

آزمایشگاه سیستم عامل

گزارش پروژه چهارم

- آراین باستانی - 810100088
- مهدیار هرندي - 810199596
- محمدرضا نعمتی - 810100226
- <https://github.com/AryanBastani/OS-Lab-4>
- Last commit hash: [d44b1f4fc836c0217c05a2ecd82baa89406afbd2](#)

1- در xv6، قفل‌های چرخشی برای کنترل دسترسی به منابع مشترک استفاده می‌شوند. وقتی یک process قفل را دریافت می‌کند، سایر process‌هایی که قصد دسترسی به همان منبع را دارند، در یک حلقه چرخشی منتظر می‌مانند تا قفل آزاد شود. اگر وقفه‌ها در هنگام دریافت قفل فعال باشند، ممکن است که یک وقفه در حین اجرای یک process که قفل را در اختیار دارد، فعال شود. اگر Interrupt handler نیز سعی در دریافت همان قفل کند، ممکن است به یک deadlock منجر شود. در یک سناریوی deadlock، فرآیندی که قفل را در اختیار دارد، منتظر وقوع یک رویداد (مثلاً پایان یک وقفه) می‌ماند که توسط خود آن فرآیند باید اتفاق بیفتد. اما چون فرآیند قادر به پیشرفت نیست (زیرا در وقفه گیر کرده)، وقفه نیز نمی‌تواند به اتمام برسد. این وضعیت می‌تواند CPU را در یک حالت blocked قرار دهد، جایی که نه process و نه Interrupt handler نمی‌توانند پیشرفت کنند. برای جلوگیری از این حالت، xv6 در هنگام دریافت قفل وقفه‌ها را غیرفعال می‌کند. این امر اطمینان می‌دهد که هیچ وقفه‌ای نمی‌تواند در حین اجرای کدی که قفل را در اختیار دارد، فعال شود و از وقوع deadlock جلوگیری می‌کند.

2- برای غیر فعال کردن وقفه‌ها در ابتدا تابع pushcli را صدا می‌زنیم که وقفه‌ها را غیر فعال میکند و پس از اجرای کامل critical area ابتدا قفل را با تابع release باز می‌کنیم. سپس popcli را صدا می‌زنیم تا وقفه‌ها دوباره فعال شوند. تابع pushcli خودش تابع cli (clear interrupt) را صدا می‌زند و وقفه‌ها را غیر فعال میکند. اما تابع popcli زمانی sti (set interrupt) را صدا می‌زند که مقدار متغیر ncli صفر باشد. در اینجا از

ncli به منظور شبیه سازی استک استفاده می شود. مقدار متغیر ncli تعداد فراخوانی هر تابع در هر پراسس ذخیره می کند و با هر pushcli یک عدد زیاد و با هر popcli یک عدد کم می شود. اگر مقدار ncli صفر بود وقفه ها غیر فعال و در غیر این صورت وقفه ها فعال می شوند.

3- تعریف توابع acquire و holding در xv6 به صورت زیر در فایل spinlock است. در acquire تابع holding فراخوانی می شود تا بررسی شود اگر cpu قفل را نگه داشته است، ادامه acquire اجرا نشود و panic کند. در غیر این صورت وارد حلقه وایل می شود تا فرایند لاک انجام شود. در واقع پیاده سازی قفل ها در سیستم عامل xv6 به صورت busy waiting است و اگر پردازنده تک هسته ای باشد، در یک لحظه میتواند یا مشغول انجام پراسس باشد و یا قفل را نگهداری کند و این دو کار باهم ممکن نیست.

```
24 void
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 }
44
45 // Check whether this cpu is holding the lock.
46 int
47 holding(struct spinlock *lock)
48 {
49     int r;
50     pushcli();
51     r = lock->locked && lock->cpu == mycpu();
52     popcli();
53     return r;
54 }
```

