آزمایشگاه سیستم عامل

گزارش کار پروژه چهارم

اعضای گروه:

امیرحسین عارف زاده 810101604

مهدی نایینی 810101536

کیارش خراسانی 810101413

Repository : https://github.com/Amir-rfz/OS-Lab

Latest Commit : 19c2af502f019a80694cca96e3da0488d0b2de64

چگونگی همگام سازی در سیستم عامل XV6

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

زیرا ممکن است به deadlock برخورد کنیم. فرض کنید ما در تکه ای از برنامه وقتی وارد critical section میشویم قفل مربوطه را به دست آوریم. حال اگر قبل از آزادسازی قفل یک interrupt رخ دهد و ما وارد interrupt handler شویم و در آنجا نیز تلاش کنیم قفل مربوطه را به دست آوریم سیستم به deadlock میرسد زیرا در قسمت قبل قفل را فعال کرده بودیم و قبل از آزاد سازی آن وارد این بخش شدیم پس در interrupt handler منتظر میماند تا قفل باز شود ولی چون وارد interrupt handler شده ایم این قفل باز نمیشود و در deadlock میمانیم.

تابع cli تابعی است که low level است و هدف آن غیر فعال کردن interrupt ها است و تابع sti تابعی low level است که به طور متقابل interrupt ها را دوباره فعال میکند.

تابع pushcli یک تابع سطح بالا است که interrupt ها را غیر فعال میکند و مقدار ncli را یکی زیاد میکند(یعنی تعداد دفعاتی که interrupt غیر فعال شده است را ذخیره میکند.) سپس تابع cli را صدا میزند تا interrupt ها غیر فعال شود. این تابع تابعی سطح بالا است.

تابع popcli به طور متقابل مقدار ncli را یکی کم میکند تا زمانی که برابر با صفر شود(یعنی تمام دفعاتی که interrupt ها فعال شده بود دوباره غیر فعال شده است) سپس اگر مقدار ncli برابر با صفر بود دوباره interrupt ها را فعال میکند. این مکانیزم برای قابل انجام کردن recursive lock است. این تابع تابعی سطح بالا است.
popcli و pushcli در تابع spinlock صدا شده و استفاده میشود.
تفاوت pushcli و popcli با cli و sti این است که دو تابع دوم سطح پایین هستند ولی دو تابع دوم سطح بالا. از طرفی دو تابع اول تعداد دفعاتی که interrupt ها غیر فعال شده است را track میکند در حالی که دو تابع دوم اینکار را نمیکنند.

2. حالات مختلف پردازه ها در 6xv را توضیح دهید. تابع sched چه وظیفه ای دارد؟

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

همانطور که در تصویر بالا قابل مشاهده است۶ وضعیت برای پردازه ها در xv6 وجود دارد.
.UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE
UNUSED: فرآیند استفاده نمی‌شود و آماده تخصیص است.
EMBRYO: فرآیند در حال ایجاد شدن و در مراحل ابتدایی تخصیص است. فرایند از UNUSED وارد این حالت میشود.
SLEEPING: فرآیند در حالت خواب است و منتظر وقوع یک رویداد است. مانند I/O یا به دست آوردن منابع مورد نیاز.
RUNNABLE: فرآیند آماده اجرا است و منتظر تخصیص CPU است.
RUNNING: فرآیند در حال اجرا توسط CPU است.
ZOMBIE: فرآیند اجرای خود را به پایان رسانده اما هنوز در جدول فرآیندها باقی مانده تا وضعیت خروج آن خوانده شود. باید توسط تابع wait دیده شود تا پردازه terminate شود و از این وضعیت بیرون بیاید.

```
510   // Enter scheduler.  Must hold only ptable.lock
511   // and have changed proc->state. Saves and restores
512   // intena because intena is a property of this
513   // kernel thread, not this CPU. It should
514   // be proc->intena and proc->ncli, but that would
515   // break in the few places where a lock is held but
516   // there's no process.
517   void
518   sched(void)
519   {
520     int intena;
521     struct proc *p = myproc();
522
523     if(!holding(&ptable.lock))
524       panic("sched ptable.lock");
525     if(mycpu()->ncli != 1)
526       panic("sched locks");
527     if(p->state == RUNNING)
528       panic("sched running");
529     if(readeflags()&FL_IF)
530       panic("sched interruptible");
531     intena = mycpu()->intena;
532     swtch(&p->context, mycpu()->scheduler);
533     mycpu()->intena = intena;
534   }
535
```

