به نام خدا

گزارش پروژه چهارم آزمایشگاه سیستم عامل

پاییز ۱۴۰۱

گروه 3: علی هدائی(810199513)، پویا صادقی(810199447)، علی عطاالهی(810199461)

.....

همگام سازی در xv6 🐶

۱) علت غیرفعال کردن وقفه چیست؟ توابع ()pushcli و ()popcli به چه منظور استفاده شده است ؟

وقفه ها را غیر فعال میکنیم تا اطمینان داشته باشیم که کدی که قرار است اجرا شود به صورت atomic اجرا میشود و وقفه ها اختلالی ایجاد نمیکنند.

غیر فعال کردن وقفه توسط دو تابع ()pushcli و ()popcli انجام میشود.

توسط تابع ()pushcli با استفاده از acquire و سپس relase ، وقفه ها را در قسمت critical کد غیر فعال میکنیم(از deadlock جلوگیری میکنیم). سپس popcli فراخوانی میشود، تا وقفه ها دوباره فعال شوند.

دو تابع گفته شد، خودشان از cli و sti استفاده میکنند تا کار های مطرح شده را انجام دهند. اما دو تابع ()pushcli و ()popcli علاوه بر آندو قابلیت های بیشتری دارند مثل قابلیت شمارش اینکه هر کدام چند بار اجرا شده اند (ncli در ساختار پردازنده)

۲) چرا قفل مذکور در سیستم‌های تک هسته‌ای مناسب نیست؟ روی کد توضیح دهید.

چون همانطور که در کد آن در شکل پایین مشخص است، شرط holding وجود دارد که یعنی به صورت busy waiting است که باعث میشود اگر پردازه ای قفل را برای مدت طولانی در اختیار بگیرد، مشکلاتی ایجاد شود.

```c
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

۳) مختصری راجع به تعامل میان پردازه‌ها توسط دو تابع مذکور توضیح دهید. چرا در مثال تولیدکننده/مصرف‌کننده استفاده از قفل‌های چرخشی ممکن نیست.

در aquiresleep تا وقتی که پردازه امکان به دست گرفتن قفل را نداشته باشد، با استفاده از تابع sleep به خواب میرود. پس میتوان نتیجه گرفت که busy waiting نداریم.

وقتی که realsesleep فراخوانی میشود، تمام پردازه هایی که روی چنل قفل به خواب رفته بودند، توسط wakeup از SLEEP به حالت RUNNABLE میروند.

چون اگر در مسئله مطرح شده از spinlock استفاده کنیم، ممکن است بلافاصله بعد از آزاد شدن قفل توسط مصرف کننده ، قفل به تولیید کننده برسد.

۴) حالات مختلف پردازه‌ها در xv6 را توضیح دهید. تابع sched() چه وظیفه‌ای دارد؟

UNUSED: پردازه ای که استفاده ای از آن نشده است و منابع و پردازنده به آن اختصاص داده نشده است

EMBYRO: بوده است، به این استیت تغییر پیدا میکند UNUSED صدا زده شود، پردازه ای که در حالت allocproc وقتی که

SLEEPING: یعنی منبع یا منابع مورد نیاز پردازه آماده نیست و پردازه در پردازنده نیست و اسکجولر از آن استفاده نمیکند

RUNNABLE: پردازه ای که تمام منابع مورد نیاز برای اجرا دارد و منتظر است تا پردازنده به آن توسط اسکجولر اختصاص داده شود

RUNNING: پردازه ای که پردازنده به آن اختصاص داده شده است و در حال اجراست

ZOMBIE: پردازه ای که کارش تمام شده است ولی هنوز در پی تیبل وجود دارد که این اتفاق زمانی می افتد که پردازه پدر ویت را صدا نکرده باشد

تابع sched در پایان کار یک پردازه صدا زده میشود و پس از چک کردن خطا های ممکن مثل گرفته نشدن لاک ptable و ... ، عمل context switch انجام میشود بگونه ای که context پردازه فعلی ذخیره میشود و context با scheduler جایگزین میشود.

۵) تغییری در توابع دسته دوم داده تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی نمایید.

همانطور که در ساختار mutex در شکل زیر مشخص است، یک owner تعریف شده است که مشخص میکند کدام پردازه قفل را در اختیار دارد پس بنابراین فقط owner قفل میتواند آن را آزاد کند و busy waiting هم نداریم و مشکل گفته شده هم به وجود نمی آید.