4- amoswap یک دستور atomic در مجموعه دستورات RISC-V است. این دستور در xv6 برای پیاده سازی قفل ها با swap کردن اتمیک مقادیر در مموری استفاده می شود و این اطمینان را فراهم می کند که دریافت و آزادسازی قفل به صورت عملیات اتمیک انجام شود. منظور از اتمیک، به این معناست که در حین انجام آن interrupt نمی تواند رخ دهد. به همین دلیل از رخ دادن race conditions در یک محیط multi-threaded جلوگیری می شود. استفاده و دستور اسمبلی آن بصورت *amoswap.w rd, rs2, (rs1)*

است. رجیستر rd مقصد است که در آن مقدار لود می شود. رجیستر rs2 مبدا است که مقدار آن با rd جابجا خواهد شد. رجیستر rs1 هم رجیستری است که آدرس خانه ای که از آن مقدار لود شود را ذخیره میکند.

5- این توابع برای مدیریت قفل‌ها زمانی که یک process در حالت sleep قرار میگیرد، استفاده می‌شوند. acquiresleep اطمینان می‌دهد که یک process قبل از قرار گرفتن در حالت sleep، قفل را دریافت کند. releasesleep، از طرف دیگر قفل را زمانی که process بیدار می‌شود، آزاد می‌کند. تفاوت اصلی نسبت به قفل‌های معمولی این است که این توابع حالت sleep برای process را مدیریت می‌کنند، از این رو از سناریوهای deadlock پراسسی که در حالت sleep قرار دارد و برای مدت زمان نامحدودی قفل را نگه دارد جلوگیری می‌کند.

6-حالات مختلف پردازش‌ها در xv6:

حالات مختلف پردازش به صورت زیر در فایل proc.h نوشته شده اند.

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

1. **UNUSED**: این وضعیت نشان میدهد که یک slot در process table توسط هیچ پردازش‌ای استفاده نمی‌شود.
2. **EMBRYO**: این وضعیت نشان میدهد که پردازش در حال ساخته شدن است.
3. **SLEEPING**: فرآیند در این حالت منتظر وقوع یک رویداد خاص است. به عنوان مثال، ممکن است منتظر داده‌های ورودی یا پایان یک عملیات I/O باشد. در حالت sleeping، پراسس منابع کمتری مصرف می‌کند زیرا فعالانه توسط CPU اجرا نمی‌شود.
4. **RUNNABLE**: در این حالت، پراسس آماده اجرا است اما به دلیل نبودن CPU در دسترس، در حال حاضر اجرا نمی‌شود. فرآیندهای در این حالت در صف انتظار scheduler قرار می‌گیرند.
5. **RUNNING**: پراسس در این حالت توسط CPU اجرا می‌شود.
6. **ZOMBIE**: این حالت زمانی رخ می‌دهد که پراسسی تمام شده باشد اما هنوز منابعی مانند process identifier را در اختیار دارد. این پراسس‌ها منتظر می‌مانند تا parent process وضعیت خروجی آنها را بررسی کند.

تابع sched در xv6 مسئول scheduling برای process ها است. این تابع حالت های مختلف process ها مانند RUNNING, SLEEPING, و RUNNABLE را مدیریت می کند. این تابع مسئول context switching بین process است. این به این معناست که زمانی که یک فرآیند از حالت RUNNING به RUNNABLE تغییر می کند، sched فرآیند بعدی را که باید اجرا شود، انتخاب می کند. این تابع با استفاده از round-robin اطمینان حاصل می کند که هر process در نوبت خود فرصت اجرا دارد. این تابع نقش کلیدی در مدیریت استفاده از CPU دارد، به گونه ای که اطمینان حاصل می شود CPU به طور موثر بین process مختلف تقسیم می شود.