مسئولیت تابع sched تعویض متن (context switch) است و بنابراین مسئولیت process scheduling را در کرنل سیستم عامل دارد. این تابع ابتدا پردازه حال حاضر را میگیرد و چک میکند که ptable قفل باشد و سپس چک میکند که فقط یک بار interrupt غیر فعال شده باشد و سپس چک میکند حالت پردازه کنونی RUNNING نباشد و اطمینان حاصل میکند که interrupt ها غیر فعال باشند و اگر این شرایط برقرار باشد ابتدا وضعیت interrupt enable را از CPU میگیرد و سپس با استفاده از تابع switch تعویض متن را انجام میدهد و وضعیت interrupt enable را در cpu را به حالت قبلی برمیگرداند.

cache coherency در سیستم عامل XV6

3. یکی از روش های سینک کردن این حافظه های نهان با یکدیگر روش Modified-Shared-Invalid است. آن را به اختصار توضیح دهید. (اسلاید های موجود در منبع اول کمک کننده شما خواهند بود)

این روش در واقع یک پروتکل از روش snooping است. در این روش ما ما یک Shared_bus داریم که پردازنده ها تغییرات ایجاد شده روی داده ها را از طریق آن نظارت می کنند و اگر داده ای که آنها دارند تغییر بکند آن را اصلاح میکنند.
روش MSI نیز همین گونه است. در این روش ما سه حالت برای هر داده موجود در حافظه نهان داریم:
۱. Invalid: در این حالت پردازنده مقدار درست را ندارد و باید آن را درخواست کند.
۲. Modified: در این حالت مقداری که پردازه دارد ممکن است با مقدار موجود در مموری متفاوت باشد و مقدار درست در دست این پردازه است.
۳. Shared: مقدار من با مقدار موجود در مموری متفاوت است.
فرض کنیم داده در آدرس 0x2000 را داشته باشیم و بخواهیم وارد حافظه نهان دو پردازنده کنیم. نحوه کار این پروتکل به این صورت است که ابتدا حافظه پردازنده ۱ و ۲ در حالت invalid هستند. سپس پردازنده ۱ و ۲ از حافظه مقدار درست را میخوانند و جفت به حالت Shared میروند. سپس پردازنده ۱ داده خود را عوض میکند پس به حالت Modified تغییر وضعیت میدهد و در Shared_bus اعلام میکند که مقدار این متغیر را عوض کرده پس در پردازنده دوم این داده تا وقتی که دوباره داده اصلی را read نکند و از حافظه دریافت نکند در حالت invalid میرود.
سپس پردازنده اول داده را در حافظه می نویسد و وارد حالت Shared میشود و بعد از آن پردازه دوم نیز داده را از حافظه دریافت میکند و وارد حالت Shared میشود.

4. یکی از روشهای همگام سازی استفاده از قفل های معروف به قفل بلیت است. این قفلها را از منظر مشکل مذکور در بالا بررسی نمایید.

روش قفل بلیت یک روش همگام برای کنترل دسترسی به یه منبع مشترک در یک محیط همگام است. تضمین میکند که تنها یک رشته یا پردازه بتواند در هر لحظه از یک منبع استفاده کند. نحوه کار آن اینگونه است که هر رشته که بخواهد از منبع استفاده کند یک بلیط می گیرد و سپس هر کدام از رشته ها به نوبت بر اساس بلیتی که گرفته اند از منبع استفاده میکنند و یک تقسیم بندی عادلانه وجود خواهد داشت.

مشکل اینجا است که در سیستم های دارای چند پردازنده وقتی یک پردازنده یکی از thread ها را نوبت میدهد تا منبع استفاده کند باید به منابع دیگر اطلاع دهد که دیگر مقدار نوبتی که الان باید ببنند و در حافظه نهان آنها است معتبر نیست و منبع مورد استفاده قابل دسترسی نیست چون بقیه در حال استفاده از آن هستند.
این انتقال پیام ها همزمان با اینکه رشته های دیگر بلیت خواهند گرفت یک ترافیک ایجاد میکند که حتی با روش های سینک کردن حافظه نهان باز هم سیستیم overhead زیادی دارد از این جهت که مانند مثال بالا باید پیام های زیادی بین پردازنده ها رد و بدل شود.

