گزارش کار پروژه چهارم آزمایشگاه سیستم عامل

گروه ۱۳:

سیده زینب پیش بین - ۸۱۰۱۹۹۵۹۷

محمد جواد بشارتی - ۸۱۰۱۹۹۳۸۶

آدرس مخزن github:

https://github.com/JavadBesharati/OS-lab-projects-UT-Spring-2023

شناسه آخرین commit:

49aadded60907afa476d36b726fd74aef6eba92e

شبیه سازی مسأله readers-writers readers-priority

در فایل proc.c تغییرات زیر را اعمال می‌کنیم:

```
12  #define MAX_SEMAPHORE           5
13
14  struct {
15    struct spinlock lock;
16    struct proc* proc[NPROC];
17    int last;
18    int v;
19    int m;
20  } semaphores[MAX_SEMAPHORE];
21
22  struct
23  {
24    struct spinlock lock;
25  } condvar;
```

```
34  int nextpid = 1;
35  int readers_cnt = 0;
```

همچنین فراخوانی‌های سیستمی خواسته شده را به شکل زیر در همان proc.c اضافه می‌کنیم:

```
888 void sem_init(int i, int v_, int m_)
889 {
890   acquire(&semaphores[i].lock);
891   semaphores[i].v = v_;
892   semaphores[i].m = m_;
893   semaphores[i].last = 0;
894   release(&semaphores[i].lock);
895 }
```

```
897  void sem_acquire(int i)
898  {
899    acquire(&semaphores[i].lock);
900    if (semaphores[i].m < semaphores[i].v)
901    {
902      semaphores[i].m++;
903    }
904    else
905    {
906      semaphores[i].proc[semaphores[i].last] = myproc();
907      semaphores[i].last++;
908      sleep(myproc(), &semaphores[i].lock);
909    }
910    release(&semaphores[i].lock);
911  }
```

```
913  void sem_release(int i)
914  {
915    acquire(&semaphores[i].lock);
916    if (semaphores[i].m < semaphores[i].v && semaphores[i].m > 0)
917    {
918      semaphores[i].m--;
919    }
920    else if (semaphores[i].m == semaphores[i].v)
921    {
922      if (semaphores[i].last == 0)
923      {
924        semaphores[i].m--;
925      }
926      else
927      {
928        wakeup(semaphores[i].proc[0]);
929        for (int j = 1; j < NPROC; j++)
930          semaphores[i].proc[j - 1] = semaphores[i].proc[j];
931        semaphores[i].last--;
932      }
933    }
934    release(&semaphores[i].lock);
935  }
```

در ادامه تعدادی تابع کمکی برای پیاده‌سازی آنچه مسأله خواسته به همان فایل proc.c اضافه شده‌اند:

