دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش پروژه پایانی درس سیستم‌های عامل

نگارش

محمد چمن مطلق

۹۶۳۱۰۱۸

استاد درس

دکتر طاهری جوان

بهمن ۹۸

- **مقدمه**

هدف نهایی این پروژه پیاده سازی قفل از طریق الگوریتم Ticket lock و همچنین پیاده سازی پردازش چند نخی (Threading) در سیستم عامل XV6 است. به همین منظور در دو بخش مجزا به توضیح مسائل ذکر شده می‌پردازیم

- **بخش اول: پیاده سازی Ticket lock**

در این بخش به پیاده سازی قفل بلیت داخل XV6 می‌پردازیم. این نوع قفل مشابه قفل Spin lock است که در حال حاضر درون سیستم عامل ما پیاده سازی شده است و تفاوت اصلی قفل بلیت با Spin lock در آن است که در قفل بلیت دیگر انتظار مشغول نداریم و فرایندی که در انتظار باز شدن قفل است، در حالت Sleep قرار خواهد گرفت.

به منظور پیاده سازی این نوع قفل، یک Struct به نام ticketLock با دو فیلد زیر خواهیم داشت:

- currentTicket: شماره بلیت مربوط به فرایندی که در حال حاضر در حال سرویس گرفتن است.
- nextTicket: شماره بلیت بعدی که در صورت درخواست در اختیار فرایند جدید قرار می‌گیرد.

در ادامه نیازمند سه تابع زیر هستیم:

۱. initTicketLock: یک Struct ticketLock می‌پذیرد و فیلد های آن را مقدار دهی اولیه (صفر) می‌کند.
۲. acquireTicketLock: در ابتدا یک Struct ticketLock می‌پذیرد، اگر مقدار فیلد currentTicket ساختار ورودی برابر currentTicket در حال سرویس دهی باشد، فرایند سرویس داده شود در غیر این صورت به خواب می‌رود.
۳. releaseTicketLock: یک Struct ticketLock می‌پذیرد و مقدار بلیت بعدی قابل سرویس دهی (در صورت وجود) را از حالت خواب بیدار می‌کند.

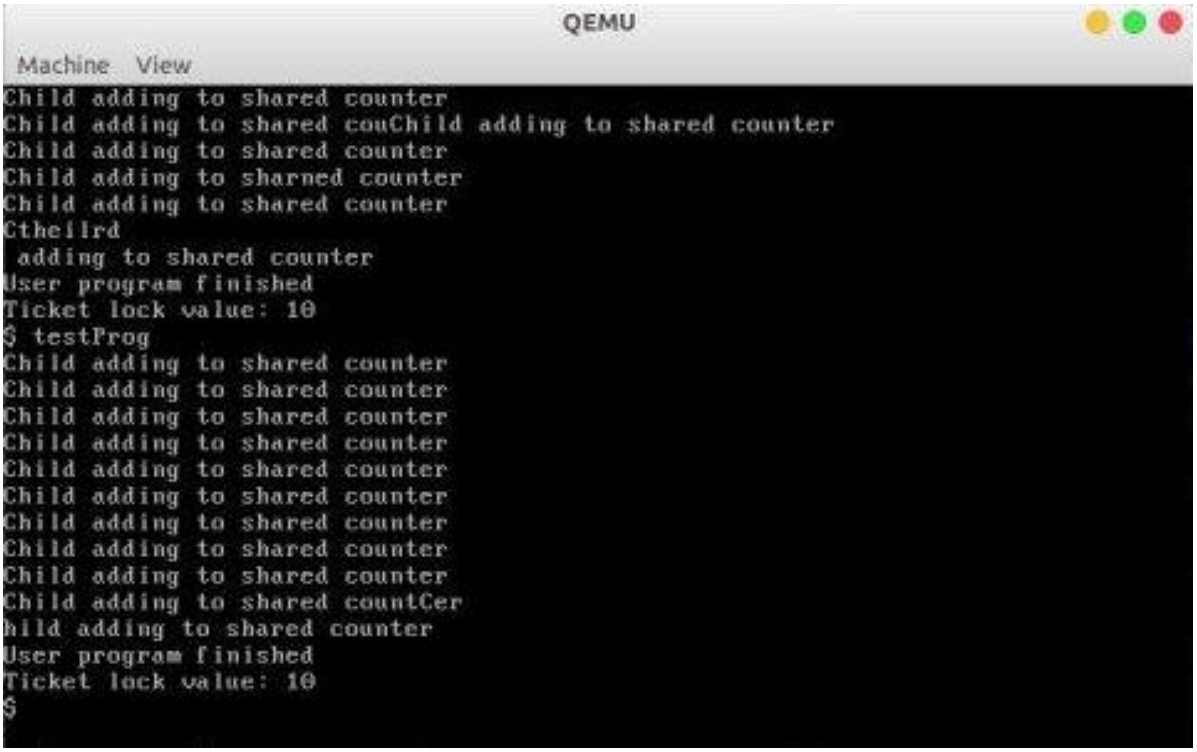
با استفاده از توابع فوق، یک تست ساده را پیاده سازی می‌کنیم و در ادامه مساله خوانندگان و نویسندگان را با استفاده از قفل ذکر شده پیاده سازی می‌کنیم.

