

به نام خدا

گزارش پروژه چهارم آزمایشگاه سیستم عامل

گروه 11:

حسین بیاتی 810198366

امیررضا کفاشان 810899064

حسین نوروزی 810899073

گیتهاب گروه:

https://github.com/HoseinBayati/UT_OS_LAB_CA4

آخرین کامیت پروژه:

eaca672f0c5dd2e8f052fa5595410408d2687400

پاییز 1402

-1

در پردازنده‌های جدیدتر برای جلوگیری از انقطاع روند کار (Preemption) اینترپت‌های سیستم را غیرفعال می‌کنیم. این کار موجب می‌شود خود وقفه‌ها باعث ایجاد race condition نشوند. اگرچه که این راهکار برای جلوگیری از این مشکلات مخصوصاً در سیستم‌های بی‌درنگ اصلاً مناسب نیست.

-2

تابع `cli`، `interrupt`‌ها را غیرفعال و `sti`، فعال می‌کند. عملکرد توابع `pushcli()` و `popcli()` هم همین است با این تفاوت که پس از یک‌بار صدا زدن `sti`، `interrupt`‌ها فعال می‌شوند در حالی که برای فعال‌سازی `interrupt`‌ها توسط دو دستور دوم باید به همان تعدادی که `pushcli()` را فراخوانی کردیم، `popcli()` را نیز فراخوانی کنیم.

-۳

```
25     acquire(struct spinlock *lk)
26     {
27         pushcli(); // disable interrupts to avoid deadlock.
28         if(holding(lk))
29             panic("acquire");
30
31         // The xchg is atomic.
32         while(xchg(&lk->locked, 1) != 0)
33             ;
34
35         // Tell the C compiler and the processor to not move loads or stores
36         // past this point, to ensure that the critical section's memory
37         // references happen after the lock is acquired.
38         __sync_synchronize();
39
40         // Record info about lock acquisition for debugging.
41         lk->cpu = mycpu();
42         getcallerpcs(&lk, lk->pcs);
43     }
```

به طور کلی `spinlock` پیاده‌سازی خوبی برای ساختار `lock` نیست زیرا باعث `busy waiting` می‌شود و زمان قابل توجهی از پردازنده در حلقه `while` بالا به هدر می‌رود. به طور خاص این ساختار در پردازنده‌های تک هسته‌ای می‌تواند موجب `deadlock` شود. در حالتی ممکن است زمانی که یک پردازنده قفل را تصاحب می‌کند، پردازنده دیگر در حلقه `while` با دستور `xchg` در تلاش برای تصاحب قفل باشد.

-4

هدف اصلی از تعریف دستور `amoswap`، ایجاد یک عملیات تبادلی است. عملیات تبادلی به معنای تعویض مقادیر دو مکان حافظه در یک عمل بدون تداخل `thread`‌ها است. به این صورت که مقدار قبلی در یک مکان حافظه با مقدار جدید تعویض می‌شود و مقدار قبلی مکان حافظه در خروجی برگردانده می‌شود. این دستور برای

جلوگیری از مشکلاتی مثل race condition کاربرد دارد. در سوال قبل دیدیم که در حلقه while یک عملیات swap به طور اتمیک انجام می‌شد؛ این دستور در چنین جایی کاربرد دارد

-5

```
22 void
23 acquiresleep(struct sleeplock *lk)
24 {
25     acquire(&lk->lk);
26     while (lk->locked) {
27         sleep(lk, &lk->lk);
28     }
29     lk->locked = 1;
30     lk->pid = myproc()->pid;
31     release(&lk->lk);
32 }
33
34 void
35 releasesleep(struct sleeplock *lk)
36 {
37     acquire(&lk->lk);
38     lk->locked = 0;
39     lk->pid = 0;
40     wakeup(lk);
41     release(&lk->lk);
42 }
```

تعامل بین پردازنده‌ها در توابع sleep و wakeup مطرح است. زمانی که تابع sleep فراخوانی می‌شود وضعیت پردازنده به sleeping تغییر پیدا می‌کند و سپس پردازنده قفل sleeplock را رها می‌کند تا زمانی که توسط پردازنده دیگری wake up شود. در تابع wakeup نیز با استفاده از ورودی void* chan پردازنده‌هایی که در انتظار گرفتن sleeplock هستند به حالت runnable تغییر پیدا می‌کنند تا در تابع scheduler فرصت گرفتن پردازنده را داشته باشند.