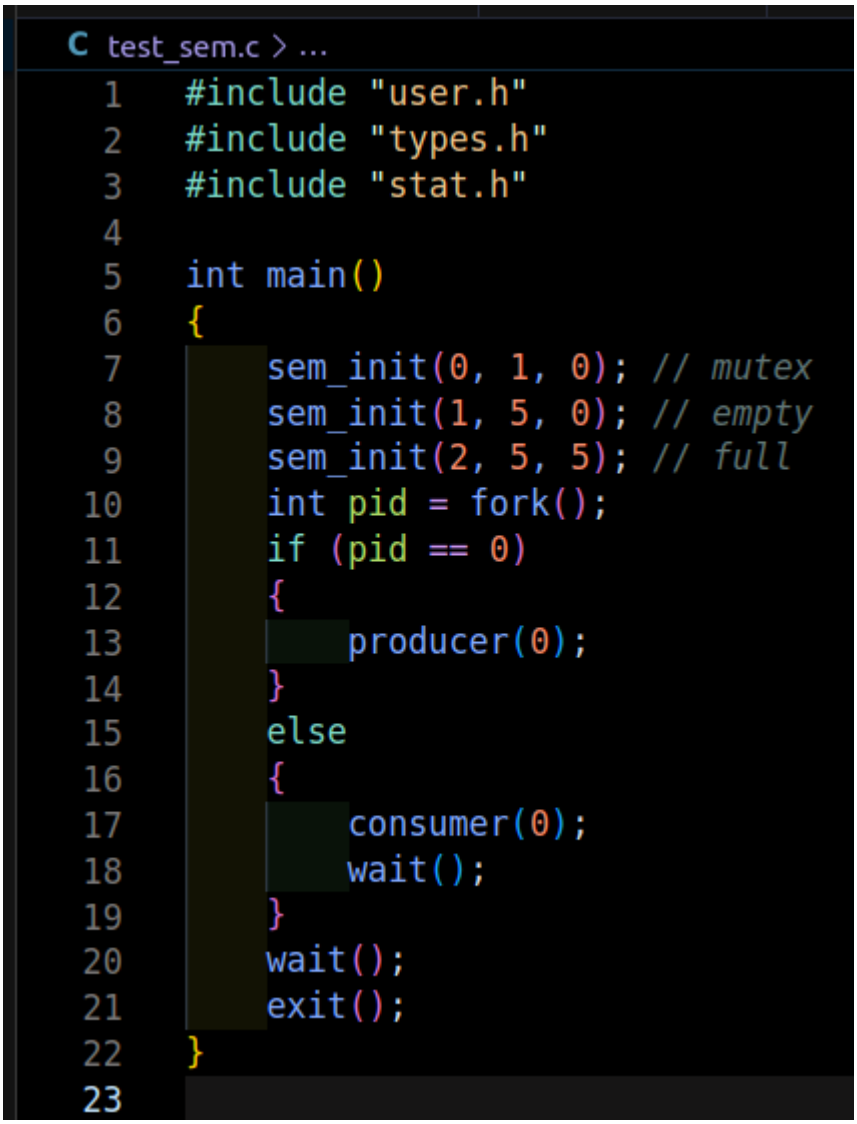
```
937  void producer(int i)
938  {
939    while (i < 10)
940    {
941      cprintf("produce an item %d in next produced\n", i);
942      sem_acquire(1);
943      sem_acquire(0);
944      cprintf("add next produced to the buffer\n");
945      sem_release(0);
946      sem_release(2);
947      i++;
948    }
949  }
950
951  void consumer(int i)
952  {
953    while (i < 10)
954    {
955      sem_acquire(2);
956      sem_acquire(0);
957      cprintf("remove an item from buffer to next consumed\n");
958      sem_release(0);
959      sem_release(1);
960      cprintf("consume the item %d in next consumed\n", i);
961      i++;
962    }
963  }
```

```
965  void to_sleep(void *chan)
966  {
967    struct proc *p = myproc();
968
969    if (p == 0)
970      panic("sleep");
971
972    acquire(&ptable.lock);
973
974    p->chan = chan;
975    p->state = SLEEPING;
976
977    sched();
978
979    p->chan = 0;
980
981    release(&ptable.lock);
982  }
983
984  void cv_wait(void *condvar)
985  {
986    to_sleep(condvar);
987  }
988
989  void cv_signal(void *condvar)
990  {
991    wakeup(condvar);
992  }
```

```
 993
 994  int test_variable = 0;
 995
 996  void reader(int i, void *condvar)
 997  {
 998    readers_cnt++;
 999    cprintf("reader %d init\n", i);
1000    if (writers_cnt)
1001    {
1002      cv_wait(&condvar);
1003    }
1004    cprintf("reader %d reads one item from buffer\n", i);
1005    cprintf("number of active readers: %d\n", readers_cnt);
1006    readers_cnt--;
1007    cv_signal(&condvar);
1008  }
1009
1010  void writer(int i, void *condvar)
1011  {
1012    cprintf("readers: %d\n", readers_cnt);
1013    cprintf("writer %d init\n", i);
1014    if (readers_cnt || writers_cnt)
1015    {
1016      cv_wait(&condvar);
1017    }
1018    writers_cnt++;
1019    test_variable++;
1020    cprintf("test varibale is %d\n", test_variable);
1021    cprintf("writer %d writes next item in buffer\n", i);
1022    cprintf("number of active writers: %d\n", writers_cnt);
1023    writers_cnt--;
1024    cv_signal(&condvar);
1025  }
```

در ادامه در فایل‌های syscall.c، defs.h، sysproc.c و ... مشابه با آزمایش‌های پیشین تغییرات مورد نیاز برای اجرای فراخوانی‌های سیستمی اعمال شد.

