به نام خدا

گزارش پروژه «۴» آزمایشگاه سیستم عامل

استاد: دکتر کارگهی

گروه ۱۱

امیرارسلان شهبازی ۸۱۰۱۰۱۴۵۱

سید محمدحسین مظهری ۸۱۰۱۰۱۵۲۰

محمدمهدی صمدی ۸۱۰۱۰۱۴۶۵

لینک مخزن https://github.com/AMIRSH1383/OS-SMS_LAB4.git

- **علت غیرفعال کردن وقفه چیست؟ توابع pushcli , popcli به چه منظور استفاده شده و چه تفاوتی با sti,cli دارند.**

زیرا می خواهیم مطمئن شویم که کدهایی که می خواهیم اجرا کنیم به صورت اتمیک اجرا شوند. یعنی کدهای وقفه را نمی توان مسدود کرد و برای محافظت از ناحیه بحرانی باید وقفه ها غیر فعال شوند.

این عملیات به کمک دو تا pushcli , popcli انجام می شود که با پوش، وقفه ها را غیرفعال کرده و سپس acquire و release را صدا می زنیم. در نهایت برای فعال سازی مجدد وقفه ها تابع popcli را صدا می زنیم.

خود این دو تابع از cli , sti نیز استفاده می کنند اما یک قابلیت اضافه ای که دارند این است که قابلیت شمارش هم دارند یعنی مشخص است که هرکدام چقدر اجرا شده است و این موضوع می تواند در کنترل کردن کمک کند.

- **حالات مختلف پردازه ها در xv6 را توضیح دهید. تابع sched چه وظیفه ای دارد.**

UNUSED: از پردازه ای استفاده نشده است.

EMBRYO: وقتی از حالت بی استفاده تغییر می کنیم به این حالت می رویم و دیگر unused نیست.

SLEEPING: پردازه در پردازنده نیست و به منبعی نیاز دارد که هنوز آماده نیست. یعنی زمان بند از آن استفاده نمی کند.

RUNNABLE: پردازه در حالت صبر است و می تواند توسط زمانبند به پردازنده اختصاص پیدا کند.

RUNNING: پردازنده به آن اختصاص داده شده است و در حال اجرا است.

ZOMBIE: پردازه ای که زامبی شده از طرف پدر رها شده است یعنی پردازه پدر wait را صدا نزده و اطلاعات آن هنوز در جدول وجود دارد.

تابع sched در پایان کار یک پردازه صدا زده می شود و عملکرد آن این است که کانتکست فعلی پردازه را ذخیره و زمانبندی پردازه بعدی انجام می شود.

- **یکی از روش‌های سینک کردن حافظه‌های نهان با یک‌دیگر، روش Modified-Shared-Invalid است. آن را به اختصار توضیح دهید. (اسلایدهای موجود در منبع اول کمک‌کننده شما خواهند بود)**

روش Modifier shared invalid که به اختصار آن را MSI می‌نامیم، پروتکلی برای حفظ انسجام کش در سیستم‌های Multicore مانند XV6 است. به طور کلی این روش سه استیت دارد هر کدام ازآنان را به اختصار توضیح می‌دهیم:

- Modified: در این حالت، دیتای جدید در کش نگه‌داری می‌شود اما هنوز در مموری نوشته نشده است. در نتیجه مموری مقدار قبلی را دارد. البته در این حالت منظور از کش، تنها آن کشی‌ست که مربوط به آن بخش پراسس است و بقیه کش‌ها مقدار جدید را ندارند.
- Shared: در این استیت، دیتا در مموری هم تغییر یافته و کش‌های دیگر نیز کپی‌ای از مقدار جدید دارند.
- Invalid: اگر بخش دیگری از پراسس در کش خود مقدار را تغییر دهد، دیگر دیتا داخل کش شما معتبر نیست.

به عنوان مثال سیستمی با سه کش در نظر بگیرید. در ابتدا کش اول یک دیتا را می‌نویسد و به حالت Modified می‌رود. حال دیتا جدید به مموری می‌رود و کش دوم و سوم از آن اطلاع یافته و تغییرش می‌دهند. پس هر سه کش در حالت Shared می‌روند. به فرض کش سوم آن دیتا را عوض می‌کند. پس کش سوم به حالت Modified رفته و دو کش اول به حالت Invalid می‌روند. پس این دو کش سعی می‌کنند از مموری دیتا جدید را بردارند.