```
struct mutex {
        atomic_long_t           owner;
        raw_spinlock_t          wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
        struct list_head        wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map      dep_map;
#endif
};
```

۶) یکی از روش‌های افزایش کارایی در بارهای کاری چندریسه‌ای استفاده از حافظه تراکنشی بوده که در کتاب نیز به آن اشاره شده است. به عنوان مثال این فناوری در پردازنده‌های جدیدتر اینتل تحت عنوان افزونه‌های همگامسازی تراکنشی (TSX) پشتیبانی می‌شود. آن را مختصراً شرح داده و نقش حذف قفل را در آن بیان کنید ؟

حافظه تراکنشی مدلی برای کنترل دسترسی های همزمان حافظه در محدوده برنامه نویسی موازی است. در برنامه نویسی موازی، کنترل همزمانی تضمین می کند که thread هایی که به صورت موازی در حال اجرا هستند، منابع یکسان را به طور همزمان آپدیت نمیکنند. در واقع دنباله ای از عملیات های نوشتن و خواندن از حافظه است که به صورت atomic انجام میشود.که این کار چند مزیت دارد. اول اینکه برنامه نویسان از استدلال در مورد درستی و قفل خود رها می شوند،ساختارهای داده مشترک تضمین شده است که حتی در صورت خرابی ، ثابت نگه داشته می شوند، تراکنش ها میتوانند به طور طبیعی انجام شوند، که توسعه نرم افزار موازی قابل ترکیبی را راحت تر میکند همچنین دیگر deadlock نخواهیم داشت. روش کار آن بدین صورت است که اگر کانفیلیکتی وجود نداشته باشد، سیستم های حافظه تراکنشی میتواند چندین تراکنش را به صورت موازی اجرا کند و در نهایت تراکنش حافظه ای کامیت می شود اما اگر کانفیلیکت داشته باشند به گونه ای که به یک بخش حافظه بخواهند دسترسی پیدا کنند و حداقل یکی از آنها بخواهد بنویسد این عملیات برگشت داده میشود. استفاده از Lock Elision سه هدف را دنبال میکند:

1) پیاده سازی را به اندازه کافی ساده نگه داشتن تا در یک wrapper گنجانده شود، به برنامه نویس اجازه می دهد تا به سادگی از عملکردهای wrapper برای LOCK و UNLOCK استفاده کند بدون نیاز به دانستن اینکه آیا قفل یا حافظه تراکنشی استفاده می شود.

2) پیاده سازی را انعطاف پذیر نگه داشتن تا اینکه پیاده سازی های مختلف قفل و پیاده سازی های مختلف حافظه تراکنشی را بتوان بدون تغییر ساختار پیش بینی استفاده کرد.

3) پیش‌بینی‌کننده را طوری طراحی کردن که تراکنش‌ها تا حد امکان استفاده شوند، سپس خاموش کنید و هر زمان که احتمال لغو قفل وجود داشت، آن را خاموش کنید و به سمت قفل بروید.

.....

مانع (barrier) 😕

۷) پیاده سازی ماکروی ()barrier در لینوکس برای معماری x86 را فقط بنویسید.

```
#define barrier() __asm__ __volatile__("":::"memory")
```

```
#define barrier() asm volatile("" ::: "memory")
```

به صورت اول زده شده اما صورت دوم هم ولید هست (مشابه همین در آزمایشگاه های قبلی پیاده شده).

۸) آیا یک دستور مانع حافظه باید مانع بهینه سازی هم باشد؟ نام ماکروی پیاده سازی سه نوع مانع حافظه در لینوکس در معماری x86 را به همراه دستورالعمل های ماشین پیاده سازی آنها ذکر کنید.

این دستور باتوجه به ماهیتش، بهینه سازی در زمان کامپایل را محدود میکند. یعنی بدین صورت که جلوی تغییر ترتیب دستورات توسط کامپایلر را میگیرد. البته توجه شود روی ترتیب دهی پردازنده تاثیری ندارد. این دستور مانع کامپایلر GCC برای بهینه سازی هایی همانند تغییر ترتیب نوشتن ها و خواندن ها حافظه و همچنین نمیگذارد GCC مقادیر را cache کند و باید از دسترسی به مموری استفاده کند. یعنی اینکه در بهینه سازی اثر دارد و محدودیت هایش را اعمال میکند اما لزومی ندارد کلا بهینه سازی را متوقف کند.

نوع اول -> mb:

```
#include <asm/system.h>

void mb(void);
```

نوع دوم -> rmb:

```
#include <asm/system.h>

void rmb(void);
```

نوع سوم -> wmb:

