به نام خدا

گزارش کار پروژه چهارم آزمایشگاه سیستم عامل

" همگام سازی"

گروه 5 :

- سینا طبسی

810199554

- سید حامد میرامیرخانی

810199500

- فاطمه محمدی

810199489

❖ بخش اول: سوالات

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

در پردازنده های مدرن، در مواردی نیاز است که از preemption ها جلوگیری شود، در این موارد بایستی وقفه ها را غیرفعال کنیم و یا چون وقفه ها بالاترین اولویت را دارند، در هر لحظه ممکن است که کد هسته متوقف شود تا interrupt handler مربوط به وقفه ایجاد شده اجرا شود در این موارد به منظور محافظت از ناحیه بحرانی و یا جلوگیری از deadlock نیاز است وقفه ها غیر فعال شوند.

به منظور فعال غیر فعال کردن وقفه ها از توابع cli و sti که به ترتیب برای غیرفعال و فعال کردن وقفه ها استفاده میشوند، استفاده میکنیم؛ به عنوان مثال در زمان Spin Lock.

توابع pushcli و popcli همانند مشابه cliو sti میباشند، درواقع میتوان انها را همچون یک wrapper برای توابع cli و sti دانست. در استفاده از pushcli و popcli می توان فرض کرد که مدیریت فعال و یا غیر فعال کردن وقفه ها با استفاده از یک استک(stack)انجام میشود به این صورت که تا زمانی که استک خالی باشد، وقفه ها فعالند و در غیر این صورت وقفه ها غیر فعالند.

توابع pushcli و popcli در واقع به ترتیب برای غیرفعال کردن و فعال کردن وقفه ها استفاده میشود با این تفاوت از cli و sti که اگر فرضا cli، دو یا بیشتر بار (مثلا 5 بار) صدا زده شود، با تنها یک بار فراخوانی sti، وقفه ها فعال میشوند، اما در puscli و popcli برای فعال سازی مجدد وقفه ها پس از فراخوانی 5 بار pushcli، باید حتما 5 بار popcli فراخوانی شود.

در واقع در xv6، برای مدیریت ناحیه های بحرانی تودرتو از این دو تابع pushcli و popcli استفاده میشود.

توابع pushcli و popcli از همان توابع cli و sti استفاده میکنند اما علاوه بر توانی فعال و غیرفعال کردن وقفه ها، قابلیت های بیشتری نیز دارند :

1- چاپ کردن خطاهایی که رخ میدهند.

2- مدیریت نواحی بحرانی تودرتو با استفاده از متغیر های intena , ncli در داده ساختار وضعیت پردازنده .

3- اطمینان حاصل کردن از اینکه تمامی قفل ها آزاد شوند و سپس وقفه ها غیرفعال شوند.

2. **مختصری راجع به تعامل میان پردازه ها توسط دو تابع مذکور توضیح دهید. چرا در مثال تولیدکننده/مصرف کننده استفاده از قفل های چرخشی ممکن نیست.**

در روش spinlock پردازه ای که منتظر است که lock آزاد شود حلقه را تکرار میکند و موجب میشود زمان cpu صرف شود. sleeplock باعث میشود که پردازه sleep شود و تا هنگامی که نوبت آن نرسیده از زمان cpu استفاده

نمیکند. هنگامی که یک دارنده sleeplock آن را رها میکند پردازه ای که spilock od sleeplock را اول به دست آورده wakeup میکند.
همچنین بقیه پردازه ها همان مکانیزم sleeplock را به کار میبرند که ترتیب waiter ها را تضمین میکند. در spinlock شرط bounded waiting برقرار نیست. فرض کنید این حالت پیش بیاید که هنوز بازه زمانی پردازه قبل تمام نشده و دوباره همان قبلی بیاید و lock را اشغال کند و مجددا احتمال دارد این اتفاق تکرار شود. به این دلیل که هیچ کرانی برای اینکه پردازه رقیب چند بار وارد این حلقه می شود وجود ندارد پس راه حل خوبی برای مسئله producer/consumer نیست. امکان دارد producer در حلقه while بماند و اجازه ندهد consumer وارد critical section شود و این مشکل اساسی است.

