بخش ۱,۱:

میخواستیم که امکان ایجاد ticketlock رو به سیستم عامل xv6 اضافه کنیم. برای این منظور ما در ابتدا با تقلید از spinlock دو فایل ticketlock.h و ticketlock.c را به پروژه اضافه کردیم که ticketlock.h مشابه spinlock.h دارای یک struct است که بخشی از attribute ها آن برای دیباگ کردن است اما بخش دیگری از آن برای هندل کردن ticketlock است که شامل دو attribute با نام‌های next و turn است که متغیر turn همانطور که از اسمش هم مشخص است نگهدارنده نوبت فعلی است و متغیر next مقداری را نگه میدارد که قرار است به پراسس بعدی که برای گرفتن لاک تلاش میکند داده شود.

اما لازمه پیاده سازی این مطالب همانطور که در تعریف پروژه هم گفته شد پیاده سازی سه تابع initTicketlock، acquireTicketlock و releaseTicketlock را میطلبد که این سه تابع در ticketlock.c پیاده سازی میشود. در ادامه به توضیح این سه تابع میپردازیم:

initTicketlock:

```
void initTicketLock(struct ticketlock *lk, char *name)
{
    lk->name = name;
    lk->next = 0;
    lk->turn = 0;
}
```

این تابع به این صورت عمل میکند که ticketlock را برای اولین بار مقدار دهی میکند به این صورت که مقدار next و turn را صفر قرار میدهد.

acquireTicketlock:

```
void acquireTicketLock(struct ticketlock *lk)
{
    int my_turn = fetch_and_add(&lk->next, 1);
    while (lk->turn != my_turn)
        ticketlockSleep(lk);
    // Record info about lock acquisition for debugging.
    getcallerpcs(&lk, lk->pcs);
}
```

مهم ترین تابع این بخش همین تابع است. در این تابع قصد داریم که قفل را بگیریم به این صورت که ابتدا فرایند مقدار نوبت بلیت خود را میگیرد و سپس باید به مقدار next اضافه شود ولی چون ممکن است در همین میان پردازنده از این فرایند گرفته شود و به فرایند دیگری داده شود و هر دو یک نوبت داشته باشند باید این مقدارگیری (fetch) و مقدار دهی جدید (add) به صورت اتمیک انجام شود. برای این موضوع همانطور که در صورت پروژه گفته شده بود بایستی که در ابتدا این دستور را در فایل x86.h پیاده سازی میکردیم که کردیم. بعد تا زمانی که نوبت آن فرایند نشده است فرایند در while میماند. و به حالت sleep میرود؛ هر بار که فرایند بیدار میشود بار دیگر شرط آن چک میشود چنانچه که هنوز نوبت آن نشده بود دوباره به حالت sleep میرود (به همین دلیل است که در این حالت انتظار مشغول نداریم) بعد از گرفتن قفل و خارج شدن حلقه تابعی را صدا میزنیم که در spinlock صدا زده میشد و کار آن اینست که مقادیر دیگر Struct که برای دیباگ کردن و... به کار میرود را set کند.

releaseTicketlock:

```
void releaseTicketLock(struct ticketlock *lk)
{
    fetch_and_add(&lk->turn, 1);
    wakeup(lk);
}
```

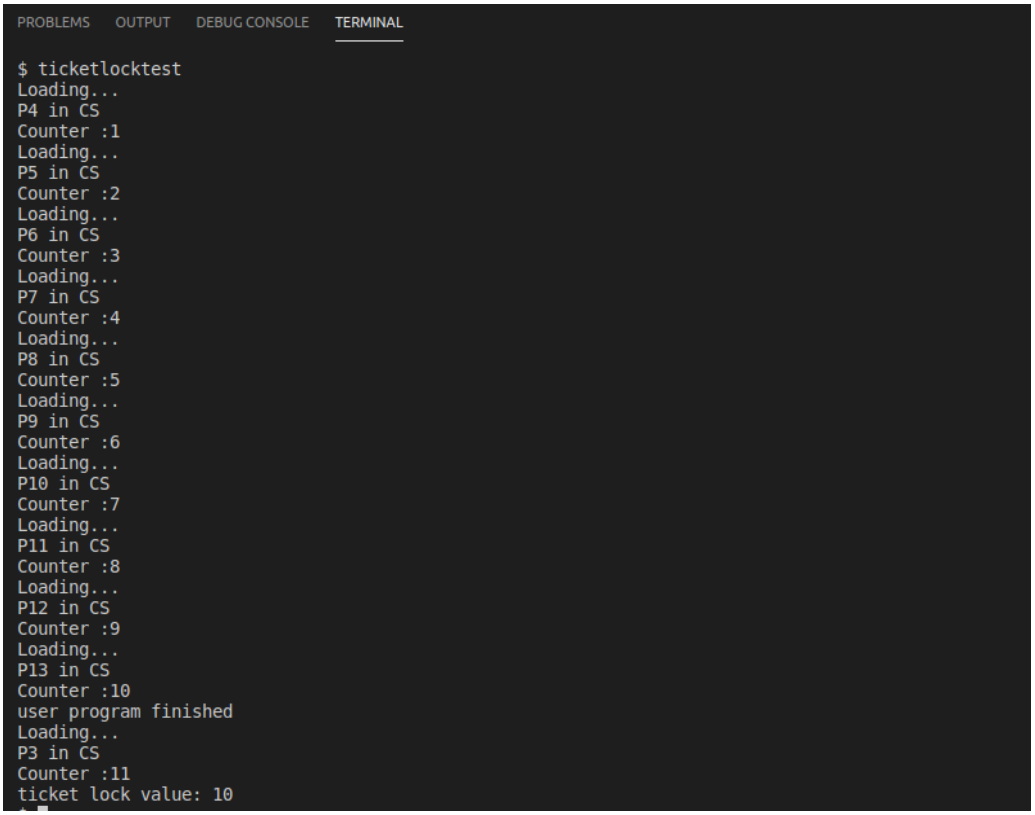
در این تابع وقتی یک فرایند میخواهد قفل را رها کند ابتدا به نوبت یا همان turn یکی اضافه میکند سپش فرایندهایی که روی آن قفل به حالت sleep رفته اند را بیدار میکند که همانطور که گفتیم اگر نوبتشان شده باشد که قفل را به دست می آورند در غیر این صورت چون در حلقه هستند دوباره شرط نوبت چک میشود و به خواب میروند.