- **یکی از روش‌های همگام‌سازی استفاده از قفل‌هایی معروف به قفل بلیط است. این قفل‌ها را از منظر مشکل بالا بررسی نمایید.**

به این علت نام قفل بلیط گذاشته شده که هر thread، در صورتی که بخواهد به یک منبع مشترک دسترسی پیدا کند، مانند صف بانک ابتدا یک بلیت می‌گیرد و هر وقت نوبتش شد می‌واند دسترسی پیدا کند. پس الگوریتم first in first served می‌باشد. این مکانیزم با اینکه منصفانه است میان بخش‌های مختلف، اما overhead بالایی برای update کردن مقادیر دارد. نحوه عملکرد این پروتکل به این بستگی دارد که در چه فاصله‌های زمانی cache invalidation و یا update cache رخ دهد.

- **دو مورد از معایب استفاده از قفل با امکان ورود مجدد را بیان نمایید.**

۱) **پیچیدگی در طراحی و عیب‌یابی**

استفاده از Reentrant Mutex می‌تواند کد را پیچیده‌تر کند، زیرا امکان ورود چندباره به قفل ممکن است موجب شود که توسعه‌دهنده به‌سادگی در مورد وضعیت قفل (قفل‌شده یا آزاد) اشتباه کند. این امر می‌تواند مشکلاتی مانند شرایط رقابتی (race condition) یا حتی درک نادرست از جریان برنامه را ایجاد کند.

۲) **کاهش کارایی و سربار محاسباتی**

Reentrant Mutex معمولاً نیاز دارد تا اطلاعاتی مانند شناسه‌ی رشته (Thread ID) و تعداد دفعات ورود به قفل را ذخیره کند. این سربار اضافی ممکن است عملکرد را نسبت به قفل‌های ساده (Non-reentrant Mutex) کاهش دهد، به‌ویژه در سناریوهایی که به قفل‌گذاری و باز کردن سریع نیاز است.

- **یکی دیگر از ابزار های همگام سازی قفل Read-Write lock است . نحوه کارکرد این قفل را توضیح دهید .و در چه مواردی این قفل نسبت به قفل با امکان ورود مجدد برتری دارد ؟**

نحوه کارکرد قفلRead-Write Lock

قفل‌های Read-Write یا قفل‌های خواندن-نوشتن به گونه‌ای طراحی شده‌اند که دسترسی همزمان به داده‌های مشترک را مدیریت کنند. در این قفل‌ها:

۱. **حالت خواندن:(Read Lock)**

- چندین رشته (Thread) می‌توانند به طور همزمان قفل را برای خواندن داده بگیرند، مشروط بر اینکه هیچ رشته‌ای قفل را برای نوشتن در اختیار نداشته باشد.
- این حالت برای مواقعی مناسب است که نیاز به دسترسی فقط خواندنی به داده‌ها وجود دارد.

۲. **حالت نوشتن:(Write Lock)**

- تنها یک رشته می‌تواند قفل را برای نوشتن داده بگیرد.
- هنگامی که قفل در حالت نوشتن است، هیچ رشته دیگری نمی‌تواند به داده‌ها دسترسی داشته باشد (چه برای خواندن و چه برای نوشتن).

برتری‌های Read-Write Lock نسبت بهReentrant Mutex

۱. **بهبود کارایی در سناریوهای با تعداد بالای عملیات خواندن:**

- در صورتی که اکثر عملیات‌ها فقط خواندن داده‌ها باشند، Read-Write Lockبه چندین رشته اجازه می‌دهد به طور همزمان داده‌ها را بخوانند، در حالی که Reentrant Mutex فقط اجازه می‌دهد یک رشته در هر لحظه قفل را در اختیار داشته باشد.
- این ویژگی کارایی را در سیستم‌هایی که عمدتاً عملیات خواندن انجام می‌دهند (مانند پایگاه‌های داده) به طور قابل توجهی افزایش می‌دهد.

۲. **مدیریت بهتر تعادل بین عملیات خواندن و نوشتن:**

- در سناریوهایی که تعداد عملیات نوشتن نسبت به خواندن کمتر است، Read-Write Lockاجازه می‌دهد عملیات نوشتن تنها در صورت نیاز انجام شود، بدون اینکه دسترسی خواندن به طور غیرضروری مسدود شود.
- این ویژگی باعث می‌شود سیستم تعادلی بهتر بین همزمانی و قفل‌گذاری داشته باشد.

