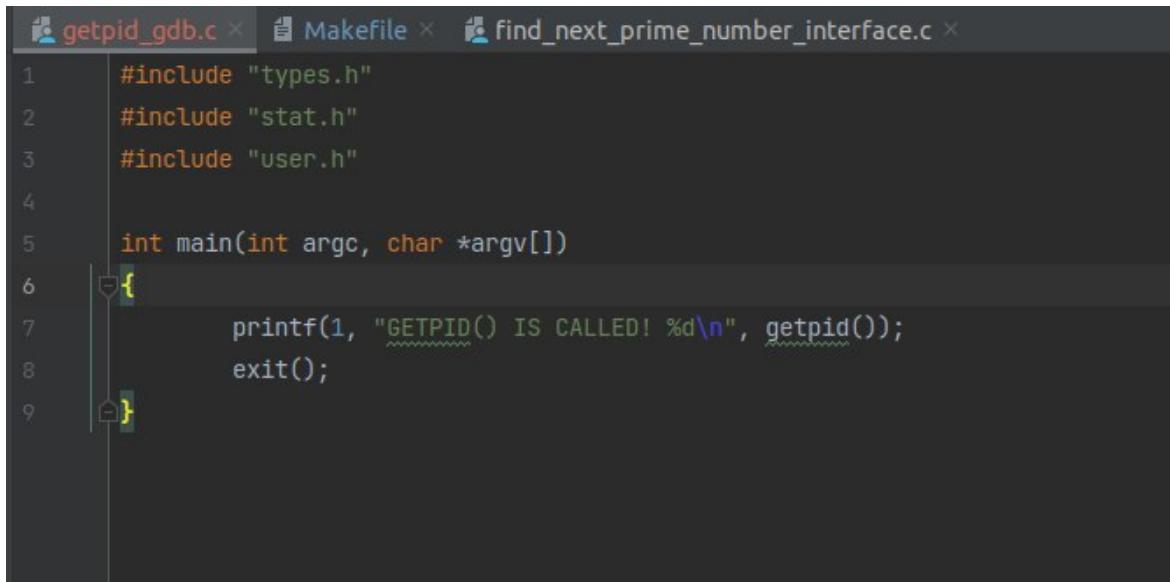
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ld -m elf_i386 -N -e main -Ttext 0 -o _wait_for_process wait_for_process.o ulib.o usys.o printf.o umalloc.o
objdump -S _wait_for_process > wait_for_process.asm
objdump -t _wait_for_process | sed '1,/SYMBOL TABLE/d; s/ .* //; /^$/d' > wait_for_process.sym
./mkfs fs.img README_sort_string_cat_echo_fortest_grep_init_kill_ln_ls_mkdir_rm_sh_stressfs_userstest
nmeta 59 (boot, super, log blocks 30 inode blocks 26, bitmap blocks 1) blocks 941 total 1000
ballof: first 732 blocks have been allocated
ballof: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,med
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
Group #22:
1. Danial Saeedi
2. Soroosh Sadeghian
3. Mohammad GharehHasanloo
$ wait_for_process
Testing wait_for_process function
Wait 9000 iterations for first child
Wait 90000 iterations for first child
Wait 900000 iterations for first child
Wait 9000000 iterations for first child
First child has finished
Second child has finished
Parent has been finished
mylinux@ubuntu:~/Desktop/OS-Lab-main/Project1$ 

```

خروجی برای همه فراخوانی های سیستمی جدید

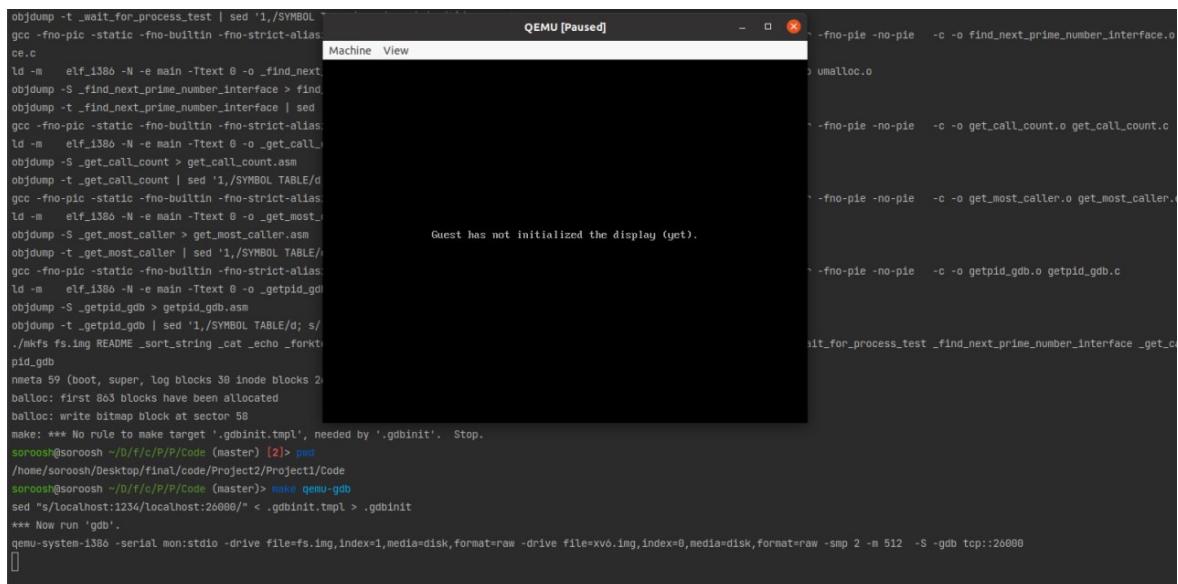
```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
init: starting sh
Group #22:
1. Danial Saeedi
2. Soroosh Sadeghian
3. Mohammad GharehHasanloo
$ get most caller
pid of the most caller(wait): 1
pid of the most caller(fork): 1
pid of the most caller(wrtie): 1
$ get call count
child fork count 0
child write count 19
child get call count count 3
wait count 0
parent fork count 1
parent write count 20
parent get call count count 3
wait count 1
$ get most call
exec: fail
exec get most call failed
$ get most caller
pid of the most caller(wait): 2
pid of the most caller(fork): 2
pid of the most caller(wrtie): 3
$ wait for process test
Testing wait for process function
Wait 9000 iterations for first child