```
#include <asm/system.h>

void wmb(void);
```

۹) یک کاربرد از مانع در پردازش موازی ارائه دهید.

شرایطی را تصور کنید که عملیاتی را به چندین بخش تقسیم کرده ایم و هر بخش بصورت یک پردازه مجزا دارد اجرا میشود، مثلا میخواهیم اطلاعاتی را از محتویات چندین فایل استخراج کنیم و فایلها را تقسیم بندی کرده هرکدام را به یکی از این پردازه ها نسبت داده ایم. حال در ادامه، به این اطلاعات بصورت تجمیعی احتیاج داریم، یعنی اطلاعات را per file یا per categori نمیخواهیم و همه ی آنها تجمیع شده مورد نیاز میباشند(مثال بررسی تعداد فایل های موجود در تمام کتابخانه برحسب هر ژانر، که هر پردازه، یک بخش از کتابخانه را جست و جو میکرد). در چنین حالتی، ما نیاز داریم تمامی این پردازه ها کارهایشان را انجام داده و به اتمام برسانند. و بدین صورت مانع در پردازش موازی به کمک ما می آید.

در مثال قبل، پردازه ها به پایان میرسیدند(مدل فورک-جوین)، اما لزوما اینگونه نیست و میتوانند پس از آن نیز ادامه داشته باشند،به عنوان مثال، تعدادی عملیات که برروی یک ماتریس قرار است انجام شود را متصور شوید؛ این عملیات ها باید به ترتیب مشخص انجام شود و برخی از آنها به داده های ماتریس حساس میباشد(به چندین درایه برای محاسبات نیاز دارد). برای اجرای عملیات های حساس به مقدار، لازم است که عملیات های قبلی به طور کامل انجام شود. حال چندین پردازه داریم که هرکدام ببروی یک قسمت ماتریس کار میکنند(نوشتن آنها مربوط به همان بخش اما خواندن لزوما نه و طبعا برروی یک کپی از نسخه اصلی نوشتن را انجام میدهند که تداخل نخورند). در چنین شرایطی و قبل از ورود یکی از این پزدازه ها به توابع ذکر شده، باید منتظر شد تا سایرین نیز برسند. در اینجا نیز از barrier استفاده میکنیم.

همچنین حالت دیگر، مربوط به ماکرو ذکر شده در لینوکس است که نمیگذارد کامپایلر از کش مموری هم استفاده کند و باید مقادیر درخواست شده بعد آن دستور مجدد خوانده بشوند. یکی از حالت های آن نیز مربوط به voletile است که ممکن است یک پوینتر مقدار یک متغیر مثلا کانستنت را تغییر بدهد اما کامپایلر بدلیل کش کردن مقدار آن، متوجه تغییر نشود.

.....

شبیه سازی مسئله فلاسفه خورنده 😸

برای شبیه سازی این مسئله طبق صورت نیازمند پیاده سازی سمافور و سه سیستم کال برای آن بودیم که بدین شرح است:

proc.c و user.h

```
+ #define MAX_SEMAPHORE 5
```

برای سمافور از یک داده ساختار استفاده شده که درون آن یک صف برای پروسس ها در نظر گرفته شده است. همچنین حداکثر تعداد این سمافور ها 5 در نظر گرفته شده است.

```
+ struct semaphore {
+   int value, front_proc_index, procs_queue_size;
+   struct proc *queue[MAX_SEMAPHORE_PROC];
+   struct spinlock lock;
+ } semtable[MAX_SEMAPHORE];
+
```

در ادامه برای آنکه یک سمافور گرفته یا آزاد شود باید value و همچنین تعداد پروسس هایی که در صف سمافور هستن مدنظر گرفته شود که بدین صورت داریم :

```
+ void sem_init(int, int);
+ void sem_acquire(int);
+ void sem_release(int);
```

```
+
+ void
+ sem_init(int i, int value)
+ {
+   semtable[i].value = value;
+   semtable[i].front_proc_index = 0;
+   semtable[i].procs_queue_size = 0;
+ }
+
```

```
+ void
+ sem_acquire(int i)
+ {
+   acquire(&semtable[i].lock);
+
+   if(semtable[i].value <= 0)
+   {
+     struct proc *p = myproc();
+     int index = (semtable[i].front_proc_index + semtable[i].procs_queue_size) %
  MAX_SEMAPHORE_PROC;
+     semtable[i].procs_queue_size++;
+     semtable[i].queue[index] = p;
+     sleep(&p->pid, &semtable[i].lock);
+   }
+   semtable[i].value--;
+
+   release(&semtable[i].lock);
+ }
```

