

بسمه تعالی



گزارش تمرین عملی دوم – سوال اول – درس سیستم‌های عامل دکتر اسدی – نیمسال اول 404-405

نویسنده‌گان:

1. سیداحمد موسوی‌اول – 402106648

2. عرفان تیموری – 402105813

گزارشکار.

(آ) هدف از این سوال بررسی وضعیت هسته (Kernel) و پروسه‌ی کاربر در لحظه‌ی ورود به یک فراخوانی سیستمی (syscall) با استفاده از GDB است.

ابتدا هسته xv6 را تحت qemu-gdb اجرا می‌کنیم. یک نقطه‌ی توقف (Breakpoint) در تابع syscall واقع در (kernel/syscall.c) تنظیم می‌کنیم و پس از اجرای برنامه (با دستور C)، اجرای برنامه را در این نقطه متوقف می‌کنیم.

```
kernel/proc.c
426 {
427     struct proc *p;
428     struct cpu *c = mycpu();
429
430     c->proc = 0;
431     for(;;){  
432         // The most recent process to run may have had interrupts  
433         // turned off; enable them to avoid a deadlock if all  
434         // processes are waiting. Then turn them back off  
435         // to avoid a possible race between an interrupt  
436         // and wfi.  
437         intr_on();  
438         intr_off();  
439
440         int found = 0;  
441         for(p = proc; p < &proc[NPROC]; p++) {  
442             acquire(&p->lock);  
443             if(p->state == RUNNABLE) {  
444                 // Switch to chosen process. It is the process's job  
445                 // to release its lock and then reacquire it  
446                 // before jumping back to us.  
447                 p->state = RUNNING;  
448                 c->proc = p;  
449                 switch(&c->context, &p->context);  
450
remote Thread 1.3 (src) In: scheduler
(gdb) L437 PC: 0x80001dce
```

برای پاسخ به سوال اول، گام‌های زیر برداشته می‌شود:

1. بررسی استک: ابتدا از دستور backtrace استفاده می‌شود. این دستور پشتی‌فراخوانی را نشان می‌دهد و تایید می‌کند که ما از یک تله (trap) در مد کاربر به تابع usertrap و سپس به syscall رسیده‌ایم.
2. شناسایی پروسه جاری: در xv6، تابع myproc() پوینتری به ساختار struct proc پروسه‌ای که در حال حاضر روی CPU اجرا می‌شود (حتی اگر در مد هسته باشد) برمی‌گرداند.

پس به کمک دستور `p` آنقدری جلو می‌رویم که به `struct p (next)` دسترسی داشته باشیم.

3. چاپ ساختار پروسه: با استفاده از دستور `(gdb) p/x *myproc()` تمام محتویات `struct proc` چاپ می‌شود.

```
kernel/syscall.c
122 [SYS_open] sys_open,
123 [SYS_write] sys_write,
124 [SYS_mknod] sys_mknod,
125 [SYS_unlink] sys_unlink,
126 [SYS_link] sys_link,
127 [SYS_mkdir] sys_mkdir,
128 [SYS_close] sys_close,
129 };
130
131 void
132 syscall(void)
B+ 133 [
134     int num;
135     struct proc *p = myproc();
136
> 137     num = p->trapframe->a7;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
142     } else {
143         printf("Nd %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
L137 PC: 0x80002856

remote Thread 1.1 (src) In: syscall
(gdb) backtrace
#0 syscall () at kernel/syscall.c:133
#1 0x000000000002602 in usertrap () at kernel/trap.c:68
#2 0x0000003fffff00c in ?? ()
(gdb) n
(gdb) n
(gdb) p/x *p
$1 = {lock = {locked = 0x0, name = 0x80007178, cpu = 0x0}, state = 0x4, chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1, parent = 0x0, kstack = 0x3fffffd000, sz = 0x4000,
pagetable = 0x87f52000, trapframe = 0x87f56000, context = {ra = 0x80001e62, sp = 0x3fffffdc80, s1 = 0x800126e8, s2 = 0x800122b8, s3 = 0x0, s4 = 0x8001dc50, s5 = 0x3,
s6 = 0x80023388, s7 = 0x0, s8 = 0x800234b0, s9 = 0x17e, s10 = 0xb0, s11 = 0x2}, ofile = {0x0 <repeats 16 times>}, cwd = 0x800207f8, name = {0x69, 0x6e, 0x69, 0x74,
0x0 <repeats 12 times>}}
(gdb)
```

نتیجه: این خروجی شامل اطلاعات حیاتی پروسه مانند `pid` (شناسه پروسه)، `sz.state` (اندازه حافظه) و مهم‌تر از همه، پوینتر `trapframe` است.