محدودیت‌هایRead-Write Lock

- در شرایطی که تعداد عملیات نوشتن زیاد است، کارایی ممکن است کاهش یابد، زیرا عملیات نوشتن باید منتظر بمانند تا تمام قفل‌های خواندن آزاد شوند.
- مدیریت و پیاده‌سازی این نوع قفل پیچیده‌تر از Reentrant Mutex است و نیاز به دقت بیشتری دارد.

نتیجه‌گیری:

در سناریوهایی که حجم عملیات خواندن بالا و عملیات نوشتن کم است، Read-Write Lock انتخاب بهتری نسبت به Reentrant Mutex است. اما در مواقعی که تنها یک رشته با داده‌ها تعامل دارد یا قفل‌گذاری چندباره توسط همان رشته لازم است ، Reentrant mutex ساده‌تر و مناسب‌تر خواهد بود.

پیاده‌سازی تسک اول

ابتدا در فایل syscall.h، برای هر سیستم‌کال جدید یک عدد تعریف می‌کنیم.

```
#define SYS_count_syscalls 22
#define SYS_init_reentrantlock    23
#define SYS_acquire_reentrantlock 24
#define SYS_release_reentrantlock 25
```

حال برای پیاده‌سازی تسک اول پروژه به struct cpu در فایل proc.h یک فیلد جدید به نام syscall_count اضافه می‌کنیم که نشان‌دهنده تعداد سیستم‌کال‌هایی است که به روی این cpu اجرا شده‌اند.

```
// Per-CPU state
struct cpu {
  int syscall_count;           // number of systcalls executed on the cpu
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
};
```

در قدم بعدی به فایل trap.c رفته و متغیرهایی را تعریف می‌کنیم.

- Main lock: قفلی که برای کل سیستم است.
- Cpu lock: قفلی که برای cpu کنونی است.
- Number of all system calls: متغیر گلوبال نشان‌دهنده تعداد کل سیستم‌کال‌های اجرا شده به روی سیستم.

```
struct spinlock main_lock;
struct spinlock cpu_lock;
int numof_all_syscalls;
void init_count(void) {
    numof_all_syscalls = 0;
    initlock(&cpu_lock, "cpu_lock");
    initlock(&main_lock, "main_lock");
}
```

این متغیرها را در فایل defs.h به صورت extern تعریف می‌کنیم تا در بقیه بخش‌ها ازشان استفاده کنیم.

```
// trap.c
extern struct spinlock tickslock;
extern int    numof_all_syscalls;
extern struct spinlock main_lock;
extern struct spinlock cpu_lock;
```

برای initialize کردن این متغیرها، به فایل main.c رفته و در تابع main آن تابع init_count که در trap.cتعریف شده بود را کال می‌کنیم.

```
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  // ------------------------------------------------
  init_count();    // vars related to count syscall
  // ------------------------------------------------
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup

}
```

حال منطق این تسک را در تابع trap فایل پیاده‌سازی کنیم. با توجه به شماره سیستم‌کال و اعداد دیفاین‌شده در فایل syscall.h، به هر سیستم‌کال یک ضریب می‌دهیم. باید لاک هرکدام را acquire کنیم، شمارنده را افزایش دهیم و سپس release کنیم.

```
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    int id = tf->eax;
    switch (id)
    {
      case 15:
          acquire(&cpu_lock); mycpu()->syscall_count += 3; release(&cpu_lock);
          acquire(&main_lock); numof_all_syscalls += 3; release(&main_lock);
          break;
      case 16:
          acquire(&cpu_lock); mycpu()->syscall_count += 2; release(&cpu_lock);
          acquire(&main_lock); numof_all_syscalls += 2; release(&main_lock);
          break;
      default:
          acquire(&cpu_lock); mycpu()->syscall_count++; release(&cpu_lock);
          acquire(&main_lock); numof_all_syscalls++; release(&main_lock);
          break;
    }
    if(myproc()->killed) { exit(); }
    return;
  }
```