3. حالات مختلف پردازه ها در xv6 را توضیح دهید. تابع sched چه وظیفه ای دارد؟

با توجه به عکس زیر میتوان گفت هر پردازه ها در xv6 حالت(استیت) های زیر را میتوان داشته باشد:

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

1) UNUSED
نبود پردازه - استیت استفاده نشده.
در واقع همانطور که میدانیم پردازه ها در یک لیست نگهداری می شوند و اگر در خانه ای پردازه ای قرار نداشته باشد با این حالت نشان داده میشود.

2) EMBRYO
تازه متولد شده - تازه ایجاد شده.
زمانی که یک پردازه ایجاد میشود ابتدا استیت ان به حالت EMBRYO قرار میگیرد.
(در واقع همانطور که در شکلی که در ادامه آمده است نشان داده شده است، وقتی allocproc صدا زده پردازه ای UNUSED بود است به این استیت تغییر حالت میدهد)

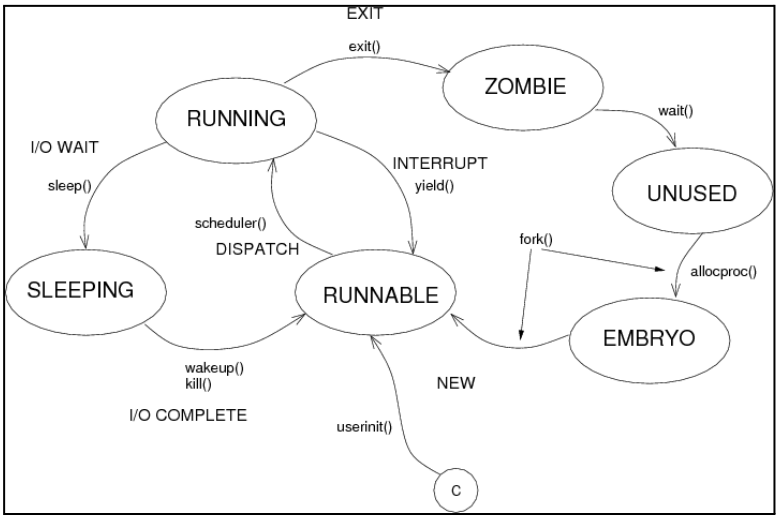
3) SLEEPING
در حالت خواب.
زمانی که به پردازه در صف اجرای scheduler ، پردازنده تخصیص داده نمیشود و بدون فعالیت میماند به این حالت در می آید. برای مثال زمانی که پردازه در انتظار دسترسی به یک منبع بماند.(منابع مورد نیاز پردازنده تامین نشده است)

4) RUNNABLE
قابل اجرا.
زمانی که پردازه در صف اجرا scheduler است و آماده اجرا است (در انتظار پردازنده قرار میگیرد).
برخلاف حالت SLEEPING برای پردازه در این حالت همه منابع مورد نیاز پردازه در اختیار قرار گرفته است.

5) RUNNING
در حال اجرا.
این حالت به پردازه، پردازنده اختصاص داده شده است و درحال اجرا توسط پردازنده میباشد.

6) ZOMBIE
حالت زامبی.
زمانی که پردازه کارش تمام میشود قبل از اینکه به حالت UNUSED برود به حالت زامبی میرود تا پردازنده پدر آن بتواند از اتمام کار پردازه فرزند (با استفاده از تابع wait) آگاه شود.(کار پردازه تمام شده است اما هنور اطلاعات ان در ptable موجود است)

در ادامه شمای کلی از حالات پردازه و چرخه تغییرات آن امده است:



- وظیفه تابع sched:

این تابع در هر زمان که نیاز به زمانبندی باشد استفاده می شود برای مثال در تابع exit و yield و همینطور این تابع هنگام زمانبندی پردازه جدید در ابتدا صدا زده میشود و پس از بررسی خطاهای که ممکن از رخ بدهند از جمله وقفه ها فعال باشند، پردازه در حال اجرا باشد، قفل ptable گرفته نشده باشد و ... در صورت عدم وجود خطا، عملیا context switch را بین پردازه های فعلی و پردازه حاضر در scheduler پردازنده را انجام میدهد.

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

4. 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.