برای پاسخ به سوال دوم، وضعیت ذخیره‌شدهی پروسه در `trapframe` بررسی می‌شود:

1. دسترسی به رजیستر `a7`: طبق قرارداد فراخوانی در RISC-V، پروسه‌ی کاربر قبل از اجرای دستور `ecall` (که منجر به تله و ورود به هسته می‌شود)، شماره‌ی فراخوانی سیستمی مورد نظر خود را در رجیستر `a7` قرار می‌دهد، هسته این رجیسترهای را در پروسه ذخیره می‌کند. همچنین دستور `p/x myproc()->trapframe->a7` (gdb) `p/x myproc()->trapframe->a7` مستقیماً این مقدار را از `trapframe` می‌خواند و نمایش می‌دهد. اگر این دستور را اجرا کنیم خروجی `0xf` حاصل می‌شود که یعنی اولین فراخوانی سیستمی `xv6` دارای آیدی `15` است. حال اگر دستور

1. `P syscall[p → trapframe → a7]`

را اجرا کنیم `syscall` موردنظر به ما خروجی داده می‌شود که همان `sys-open` است؛ بنابراین اولین `syscall` اجرا شده در دستور `xv6` است.

2. تفسیر مقدار `a7`: مقدار نمایش داده شده (مثلًا `0x7` برای `fork` یا `0x1` برای `exec`) مشخص می‌کند که پروسه‌ی کاربر کدام سرویس را از هسته درخواست کرده است (این همان مقداری است که در کدهایی مانند `user/initcode.S` قبل از `ecall` بارگذاری می‌شود).

3. بررسی **sstatus** رجیستر (Status Register) وضعیت پردازنده مانند فعال بودن وقفه‌ها یا مد قبلی را قبل از وقوع تله نشان می‌دهد. این مقدار نیز در **trapframe** ذخیره می‌شود و با دستور **User/Supervisor**)

1. (gdb) p/x myproc()->trapframe->sstatus

قابل مشاهده است (همچنین می‌توان مستقیماً با **(gdb) p/x \$sstatus** CSR دسترسی پیدا کرد).

برای پاسخ به سوال سوم نیز کافیست دستور **(gdb) p/x \$sstatus** را اجرا کنیم. خروجی این دستور یک عدد 32 بیتی است که بیت 8 آن نشان‌دهنده مد قبلی سیستم قبل از وقوع تله است.

```
kernel/syscall.c
122 SYS_open| sys_open,
123 SYS_write| sys_write,
124 SYS_mknod| sys_mknod,
125 SYS_unlink| sys_unlink,
126 SYS_link| sys_link,
127 SYS_mkdir| sys_mkdir,
128 SYS_close| sys_close,
129 }
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *p = myproc();
136
> 137     num = p->trapframe->a7;
138     if(num > 0 && num < NELEM(syscalls)) {syscalls[num]} {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
142     } else {
143         printf("Md %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
}
remote Thread 1.2 (src) In: syscall
(gdb) n
(gdb) n
(gdb) p/x $sstatus
$1 = 0x200000022
(gdb)
```

اگر این بیت 0 باشد یعنی در حالت **user** بوده‌ایم و اگر 1 باشد یعنی در حالت **kernel** بوده‌ایم. حال با توجه به تصویر بالا این بیت 0 است؛ بنابراین قبل از وقوع وقفه در حالت **user** بوده‌ایم.