7- تنها کاری که باید انجام داد این است که در تابع releasesleep قبل از رهاسازی قفل، pid پردازنده را چک کنیم و اگر برابر با شماره پردازنده ی صاحب قفل بود آن را رها کنیم و در غیر این صورت کاری نمیکنیم:

```
1 void
2 releasesleep(struct sleeplock *lk)
3 {
4     acquire(&lk->lk);
5
6     if(myproc()->pid == lk->pid)
7     {
8         lk->locked = 0;
9         lk->pid = 0;
10        wakeup(lk);
11    }
12
13    release(&lk->lk);
14 }
```

در لینوکس نیز در فایل `mutex.h` در استراکت `mutex` متغیری به نام `owner` داریم که نشان دهنده ی صاحب قفل است و همانند کاری که کردیم، در حین رهاسازی بررسی میشود که این پردازش، همان `owner` است یا خیر:

```
1 struct mutex {
2     atomic_long_t    owner;
3     raw_spinlock_t   wait_lock;
4     #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
5         struct optimistic_spin_queue osq; /* Spinner MCS lock */
6     #endif
7     struct list_head  wait_list;
8     #ifdef CONFIG_DEBUG_MUTEXES
9         void          *magic;
10    #endif
11    #ifdef CONFIG_DEBUG_LOCK_ALLOC
12        struct lockdep_map  dep_map;
13    #endif
14 };
```

8- الگوریتم های lock-free یا بدون قفل، روشی هستند که در آن ها چندین نخ (thread) می توانند به طور همزمان به یک داده مشترک دسترسی داشته باشند، بدون اینکه نیاز به استفاده از قفل (lock) یا مکانیزم های همگام سازی دیگر داشته باشند. این روش مزایایی مانند بهبود عملکرد، کاهش مصرف منابع و جلوگیری از بروز مشکلاتی مانند deadlock یا starvation را دارد. اما همچنین معایبی نیز دارد، مانند پیچیدگی بیشتر، نیاز به استفاده از دستورات اتمیک (atomic) یا مقاوم در برابر خطا (fault-tolerant)، و احتمال بروز مشکلاتی مانند ABA problem یا memory reclamation.

برای مقایسه، برنامه نویسی با lock روشی است که در آن ها یک نخ برای دسترسی به یک داده مشترک، آن را قفل می کند تا نخ های دیگر نتوانند به آن دسترسی داشته باشند. این روش مزایایی مانند سادگی بیشتر، امکان استفاده از دستورات عادی، و جلوگیری از مشکل ABA را دارد. اما همچنین معایبی نیز دارد، مانند کاهش عملکرد، افزایش مصرف منابع و احتمال بروز مشکلاتی مانند deadlock یا priority inversion.

پیاده سازی متغیرهای مختص هر هسته پردازنده

الف) یکی از روشهای حل مشکل نامعتبر شدن مقادیر حافظه نهان در سطح سخت افزار، استفاده از پروتکل‌های هماهنگی حافظه نهان است. این پروتکل‌ها مکانیزم‌هایی را فراهم می‌کنند که با استفاده از آن‌ها، حافظه‌های نهان می‌توانند به صورت خودکار یا با درخواست پردازنده، مقادیر خود را با حافظه اصلی یا حافظه‌های نهان دیگر همگام‌سازی کنند. برای مثال، یکی از پروتکل‌های معروف پروتکل MSI است که هر بلوک حافظه نهان را در یکی از سه حالت Invalid یا Modified, Shared قرار می‌دهد. این حالت‌ها نشان می‌دهند که مقدار بلوک حافظه نهان چقدر با مقدار متناظر آن در حافظه اصلی یا حافظه‌های نهان دیگر همخوانی دارد. با استفاده از این پروتکل، هرگاه یک پردازنده بخواهد در یک بلوک حافظه نهان بنویسد، ابتدا باید از حالت بلوک حافظه نهان مطلع شود و در صورت لزوم، پیام‌هایی را به پردازنده‌های دیگر بفرستد تا مقادیر حافظه نهان را به‌روزرسانی کند.

