آزمایشگاه سیستم عامل

پروژه شماره 4

علی پادیاو – اولدوز نیساری – کسری حاجی حیدری

بهار 1402

Repository Link: https://github.com/alumpish/OS-Lab-Projects

Latest Commit Hash: bb3f536ad113fabc63997372299ced56a1ba389d

1- علت غیر فعال کردن وقفه این است که بتوانیم کدهایی در ناحیه بحرانی هستند را بدون اختلال و وقفه و به صورت پیوسته اجرا کنیم تا مشکلی در رابطه با متغیرهای مشترکی در این ناحیه وجود دارند پیش نیاید و اجرا را به صورت atomic داشته باشیم.
تابع pushcli به منظور غیر فعال کردن وقفه‌ها قبل از ورود به ناحیه بحرانی استفاده می‌شود و تابع popcli برای فعال کردن مجدد وقفه‌ها پس از عبور از ناحیه بحرانی استفاده می‌شود.

تفاوت این توابع با cli و sti این است که در cli و sti در حالتی که چند قفل داشته باشیم با تغییر دادن یک قفل هم وقفه‌ها از فعال و غیر فعال تغییییر وضعیت می‌دهند در حالتی که توابع pushcli و popcli تنها در زمانی که قفل‌ها آزاد باشند وقفه‌ها فعال می‌شوند.
هم چنین از جمله دیگر تفاوت‌ها این است که داخل توابع pushcli و popcli توابع cli و sti فراخوانی شده‌اند و اگر به بدنه توابع دقت کنیم متوجه می‌شویم قابلیت مدیریت بخش‌های کد و تعیین دفعات اجرا در این حالت وجود دارد. (در pushcli به یک تعداد معین می‌توان وقف‌ها را غیر فعل کرد که آن تعداد با ncli تعیین می‌شود)

```
void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}

```

```
static inline void
cli(void)
{
  asm volatile("cli");
}

static inline void
sti(void)
{
  asm volatile("sti");
}
```

2- تابع acquiresleep آدرس قفل را به عنوان ورودی دریافت می‌کند و سپس در داخل بدنه تابع تا زمانی که شرایط برای به اختیار گرفتن تابع به آن داده نشده است دستور sleep اجرا می‌شود.

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if (p == 0)
    panic("sleep");

  if (lk == 0)
    panic("sleep without lk");

  if (lk != &ptable.lock)
  {                        // DOC: sleeplock0
    acquire(&ptable.lock); // DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if (lk != &ptable.lock)
  { // DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

به تابع releasesleep هم آدرس قفل به عنوان ورودی داده می‌شود. در این تابع پردازه ای که پیش تر قفل شده بود ()wakeup را صدا می‌زند. با این کار پردازه‌هایی که روی آن قفل خاص در وضعیت sleep هستند بیدار می‌شوند و در وضعیت RUNNABLE قرار می‌گیرند.

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);

  if (lk->pid != myproc()->pid) {
    release(&lk->lk);
    return;
  }

  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

در مثال تولید کننده / مصرف کننده نمی‌توانیم از قفل‌های چرخشی استفاده کنیم چون شرطbounded waiting را ندارد و ممکن است تولید کننده بلافاصله بعد از آزاد شدن قفل خودش دوباره آن را در دست بگیرد و چون این کار محدودیت و کرانی ندارد، ممکن است قفل هیچ وقت به مصرف کننده نرسد و وارد ناحیه بحرانی نشود.

-3

اگر در فایل‌ها جست و جو کنیم متوجه می‌شویم که در فایل proc.h وضعیت پردازه‌ها را به صورت زیر نوشته است:

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

که حال به تشریح هر یک می‌پردازیم:

1) UNUSED: در این حالت استفاده ای از پردازه نمی‌شود.
2) EMBRYO: وضعیتی است که پردازه ای که تازه ایجاد شده وارد آن می‌شود. در حقیقت وقتی از حالت unused تغییر حالت می‌دهیم به این وضعیت می‌رسیم.
3) SLEEPING: وضعیتی است که وقتی پردازه به منابعی نیاز دارد که برای آن قابل دسترسی نیست ، پردازه در این وضعیت قرار می‌گیرد. این آماده نبودن به دلیل در اختیار داشتن منابع توسط پردازه یا عملیات I/O یا ... است.
4) RUNNABLE: حالتی است که پردازه آماده اجرا است و آماده است که پردازنده به آن اختصاص داده شود.
5) RUNNING: پردازه در حال اجرا است و پردازنده به آن اختصاص داده شده است.
6) ZOMBIE: حالتی است که کار پردازه به اتمام رسیده باشد اما والدش wait را صدا نکرده باشد پردازه در این حالت قرار دارد. در این حالت با وجود پایان پردازه اطلاعات آن هنوز در ptable باقی مانده است.