-6

- Runnable: در صف اجرای scheduler قرار دارد و آماده اجرا است.
- Running: در حال اجرا توسط پردازنده.
- Sleeping: پردازنده به هدف خاصی در انتظار است و فعلاً به آن پردازنده تخصیص داده نمی‌شود.
- Embryo: پردازنده‌ای که تازه متولد شده است.

- **Unused**: تعداد مشخصی جایگاه برای پردازنده‌ها داریم که در صورتی که حاوی پردازنده ای نباشند وضعیت آن‌ها Unused است.
 - **Zombie**: پردازنده پیش از اتمام کار و تغییر وضعیت به Unused در این حالت قرار می‌گیرد تا والد آن از اتمام کار آن با خبر شود.
- هدف تابع sched سپردن پردازنده به دست زمانبند است را پردازنده مناسب بعدی را برای اجرا انتخاب کند. پس از اتمام شدن کار یک پردازنده به هردلیل این تابع فراخوانی می‌شود.

```

sched(void)
{
    int intena;
    struct proc *p = myproc();
    |
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags() & FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
    mycpu()->intena = intena;
}

```

-7

برای این منظور کافی است در ابتدای تابع releasesleep، به کمک تابع holdingsleep و با استفاده از pid بررسی کنیم که قفل در اختیار کسی است که درخواست release کردن آن را دارد یا خیر. به طور کلی پیاده‌سازی قفل‌ها در لینوکس پیچیدگی بسیار بیشتری دارد. توابع spinlock() و spinunlock() مشابه توابع acquire و release در 6xv هستند. توابع mutex_lock() و mutex_unlock() نیز مشابه acquiresleep و releasesleep هستند.

-8

برای جلوگیری از هزینه‌های مرتبط با قفل‌ها، بسیاری از سیستم‌عامل‌ها از ساختارهای و الگوریتم‌های بدون قفل استفاده می‌کنند. به عنوان مثال، می‌توان یک لیست پیوندی پیاده‌سازی کرد که برای جستجو در لیست احتیاجی به قفل ندارد و برای درج یک مورد در لیست از یک دستور اتمی استفاده می‌کند. برنامه‌نویسی بدون قفل پیچیده‌تر است اما نیاز به توجه درباره

ترتیب اجرای دستورات و بازسازی حافظه دارد. این ساختارها معمولاً در مواردی که تعداد نخ‌ها زیاد است و از تعداد کمی منابع مشترک استفاده می‌شود، کارایی بالاتری نسبت به روش‌های blocking همگام‌سازی دارند.

قسمت دوم

(الف)

برای حل مشکل مزبور در سطح سخت‌افزار، از یک تکنیک به نام Cache Coherence استفاده می‌شود.

این تکنیک برای اطمینان از هماهنگی محتوای حافظه‌های نهان در سطح مختلف پردازنده‌ها به کار می‌رود.

1- پروتکل همگانی:

از یک پروتکل همگانی برای دسترسی و به‌روزرسانی حافظه نهان در سطح مختلف استفاده می‌شود.

- این پروتکل مشخص می‌کند که چگونه دسترسی به حافظه نهان انجام می‌شود و چگونه تغییرات در یک پردازنده باید به سایر پردازنده‌ها اعلام شود.

2- invalidation and Update

- وقتی یک پردازنده مقدار حافظه نهان را تغییر می‌دهد، این تغییر به سایر پردازنده‌ها اعلام می‌شود.

- یکی از روش‌های اعلام تغییر، استفاده از عملیات invalidation است، که به معنای این است که داده موجود در حافظه نهان دیگر معتبر نیست.

روش دیگر عملیات به‌روزرسانی است، که در آن داده جدید به سایر حافظه‌های نهان ارسال می‌شود.