برنامه‌های سطح کاربر اضافه شده:



C test_condvar.c > main()

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

struct condvar
{
    struct spinlock lock;
};

int main()
{
    struct condvar c;
    int pid = fork();
    if (pid < 0)
    {
        printf(2, "Error forking first child!\n");
    }
    else if (pid == 0)
    {
        sleep(100);
        printf(1, "Child 1 Executing.\n");
        cv_signal(&c);
    }
    else
    {
        pid = fork();
        if (pid < 0)
        {
            printf(2, "Error forking second child!\n");
        }
        else if (pid == 0)
        {
            cv_wait(&c);
            printf(1, "Child 2 Executing.\n");
        }
        else
        {
            int i;
            for (i = 0; i < 2; i++)
            {
                wait();
            }
        }
    }
    exit();
}
```

C test_readers_writers.c > main()

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

struct condvar
{
    struct spinlock lock;
};

int main()
{
    struct condvar c;

    int pid = fork();
    if (pid < 0)
    {
        printf(2, "Error forking child!\n");
    }
    else if (pid == 0)
    {
        int pid1 = fork();
        if (pid1 == 0)
        {
            int pid1 = fork();
            if (pid1 == 0)
            {
                writer(1, &c);
            }
            else
            {
                reader(2, &c);
                wait();
            }
        }
```


C test_readers_writers.c > ...
35 }
36 else
37 {
38 reader(1, &c);
39 wait();
40 }
41 }
42 else
43 {
44 int pid1 = fork();
45
46 if (pid1 == 0)
47 {
48 writer(0, &c);
49 }
50 else
51 {
52 reader(0, &c);
53 wait();
54 }
55 wait();
56 }
57 exit();
58 return 0;
59 }
60

پاسخ به سؤالات تشریحی:

۱. علت این مسئله اطمینان از اجرا شدن کدها به صورت atomic است. یعنی کدهای وقفه را نمی‌توان مسدود کرد و برای حفاظت از ناحیه critical باید وقفه‌ها غیرفعال شده باشند.

این عملیات به کمک دو تابع pushcli و popcli انجام می‌شود. بدین صورت که با استفاده از pushcli وقفه‌ها را غیرفعال می‌کنیم و از acquire استفاده می‌کنیم و سپس پس از اتمام ناحیه critical و release تابع popcli صدا می‌شود تا وقفه‌ها دوباره فعال شوند.
در ساختار pushcli و popcli از cli و sti استفاده شده اما نکته مهم این است که فقط توابعی بر روی این‌ها نیستند و قابلیت‌های دیگری هم دارند. تفاوت این است که pushcli و popcli قابلیت شمارش هم دارند، یعنی مشخص است که هر کدام چقدر اجرا شده‌اند که می‌تواند در کنترل کردن کمک کند.

۲. در acquiresleep یک پردازنده روی آدرس قفلی که به آن پاس داده شده، sleep می‌کند تا موقعی که فرصتی برای به درست گرفتن قفل مورد نظر پیدا کند و در نتیجه busy waiting نداریم. در releasesleep، ریسه‌ای که sleeplock را نگه‌داشته، نام پردازه‌هایی که روی چنل قفل sleep کرده‌اند را بیدار می‌کند و وضعیت آن پردازه‌ها از sleeping به runnable تغییر می‌کند.

در مسئله producer/consumer اگر فقط از spinlock بخواهیم استفاده کنیم (از semaphore استفاده نکنیم) نمی‌توانیم تضمین کنیم که بلافاصله بعد از آزاد شدن قفل توسط مصرف کننده، قفل به مصرف کننده برسد. یعنی ممکن است شرایطی پیش بیاید که buffer خالی است اما تولید کننده قفل را برای مدتی در دست نمی‌گیرد.

۳. در سیستم‌عامل xv۶، پردازه‌ها به صورت چرخه‌ای اجرا می‌شوند و هر پردازه در یکی از حالت‌های مختلف قرار می‌گیرد. حالت‌های مختلف پردازه‌ها در xv۶ عبارتند از:

- Running: پردازه در حال حاضر در حال اجرا در پردازنده است.
- Runnable: پردازه آماده به اجرا است، در صف اجرا قرار دارد و منتظر اجرا توسط پردازنده است.
- Sleeping: منابع مورد نیاز پردازه تامین نشده است. به عبارتی scheduler از این پردازه استفاده نمی‌کند.
- Zombie: پردازه‌ای که اجرایش به پایان رسیده است، اما هنوز پردازه پدر آن wait را صدا نکرده است و اطلاعات این پردازه هنوز در ptable موجود است.
- Unused: پردازه‌ای که هنوز در حالت استفاده قرار نگرفته است یا قبلاً استفاده شده ولی حالا از دسترس خارج شده است.
- Embryo: وقتی allocproc صدا زده شد، پردازه ای که unused بوده است، به این استیت تغییر می کند یعنی پردازه می تواند ادامه مراحل را طی کند تا به پایان برسد.