وظیفه تابع ()sched:

هنگامی‌که اختصاص پردازنده به یک پردازه تمام می‌شود ، این تابع صدا زده می‌شود . در این تابع با ذخیره سازی context فعلی و جایگزین شدن context مربوط به scheduler عملیات context switch انجام می‌شود. هم چنین از آنجایی که با پایان یافتن هر پردازه باید lock مربوط به ptable توسط آن در اختیار گرفته شود و سپس با رها کردن قفل‌های دیگر از وضعیت RUNNING خارج شود تابع sched شرایط مربوط به این عملیات را بررسی می‌کند و swtch را صدا می‌زند.

```
1  void sched(void)
2  {
3    int intena;
4    struct proc *p = myproc();
5
6    if (!holding(&ptable.lock))
7      panic("sched ptable.lock");
8    if (mycpu()->ncli != 1)
9      panic("sched locks");
10   if (p->state == RUNNING)
11     panic("sched running");
12   if (readeflags() & FL_IF)
13     panic("sched interruptible");
14   intena = mycpu()->intena;
15   swtch(&p->context, mycpu()->scheduler);
16   mycpu()->intena = intena;
17 }
```

4-به منظور دیباگ کردن داخل استراکت sleeplock یک مغتیر pid قرار دارد و برای چک کردن صاحب قفل در تابع releasesleep میتوان از این متغیر استفاده کرد.بدین منظور تابع جدید به شکل زیر می‌باشد:

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);

  if (lk->pid != myproc()->pid) {
    release(&lk->lk);
    return;
  }

  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

معادل این قفل در هسته لینوکس، mutex میباشد. در لحظه فقط یک تسک میتواند این قفل را نگه دارد. همچنین برای آزادسازی آن در تعریف استراکت mutex یک فیلد به نام owner ایجاد شده است و فقط صاحب قفل قادر به آزاد سازی آن می‌باشد. (تعریف این استراکت در فایل mutex.h قرار دارد)

```
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
};
```

5- ایده و کانسپت transactional memory از حوزه database به وجود آمده و از آن در همگام سازی نیز استفاده میشود. memory transaction توالی ای از عملیات‌های خواندن و نوشتن حافظه است که به صورت atomic صورت می‌گیرد.

اگر تمام عملیات کامل صورت بگیرند تغییرات صورت میگیرد و در غیر اینصورت عملیات متوقف میشود و تغییری صورت نمیگیرد و به عقب برمیگردیم. (Lock elision)

در روش‌های عادی همگام‌سازی (mutex, semaphores) احتمال بروز خطاهای از جمله deadlock بالا است همچنین اگر تعداد پردازنده‌ها افرایش بیابد، جنگ بر سر بدست آوردن قفل نیز افزایش می‌یابد و همین باعث کند شدن عملیات می‌شود.

مزیت این روش نسبت به قفل‌ها این است که در این روش توسعه دهنده مسئولیتی در قبال atomic بودن ندارد و همچنین چون قفل نداریم امکان ایجاد deadlock نیز از بین میرود.همچنین این سیستم تشخیص میدهد که کدام گزاره‌ها در یک atomic block میتوانند به صورت موازی اجرا شوند که تشخیص چنین موضوعی برای برنامه نویس کار پیچیده ای می‌باشد.

Transactional memory میتواند به صورت نرم افزاری یا سخت افزاری پیاده سازی شود.

Software transactional memory (STM) توسط کامپایلر کدهایی را در بلاک مد نظر اضافه میکند.

Hardware transactional memory (HTM) در سطح سخت افزار پیاده سازی میشود و از آنجایی که سربار STM را ندارد از آن سریعتر است.

Transactional synchronization extension (TSX) یک افرونه برای x86 ISA میباشد که قابلیت HTM را اضافه میکند و با استفاده از lock elision باعث افزایش سرعت اجرای برنامه‌های مولتی‌ترد میشود. (تا 40% افزایش سرعت) این افزونه علت مشکلات امنیتی در اکثر پردازنده‌های اینتل غیر فعال شده است.