ب) قفل‌های بلیت یا ticket lock نوعی از قفل‌های اسپین‌لوک هستند که برای همگام‌سازی دسترسی به منابع اشتراکی در سیستم‌های چند پردازنده استفاده می‌شوند. این قفل‌ها از دو متغیر ticket و turn استفاده می‌کنند که هر دو از نوع عدد صحیح هستند. هرگاه یک پردازنده بخواهد به منبع اشتراکی دسترسی پیدا کند، ابتدا یک بلیت از متغیر ticket می‌گیرد و سپس منتظر می‌ماند تا متغیر turn با مقدار بلیتش برابر شود. در این صورت، پردازنده می‌تواند به منبع اشتراکی دسترسی داشته باشد. پس از اتمام کار، پردازنده مقدار متغیر turn را یک واحد افزایش می‌دهد تا پردازنده بعدی بتواند به نوبت خود دسترسی داشته باشد.

قفل‌های بلیت می‌توانند مشکل نامعتبر شدن مقادیر حافظه نهان را حل کنند. زیرا این قفل‌ها از مکانیزمی به نام write invalidate استفاده می‌کنند. این مکانیزم باعث می‌شود که هرگاه یک پردازنده بخواهد در یک بلوک حافظه نهان بنویسد، مقادیر متناظر آن بلوک در حافظه‌های نهان دیگر پردازنده‌ها نامعتبر شوند. بدین ترتیب، هیچ پردازنده‌ای نمی‌تواند از مقادیر قدیمی حافظه نهان استفاده کند و باید مقادیر جدید را از حافظه اصلی یا پردازنده‌ای که در حال نوشتن است دریافت کند.

ج) یکی از روش‌های تعریف داده‌های مختص هر هسته در لینوکس، استفاده از ماکروی `per_cpu` است. این ماکرو باعث می‌شود که یک متغیر به ازای هر هسته پردازنده یک نسخه جداگانه داشته باشد. برای استفاده از این ماکرو، باید متغیر را با کلمه کلیدی `percpu__` تعریف کرد و سپس با استفاده از تابع `per_cpu` به مقدار آن دسترسی پیدا کرد. برای مثال، کد زیر یک متغیر از نوع عدد صحیح به نام `counter` را به ازای هر هسته تعریف می‌کند و مقدار آن را با یک واحد افزایش می‌دهد:

```
1 #include <linux/percpu.h>
2
3 static int __percpu counter;
4
5 void increment_counter(void)
6 {
7     per_cpu(counter, smp_processor_id())++;
8 }
```

بخش کد:

برای متغیرهای محلی هر هسته، در struct سی پی یو یک متغیر جدید برای نگهداری تعداد فراخوانی های سیستمی تعریف میکنیم:

...	@@ -1,15 +1,17 @@		
1	// Per-CPU state	1	// Per-CPU state
2	struct cpu {	2	struct cpu {
3	uchar apicid; // Local APIC ID	3	uchar apicid; // Local APIC ID
4	struct context *scheduler; // switch() here to enter scheduler	4	struct context *scheduler; // switch() here to enter scheduler
5	struct taskstate ts; // Used by x86 to find stack for interrupt	5	struct taskstate ts; // Used by x86 to find stack for interrupt
6	struct segdesc gdt[NSEGs]; // x86 global descriptor table	6	struct segdesc gdt[NSEGs]; // x86 global descriptor table
7	volatile uint started; // Has the CPU started?	7	volatile uint started; // Has the CPU started?
8	int ncli; // Depth of pushcli nesting.	8	int ncli; // Depth of pushcli nesting.
9	int intena; // Were interrupts enabled before pushcli?	9	int intena; // Were interrupts enabled before pushcli?
10	struct proc *proc; // The process running on this cpu or null	10	struct proc *proc; // The process running on this cpu or null
		11 +	int num_of_syscalls; // number of system call for each core
11	};	12	};
12		13	

و از آنجایی که میخواهیم تعداد هسته ها برابر با 4 تا باشد، باید در makefile متغیر NCPU و CPUS در param.h هر دو را برابر 4 قرار دهیم:

▼ 2	param.h	...	
...	@@ -1,6 +1,6 @@		
1	#define NPROC 64 // maximum number of processes	1	#define NPROC 64 // maximum number of processes
2	#define KSTACKSIZE 4096 // size of per-process kernel stack	2	#define KSTACKSIZE 4096 // size of per-process kernel stack
3 +	#define NCPU 8 // maximum number of CPUs	3 +	#define NCPU 4 // maximum number of CPUs

```
221         else
222         #ifndef CPUS
223 + CPUS := 4
224         #endif
```


