به نام خدا

گزارشکار آزمایش چهار آزمایشگاه سیستم عامل

اعضای گروه:

علی زمانی **810101436**

برنا فروهری **810101480**

نیما خنجری **810101416**

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

هنگامی که در یک ناحیه‌ی بحرانی از یک قفل استفاده می‌کنیم، هدف این است که یک thread بتواند به این ناحیه دسترسی داشته باشد. اگر وقفه را غیر فعال نکنیم ممکن است مشکلات متعددی در هنگامی که critical section حال اجرا است رخ بدهد. مثلا فرض کنید که در طول اجرای critical section, وقفه ای رخ بدهد و در نتیجه آن, برنامه به یک ناحیه‌ی دیگر هدایت شود. این گونه وضعیت lock ممکن است که دچار ناسازگاری شود. برای نمونه, اگر lock توسط thread جاری گرفته شده باشد، اما threadای دیگر سعی کند همان قفل را دوباره بگیرد, سیستم درگیر deadlock خواهد شد. از طرفی, اگر وقفه‌ها فعال باشند و چندین thread به صورت غیرهمزمان اجرا شوند، ممکن است thread دیگری بدون توجه به وضعیت lock وارد critical section شود و به داده‌های محافظت‌شده دسترسی پیدا کند.

حال به سراغ توابع pushcli و popcli می رویم و تفاوت آن ها را با cli , sti بیان می کنیم.

به شور کلی, این توابع برای مدیریت وقفه‌ها استفاده می‌شوند. اما توابع cli و sti سطح پایین ترند. تابع pushcli مانند cli عمل می‌کند، اما یک شمارنده‌ی داخلی نگهداری می‌کند تا تعداد دفعاتی که وقفه غیرفعال شده است، ثبت شود. این کار باعث می‌شود که غیرفعال کردن وقفه‌ها به صورت تو در تو (nested) مدیریت شود.

از طرفی, تابع popcli شمارنده را کاهش می‌دهد. اگر شمارنده به صفر برسد، وقفه‌ها دوباره فعال می‌شوند. این امر با استفاده از تابع سطح پایینتر sti صورت می گیرد.

اگر در یک تابع یا thread، وقفه غیرفعال شود و سپس یک تابع دیگر نیز دوباره وقفه‌ها را غیرفعال کند، هنگام بازگشت از آن توابع باید به درستی وقفه‌ها را دوباره فعال کرد. استفاده مستقیم از cli و sti می‌تواند باعث شود که وقفه‌ها به اشتباه **زودتر از موعد** فعال شوند. پس در این شرایط می توان از توابع pushcli و popcli استفاده کرد تا تضمین شود که وقفه‌ها در زمان درست فعال خواهند شد چون pushcli و popcli با استفاده از یک شمارنده، امکان مدیریت تو در توی وقفه‌ها را فراهم می‌کنند، در حالی که cli و sti مستقیماً بیت وقفه یا همان (IF) را تنظیم یا پاک می‌کنند و قابلیت مدیریت تو در تو را ندارند.

پرسش 2: حالات مختلف پردازهها در xv6 را توضیح دهید تابع schede چه وظیفه ای دارد؟

وضعیت های مختلفی که در سیستم عامل xv6 برای یک پردازه تعریف شده به این ترتیب هستند:

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

1- UNUSED : این وضعیت معادل وضعیت terminated می باشد یعنی نماینده حالتی است که پردازه خاتمه یافته یا اصلا هنوز ایجاد نشده است.

2- EMBRYO : این وضعیت نماینده حالتی است که پردازه تازه ایجاد شده ولی هنوز آماده اجرا نیست.(عملا مشابه وضعیت new در شکل می باشد)

3- RUNNABLE : اگر پردازه در این وضعیت باشد یعنی آماده اجرا است و صرفا باید منتظر بماند تا به cpu اختصاص یابد. می توان گفت این وضعیت در xv6 معادل وضعیت ready است.

4- RUNNING : در این وضعیت پردازه در حال اجرا شدن روی cpu می باشد.مشابه running در شکل بالا.

5- SLEEPING : در این وضعیت پردازه منتظر تکمیل شدن یک فرآیند I/O یا رویدادی خاص است و عملا معادل وضعیت waiting در شکل بالا می باشد.

6- ZOMBIE : این وضعیت نماینده حالتی است که پردازه موردنظر خاتمه پیدا کرده ولی هنوز resource های آن به طور کامل آزاد نشده اند.

تابع sched همانطور که از نامش می توان حدس زد, مسئول schedule کردن پردازه های فعال در سیستم می باشد. از جمله وظایفی که این تابع دارد:

ذخیره‌ی وضعیت پردازه جاری: برای مثال هنگامی که پردازه ای به حالت sleep میرود و cpu را رها می کند و باید وضعیت آن ذخیره شود, تابع sched وضعیت آن را ذخیره میکند.