تضمینی برای موفقیت آمیز بودن HTM وجود ندارد (البته در اکثر موارد معقول موفقیت آمیز است) به همین دلیل باید مسیر بازگشتی وجود داشته باشد. این مسیر بازگشت به وسیله lock elision ایجاد می‌شود. بدین صورت که اگر عملیات موفقیت آمیز بود آن را انجام میدهد و اگر نبود به عقب باز می‌گردد.

شبیه سازی مسئله readers priority readers-writers

پیاده سازی سمافور

استراکت سمافور ردا در فایل semaphore.h پیاده سازی کردیم:

```
struct semaphore {
  int value;                  // The current value of the semaphore.
  struct spinlock lk;         // Spinlock protecting the semaphore.

  struct proc* waiting[NPROC]; // Queue of processes waiting on the semaphore.
  struct proc* holding[NPROC]; // List of processes holding the semaphore.
  int wfirst;                 // The head of the waiting queue.
  int wlast;                  // The tail of the waiting queue.

  char* name;                 // Name of the semaphore.
};
```

صف waiting برای این است که پردازه‌ها اگر منتظرند، به ترتیب زمان ورودشان، از صف خارج شوند.

سیستم‌کال‌ها

1) sem_init(i, v)

```
void
sem_init(int id, int value, const char* name)
{
  semaphore_init(&sems[id], value, name);
}
```

```
void
semaphore_init(struct semaphore* sem, int value, char* name)
{
  sem->value = value;
  sem->wfirst = 0;
  sem->wlast = 0;
  initlock(&sem->lk, "semaphore");
  memset(sem->waiting, 0, sizeof(sem->waiting));
  memset(sem->holding, 0, sizeof(sem->holding));
  sem->name = name;
}
```

این سیستم‌کال به آن سمافوری در آرایه که میخواهیم، مقادیر اولیه را می‌دهد.

2) sem_aquire(i)

```
1  void
2  sem_acquire(int id)
3  {
4    semaphore_acquire(&sems[id]);
5  }
```

```
1  void
2  semaphore_acquire(struct semaphore* sem)
3  {
4    acquire(&sem->lk);
5    --sem->value;
6    // cprintf("%s %d\n", sem->name, sem->value);
7    if(sem->value < 0){
8      sem->waiting[sem->wlast] = myproc();
9      sem->wlast = (sem->wlast + 1) % NELEM(sem->waiting);
10     sleep(sem, &sem->lk);
11   }
12   struct proc* p = myproc();
13   for(int i = 0; i < NELEM(sem->holding); ++i){
14     if(sem->holding[i] == 0){
15       sem->holding[i] = p;
16       break;
17     }
18   }
19   release(&sem->lk);
20 }
```

در این سیستم کال اگر مقدار سمافور 0 یا کمتر باشد، به صف انتظار اضافه می‌شود. همینطور پردازه کنونی به عنوان نگهدارنده این سمافور ست می‌شود. در ابتدای این تابع از اسپین‌لاک استفاده میشود تا اطلاعات استراکت سمافور حفظ شود.

3) sem_release(i)

```
1  void
2  sem_release(int id)
3  {
4    semaphore_release(&sems[id]);
5  }
```

```
void
semaphore_release(struct semaphore* sem)
{
  acquire(&sem->lk);
  ++sem->value;
  if(sem->value <= 0){
    wakeupproc(sem->waiting[sem->wfirst]);
    sem->waiting[sem->wfirst] = 0;
    sem->wfirst = (sem->wfirst + 1) % NELEM(sem->waiting);
  }
  struct proc* p = myproc();
  for(int i = 0; i < NELEM(sem->holding); ++i){
    if(sem->holding[i] == p){
      sem->holding[i] = 0;
      break;
    }
  }
  release(&sem->lk);
}
```

در این سیستم‌کال اگر مقدار سمافوری 1- باشد، آن را از حالت sleep درمی‌آورد و همینطور سمافورهایی که پردازه کنونی نگهدارنده‌اش بودند، به حالت قبل برمی‌گردند. از اسپین‌لاک در اینجا هم استفاده می‌شود.