و حالا برای نگهداری تعداد فراخوانی های سیستمی بصورت global یک struct تعریف میکنیم و در آن به قفل نیز نیاز داریم چرا که ممکن است چند پردازش همزمان با آن کار کنند.(همچنین برای استفاده از این struct نیاز داریم از extern نیز نیاز داریم)

```
+ #include "spinlock.h"
+
+ struct total_syscls
+ //struct for save number systemcalls globally between cores
+ {
+     int the_number;
+     struct spinlock lk;
+ };
+ extern struct total_syscls tot_syscls;
```

حالا در فایل syscall.c ، از آن متغیر برای ذخیره سازی داده بصورت globally در اینجا استفاده میکنیم:

```
8      #include "syscall.h"
9      + #include "mp.h"
10     +
11
12     // User code makes a system call with INT T_SYSCALL.
13     // System call number in %eax.
14     // Arguments on the stack, from the user call to the C
15     // library system call function. The saved user %esp points
16     // to a saved program counter, and then the first argument.
17
18     + struct total_syscls tot_syscls;
```

برای initialize کردن این متغیرها، در تابع exec() آنها را صفر میکنیم تا وقتی برنامه شروع شد این متغیرها نیز initialize شوند:

```
6      #include "defs.h"
7      #include "x86.h"
8      #include "elf.h"
9  + #include "mp.h"
10
11      int
12      exec(char *path, char **argv)
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102      curproc->tf->esp = sp;
103      switchvm(curproc);
104      freevm(oldpgdir);
105  +
106  +      acquire(&tot_syscls.lk);
107  +      tot_syscls.the_number = 0;
108  +      release(&tot_syscls.lk);
109  +
110  +      cpus[0].num_of_syscalls = 0;
111  +      cpus[1].num_of_syscalls = 0;
112  +      cpus[2].num_of_syscalls = 0;
113  +      cpus[3].num_of_syscalls = 0;
114      return 0;
```

برای پیاده سازی sys_print_syscalls در فایل sysproc.c نیز فقط کافیست تابع print_syscalls را صدا کنیم:

```
101     }
102     +
103     + int sys_print_syscalls(void)
104     + {
105     +     return print_syscalls();
106     + }
```

حالا در تابع syscall در این فایل، ابتدا با cli() اینترایت ها را غیرفعال کرده و cpuid را بدست آورده و با sti() دوباره اینترایت ها را فعال میکنیم.

سپس تعداد فراخوانی های سیستمی هسته ی با این cpuid را یکی افزایش میدهیم و متغیر داخل total_syscls را نیز همینطور!

```
155     +
156     + cli();
157     + int CPUid = cpuid();
158     + sti();
159     +
160     + cpus[CPUid].num_of_syscalls++;
161     +
162     + acquire(&tot_syscls.lk);
163     + tot_syscls.the_number++;
164     + release(&tot_syscls.lk);
165     + }
```

حالا در همین فایل، با یک تابع متغیرهای محلی و متغیر سراسری را چاپ میکنیم (زیرا در فایل proc.c دسترسی به آنها نداریم و باید در همین فایل syscall.c اینکار را کنیم)

```
167 + void show_syscalls(void)
168 + {
169 +     cprintf("\ncore 1: %d , core 2: %d , core 3: %d , core 4: %d\n\n",
170 +         cpus[0].num_of_syscalls, cpus[1].num_of_syscalls,
171 +         cpus[2].num_of_syscalls, cpus[3].num_of_syscalls);
172 +
173 +     acquire(&tot_syscls.lk);
174 +     cprintf("          and the global value: %d\n", tot_syscls.the_number);
175 +     release(&tot_syscls.lk);
176 }
```

بنابراین در تابع داخل proc.c، فقط تابع show_syscalls داخل syscall.c را صدا میزنیم:

```
int print_syscalls(void)
{
    show_syscalls();
    return(0);
}
```

خروجی این بخش:

```
init: starting sh
$ syscalls_count
core 1: 22 , core 2: 9 , core 3: 10 , core 4: 1
          and the global value: 42
$
```

و بنابراین طبق انتظار ما، مجموع داده‌های هسته‌ها با مقدار متغیر سراسری برابر است.