ب) در این قسمت از ما خواسته شده است دستور **top** را به سیستم‌عامل اضافه کنیم. برای اینکار ابتدا قطعه کد زیر را به کد **proc.c** اضافه کنیم:

این کد به ترتیب **PID**, **process**, وضعیت **STATE**, نام آن و سایز آن را نشان می‌دهد.

```
inefficient.cpp  C hello.c U  C top.c U  C proc.c M X  C user.h M  ♡ usys.pl M  M Makefile M
kernel > C proc.c
667 {
680     for(p = proc; p < &proc[NPROC]; p++){
688         printf("\n");
689     }
690
691     void
692     top(void)
693     {
694         static char *states[] = {
695             [UNUSED]    "unused",
696             [USED]      "used",
697             [SLEEPING]  "sleep",
698             [RUNNABLE]   "runble",
699             [RUNNING]    "run",
700             [ZOMBIE]    "zombie"
701         };
702         struct proc *p;
703         char *state;
704
705         printf("\n--- Process List ---");
706         printf("PID\tSIZE\tSTATE\tNAME\n");
707
708         for(p = proc; p < &proc[NPROC]; p++){
709             if(p->state == UNUSED)
710                 continue;
711             if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
712                 state = states[p->state];
713             else
714                 state = "???";
715             printf("%d\t%s\t%s\t%lu", p->pid, state, p->name, p->sz);
716             printf("\n");
717         }
718     }
719 }
720 }
```

```

kernel > C sysfile.c
479 {
496     if(copyout(p->pagetable, fdarray, (char*)&fd0, sizeof(fd0)) < 0 || 
497         p->write[ui] = 0,
500         fileclose(rf);
501         fileclose(wf);
502         return -1;
503     }
504     return 0;
505 }
506
507 uint64
508 sys_top(void)
509 {
510     top();
511     return 0;
512 }

```

مشابه تمرین 1 به بعضی از فایل‌ها باید این دستور جدید را اضافه کنیم؛ برای مثال به فایل‌های `syscall.h` دستور `top` را به همراه id آن اضافه می‌کنیم، یا در `syscall.c` فراخوانی سیستمی موردنظر را اضافه می‌کنیم و ...

```

kernel > C def.h
153 void          kvm_inithart(void);
154 void          kvmmap(pagetable_t, uint64, uint64, uint64, int);
155 int           mappages(pagetable_t, uint64, uint64, uint64, int);
156 pagetable_t   uvmcreate(void);
157 uint64        uvmalloc(pagetable_t, uint64, uint64, int);
158 uint64        uvmdealloc(pagetable_t, uint64, uint64);
159 int           uvmcopy(pagetable_t, pagetable_t, uint64);
160 void          uvmfree(pagetable_t, uint64);
161 void          uvmunmap(pagetable_t, uint64, uint64, int);
162 void          uvmclear(pagetable_t, uint64);
163 pte_t *       walk(pagetable_t, uint64, int);
164 uint64        walkaddr(pagetable_t, uint64);
165 int           copyout(pagetable_t, uint64, char *, uint64);
166 int           copyin(pagetable_t, char *, uint64, uint64);
167 int           copyinstr(pagetable_t, char *, uint64, uint64);
168 int           ismapped(pagetable_t, uint64);
169 uint64        vmfault(pagetable_t, uint64, int);
170
171 // plic.c
172 void          plicinit(void);
173 void          plicinithart(void);
174 int           plic_claim(void);
175 void          plic_complete(int);
176
177 // virtio_disk.c
178 void          virtio_disk_init(void);
179 void          virtio_disk_rw(struct buf *, int);
180 void          virtio_disk_intr(void);
181 void          top(void);
182
183 // number of elements in fixed-size array
184 #define NELEM(x) (sizeof(x)/sizeof((x)[0]))
185
186
187

```

در نهایت با اضافه کردن دستور `top` به تمام بخش‌هایی که لازم است خروجی دستور به شکل زیر می‌شود:

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ top
--- Process List ---PID SIZE STATE NAME
1 sleep init 16384
2 sleep sh 20480
3 run top 16384
$
```

مشاهده می‌کنیم که `PID` پردازه‌ها به همراه `SIZE`, `STATE` و دستوری که باعث اجرای آن شده است بعد از زدن دستور `top` که خودمان آن را تعریف کردیم به نمایش در می‌آید.