```
+ void
+ sem_release(int i)
+ {
+   acquire(&semtable[i].lock);
+   semtable[i].value++;
+
+   if(semtable[i].value > 0 && semtable[i].procs_queue_size > 0)
+   {
+       int temp_index = semtable[i].front_proc_index;
+       semtable[i].front_proc_index = (semtable[i].front_proc_index + 1) % NPROC;
+       semtable[i].procs_queue_size--;
+       wakeup(&semtable[i].queue[temp_index]->pid);
+   }
+
+   release(&semtable[i].lock);
+ }
```

usys.S و syscall.h و syscall.c :

حال باید برای این سیستم کال‌های مورد نیاز پیاده‌سازی شوند که داریم :

```
+ extern int sys_sem_init(void);
+ extern int sys_sem_acquire(void);
+ extern int sys_sem_release(void);
```

```
+ [SYS_sys_sem_init]          sys_sem_init,
+ [SYS_sys_sem_acquire]       sys_sem_acquire,
+ [SYS_sys_sem_release]       sys_sem_release,
  };
```

```
+ #define SYS_sys_sem_init          31
+ #define SYS_sys_sem_acquire       32
+ #define SYS_sys_sem_release       33
```

```
+ void
+ sys_sem_init(void)
+ {
+   int i, v;
+   argint(0, &i);
+   argint(1, &v);
+
+   sem_init(i, v);
+ }
+
+ void
+ sys_sem_acquire(void)
+ {
+   int i;
+   argint(0, &i);
+
+   sem_acquire(i);
+ }
+
+ void
+ sys_sem_release(void)
+ {
+   int i;
+   argint(0, &i);
+
+   sem_release(i);
}
```

```
SYSCALL(print_all_procs)
+ SYSCALL(sem_init)
+ SYSCALL(sem_acquire)
+ SYSCALL(sem_release)
```

phillsofs.c :

حال در ادامه سراغ این می‌رویم که مسئله را شبیه سازی کنیم.

برای این‌کار ابتدا 5 سمافور را آغاز میکنیم که هر سمافور نماینده یک قاشق روی میز است (یا از دیدگاهی دیگر یک نفر روی میز که فرقی ندارد).

```
+ int main(int argc, char *argv[])
+ {
+     init_game();
+     start_game();
+ }
```

```
void init_game() {
    for(int i = 0; i < MAX_SEM; i++)
        sem_init(i, 1);
}
```

در ادامه برای آنکه هر فیلسوف کار خود را شروع کند، پنج بار فورک میکنیم که هر پروسس برای یکی از فیلسوف ها خواهد بود. برای هر فیلسوف تابع philosopher را انجام می‌دهیم که در آن به تعداد راندهای مشخصی باید غذا بخورد. در ادامه سراغ الگوریتم آن خواهیم رفت.

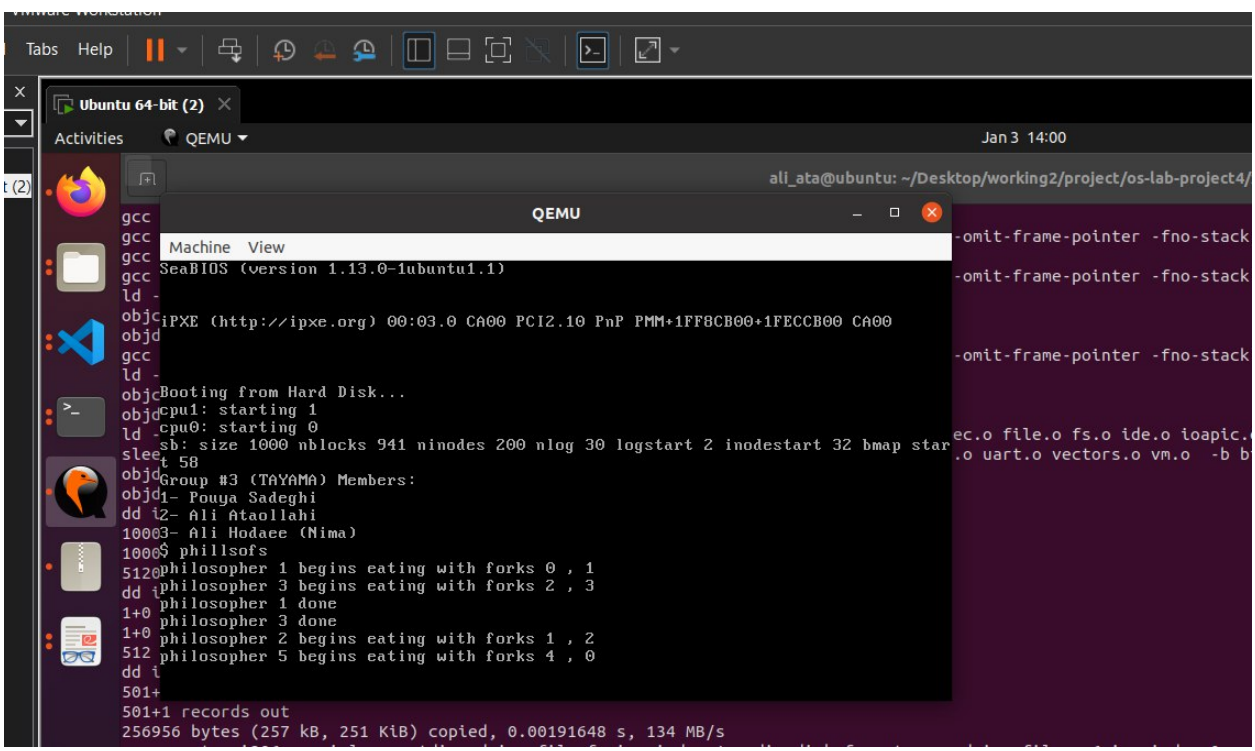
```
void start_game() {
    for(int i = 0; i < MAX_SEM; i++) {
        int pid = fork();