انتخاب پردازه‌ی بعدی برای اجرا: این تابع با استفاده از الگوریتم های زمان بندی که در آزمایش قبلی هم دیدیم, پردازه‌ی بعدی برای اجرا را از میان پردازه‌های RUNNABLE انتخاب می‌کند.

Context Switching: sched عملیات context switching را انجام می‌دهد تا کنترل cpu به پردازه‌ی جدید منتقل شود. این امر شامل تنظیم مقادیر رجیسترها، تغییر جدول صفحات و سایر منابع مرتبط است.

تخصیص cpu به صورت مساوی : این تابع تلاش می‌کند با استفاده از الگوریتم زمان‌بندی زمان استفاده از cpu بین پردازه‌ها به طور منصفانه تقسیم شود.

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

یکی از روش های سینک کردن این حافظه های نهان با یکدیگر روش Modified-Shared-Invalid می باشد. طبق این روش هر بلوک از داده‌ها در حافظه نهان می‌تواند در یکی از سه حالت زیر قرار داشته باشد:

Modified: داده در حافظه نهان محلی تغییر یافته و با نسخه‌ی اصلی آن در حافظه اصلی (Main Memory)همگام نیست.

Shared: داده بین چندین حافظه نهان مشترک است و هیچ تغییری در آن داده اعمال نشده است.

Invalid: داده در این حافظه نهان نامعتبر است، یعنی نسخه‌ی معتبری از آن وجود ندارد.

حال این روش برای اطمینان از سینک بودن حافظه های نهان با یکدیگر, از پیام‌هایی بین هسته‌ها و حافظه اصلی استفاده می‌کند. این پیام ها بر چند نوع هستند:

Read Miss: اگر پردازنده بخواهد داده‌ای را بخواند که در حالت Invalid باشد، درخواست به حافظه اصلی یا حافظه نهان دیگری که داده‌ی معتبر دارد، ارسال می‌شود . پس از بارگذاری، داده به حالت Shared تغییر می‌کند.

Write Miss: اگر پردازنده بخواهد داده‌ای را بنویسد که در حالت Invalid یا Shared باشد, ابتدا یک پیام Broadcast به سایر حافظه‌های نهان ارسال می‌شود تا کپی‌های Shared یا Modified را Invalidکند. سپس داده به حالت Modified تغییر می‌یابد.

Read Hit/Write Hit: اگر داده در حافظه نهان موجود و در حالت مناسب باشد, عملیات بدون نیاز به هماهنگی با حافظه اصلی یا سایر حافظه‌های نهان انجام می‌شود.

Invalidate: وقتی یک پردازنده داده‌ای را در حالت Modified تغییر می‌دهد، پیام‌هایی به سایر هسته‌ها ارسال می‌کند تا نسخه‌های Shared آن داده در حافظه‌های نهان دیگر نامعتبر شوند.

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

قفل بلیت یکی از روش‌های همگام‌سازی بین پردازنده‌ها است که برای حل مشکل دسترسی هم زمان به منابع مشترک استفاده می‌شود. این روش چند ویژگی اصلی دارد که بدین صوریت هستند:

مهم ترین ویژگی این روش, مکانیزمی است که برای دسترسی پردازنده ها به منابع پیاده سازی شده. با استفاده از ساختاری شبیه به صف, قفل بلیت تضمین می‌کند که پردازنده‌ها بر اساس ترتیب درخواست‌شان به منابع دسترسی پیدا می‌کنند. به این گونه که انگار هر پردازنده بلیتی دریافت میکند و منتظر نوبتش می ماند.

این گونه, این قفل به پردازنده‌ها اجازه می‌دهد منتظر نوبت خود شوند و در صورت امکان به کار دیگری بپردازند. در نتیجه این مکانیزم باعث کاهش سربار انتظار می‌شود.از طرفی چون پردازنده‌ها به صورت صف وارد ناحیه بحرانی می‌شوند، امکان Starvation وجود ندارد.

در مثالی که زده شده, پردازنده‌های CPU1 و CPU2 به یک آدرس حافظه مشترک دسترسی دارند (0xA300). در صورتی که CPU1 مقدار این آدرس را به 101 تغییر دهد، دو مشکل ممکن است به وجود بیاید:

1. کش پردازنده CPU2 همچنان مقدار قدیمی (100) را نگه دارد.
2. مقدار در حافظه اصلی (MEM1) نیز هنوز به‌روز نشده باشد.