پیاده سازی سازوکار همگام سازی با قابلیت اولویت دادن

برای پیاده سازی `prioritylock` همانند `sleeplock` در `xv6` عمل میکنیم و در ابتدا به یک `struct` نیاز خواهیم داشت که در آن اطلاعات مورد نیاز قفل و صف پردازش ها را ذخیره میکنیم:

```
1 struct prioritylock
2 {
3     char* name;           // lock's name
4     int pid;              // this is for the process is holding the lock
5     struct spinlock lk;   // lock for spinlock
6     uint locked;          // is lock holding?
7     int processes[NPROC]; // pid of members of queue
8     int num_of_procs;     // this is a counter for queue members
9 };
10
```

حالا برای پیاده سازی `acquire` کردن این قفل، ابتدا چک میکنیم که توسط همین پردازش قبلا `acquire` نشده باشد. چراکه هرچقدر هم منتظر `release` شدن قفل بمانیم اتفاقی نمی افتد و این حالت همانند ارور هندل خواهد شد:

```

1  int acquire_pr(struct prioritylock* lk)
2  {
3
4      int is_holding;
5      acquire(&lk->lk);
6      is_holding = lk->locked && (lk->pid == myproc()->pid);
7      if(is_holding)
8      {
9          release(&lk->lk);
10         return(-1);
11     }

```

سپس اگر صف پر نبود، پردازش جدید را در آن insert میکنیم و تا زمانی که قفل در اختیار پردازش دیگریست یا بالاترین اولویت برابر پردازش فعلی نیست صبر میکنیم:

```

1  if (lk->num_of_procs < NPROC)
2  {
3      int new_proc_indx = 0;
4      while (lk->processes[new_proc_indx] > myproc()->pid)
5          new_proc_indx++;
6      for (int i = lk->num_of_procs; i > new_proc_indx; i--)
7          lk->processes[i] = lk->processes[i - 1];
8      lk->processes[new_proc_indx] = myproc()->pid;
9      lk->num_of_procs++;
10 }
11
12 while (lk->locked == 1 || !priority(lk))
13 {
14     release(&lk->lk);
15     acquire(&lk->lk);
16 }
17

```

در نهایت؛ صف را پرینت کرده و بالاخره قفل را acquire میکنیم:

```
1  cprintf("This is the priority list:\n");
2  for (int i = 0; i < lk->num_of_procs; i++)
3  {
4      cprintf("%d", lk->processes[i]);
5      if (i != lk->num_of_procs - 1)
6          cprintf(", ");
7      else
8          cprintf("\n");
9  }
10
11  cprintf("process %d is acquiring the lock\n", myproc()->pid);
12
13  lk->locked = 1;
14  lk->pid = myproc()->pid;
15  release(&lk->lk);
16
17  return(lk->pid);
18 }
```

برای release کردن نیز ابتدا چک میکنیم که قفل در اختیار پردازش ی فعلی است یا خیر(حتی ممکن است قفل آزاد باشد و در اختیار پردازش ای نباشد) و اگر در اختیار پردازش ی فعلی بود release میکنیم:

```
1  int release_pr(struct prioritylock* lk)
2  {
3
4      int is_holding;
5      acquire(&lk->lk);
6      is_holding = lk->locked && (lk->pid == myproc()->pid);
7      if(!is_holding)
8      {
9          release(&lk->lk);
10         return(-1);
11     }
12
13     cprintf("process %d is releasing the lock\n", lk->pid);
14
15     lk->locked = 0;
16     lk->pid = 0;
17     release(&lk->lk);
18
19     return(0);
20 }
21
```