3- Write back & Write through

در روش Write-Back تغییرات تنها در صورت نیاز مانند بارگذاری داده جدید به حافظه اصلی اعمال می‌شود. در غیر این‌صورت تغییرات در حافظه نهان نگه داشته خواهند شد.

در روش Write Through هر تغییر به صورت فوری در حافظه اصلی نیز اعمال می‌شود.

۴- مدیریت تداخل

برنامه‌های کاربردی باید به دقت مدیریت شوند تا از تداخلات در دسترسی به حافظه نهان جلوگیری شود.
استفاده از متغیرهای همگانی، قفل‌های نرم افزاری، یا تنظیمات خاص می‌تواند در مدیریت تداخلات مؤثر باشد.

(ب)

1.

- هر پردازنده که می‌خواهد به قفل دسترسی پیدا کند، یک بلیت افزایش می‌یابد.
 - هنگامی که قفل آزاد می‌شود، پردازنده با کمترین بلیت (به عبارت دیگر، کوچکترین شماره) می‌تواند قفل را بگیرد.
2. تداخل کمتر:

- در مقایسه با قفل‌های دیگر، تداخل بین پردازنده‌ها برای درخواست دسترسی به قفل را کاهش می‌دهد.
 - این امکان را فراهم می‌کند که پردازنده‌ها کمتر به یکدیگر وابسته باشند و همگام‌سازی بهبود یابد.
3. عدم انتظار فعال:

- پردازنده‌ها برای دستیابی به قفل منتظر نمی‌مانند و فوراً یک بلیت به آن‌ها اختصاص داده می‌شود.
- عدم وجود Active Spinning باعث بهبود عملکرد سیستم می‌شود.

(ج)

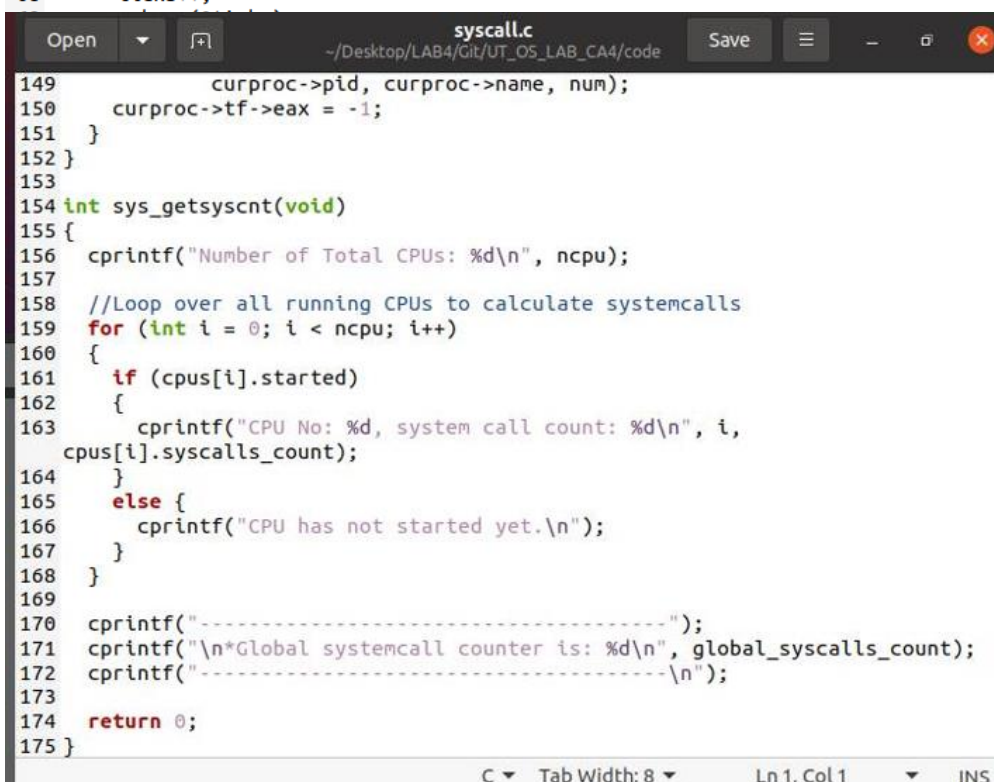
1. استفاده از ماکروهای پیش‌پردازنده: در فایل‌های سرآیند (header) لینوکس، می‌توانیم از ماکروهای پیش‌پردازنده مانند `__percpu` استفاده کنیم. این ماکرو، تعریف متغیرهایی را که به صورت مخصوص برای هر هسته هستند، انجام می‌دهد. با استفاده از این ماکرو، می‌توانیم متغیرها را به صورت مخصوص برای هر هسته تعریف کنیم.
2. استفاده از متغیرهای خصوصی هسته: (Kernel Thread Local Storage) در برخی معماری‌ها، می‌توانیم از متغیرهای خصوصی هسته استفاده کنیم. این متغیرها برای هر هسته مجزا تعریف می‌شوند و به طور خاص برای هسته فعلی قابل دسترسی هستند.
3. استفاده از توابع و ساختارهای خصوصی هسته: در برخی موارد، می‌توانیم از توابع و ساختارهای خصوصی هسته استفاده کنیم که تعیین کننده هسته فعلی است. این توابع و ساختارها اطلاعاتی مانند شماره هسته فعلی را در اختیار ما قرار می‌دهند و می‌توانیم بر اساس آنها عملیات خاصی را انجام دهیم.
4. استفاده از متغیرهای `Global` مختص هسته: در برخی موارد، می‌توانیم از متغیرهای `Global` استفاده کنیم و با استفاده از توابع و مکانیزم‌های هسته، به آنها دسترسی مختص هسته‌ها را فراهم کنیم. برای مثال، با استفاده از توابع هسته مانند `smp_processor_id()` می‌توانیم شماره هسته فعلی را دریافت کنیم و بر اساس آن به داده‌های مختص هر هسته دسترسی داشته باشیم.