۱,۱- تست قفل بلیت

دو فراخوانی سیستم به نام ticketLockInit و ticketLockTest را ایجاد می‌کنیم. سیستم کال اول صرفا یک ساختار قفل بلیت ایجاد کرده و مقدار های اولیه آن را برابر صفر قرار می‌دهد و مقدار متغیر عددی testLock را برابر صفر قرار می‌دهد.

سیستم کال ticketLockTest با هر بار صدا زده شدن، در انتظار قفل بلیت ذکر شده می‌ماند و در صورتی که اجازه کار پیدا کند، مقدار یک را به testLock اضافه می‌کند و مقدار جدید این متغیر را بازگردانی می‌کند.

در سطح کاربر برنامه ای به نام testProg نوشته ایم که با استفاده از fork به صورت همزمان ده بار فراخوانی سیستمی ticketLockTest را اجرا می‌کند و در نهایت مقدار نهایی testLock نمایش داده می‌شود. خروجی این برنامه به شکل زیر خواهد بود:



نصویر ۱- خروجی برنامه تست قفل بلیت، مقدار نهایی متغیر testLock برابر ۱۰ است

۱,۲- پیاده سازی الگوریتم خوانندگان و نویسندگان

پیاده سازی این بخش مشابه بخش قبل است. یک متغیر sharedCounter با مقدار اولیه صفر داریم و خوانندگان تنها می‌توانند این مقدار را بخوانند و نویسندگان تنها می‌توانند عدد یک را به متغیر فوق اضافه کنند. خواننده ها می‌توانند به صورت همزمان داده را بخوانند ولی در در هر لحظه یک نویسنده می‌تواند مقدار sharedCounter را عوض کند (در حالتی که خواننده ای هم درون سیستم نیست).

دو فراخوانی سیستمی به نام های rwinit و rwtest پیاده سازی می‌کنیم. سیستم کال اول صرفا به مقدار دهی اولیه متغیر ها و ساختار قفل بلیت می‌پردازد. سیستم کال rwtest یک مقدار ورودی به نام pattern قبول می‌کند که اگر مقدار آن ۰ باشد به معنی آن است که قصد خواندن داریم و اگر مقدار آن برابر ۱ باشد قصد نوشتن داریم. پیاده سازی خواننده و نویسنده به شکل زیر است:

```
int reader(){
  // Reader wants to enter the critical section
   acquireTicketLock(&mutex);

   // The number of readers has now increased by 1
   readcnt++;

   // there is atleast one reader in the critical section
   // this ensure no writer can enter if there is even one reader
   // thus we give preference to readers here
   if (readcnt==1)
      acquireTicketLock(&wrt);

   // other readers can enter while this current reader is inside
   // the critical section
   releaseTicketLock(&mutex);

   int out = sharedCounter;// current reader performs reading here
   acquireTicketLock(&mutex);   // a reader wants to leave

   readcnt--;

   // that is, no reader is left in the critical section,
   if (readcnt == 0)
       releaseTicketLock(&wrt);         // writers can enter

   releaseTicketLock(&mutex); // reader leaves
   return out;
}
```