تابع sched در سیستم‌عامل xv٦ وظیفه تعیین و تنظیم پردازه‌ای را دارد که باید در حالت running قرار بگیرد. این تابع بر اساس scheduler مشخص شده، پردازه‌ای را انتخاب می‌کند که باید در اجرا قرار بگیرد و برنامه‌اش ادامه یابد. از جمله وظایف تابع sched می‌توان به ایجاد و بازسازی محیط پردازه (context)، تغییر حالت پردازه و تعیین ترتیب اجرای پردازه‌ها اشاره کرد.

٤. لینک زیر تعریف mutex در لینوکس را نشان می دهد:

https://github.com/torvalds/linux/blob/master/include/linux/mutex.h

و این استراکت به شکل زیر تعریف شده است:

```
63  struct mutex {
64      atomic_long_t       owner;
65      raw_spinlock_t      wait_lock;
66  #ifdef CONFIG_MUTEX_SPIN_ON_OWNER
67      struct optimistic_spin_queue osq; /* Spinner MCS lock */
68  #endif
69      struct list_head    wait_list;
70  #ifdef CONFIG_DEBUG_MUTEXES
71      void            *magic;
72  #endif
73  #ifdef CONFIG_DEBUG_LOCK_ALLOC
74      struct lockdep_map  dep_map;
75  #endif
76  };
```

در لینوکس mutex به این صورت است که در ساختار آن owner تعریف شده است که مشخص کند کدام پردازه قفل را در اختیار دارد. این قفل ویژگی‌های زیر را دارد:

- تنها نگهدارنده قفل می تواند آن را رها کند.
- رها سازی قفل‌های متعدد در یک زمان مجاز نیست.
- انتظار مشغول شده نداریم.

دراستراکت sleeplock که در فایل sleeplock.c قرار دارد، متغیر pid را داریم که هدف اصلی ایجاد آن برای دیباگ کردن بوده است. از همین متغیر برای چک کردن صاحب قفل درreleasesleep استفاده می‌کنیم. قطعه کد زیر تابع تغییر یافته است که قابلیت خواسته شده در آن ایجاد شده است:

```
34  void
35  releasesleep(struct sleeplock *lk)
36  {
37    acquire(&lk->lk);
38
39    if (lk->pid != myproc()->pid) {
40      release(&lk->lk);
41      return;
42    }
43
44    lk->locked = 0;
45    lk->pid = 0;
46    wakeup(lk);
47    release(&lk->lk);
48  }
```

۵. حافظه تراکنشی (Transactional Memory) دنباله‌ای از عملیات‌های خواندن و نوشتن در حافظه است، که به صورت atomic انجام می شوند. اگر تمام عملیات‌ها در تراکنش با موفقیت به اتمام برسند، تراکنش حافظه‌ای ثبت می شود (commit)؛ در غیر اینصورت عملیات متوقف و برگشت داده می‌شود(rolled back). این روش کار برنامه‌نویسان را برای همگام سازی راحت می‌کند، زیرا کافی است که مثلا ساختاری مانند {S} atomic به زبان اضافه شود و اطمینان حاصل شود که عملیات S مانند یک تراکنش اجرا شود. به این ترتیب اطمینان حاصل می شود که ناحیه بحرانی محافظت می شود.

فواید Transactional Memory به شرح زیر است:

- وظیفه atomic کردن عملیات‌ها از عهده برنامه‌نویس خارج می‌شود.
- چون از قفل استفاده نمی‌شود، بنابراین deadlock هم نخواهیم داشت.

- با افزایش ریسه‌ها، مدیریت همگام‌سازی با استفاده از قفل‌های سنتی مانند mutex و semaphore دشوار خواهد شد، زیرا سربار و اختلاف ریسه‌ها برای نگهداری قفل (lock ownership) بسیار زیاد خواهد شد.