5. استفاده از نخ‌های خصوصی هسته (Kernel Thread) در لینوکس، می‌توانیم نخ‌های خصوصی هسته ایجاد کنیم که به هر هسته مربوط می‌شوند. این نخ‌ها می‌توانند داده‌های خصوصی را برای هر هسته ایجاد کنند و از آنها در زمان اجرا استفاده کنند

پیاده سازی فراخوانی سیستمی متغیرهای مختص هر پردازنده:

برای اینکار تغییرات لازم را در فایل های trap.c و syscall.c اعمال خواهیم کرد:

```
37 void idtinit(void)
38 {
39     lidt(idt, sizeof(idt));
40 }
41
42 // PAGEBREAK: 41
43 void trap(struct trapframe *tf)
44 {
45     if (tf->trapno == T_SYSCALL)
46     {
47         if (myproc()->killed)
48             exit();
49         myproc()->tf = tf;
50         syscall();
51         acquire(&global_syscalls_count_lock);
52         mycpu()->syscalls_count++;
53         global_syscalls_count++;
54         wakeup(&ticks);
55         release(&global_syscalls_count_lock);
56
57         if (myproc()->killed)
58             exit();
59         return;
60     }
61
62     switch (tf->trapno)
63     {
64     case T_IRQ0 + IRQ_TIMER:
65         if (cpuid() == 0)
66         {
67             acquire(&tickslock);
68             ticks++;
```