نصویر ۳- الگوریتم مربوط به هر خواننده

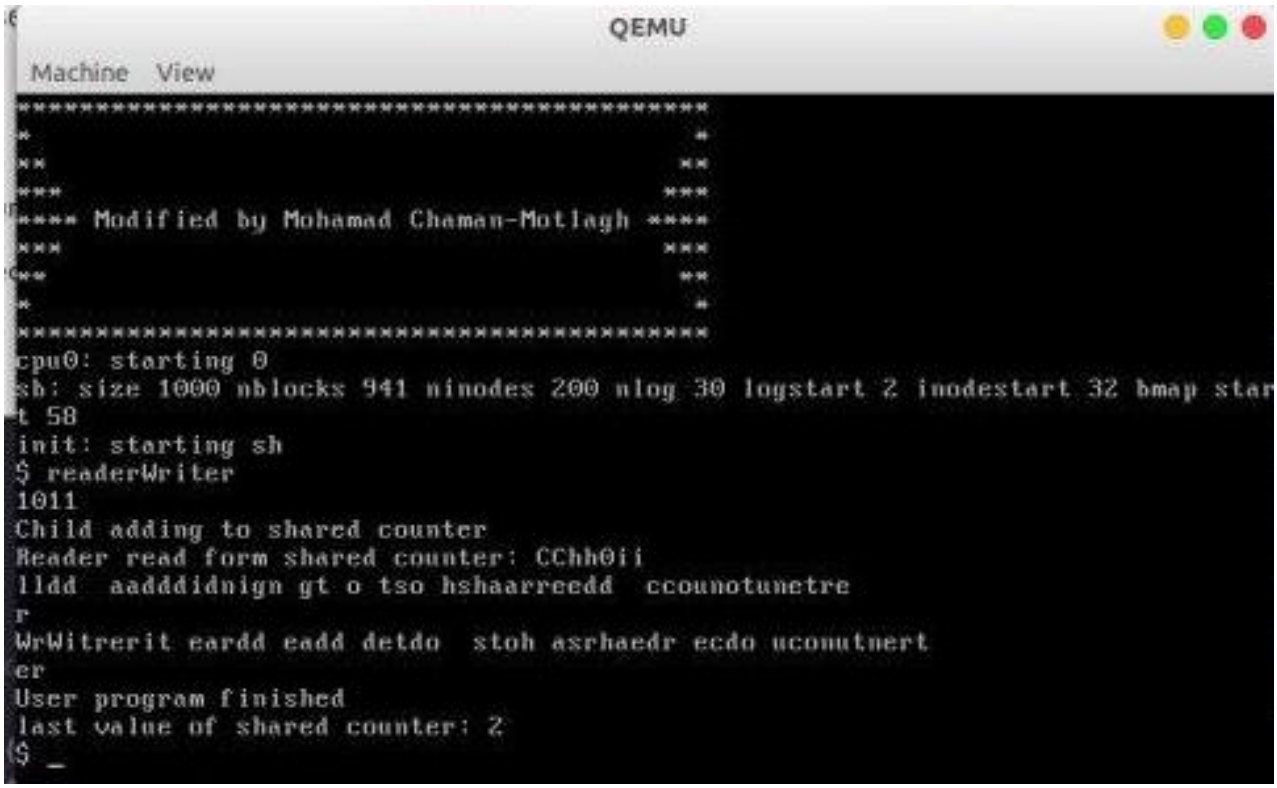
```
void writer(){
  // writer requests for critical section
    acquireTicketLock(&wrt);

    sharedCounter++;// performs the write

    // leaves the critical section
    releaseTicketLock(&wrt);
}
```