برای پیاده سازی تابعی که تشخیص دهد پردازده ی فعلی نسبت به پردازده های دیگر اولویت دارد یا خیر نیز، چک میکنیم اگر صف خالی بود یا پردازده ی اول در صف برابر با پردازده ی فعلی نبود، یعنی پردازده ی فعلی در حال حاضر ارجحیت ندارد ولی در غیر این صورت این پردازده را از صف pop میکنیم و این پردازده نسبت به بقیه اولویت دارد:

```
1 int priority(struct prioritylock* lk)
2 {
3     if (lk->num_of_procs == 0 || lk->processes[0] != myproc()->pid)
4         return 0;
5
6     for (int i = 0; i < lk->num_of_procs - 1; i++)
7         lk->processes[i] = lk->processes[i + 1];
8     lk->processes[lk->num_of_procs - 1] = 0;
9     lk->num_of_procs--;
10    return 1;
11 }
```

حالا باید به فایل proc.c نیز چیزهایی اضافه کرد:

-ابتدا struct این قفل جدید را در این فایل تعریف میکنیم تا از آن در پردازش استفاده کنیم:

```
1 struct prioritylock pr_lock;
```


-در pinit باید این قفل را initialize کنیم:

```
1 void
2 pinit(void)
3 {
4     initlock(&ptable.lock, "ptable");
5     init_pr(&pr_lock, "priority-table");
6 }
```

-توابع acquire_prior و release_prior را نیز به آخر این فایل اضافه میکنیم:

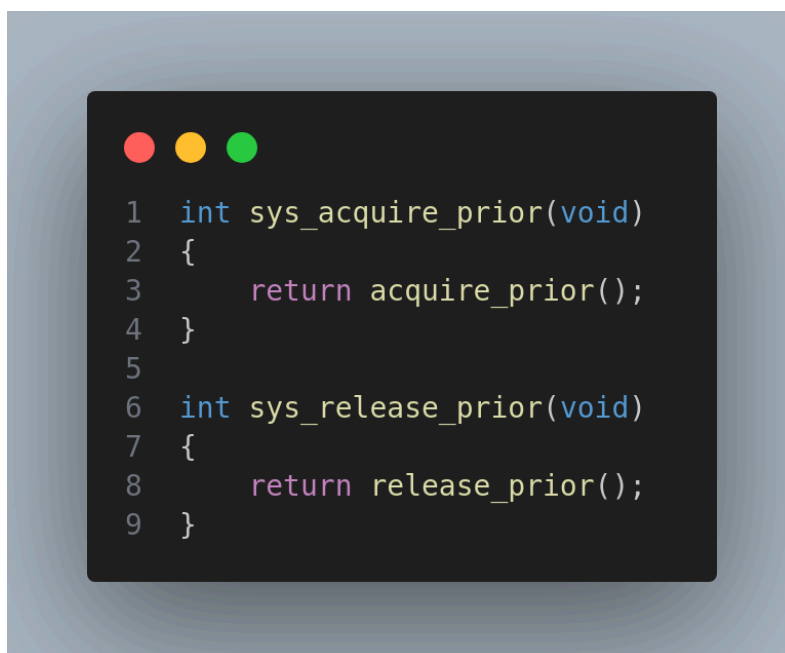
```
1 int acquire_prior(void)
2 {
3     return((acquire_pr(&pr_lock)));
4 }
5
6 int release_prior(void)
7 {
8     return(release_pr(&pr_lock));
9 }
```

-در تابع `exit` باید قبل از خارج شدن، اگر `priorlock` دست پرده ای بود، آن را `release` کنیم:(دقت شود اگر قفل آزاد باشد با توجه به پیاده سازی `release_pr`، بدون انجام کاری ریترن میشود)

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays a single line of C code: `1 release_pr(&pr_lock);`

```
1 release_pr(&pr_lock);
```