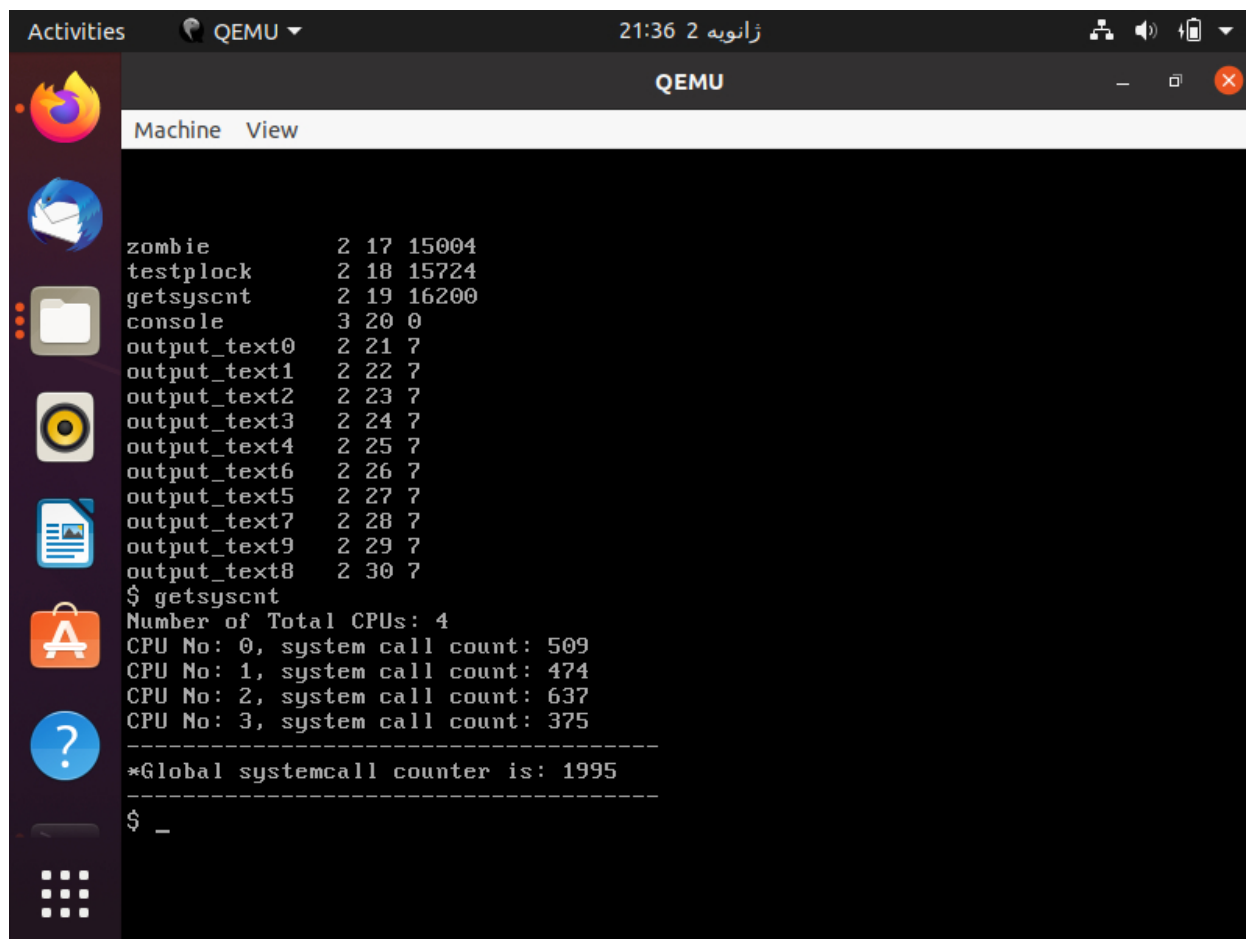
```
149         curproc->pid, curproc->name, num);
150         curproc->tf->eax = -1;
151     }
152 }
153
154 int sys_getsyscnt(void)
155 {
156     cprintf("Number of Total CPUs: %d\n", ncpu);
157
158     //Loop over all running CPUs to calculate systemcalls
159     for (int i = 0; i < ncpu; i++)
160     {
161         if (cpus[i].started)
162         {
163             cprintf("CPU No: %d, system call count: %d\n", i,
164                 cpus[i].syscalls_count);
165         }
166         else {
167             cprintf("CPU has not started yet.\n");
168         }
169     }
170     cprintf("-----\n");
171     cprintf("\n*Global syscall counter is: %d\n", global_syscalls_count);
172     cprintf("-----\n");
173
174     return 0;
175 }
```