در مرحله بعد برنامه user level می‌نویسیم که کد پیاده‌سازی‌شده را تست کند. این کد در فایل test_count_syscalls قرار دارد. در این برنامه به تعداد مشخصی پراسس fork می‌شود و هر کدام از آنان پیامی را در فایلی می‌نویسد. در انتها سیستم‌کال این بخش صدا زده می‌شود تا نتیجه بررسی شود. به این علت به روی فایل‌ها می‌نویسیم که علاوه بر سیستم‌کال‌های معمول، از open و write نیز استفاده شده باشد.

```
void write_to_file(const char *filename, const char *content) {
    int fd = open(filename, O_CREATE | O_RDWR);
    if (fd < 0) {
        printf(1, "Error: Could not open file %s\n", filename);
        exit();
    }
    if (write(fd, content, strlen(content)) < 0) {
        // printf(1, "Error: Could not write to file %s\n", filename);
        close(fd);
        exit();
    }
    close(fd);
}

int main() {
    for (int i = 0; i < NUM_PROCESSES; i++) {
        int pid = fork();
        if (pid < 0) {
            printf(1, "Error: fork failed\n");
            exit();
        }
        if (pid == 0) {
            // Child process
            char filename[20] = FILE_PREFIX;
            filename[13] = i + '0';
            char content[50] = "This is process writing to its file.";
            write_to_file(filename, content);
            exit();
        }
    }

    for (int i = 0; i < NUM_PROCESSES; i++) {
        wait();
    }

    printf(1, "All processes have completed writing to their files.\n");
    count_syscalls();
    exit();
}
```

نتیجه چندین بار اجرای برنامه بالا را در زیر مشاهده می‌کنید که همواره جواب دو روش تطابق دارند.

```
All processes have completed writing to their files.
Global counter says: 131
Number of system calls used is: 131
$ test_count_syscalls
All processes have completed writing to their files.
Global counter says: 236
Number of system calls used is: 236
$ test_count_syscalls
All processes have completed writing to their files.
Global counter says: 345
Number of system calls used is: 345
$ test_count_syscalls
All processes have completed writing to their files.
Global counter says: 452
Number of system calls used is: 452
$ test_count_syscalls
All processes have completed writing to their files.
Global counter says: 557
Number of system calls used is: 557
$ test_count_syscalls
All processes have completed writing to their files.
Global counter says: 662
Number of system calls used is: 662
$
```

پیاده‌سازی تسک دوم

برای پیاده‌سازی این قفل یک فایل هدر و یک فایل کد جدید با نام‌های reentrantlock.h و reentrantlock.c می‌سازیم.
محتوای فایل هدر به صورت زیر است:

```
struct reentrantlock {
    struct spinlock lock;
    int locked;           // is locked?
    int recursive;        // number of recursive calls so far
    struct proc* owner;   // pointer to current owner of the lock
};

void init_reentrantlock(struct reentrantlock *rlock);
void acquire_reentrantlock(struct reentrantlock *rlock);
void release_reentrantlock(struct reentrantlock *rlock);
```

در فایل کد سه تابع تعریف شده‌اند که در ادامه به توضیح آنان می‌پردازیم.

Init: متغیرهای لاک ر مقداردهی اولیه می‌کند.

```
void init_reentrantlock(struct reentrantlock *rlock) {
    initlock(&rlock->lock, "reentrantlock");
    rlock->locked = 0; rlock->owner = 0; rlock->recursive = 0;
}
```

Acquire: ابتدا پراسس کنونی با myproc گرفته می‌شود. حال قفل spin که در استراکت بالا ساخته بودیم acquire می‌شود. سپس چک می‌شود که اگر صاحب کنونی قفل همان p بود و همچنین قفل لاک بود، مقدار متغیر recursive که نشان‌دهنده تعداد فراخوانی‌های تودرتو است افزایش میابد. زیرا بدین معناست که فراخوانی جدید انجام شده. اما اگر این‌چنین نبود، تا وقتی که قفل لاک است در حالت sleep بماند. وقتی ار حلقه وایل بیرون بیاید یعنی قفل دیگر لاک نیست. در این حالت آن ان لاک کرده و متغیر recursive را برابر یک می‌گذاریم زیرا تمام فراخوانی‌های بازگشتی تمام شده‌اند. صاحب قفل را همان p می‌گذاریم و در نهایت قفل spin را release می‌کنیم.

```
void acquire_reentrantlock(struct reentrantlock *rlock) {
    struct proc* p = myproc();
    acquire(&rlock->lock);
    if (rlock->locked && rlock->owner == p) {
        rlock->recursive++; release(&rlock->lock);
        return;
    }

    while (rlock->locked) { sleep(rlock, &rlock->lock); }

    rlock->locked = 1; rlock->recursive = 1; rlock->owner = p;
    release(&rlock->lock);
}
```