در `sysproc.c` نیز برای پیاده سازی `sys_acquire_prior` و `sys_release_prior` کافیهست فقط `release_prior` و `acquire_prior` را صدا بزنیم:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays two C functions: `sys_acquire_prior` and `sys_release_prior`.

```
1 int sys_acquire_prior(void)
2 {
3     return acquire_prior();
4 }
5
6 int sys_release_prior(void)
7 {
8     return release_prior();
9 }
```

و در آخر همانند اضافه کردن دیگر فراخوانی های سیستمی که در بخش های قبلی آزمایشگاه بررسی شد، تغییرات لازم را برای این فراخوانی سیستمی ها (`acquire_prior` , `release_prior`) در مورد نیاز انجام دادیم.

خروجی این بخش:

```
This is the priority list:
9, 8, 7, 6, 5
process 10 is acquiring the lock
process 10 is releasing the lock
This is the priority list:
8, 7, 6, 5
process 9 is acquiring the lock
process 9 is releasing the lock
This is the priority list:
7, 6, 5
process 8 is acquiring the lock
process 8 is releasing the lock
This is the priority list:
6, 5
process 7 is acquiring the lock
process 7 is releasing the lock
This is the priority list:
5
process 6 is acquiring the lock
process 6 is releasing the lock
This is the priority list:
process 5 is acquiring the lock
process 5 is releasing the lock
$
```

امکان رخ دادن گرسنگی در این پیاده سازی:

در این پیاده سازی، پردازش ها با pid کوچکتر همیشه اولویت بالاتری دارند. یعنی اگر یک پردازش با pid بزرگتر درخواست lock کند، اما پردازش هایی که pid کوچک تر دارند مداوم وارد صف شوند و درخواست lock کنند، آن پردازش با pid بزرگتر ممکن که برای همیشه در حال wait بماند و دچار گرسنگی شود. این مشکل رایجی در سیستم های زمانبندی با اولویت (priority scheduling) و قفل با اولویت (priority lock) است. یکی از راه حل های این مشکل این است که یک ساز و کار aging برای پردازش ها پیاده کنیم. به این صورت که هر چقدر پردازش های بیشتر wait کنند اولویت آنها بیشتر میشود. (همانند پروژه قبلی آزمایشگاه سیستم عامل که برای هر پردازش یک age ایجاد کردیم و به ازای هر tick به آن age اضافه میشد و در صورتی که age پراسس از یک حد

مشخصی عبور میکرد، به بالاترین سطح اولویت برای زمانبندی انتقال داده میشد). البته برای پیاده سازی این ساز و کار باید پیاده سازی قبلی را که توضیح دادیم تغییر دهیم.

یک راه حل دیگر میتوان به محدود کردن تعداد دفعات مجاز یک پردازش برای درخواست های پی در پی برای lock اشاره کرد که البته ممکن است برای برخی از سناریو ها پاسخگو نباشد.

مقایسه قفل بلیت و قفل اولویت:

قفل اولویت:

این قبل به هر thread یک اولویت اختصاص میدهد و با آنها زمانبندی انجام می شود. زمانی که یک thread میرسد بر اساس اولویتش در صف قرار میگیرد و هر thread ای که اولویت بیشتری دارد وارد ناحیه بحرانی می شود که این ممکن باعث گرسنگی شود که در بخش قبلی بررسی شد.

قفل بلیت:

این قفل از نوع spinlock است که به هر thread یک بلیت می دهد که بر اساس آن میتوانند وارد ناحیه بحرانی شوند. برای اینکه پردازش ای دچار گرسنگی نشود از یک FIFO استفاده می شود که از اختصاص دادن lock ها بین آن ها عادلانه باشد.

پیاده سازی آن به این صورت است که زمانی که یک thread میرسد به طور atomic مقدار queue ticket را زیاد میکند. سپس ticket value قبل از اضافه شدن را با dequeue ticket مقایسه میکند. اگر برابر بودند یعنی thread میتواند وارد ناحیه بحرانی شود. در غیر این صورت باید به وضعیت busy-wait یا yield برود تا زمانی که مجوز وارد شدن را دریافت کند.