        if(pid == 0) {
            philosopher(i + 1, i, (i + 1) % MAX_SEM);
            exit();
        }
    }

    while (wait());      You, 3 hours ago • fix error …
    exit();
}
```

روش انجام شده بدین صورت است که برای فیلسوف های با شماره زوج ابتدا قاشق سمت چپ سپس سمت راست را برمی‌داریم و برای فردها برعکس عمل می‌کنیم. این عمل باعث می‌شود که به بن بست نخوریم و هر عکس فقط یک قاشق نداشته باشد.

در ادامه بعد از مدت کوتاهی قاشق با sem_release رها می‌شود.

```c
void philosopher(int id, int left, int right) {
    for(int i = 0; i < ROUND; i++) {
        if(id % 2 == 0) {
            sem_acquire(left);
            sem_acquire(right);
        }
        else {
            sem_acquire(right);
            sem_acquire(left);
        }
        sleep(30 * id);
        printf(1, "philosopher %d begins eating with forks %d , %d\n", id, left + 1, right + 1);
        sleep(300);
        sem_release(left);
        sem_release(right);
        printf(1, "philosopher %d done\n", id);
    }
}
```

خروجی نمونه :



os-lab-project4

QEMU

Machine View

```
Group #3 (TAYAMA) Members:
1- Pouya Sadeghi
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ phillsofs
philosopher 1 begins eating with forks 1 , 2
philosopher 3 begins eating with forks 3 , 4
philosopher 1 done
philosopher 3 done
philosopher 2 begins eating with forks 2 , 3
philosopher 5 begins eating with forks 5 , 1
philosopher 2 done
philosopher 5 done
philosopher 1 begins eating with forks 1 , 2
philosopher 4 begins eating with forks 4 , 5
philosopher 1 done
philosopher 2 begins eating with forks 2 , 3
philosopher 4 done
philosopher 5 begins eating with forks 5 , 1
philosopher 2 done
philosopher 3 begins eating with forks 3 , 4
philosopher 5 done
philosopher 1 begins eating with forks 1 , 2
```

```
philosopher 3 begins eating with forks 3 , 4
philosopher 5 done
```

QEMU

Machine View

```
2- Ali Ataollahi
3- Ali Hodaee (Nima)
$ phillsofs
philosopher 1 begins eating with forks 1 , 2
philosopher 3 begins eating with forks 3 , 4
philosopher 1 done
philosopher 3 done
philosopher 2 begins eating with forks 2 , 3
philosopher 5 begins eating with forks 5 , 1
philosopher 2 done
philosopher 5 done
philosopher 1 begins eating with forks 1 , 2
philosopher 4 begins eating with forks 4 , 5
philosopher 1 done
philosopher 2 begins eating with forks 2 , 3
philosopher 4 done
philosopher 5 begins eating with forks 5 , 1
philosopher 2 done
philosopher 3 begins eating with forks 3 , 4
philosopher 5 done
philosopher 3 done
philosopher 4 begins eating with forks 4 , 5
philosopher 4 done
$
```