Release: ابتدای همه چیز قفل spin را acquire می‌کنیم. حال چک می‌کنیم که صاحب قفل پراسس کنونی است یا نه. اگر بود، باید قفل spin را release کنیم و کار تمام می‌شود. اما اگر این‌چنین نبود یعنی در یکی از توابع بازگشتی هستیم و قرار است از آن بیرون بیاییم پس recursive را یکی کم کرده. اگر recursive برابر صفر شد یعنی کار تمام شده (در واقع یعنی همه فرخوانی‌های تودرتو اتمام یافتند) پس قفل را از حالت لاک در میاوریم رو صاحب قفل را null می‌کنیم. سپس باید قفل را بیدار کنیم. بعد از تمام این فرایندها قفل spin که در اول کار acquire کرده بودم را release می‌کنیم.

```
void release_reentrantlock(struct reentrantlock *rlock) {
    acquire(&rlock->lock);
    if (rlock->owner != myproc()) { release(&rlock->lock); return; }

    rlock->recursive--;
    if (rlock->recursive == 0) {
        rlock->locked = 0; rlock->owner = 0;
        wakeup(rlock);
    }
    release(&rlock->lock);
}
```

جهت استفاده در توابع بالا به صورت سیستم‌کال، توابع زیر را در sysproc.c تعریف می‌کنیم. در هر کدام ابتدا پوینتر به قفل از طریق argptr گرفته می‌شود و سپس تابع ساخته‌شده صدا زده می‌شود.

```
extern struct cpu cpus[NCPU];
extern int ncpu;
extern struct spinlock main_lock;
extern struct spinlock cpu_lock;
extern int numof_all_syscalls;

int sys_init_reentrantlock(void) {
    struct reentrantlock *rlock;
    if (argptr(0, (void*)&rlock, sizeof(*rlock)) < 0) { return -1; }
    init_reentrantlock(rlock);
    return 0;
}

int sys_acquire_reentrantlock(void) {
    struct reentrantlock *rlock;
    if (argptr(0, (void*)&rlock, sizeof(*rlock)) < 0) { return -1; }
    acquire_reentrantlock(rlock);
    return 0;
}

int sys_release_reentrantlock(void) {
    struct reentrantlock *rlock;
    if (argptr(0, (void*)&rlock, sizeof(*rlock)) < 0) { return -1; }
    release_reentrantlock(rlock);
    return 0;
}
int sys_count_syscalls(void) {
    int num_syscalls = 0;
    for (int i = 0; i < ncpu; i++) {  // `ncpu` is the total number of CPUs
        struct cpu *c = &cpus[i];
        num_syscalls += c->syscall_count;
    }
    cprintf("Global counter says: %d\n", numof_all_syscalls);
    cprintf("Number of system calls used is: %d\n", num_syscalls);
    return 0;
}
```

حالا باید اطلاعات سیستم‌کال‌های تعریف شده را به syscall.h نیز اضافه کنیم.

```
extern int sys_count_syscalls(void);
extern int sys_init_reentrantlock(void);
extern int sys_acquire_reentrantlock(void);
extern int sys_release_reentrantlock(void);

static int (*syscalls[])(void) = {
[SYS_count_syscalls]        sys_count_syscalls,
[SYS_init_reentrantlock]    sys_init_reentrantlock,
[SYS_acquire_reentrantlock] sys_acquire_reentrantlock,
[SYS_release_reentrantlock] sys_release_reentrantlock,
```

برای تست تسک دوم برنامه user level زیر نوشته شده است. در این برنامه تابع فاکتوریل با استفاده از قفل پیاده‌سازی شده انجام می‌شود. در هر قدم اطلاعات پرینت می‌شود که ترتیب اجرا را مشاهده کنیم. یک عدد از طریق command line داده می‌شود تا فاکتویل آن حساب شود.

```
int fact(int i) {
    if (i <= 0) { return 1; }

    acquire_reentrantlock(&rlock);
    printf(1, "current i %d: acquired lock\n", i);
    int res = fact(i-1);
    release_reentrantlock(&rlock);
    printf(1, "current i %d: released lock\n", i);
    return i * res;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf(1, "please enter a number\n");
        exit();
    }
    int i = atoi(argv[1]);
    printf(1, "i: %d\n", i);
    init_reentrantlock(&rlock);
    printf(1, "calculating %d!...\n", i);
    int res = fact(i);
    printf(1, "finished calculating %d! and the result is %d\n", i, res);
    exit();
}
```

نتیجه اجرای برنامه بالا را در زیر برای فاکتوریل عدد ۷ مشاهده می‌کنیم.