به کمک این سه تابعی که نوشتیم میخواهیم دو سیستم کال تعریف کنیم ticketLockInit و ticketLockTest و برای تست آن از همان کد داخل تعریف پروژه استفاده کردیم.

لازم بود که برای پیاده سازی این قسمت ها (تابع acquireTicketlock) یک تابعی تعریف شود به نام ticketlockSleep که این تابع استیت فرایندها را به حالت sleep میبرد(این تابع با تقلید از تابع sleep و کدهای موجود در github پیاده سازی شده است)

بررسی خروجی برنامه تست: همانطور که میدانیم در برنامه تست فرایندهایی را میسازیم و همگی با هم تحت شرایط رقابتی تابع ticketlockTest را فراخوانی میکنند که در آن حافظه اشتراکی counter تغییر میکند به همین دلیل وجود lock ضروری میشود. زیرا اگر لاک نگذاریم ممکن است جواب از ۱۰ کوچکتر شود.

نتایج را در ادامه میبینیم:




بخش۱,۲:

میدانیم که از این lock ها میتوان مانند سمافور استفاده کرد(شبیه سمافور باینری) میخواهیم مسئله خوانندگان و نویسندگان را حل کنیم با این تبصره که خوانندگان دارای اولویت بیشتری از نویسندگان هستند. در صورت پروژه خواسته شده بود که سیستم کال‌های sys_rwinit و sys_rwtest را پیاده سازی کنیم.

sys_rwinit: برای این تابع کافیست که دو تا سمافور لازم برای مسئله را پیاده سازی کنیم.

sys_rwtest: در این سیستم کال اگر ورودی عدد صفر بود تابع خواندن را صدا میزنم و اگر یک بود تابع نوشتن (که در proc.c پیاده سازی میشود)

برای پیاده سازی توابع نوشتن و خواندن دقیقا مشابه شبه کد استاد عمل میکنیم :

```
int reading()
{
  int returnValue = 0;
  acquireTicketLock(&mutex);
```

```
  if (numberOfReader == 0)
    acquireTicketLock(&wl);
  numberOfReader += 1;
  releaseTicketLock(&mutex);
  //read data
  returnValue = sharedMemory;
  acquireTicketLock(&mutex);
  numberOfReader -= 1;
  if (numberOfReader == 0)
    releaseTicketLock(&wl);
  releaseTicketLock(&mutex);
  return returnValue;
}

int writing()
{
  acquireTicketLock(&wl);
  sharedMemory += 1;
  releaseTicketLock(&wl);
  return 0;
}
```

```
enter pattern for readers/writers test
100110
child adding to shared counter
reader read from shared counchilter: 0
child da adding to shared counter
reader read from shared counter: 0
dding tchild addchild adding to shared counter
reader read fromoing to shared counter
writer added to s shared counter
writer added to shared counter
 shared counter: 0
hared counter
user program finished
last value of shared counter: 2
```

برنامه تست به این شکل است که همان رشته توضیح داده شده در صورت سوال را میگیرد و مطابق آن توابع خواندن و نوشتن را صدا میزند. با توجه به پیاده سازی ها انجام شده مقداری که در نهایت در sharedMemory میماند بایستی برابر باشد با تعداد درخواست های نوشتن در رشته اولیه (تعداد یک ها به غیر از یک اولیه)