ساختار sleeplock در خود مقدار عددی به نام pid دارد که شناسه پردازه نه دارنده را در خود ذخیره میکند. هنگام صدا زدن تابع aquiresleep این شناسه در lk->pid ذخیره میشود. در شکل پایین در تابع releasesleep با افزودن شرط نشان داده شده موجب میشود تنها در صورتی که پردازنده فعلی که در پردازنده در حال اجراست و releasesleep را فراخوانی کرده است اگر شناسه یکتایی با نگهدارنده sleeplock داشته باشد علیات آزادسازی قفل انجام میشود. قفل های mutex در لینوکس کارکرد مشابهی دارند به شکلی که وظیفه ای که میخواهد وارد critical section شود باید ابتدا mutex_lock و هنگام خروج تابع mutex_unlock را فراخوانی کند. در صورتی که

mutex lock قابل دسترسی نبود و توسط تسک دیگری گرفته شده بود. تسک فعلی به وضعیت sleep وارد میشود mutex lock توسط نگهدارنده آن آزاد میشود. لینوکس قفل های semaphore را که براساس توضیحات قبل با وارد کردن تسک ها به وضعیت sleep کار میکنند را نیز دارد.

```
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
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

5. یکی از روش های افزایش کارایی در بارهای کاری چند ریسه ای استفاده از حافظه تراکنشی بوده که در کتاب نیز به آن اشاره شده است. به عنوان مثال این فناوری در پردازنده های جدیدتر اینتل تحت عنوان افزونه های همگام سازی تراکنشی (TSX) پشتیبانی میشود. آن را مختصرا شرح داده و نقش حذف قفل را در آن بیان کنید.

مفهوم حافظه تراکنشی در دیتابیس ها مطرح میشود و با الهام از آن میتوان همگام سازی را در سطح نرم افزار و یا سخت افزار انجام داد درواقع یک مدل جایگزین برای lock ها میباشد که جهت کنترل دسترسی به حافظه به صورت همروند در برنامه نویسی موازی استفاده میشود و ایده حافظه تراکنشی با استفاده از تراکنش های حافظه است.

روند کار

** تراکنش حافظه در واقع دنباله ای از عملیات های خواندن/نوشتن در حافظه میباشد و به صورت پشت سرهم و اتمی انجام میشود؛ و در صورتی تمام عملیاتها با موفقیت انجام شود، تراکنش حافظه ای ثبت می شود و در غیر این صورت متوقف میشود و بازگشت میخورد.

همانطور که اشاره شد این مدل جایگزینی برای lock ها میباشد و

از جمله مزایای استفاده از این فناوری می توان به موارد زیر اشاره کرد:

- با توجه به اینکه جایگزین قفل شده و دیگر از قفل استفاده نمیشود، deadlock یا همان بن بست نخواهیم داشت.
- وظیفه atomic (اتمی) کردن عملیات ها دیگر به عده برنامه نویس نخواهد بود.
- همگام سازی، با افزایش ریسه ها با استفاده از قفل های ستی دشوارتر است و سربار برای نگهداری قفل در استفاده از قفل های سنتی بسیار زیاد است.

- در روش های عادی قفل کردن، وقتی رقابت (contention) زیاد میشود علاوه بر موارد deadlock که اشاره شد موجب کند شدن نیز میشود.

❖ بخش دوم: پیاده سازی Semaphore

```
struct semaphore {
  int value;
  struct spinlock lk;

  struct proc* waiting[NPROC];
  struct proc* holding[NPROC];
  int wait_first;
  int wait_last;
};
```

استراکت semaphore که در عکس بالا قابل مشاهده است از چند فیلد تشکیل شده است که توضیحی مختصر در مورد آن ها می دهیم:

value: مقداری برای semaphor ما می باشد که در زمان لاک شدن مقدار -1 و با آزاد شدن آن مقدار ۱ را می گیرد.

lk: یک spinlock می باشد.

waiting: صفی دایره ای می باشد. این صف حاوی پردازه ها که در استیت انتظار برای آزاد شدن semaphore و وارد شدن به بخش critical خود می باشد هست.

holding: صفی از پردازه ها که semaphore را acquire کرده اند.

wait_first: متغیری است که اول صف waiting را نشان می دهد.

wait_last: متغیری است که انتها صف waiting را نشان می دهد.

سه سیستم کال خواسته شده در ادامه آورده شده است و این سه سیستم کال در فایل semaphore.c آورده شده است:

```
void
semaphore_init(struct semaphore* sem, int value)
{

  initlock(&sem->lk, "semaphore");
  memset(sem->waiting, 0, sizeof(sem->waiting));
  memset(sem->holding, 0, sizeof(sem->holding));
  sem->value = value;
  sem->wait_first = 0;
  sem->wait_last = 0;
}
```

این سیستم کال عملیات init سمافور را انجام می دهد. تابع استفاده شده initlock در این سیستم کال در فایل spinlock.c قرار دارد که عملیات init برای لاک را انجام می دهد.

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

در قسمت semaphore acquire ابتدا استراکچر spinlock گرفته می شود و مقدار value یک واحد کم شده. حال اگر مقدار منفی نباشد به این معنی است که سمافور می تواند اجازه ورود به critical را بدهد. بنابراین پردازه در اولین قسمت لیست holding قرار می گیرد. حال اگر این مقدار منفی باشد یعنی اجازه ورود صادر نمی شود و باید تا زمانی که نوبت به آن برسد به sleep برود. بنابراین این پردازه در صف waiting قرار می گیرد.

```
void
semaphore_release(struct semaphore* sem)
{
  acquire(&sem->lk);
  ++sem->value;
  if(sem->value <= 0){
    wakeupproc(sem->waiting[sem->wait_first]);
    sem->waiting[sem->wait_first] = 0;
    sem->wait_first = (sem->wait_first + 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 قسمت semaphore release ابتدا استراکچر spinlock گرفته می شود و مقدار value یک واحد زیاد می شود. اگر مقدار value صفر بود و یا از صفر کمتر بود به این معنی است که سمافوری وجود دارد که در انتظار گرفتن semaphore است و باید آن را wakeup کرد. پس از آن از لیست holding حذف می شود.

از آنجا که برای wakup کردن سمافور کل پردازه های sleep می شوند بنابراین ما تابعی در proc.c زدیم که پردازه مشخصی را wakeup می کند. کد آن در زیر آمده است:

```
void
wakeupproc(struct proc* p)
{
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
}
```

در نهایت ۳ reader و ۲ writer را با استفاده از سه سیستم کال یاد سده پیاده سازی می کنیم. اندیس i موجود در این سه سیستم کال اندیس سمافور کرنل می باشد.

❖ شبیه سازی Reader-writer problem

در این بخش ما از راه حل گفته شده در کتاب استفاده کردیم که این راه حل مشکل deadlock ندارد. و همچنین با توجه به استفاده صف در برای لیست waiting دیگر مشکل starvation رخ نمی دهد. از آنجا که فرآیند print در این برنامه به صورت atomic نمی باشد بنابراین یک سمافور دیگر به برنامه اضافه کرده و آن را به print اختصاص می دهیم و آن را mutex می نامیم.
در ادامه نمونه ای از اجرای کد آورده شده است:

```
Group #5 Members:
1- Fatemeh Mohammadi
2- Sina Tabasi
3- Hamed Miramirkhani
$ rwtest
Reader PID 4: behind while
Reader PID 4: passed while
Reader PID 5: behind while
Reader PID 5: passed while
Reader PID 4: reading started
Reader PID 6: behind while
Reader PID 6: passed while
Reader PID 5: reading started
Reader PID 6: reading started
Reader PID 4: reading done
Reader PID 4: exiting
Writing PID 7: behind while
Reader PID 5: reading done
Reader PID 6: reading done
Reader PID 5: exiting
Reader PID 6: exiting
Writing PID 7: passed while
Writing PID 7: writing started
Reader PID 4: behind while
Reader PID 5: behind while
Reader PID 6: behind while
Writing PID 8: behind while
Writing PID 7: writing done
Reader PID 4: passed while
```

همانطور که در نتیجه حاصل شده مشاهده می کنیم چند reader می توانند همزمان عمل read را انجام بدهند و همچنین در زمان write کردن یکی از writer ها باقی پردازه ها در انتظار خالی شدن سمافور می باشند.