Wait 900000 iterations for first child
Wait 9000000 iterations for first child
Wait 90000000 iterations for first child
First child has finished
Second child has finished
Parent has been finished
$ find next prime number 2134
Kernel: sys find next prime number() called for number 2134
The next prime number is: 2137
$ get most caller
pid of the most caller(wait): 2
pid of the most caller(fork): 2
pid of the most caller(wrtie): 9
$ get call count
child fork count 0
child write count 19
child get call count count 3
wait count 0
parent fork count 1
parent write count 20
parent get call count count 3
```

بررسی گام های اجرای فراخوانی سیستمی در سطح کرنل توسط gdb



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

int main(int argc, char *argv[])
{
    printf("GETPID() IS CALLED! %d\n", getpid());
    exit();
}
```



```
objdump -t _wait_for_process_test | sed '1,/SYMBOL
ce.c
ld -m elf_i386 -N -e main -Ttext 0 -o _find_next_
objdump -S _find_next_prime_number_interface > find_
objdump -t _find_next_prime_number_interface | sed
gcc -fno-pic -static -fno-builtin -fno-strict-alias
ld -m elf_i386 -N -e main -Ttext 0 -o _get_call_
objdump -S _get_call_count > get_call_count.asm
objdump -t _get_call_count | sed '1,/SYMBOL TABLE/d
gcc -fno-pic -static -fno-builtin -fno-strict-alias
ld -m elf_i386 -N -e main -Ttext 0 -o _get_most_
objdump -S _get_most_caller > get_most_caller.asm
objdump -t _get_most_caller | sed '1,/SYMBOL TABLE/d
gcc -fno-pic -static -fno-builtin -fno-strict-alias
ld -m elf_i386 -N -e main -Ttext 0 -o _getpid_gdb
objdump -S _getpid_gdb > getpid_gdb.asm
objdump -t _getpid_gdb | sed '1,/SYMBOL TABLE/d; s/
./mkfs fs.img README .sort_string.cat .echo .forkt
pid_gdb
nmeta 59 (boot, super, log blocks 30 inode blocks 2
balloc: first 8&3 blocks have been allocated
balloc: write bitmap block at sector 58
make: *** No rule to make target '.gdbinit.tmp', needed by '.gdbinit'. Stop.