تعداد سیستم کال های اجرا شده توسط همه ی CPU:

سیستم کال:

```
// Per-CPU state
struct 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
  int cpu_ticks;
  int syscall_count;
};
```

```
struct {
  struct spinlock lock;
  int count;
} total_syscallcount;
```

```
int syscallcount(int cpu){
  if(cpu < 0 || cpu >= ncpu)
    return -1;
  return cpus[cpu].syscall_count;
}
```

```
void
init_total_syscall_count(void)
{
  initlock(&total_syscallcount.lock, "total_syscallcount");
  total_syscallcount.count = 0;
}

int
get_total_syscallcount(void)
{
  return total_syscallcount.count;
}
```

```c
int get_coefficient(int pid) {
  if (pid == 15) {
    return 3;
  }
  else if (pid == 16) {
    return 2;
  }
  else {
    return 1;
  }
}

void syscall(void) {
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  int coeff = get_coefficient(num);

  pushcli();
  mycpu()->syscall_count += coeff;
  popcli();

  acquire(&total_syscallcount.lock);
  total_syscallcount.count += coeff;
  release(&total_syscallcount.lock);


  if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    if (num < MAX_SYSCALLS) {
        record_syscall(curproc, num);
    }

    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n", curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

```c
int sys_getsyscallcount(void)
{
  int i, sum_count = 0, total_count;
  for (i = 0; i < ncpu; i++) {
    int count = syscallcount(i);
    if(count >= 0) {
      cprintf("System call count for core %d is %d\n", i, count);
      sum_count += count;
    }
  }

  total_count = get_total_syscallcount();
  cprintf("Total syscall count is %d\n", total_count);
  cprintf("Sum of syscall count is %d\n", sum_count);
  return sum_count;
}
```

تست این سیستم کال:

```c
#include "types.h"
#include "user.h"
#include "stat.h"
#include "fcntl.h"
#define NUM_OF_FORKS 5

void acuire_user() {
    while ((open("lockfile", O_CREATE  | O_WRONLY)) < 0) ;
}

void release_user() {
    unlink("lockfile");
}

int main(int argc, char* argv[]) {
    int fd=open("file.txt",O_CREATE|O_WRONLY);

    for (int i = 0; i < NUM_OF_FORKS; i++){
        int pid = fork();
        if (pid == 0) {
            acuire_user();

            char* write_data = "Writing On File";
            int max_length = 15;
            write(fd,write_data,max_length);
            write(fd,"\n",1);

            release_user();
            exit();

        }
    }

    while (wait() != -1);
    close(fd);
    getsyscallcount();
    exit();
}
```

میوتکس پردازه مالک (Reentrant Mutex)

پیاده سازی میوتکس با قابلیت ورود مجدد:

پیاده سازی:

```
struct reentrantlock {
  struct spinlock lock;   // Underlying spinlock for atomicity
  struct proc *owner;     // Current owner of the lock
  int recursion;          // Recursion depth for reentrancy
};
```

```
void initreentrantlock(struct reentrantlock *lock, char *name) {
  initlock(&lock->lock, name);
  lock->owner = 0;
  lock->recursion = 0;
}

void acquirereentrantlock(struct reentrantlock *lock) {
  pushcli();
  if (lock->owner == myproc()) {
    lock->recursion++;
  } else {
    acquire(&lock->lock);
    lock->owner = myproc();
    lock->recursion = 1;
  }
  popcli();
}

void releasereentrantlock(struct reentrantlock *lock) {
  pushcli();
  if (lock->owner != myproc()) {
    panic("releasereentrantlock: not owner");
  }
  lock->recursion--;
  if (lock->recursion == 0) {
    lock->owner = 0;
    release(&lock->lock);
  }
  popcli();
}
```

سیستم کال:

```
void recursive_lock(int depth, struct reentrantlock *test_lock) {
  if (depth == 0) return;
  acquirereentrantlock(test_lock);
  cprintf("Acquired lock at depth %d\n", depth);
  recursive_lock(depth - 1, test_lock);
  cprintf("Releasing lock at depth %d\n", depth);
  releasereentrantlock(test_lock);
}

int sys_testreentrantlock(void) {
  static struct reentrantlock test_lock;
  static int initialized = 0;
  if (!initialized) {
    initreentrantlock(&test_lock, "reentrantlock");
    initialized = 1; // Ensure that initialize only once
  }
  recursive_lock(5, &test_lock);
  return 0;
}
```

تست این سیستم کال:

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


int main() {
    printf(1, "Testing recursive functionality of reentrant lock...\n");

    if (testreentrantlock() == 0) {
        printf(1, "Recursive test passed!\n");
    }
    else {
        printf(1, "Recursive test failed!\n");
    }

    exit();
}
```

5. دو مورد از معایب استفاده از قفل با امکان ورود مجدد را بیان نماید.