C Tab Width: 8 Ln 1, Col 1 INS

برنامه سطح کاربر برای تست فراخوانی سیستمی مورد نظر:

```
6
7 #define NUM_PROCESSES 10
8
9 void writeToFiles(int p_order) {
10     char filename[15] = "output_text";
11
12     filename[11] = '0' + p_order;
13     filename[12] = '\\0';
14
15     int fd = open(filename, O_WRONLY | O_CREATE);
16     if (fd >= 0) {
17         write(fd, "content", 7);
18         close(fd);
19     }
20 }
21
22 int main(int argc, char *argv[]) {
23     for (int i = 0; i < NUM_PROCESSES; i++) {
24         int pid = fork();
25         if (pid == 0) {
26             // Child process
27             writeToFiles(i);
28             exit();
29         }
30     }
31
32     for (int i = 0; i < NUM_PROCESSES; i++) {
33         wait();
34     }
35
36     getsyscnt();
37
38     exit();
39 }
```

خروجی برنامه:



The screenshot shows a QEMU terminal window with a dark background and light-colored text. The window title is 'QEMU'. The terminal output lists system call counts for several processes, followed by a summary of system call counts for all CPUs.

```
Machine View

zombie          2 17 15004
testptlock      2 18 15724
getsyscnt       2 19 16200
console         3 20 0
output_text0    2 21 7
output_text1    2 22 7
output_text2    2 23 7
output_text3    2 24 7
output_text4    2 25 7
output_text6    2 26 7
output_text5    2 27 7
output_text7    2 28 7
output_text9    2 29 7
output_text8    2 30 7
$ getsyscnt
Number of Total CPUs: 4
CPU No: 0, system call count: 509
CPU No: 1, system call count: 474
CPU No: 2, system call count: 637
CPU No: 3, system call count: 375
-----
*Global systemcall counter is: 1995
-----
$ _
```

- در این تصویر برای بار دوم فراخوانی سیستمی `getsyscnt` صدا زده شده است و با دستور `ls` فایل های جدید نمایش داده شده اند.

همانطور که مشاهده می شود مقدار Global برابر است با جمع مقادیر سیستم کال های همه CPU های سیستم.

پیاده سازی قفل با اولویت:

```
#define MAX_QUEUE_LENGTH 20

int priority_queue[MAX_QUEUE_LENGTH];
int pq_size = 0;
int highest_priority = 0;

struct prioritylock plocks[PLOCKS_COUNT];
You, 2 hours ago • Uncommitted changes
void init_priority_lock(struct prioritylock *lk, char *name)
{
    initlock(&lk->lk, "priority lock");
    lk->name = name;
    lk->locked = 0;
    lk->pid = 0;
}

int remove_from_priority_queue()
{
    int target_pid = myproc()->pid;

    for (int i = 0; i < pq_size; i++)
    {
        if (priority_queue[i] == target_pid)
        {
            for (int j = i; j < pq_size; j++)
            {
                priority_queue[j] = priority_queue[j+1];
            }
            return 1;
        }
    }

    return 0;
}
```

```

int add_to_priority_queue(){
    int pid = myproc()->pid;
    if (pq_size == MAX_QUEUE_LENGTH)
    {
        return 0;
    }

    int i = 0;
    while(pid > priority_queue[i] && i < pq_size)
    {
        i++;
    }

    for (int j = pq_size; j > i; j--)
    {
        priority_queue[j] = priority_queue[j-1];
    }

    priority_queue[i] = pid;

    return 1;
}

void acquire_priority_lock(struct prioritylock *lk)
{
    acquire(&lk->lk);

    int pid = myproc()->pid;
    add_to_priority_queue();

    while (lk->locked || !(pid == priority_queue[0]))
    {
        sleep(lk, &lk->lk);
    }

    lk->locked = 1;
    lk->pid = myproc()->pid;
    release(&lk->lk);
}

```

```

void release_priority_lock(struct prioritylock *lk)
{
    acquire(&lk->lk);
    pq_size--;
    remove_from_priority_queue();
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
    release(&lk->lk);
}

int holding_priority_lock(struct prioritylock *lk)
{
    int r;

    acquire(&lk->lk);
    r = lk->locked && (lk->pid == myproc()->pid);
    release(&lk->lk);
    return r;
}