sorosh@sorosh ~/D/f/c/P/P/Code (master) [2]> pnd
/home/sorosh/Desktop/final/code/Project2/Project1/Code
sorosh@sorosh ~/D/f/c/P/P/Code (master)> make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmp > .gdbinit
*** Now run 'gdb'.
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512 -S -gdb tcp::26000

```

```
soroosh@soroosh ~/Desktop/final/code/Project2/Project1/Code
% fish
Welcome to fish, the friendly interactive shell
Type 'help' for instructions on how to use fish
soroosh@soroosh ~/Desktop/final/code (master)> gnu kernel
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel...
warning: File "/home/soroosh/Desktop/final/code/Project2/Project1/Code/.gdbinit" auto-loading has been declined by your 'auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
  add-auto-load-safe-path /home/soroosh/Desktop/final/code/Project2/Project1/Code/.gdbinit
line to your configuration file "/home/soroosh/.gdbinit".
To completely disable this security protection add
  set auto-load safe-path /
line to your configuration file "/home/soroosh/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual. E.g., run from the shell:
  info "(gdb)Auto-loading safe path"
(gdb) target remote tcp::26000
```

The screenshot shows the CLion IDE interface. On the left, there's a project tree with files like mkfs.c, mmu.h, mp.c, mp.d, mp.h, mp.o, Notes, param.h, picirq.c, picirq.d, picirq.o, pipe.c, pipe.d, pipe.o, pr.pl, and printf.c. In the center, there are two tabs: 'syscall.c' and 'getpid_gdb.asm'. The 'syscall.c' tab contains C code, including a breakpoint at line 144. The 'getpid_gdb.asm' tab contains assembly code. On the right, a terminal window titled 'QEMU [Paused]' shows the boot process of SeaBIOS (version 1.13.0-lubuntu1.1) from iPXE. It lists boot devices and users. Below the terminal, the CLion interface includes tabs for Git, TODO, Problems, Terminal, Python Packages, and Python Console.

```
138
139     int caller_count[100][100] = {0};

Machine View
SeaBIOS (version 1.13.0-lubuntu1.1)

IPXE (http://ipxe.org) 00:03.0 C000 PCI2.10 PnP PMM+1FF8CB00+1FECCB00 C000

Booting from Hard Disk...
cpu0: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
Group #22:
1. Danial Saeedi
2. Soroosh Sadeghian
3. Mohammad GharrehHasanloo
$ getpid_gdb

Terminal: Local x Local(2) x + v
144     struct proc *curproc = myproc();
(gdb) continue
Continuing.

^[[A
^[[A
Thread 2 hit Breakpoint 1, syscall () at syscall.c:144
144     struct proc *curproc = myproc();
(gdb) continue
Continuing.

Thread 2 hit Breakpoint 1, syscall () at syscall.c:144
144     struct proc *curproc = myproc();
(gdb) bt
#0  syscall () at syscall.c:144
#1  0x801069d1 in trap (tf=0x1010101) at trap.c:43
#2  0x801051f5 in popcli () at spinlock.c:117
#3  popcli () at spinlock.c:117
#4  0x1010101 in ?? ()
#5  0x00000000 in ?? ()
(gdb) []
```

دستورات را مطابق عکس‌های گذاشته شده و به ترتیب برای این مرحله اجرا می‌کنیم و نتیجه هم در ترمینال قابل مشاهده است.