حال اگر از قفل بلیت برای هندل کردن این شرایط استفاده کنیم, CPU1 ابتدا قفل را دریافت می‌کند و دسترسی به critical section که همان(0xA300) میباشد, برای سایر پردازنده‌ها ممنوع می‌شود . بعد از پایان تغییر مقدار توسط CPU1 ، قفل آزاد می‌شود. هم زمان، پروتکل‌هایی مانند Modified-Shared-Invalid در معماری کش پردازنده تضمین می‌کنند که مقدار جدید در کش CPU2 و MEM1 به روزرسانی شود. در نتیجه, با جلوگیری از دسترسی همزمان CPU2، مشکل تضاد و ناسازگاری داده‌ها حل می‌شود.

پرسش 5: دو مورد از معایب استفاده از قفل با امکان ورود مجدد را بیان نمایید.

1- قفل‌های با امکان ورود مجدد نیاز به نگهداری اطلاعاتی درباره تعداد دفعات قفل شدن توسط یک thread خاص دارند. برای این منظور، معمولاً از یک شمارنده و یک شناسه (ID) برای شناسایی thread استفاده می‌شود. این موضوع باعث پیچیدگی در پیاده‌سازی و افزایش سربار (Overhead) می‌شود. در نتیجه, عملکرد سیستم را در مقایسه با سیستم هایی با قفل‌های ساده تر, کاهش می دهد.

2- اگر برنامه نویس به درستی قفل‌ها را release نکند، ممکن است تعداد دفعات باز کردن قفل با تعداد دفعات قفل کردن تطابق نداشته باشد. این موضوع می‌تواند منجر به قفل شدن دائمی منبع (Deadlock) شود. برای مثال, اگر یک Thread قفل را چند بار کسب کند اما به اشتباه فقط یک بار آن را آزاد کند، سایر رشته‌ها نمی‌توانند به منبع دسترسی پیدا کنند، زیرا قفل همچنان فعال باقی می‌ماند.

پرسش 6: یکی دیگر از ابزارهای 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.

به طور کلی قفل های Read-Write lock دو حالت اصلی دارند:

حالت Read: در این حالت, چندین thread به طور همزمان می‌توانند منبع را بخوانند. این حالت زمانی استفاده می‌شود که داده تغییر نمی‌کند و فقط نیاز به خواندن داده است.

حالت Write : در این حالت, تنها یک thread می‌تواند به منبع دسترسی داشته باشد. در نتیجه دسترسی سایرThread ها (خواندن یا نوشتن) به منبع قفل می‌شود تا از بروز شرایط رقابتی Race Conditions جلوگیری شود.

پس در هنگام استفاده از قفل Read-Write lock, اگر یک Thread درخواست خواندن (Read) دهد و هیچ thread دیگری در حال نوشتن نباشد، اجازه دسترسی به آن داده می‌شود. از طرفی اگر یک thread درخواست نوشتن (Write) دهد، باید صبر کند تا همه عملیات خواندن فعلی به پایان برسند و هیچ thread دیگری در حال خواندن یا نوشتن نباشد.

در نتیجه مکانیزمی که قفل های Read-Write lock دارند, اگر با سیستمی سر و کار داشته باشیم که تعداد عملیات خواندن در آن, خیلی بیشتر از عملیات نوشتن باشد, قفل‌های Read-Write می‌توانند بهبود قابل توجهی در عملکرد ایجاد کنند. زیرا قفل‌های با امکان ورود مجدد, حتی برای خواندن داده نیز دسترسی انحصاری می‌دهند، در حالی که قفل‌های Read-Write اجازه دسترسی اشتراکی به چندین thread برای خواندن داده را می‌دهند.

از طرفی قفل‌های Read-Write با کاهش تعداد thread هایی که به دلیل قفل انحصاری متوقف شده‌اند، امکان استفاده بهتر از پردازنده و منابع را فراهم می‌کنند.

شرح پروژه:

Cache coherency در سیستم عامل xv6

برای محاسبه تعداد سیستم کال های فراخوانی شده در هر یک از cpuها , متغیر count_syscalls را اضافه کرده ایم.

```
// 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 rr;
  int sjf;
  int fcfs;
  uint count_syscalls;
};
```

برای شمارش تمام سیستم کال ها به صورت global , متغیر syscalls_count و قفل syscallslock تعریف شده اند. همچنین قفل syscallslock در تابع tvinit مقداردهی اولیه می شود.

```
struct spinlock tickslock;
struct spinlock syscallslock;
uint ticks;
uint syscalls_count;

void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
  initlock(&syscallslock, "syscall_count");
}
```

با فراخوانی trap و با گرفتن شناسه سیستم کال فعلی, از طریق متغیر eax ضریب مربوطه را اختصاص می دهیم و به هر دو روش global و مختص به cpu, مقادیر را به روزرسانی می کنیم.