برای تعریف آرایه سمافورها، چون xv6 ترد ندارد، باید آن را در کرنل تعریف کنیم تا همه پردازه‌ها به آن دسترستی داشته باشند.در فایل semaphore.c :

```
struct semaphore sems[NSEM];
```

برنامه سطح کاربر

برای این برنامه نیاز به متغیر مشترک readcount بین پردازه‌های reader و writer داریم. از آنجایی که xv6 مولتی‌ترد نیست، نمیتوان از shmget استفاده کرد. برای همین متغیر my_variable را در فایل sysproc.c به صورت گلوبال تعریف کردیم. این متغیر با استفاده از سیستم‌کال setvar ست می‌شود. با سیستم‌کال modvar کاهش یا افزایش می‌یابد و با getvar مقدارش دریافت می‌شود.

در برنامه سطح کاربر ابتدا سمافورها را initial می‌کنیم.

```
void init_sems()
{
    sem_init(WRT, 1, "wrt");
    sem_init(MUTEX, 1, "mtx");
    sem_init(PRINT_MUTEX, 1, "prmtx");
}
```

سپس 3 پردازه reader و 2 پردازه writer می‌سازیم:

```
void start()
{
    for (int i = 0; i < NREADERS; i++)
    {
        if (fork() == 0)
        {
            reader(i);
            exit();
        }
    }

    for (int i = 0; i < NWRITERS; i++)
    {
        if (fork() == 0)
        {
            writer(i);
            exit();
        }
    }

    for (int i = 0; i < NREADERS + NWRITERS; i++)
        wait();
}
```

پردازه reader:

```
void reader(int id)
{
    int i = 5;
    while (i--)
    {
        ATOMIC(printf(1, "Reader %d wants to access readcount\n", id))
        sem_acquire(MUTEX);
        modvar(1);
        ATOMIC(printf(1, "Reader %d increased readcount to %d\n", id, getvar()))
        if (getvar() == 1)
        {
            ATOMIC(printf(1, "Reader %d wants to get WRT\n", id))
            sem_acquire(WRT);
        }
        sem_release(MUTEX);

        ATOMIC(printf(1, "Reader %d read\n", id))

        sem_acquire(MUTEX);
        modvar(-1);
        ATOMIC(printf(1, "Reader %d decreased readcount to %d\n", id, getvar()))

        if (getvar() == 0)
        {
            ATOMIC(printf(1, "Reader %d released WRT\n", id))
            sem_release(WRT);
        }
        sem_release(MUTEX);

        sleep(10);
    }
}
```

در اینجا پس به دست گرفتن mutex با modvar(1) متغیر مشترکمان را یکی زیاد می‌کنیم و پس از انجام خواندن با modvar(-1) یکی کم می‌کنیم.

پردازه writer:

```
void writer(int id)
{
    int i = 5;
    while (i--)
    {
        ATOMIC(printf(1, "Writer %d wants to get WRT\n", id))
        sem_acquire(WRT);
        ATOMIC(printf(1, "Writer %d wrote\n", id))
        sem_release(WRT);

        sleep(10);
    }
}
```

پس از اجرای برنامه سطح کاربر، خروجی زیر قابل مشاهده است:

```
$ readers_writers
Reader 0 wants to access readcount
Reader 0 increased readcount to 1
Reader 0 wants to get WRT
Reader 1 wants to access readcount
Reader 2 wants to access readcount
Reader 0 read
Writer 1 wants to get WRT
Writer 0 wants to get WRT
Reader 1 increased readcount to 2
Reader 1 read
Reader 2 increased readcount to 3
Reader 2 read
Reader 0 decreased readcount to 2
Reader 1 decreased readcount to 1
Reader 2 decreased readcount to 0
Reader 2 released WRT
Writer 1 wrote
Writer 0 wrote
Reader 0 wants to access readcount
Reader 1 wants to access readcount
Reader 0 increased readcount to 1
Reader 2 wants to access readcount
Reader 0 wants to get WRT
Writer 1 wants to get WRT
Writer 0 wants to get WRT
Reader 0 read
Reader 1 increased readcount to 2
Reader 1 read
Reader 2 increased readcount to 3
Reader 2 read
Reader 0 decreased readcount to 2
Reader 1 decreased readcount to 1
Reader 2 decreased readcount to 0
Reader 2 released WRT
Writer 1 wrote
Writer 0 wrote
Reader 0 wants to access readcount
Reader 0 increased readcount to 1
Reader 1 wants to access readcount
Reader 0 wants to get WRT
```

مشخص است که پردازه‌های reader به کمک mutex همزمان به readcount (در اینجا my_variable) دسترسی ندارند. همینطور تا زمانی که readcount 0 نشده writer-ها به wrt دست پیدا نمی‌کنند.