```

سیستم‌کال‌های مربوطه:

```

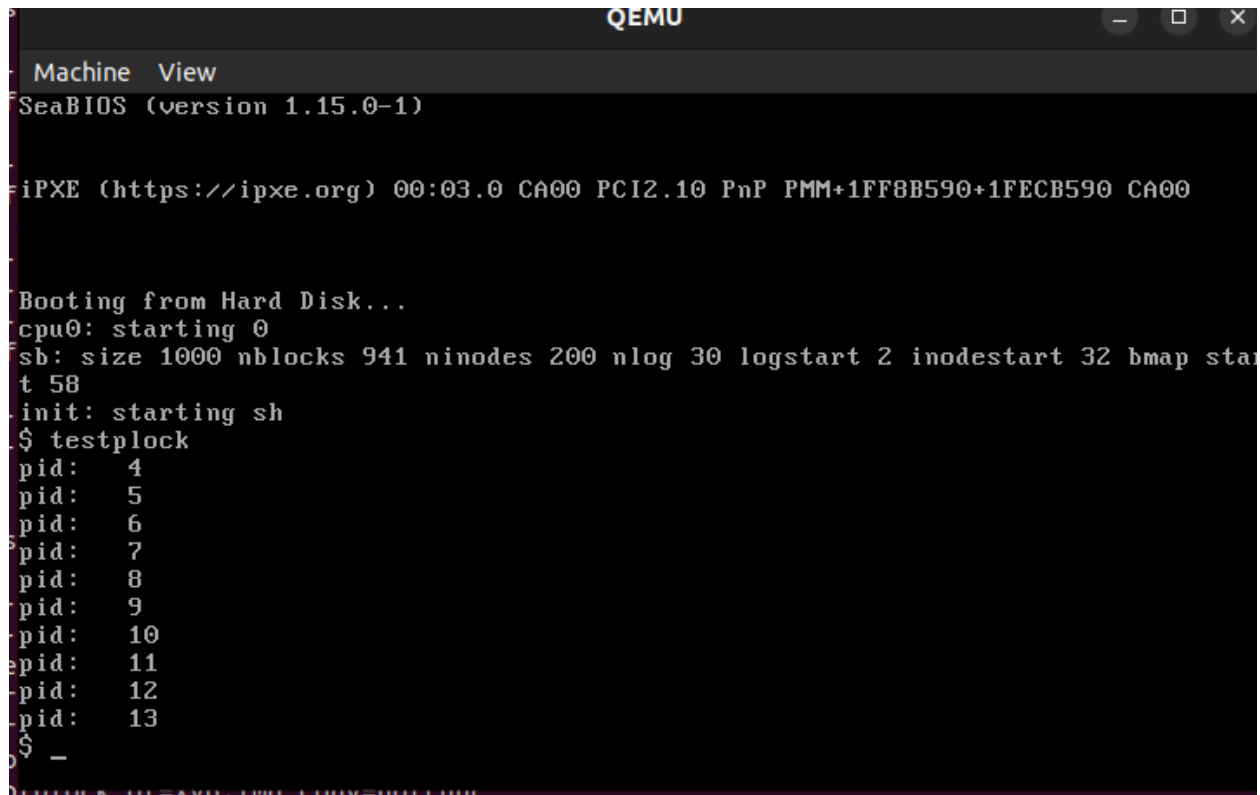
void plock_init(int plock_id)
{
    init_priority_lock(&(plocks[plock_id]), "priority_lock");
}

void plock_acquire(int plock_id)
{
    acquire_priority_lock(&(plocks[plock_id]));
}

void plock_release(int plock_id){
    release_priority_lock(&(plocks[plock_id]));
}

```

برنامه‌ی تست: testplock.c اجرای ده ریسه که مقدار pid خود را چاپ می‌کنند. (در ناحیه بحرانی)
این ده پردازش، با موفقیت، به ترتیب pid به ناحیه بحرانی دسترسی پیدا می‌کنند و بنابراین به ترتیب pid خود را چاپ می‌کنند.



```
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
$ testplock
pid: 4
pid: 5
pid: 6
pid: 7
pid: 8
pid: 9
pid: 10
pid: 11
pid: 12
pid: 13
$ -
```