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){


    if(myproc()->killed)
      exit();

    uint my_eax = tf->eax;

    acquire(&syscallslock);
    if(my_eax == SYS_open)
      syscalls_count+=3;
    if(my_eax == SYS_write)
      syscalls_count+=2;
    else
      syscalls_count++;
    add_cpu_syscalls(my_eax);
    release(&syscallslock);


    myproc()->tf = tf;
    syscall();
```

```
void add_cpu_syscalls(uint my_eax)
{
  if(my_eax == SYS_open)
    mycpu()->count_syscalls += 3;
  if(my_eax == SYS_write)
    mycpu()->count_syscalls += 2;
  else
    mycpu()->count_syscalls += 1;

}
```

حال, دو سیستم کال مجزا برای هر دو روش در نظر میگیریم. در روش اول مقادیر ذخیره شده در تمام cpuها را با هم جمع می زنیم و باز میگردانیم. در روش دوم, صرفا نیاز است global variable مربوطه را بازگردانیم.

```c
int sum_all_cpus_syscalls()
{
  int count = 0;
  for(int i=0; i<NCPU; i++)
    count += cpus[i].count_syscalls;
  return count;
}
```

روش اول

```c
int sys_count_syscalls_all_cpus(void)
{
  int count;

  acquire(&syscallslock);
  count = syscalls_count;
  release(&syscallslock);

  return count;

}
```

روش دوم

تست برنامه:

trap.c | sysproc.c | get_count_syscalls_all_cpus.c

get_count_syscalls_all_cpus.c > ...

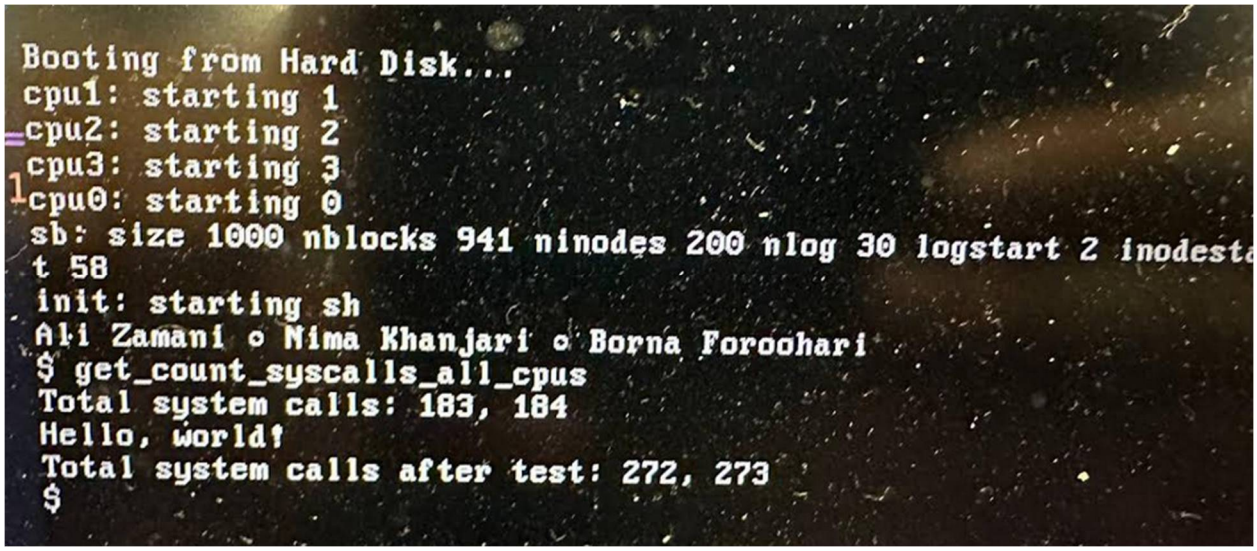
```c
#include "user.h"
#include "types.h"

int main(void)
{
    int count = count_syscalls_all_cpus();
    int count2 = sum_all_cpus_syscalls();
    printf(1, "Total system calls: %d, %d\n", count, count2);

    // Test by making some system calls
    printf(1, "Hello, world!\n");
    count_syscalls_all_cpus(); // Call the system call again

    count = count_syscalls_all_cpus();
    count2 = sum_all_cpus_syscalls();
    printf(1, "Total system calls after test: %d, %d\n", count, count2);

    exit();
}
```



همانطور که در برنامه مشاهده می کنیم با فراخوانی هر دو سیستم کال, نتایج یکسانی را میگیریم.

توجه: همانطور که میبینیم, یک عدد در نتیجه دو روش, تفاوت داریم که به این دلیل است که خود سیستم کال count_syscalls_all_cpus یک سیستم کال حساب می شود و در سیستم کال بعدی محاسبه می شود. فلذا در واقع هر دو سیستم کال عملکرد یکسانی دارند.