نصویر ۲- الگوریتم مربوط به هر نویسنده

درون برنامه کاربر readerWriter از این الگوریتم اســتفاده شــده اســت و با اســتفاده از Fork فرایند ها به صورت همزمان تلاش به خواندن یا نوشتن می‌کنند. خروجی این برنامه به ازای ورودی ۱۰۱۱ (به معنای خواندن و دوبار نوشتن) به صورت زیر خواهد بود: (مقدار نهایی باید برابر ۲ باشد زیرا دو بار نوشتن انجام شده است)



نصویر ۴خروجی الگوریتم خوانندگان نویسندگان با ورودی ۱۰۱۱

(از آنجایی که عملیات printf اتمیک نیست، جملات خروجی میانی به صورت به هم ریخته چاپ می‌شوند.)

- **بخش دوم: پیاده سازی Threading**

در این بخش به پیاده سازی پردازش چند نخی درون XV6 می‌پردازیم. پیاده سازی ما به صورت Kernel-level threading است. تفاوت میان Thread های سطح kernel و User این است که نخ های کرنل توسط سیستم عامل ساخته و مدیریت می‌شوند در حالی که نخ های سطح کاربر توسط زبان های برنامه نویسی ساخته و مدیریت می‌شوند. کار کردن با نخ های کرنل به شدت پیچیده است چرا که تمام کد های مورد نیاز برای مدیریت درست داده ها و روند برنامه باید توسط برنامه نویس پیاده شود، در حالی در نخ های کاربر زبان برنامه نویسی خیلی از مدیریت های فرایند چند نخی را انجام می‌دهد و استفاده از آن ها ساده تر است.

در مقابل نخ های کرنل کارایی بهتری دارند و به راحتی از پردازنده های چند هسته ای پشتیبانی می‌کنند، درحالی که در نخ های سطح کاربر پشتیبانی از چند هسته به صورت پیش‌فرض ممکن نمی‌باشد.

به منظور پیاده سازی این بخش نیازمند توابع زیر هستیم:

createThread(), exitThread, joinThread, getThreadID

در ابتدا باید ساختار Thread به فایل proc.h اضافه شود:

```
#define MAX_THREADS 16
enum threadstate { TUNSED, TEMBRYO, TSLEEPING, TRUNNABLE, TRUNNING, TZOMBIE };

struct thread {
    int tid;                    // Thread ID
    struct thread *tparent;     // Parent Thread
    struct proc *tproc;         // Thread parent process
    enum threadstate tstate;    // Thread state
    void *chan;                 // If non-zero, sleeping on chan
    struct context *context;    // swtch() here to run process
    char *kstack;               // Bottom of kernel stack for this process
    struct trapframe *tf;       // Trap frame for current syscall

};
```

نصویر ۵- ساختار Thread

در ادامه باید آرایه نخ های هر فرایند به ساختار فرایند اضافه شود همچنین باید ID هر نخ در حال پردازش به ساختار CPU اضافه شود.

تا مرحله فعلی پیاده سازی تنها وجود یک نخ برای هر فرایند پیاده سازی شده است. به این منظور هنگام هر allocProcess، یک نخ به صورت فرزند فرایند فعلی تعریف می‌شود و والد آن نیز فرایند ذکر شده قرار می-گیرد. هنگام Scheduling نیز مقدار متغیر thread ساختار cpu برابر نخ مورد نظر قرار می‌گیرد.

تابع mythread نیز به این صورت پیاده می‌شود که پس از متوقف کردن وقفه ها، مقدار ساختار thread موجود در ساختار cpu را باز می‌گرداند.

در نهایت فراخوانی سیستمی getThreadID با فراخواندن تابع mythread، مقدار tid را از ساختار thread باز می‌گرداند.

برنامه کاربر testThreadSystemCalls تنها به تست همین سیستم کال می‌پردازد و با فراخوانی آن، ID نخ در حال پردازش را چاپ می‌کند.