معایب زیادی وجود دارد که میتوان به چند مورد اشاره کرد.

۱. ایجاد deadlock: باید قفل ها به ترتیبی که گرفته شده اند آزاد شوند. اگر در شرطی از خروج از تابع بازگشتی سیستم قفل شود ولی قفل باز نشود قبل از خروج، باعث میشود که سیستم وارد حالت deadlock برود زیرا تابع منتظر قفلی میماند که قبلا گرفته است یا قفل آزاد نمیشود.

۲. افزایش سربار: کنترل تعداد دفعات فراخوانی تابع بازگشتی و کنترل depth هزینه محاسباتی زیادی را به سیستم می تواند تحمیل کند. و کد را پیچیده تر میکند.

۳. نقض resource utilization: توابع بازگشتی معمولا زمان زیادی طول میکشد تا کار خود را تمام کنند و به همین دلیل وقتی در این توابع قفل را مدام می گیریم باعث میشود که thread های دیگر فرصتی برای استفاده از منابع پیدا نکنند و منبع به طور انحصاری در اختیار یک رشته قرار گیرد. از طرفی ممکن است در سیستم قفل بازگشتی به infinite loop برخورد کنیم که در این صورت سیستم مدام قفل را دریافت میکند ولی آزاد نمیکند و به همین دلیل از تابع خارج نمیشود و این میتواند به این منجر شود که سیستم در وضعیتی بدون امکان recovery قرار گیرد و نتواند از آن خارج شود.

۴. سختی دیباگ کردن: دیباگ کردن کد های بازگشتی سخت است و وقتی که سیستم lock را در این توابع در اختیار میگیرد دیباگ کردن آن به شدت سخت تر میشود.

6. 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.

دو نوع قفل ما داریم:

1. Read lock: مجوز برای خواندن(مثلا از یک فایل). چند پردازه با هم میتوانند از یک فایل بخوانند و باعث آسیب به محتوای فایل نشوند به همین دلیل این امکان وجود دارد که چند پردازه همزمان این قفل را در اختیار داشته باشند و بتوانند از فایل بخوانند.
2. Write lock: این مجوز برای نوشتن است. برای جلوگیری از data corruption تنها یک پردازه اجازه دارد این قفل را دریافت کند و تا وقتی که این قفل را دارد اجازه دارد که در فایل مورد نظر بنویسد.

نکته این است تا زمانی که یک پردازه قفل write lock را دارد هیچ پردازه دیگری اجازه خواندن یا نوشتن را ندارد. و تا زمانی که یک پردازه قفل read lock را دارد هیچ پردازه دیگری اجازه نوشتن را ندارد.

کاربرد این نوع قفل برای زمانی است که دسترسی های زیاد برای خواندن نیاز داریم و تعداد این دسترسی ها بر تعداد دسترسی های که برای نوشتن داریم برتری دارد. (مثلا دسترسی به دیتابیس) ولی دسترسی های کمی برای نوشتن در فایل نیاز داریم.

به طور کلی زمانی که تعدادی دسترسی برای نوشتن به ندرت اتفاق می افتد ولی مرتب دسترسی برای خواندن نیاز است این قفل برتری دارد از آن جهت که اجازه خواندن همروند را خواهد داد.

مقایسه کارایی این دو را میتوان به صورت زیر بیان کرد.

Read-Write Lock: زمانی که با حجم زیادی از خواندن مواجه هستیم که چندین thread به طور همزمان داده‌های مشترک را می‌خوانند و عملیات نوشتن به ندرت انجام می‌شود.

Re-Entrant Lock: زمانی که نیاز دارید مدیریت مجدد ورود به قفل را در یک thread واحد انجام دهید، مانند در توابع بازگشتی یا هنگامی که یک thread نیاز دارد چندین بار به همان قفل دسترسی پیدا کند.