دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

گزارش پروژه دوم سیستم عامل

نگارش

آرش حاجی صفی - 9631019

استاد درس

استاد طاهری جوان

بهمن 1398

گزارش سوال اول:

۱.۱ پیاده سازی قفل بلیت (Ticket Lock)

برای این قسمت باید قفل Ticketlock را شبیه‌سازی می‌کردیم.

برای این منظور دو فایل Ticketlock.c و Ticketlock.h را ایجاد کرده‌ام که فیلدها و توابع مربوط به این قفل داخل آنها ذخیره شده‌اند.

استراکت این قفل دارای فیلدهای زیر است که در فایل Ticketlock.h مشخص کرده ام:

```
ticketlock.h ×
home > arash > Desktop > FinalProject > ticketlock.h > ...
// Mutual exclusion ticket lock.
struct ticketlock {
  int turn;        // which ticket is currently processing?
  int ticket;        // ticket value

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

برای پیاده سازی این قفل فقط به turn و ticket نیاز است. turn مشخص می‌کند که نوبت کدام process با شماره تیکت برابر با turn است که وارد Critical Section شود و ticket هم شماره‌ای است که به هر پراسسی که خواستار ورود به C.S است داده می‌شود و هربار یکی زیاد می‌شود.

بقیه‌ی فیلدها برای debug کردن به کار می‌روند که چون در قفل های دیگر XV6 مثل spinlock استفاده شده بودند من هم آنهارا نگهداری کردم برای استفاده در آینده.

در فایل Ticketlock.c توابع مربوط به این قفل را پیاده‌سازی کرده ام:

tlock.h C ticketlock.c ×

arash > Desktop > FinalProject > C ticketlock.c > ...

```c
void
initTicketlock(struct ticketlock *lk, char *name)
{
  lk->name = name;
  lk->turn = 0;
  lk->ticket = 0;
  lk->cpu = 0;
}

// Acquire the lock, get a ticket.
void
acquireTicketlock(struct ticketlock *lk)
{
  int ticket;
  ticket = fetch_and_add(&lk->ticket, 1);
//  cprintf("Ticket: %d, Turn: %d\n", ticket, lk->turn);
  while(ticket != lk->turn)
  ;

  // Record info about lock acquisition for debugging.
  //lk->cpu = mycpu();
}
```

در init همه‌ی فیلدهای این قفل مقدار اولیه می‌گیرند.

در acquireTicketlock به پروسه‌ای که آنرا فراخوانده، یک ticket اختصاص پیدا می‌کند که این تیکت پس از اختصاص به این پروسه به صورت atomic یکی زیاد می‌شود. تا زمانی که turn برابر با ticket آن پروسه نشود در while می‌ماند و وقتی turn برابر با آن شد، از while خارج شده و پروسه کدهای بعد از فراخوانی این تابع را اجرا کرده (مثلاً C.Sرا).

برای releaseTicketlock کد زیر را نوشته‌ام:

```c
// Release the lock, give turn to the next ticket
void
releaseTicketlock(struct ticketlock *lk)
{
  lk->turn++;
//  cprintf("program finished. Turn: %d\n", lk->turn);
}
```

همینطور که مشخص است، تنها turn یکی زیاد می‌شود و در نتیجه تیکت بعدی از while خارج می‌شود.

سیستم کال های خواسته شده را در تمامی 5 فایل زیر تعریف کرده‌ام:

```
1. syscall.h
2. syscall.c
3. sysproc.c
4. usys.S
5. user.h
```

یک Ticketlock در Proc.c تعریف کرده ام که با سیستم کال ticketlockInit مقداردهی اولیه می‌شود. یک متغیر هم در Proc.c گرفته ام که بین همه‌ی پروسه‌ها مشترک است و با فراخوانی سیستم کال tickelockTest یکی زیاد می‌شود.

```
ticketlock.h   ticketlock.c   proc.c
me > arash > Desktop > FinalProject > proc.c > rwinit(void)

int
ticketlockInit(void)
{
  initTicketlock(&tl, "ticketlock");
  sharedValue = 0;
  return 1;
}

int
ticketlockTest(void)
{
  acquireTicketlock(&tl);
  sharedValue++;
  releaseTicketlock(&tl);
  return sharedValue;
}
```

فایل ticketlockTest.c را ساخته ام و در makefile اضافه کرده ام. خروجی برنامه‌ی تست این قسمت به صورت زیر حاصل می‌شود:

```
arash@ubuntu: ~/Desktop/FinalProject

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8D100+1FECD100 C980



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ TicketlockTest
child adding to shared counter
child adding to shared counter
child adding to shared counter
child adding to shared counter
child adding to shared counter
child adding to shared counter
child adding to shared counter
chichilld adding to sharedcdh ila d ddcainodg utdnoite srnh
ga tore dsh arecdo ucnotuenrt
er
user program finished
ticket lock value: 10
$
```

که نتیجه‌ی درست و رضایت بخشی است. چون تمامی 10 فرآیند به صورت Mutually Exclusive مقدار متغیر مشترک را زیاد کرده اند که مقدار نهایی آن 10 شده. چون خود دستور Printf خارج از قفل اجرا می‌شود و M.E نیست، پرینت‌ها کمی درهم شده اند.

۲.۱ پیاده سازی قفل خوانندگان و نویسندگان :

برای این قسمت سیستم کال های rwinit و rwtest(uint pattern) را پیاده سازی کرده ام. در proc.c 2تا tickelock برای این سیستم کال ها گرفته ام، یکی wrt و دیگری mutex. یک متغیر مشترک هم به صورت جدا برای این مسئله گرفته ام.

در rwinit این قفل‌ها و متغیر مشترک مقداردهی اولیه می‌شوند. در rwtest اگر 1 گرفته شود پروسه وارد قسمت نویسنده شده و یکی به متغیر مشترک اضافه می‌کند با استفاده از قفل و اگر 0 باشد مقدار متغیر مشترک را می‌خواند. برای خواندن و نوشتن دقیقاً همان خوانندگان و نویسندگان استفاده شده با این تفاوت که جای wait و signal روی هر سمافور، از

acquireTicketlock و releaseTicketlock روی قفل بلیت متناظر آن اتسفاده کرده‌ام که دقیقاً شرایط مسئله را برقرار می‌سازد.

برای تست برنامه‌ی ReadersWriterTest.c را به makefile اضافه کرده ام.

به عنوان نمونه pattern 110010 را به برنامه اگر بدهیم خروجی به این صورت می‌شود:

```
arash@ubuntu: ~/Desktop/FinalProject

Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ ReadersWritersTest
enter pattern for readers/writers test
110010
child adding to shared counter
writer added to sharedchild a ccchdding to shared counter
reader reaohuniterd
ild  ladddinf g ratood sm hsahadrerid encougd  nctoeunrtteo
crr heas :de rhar1
reiedla ddco uf anrtoemdrdi
nwsgrih taterore s ad cdohudnateder ret do :  c1os
haurnetde r
counter
reader read from shared counter : 2
user program finished
last value of shared counter: 2
$
```

که می‌بینیم دقیقاً مطابق pattern، اول writer وارد شده و یکی اضافه کرده، در نتیجه 2تا reader اول مقدار 1 را خوانده اند و سپس writer بعدی وارد شده و یکی دیگر اضافه کرده و نهایتاً reader آخر مقدار 2 را خوانده.

چون خود printf ها بیرون قفل اجرا می‌شوند و M.E نیستند نوشته‌ها به هم ریخته ولی ترتیب اجرا و مقادیر که درون قفل اجرا شده‌اند می‌بینیم که درستند.

گزارش سوال دوم: پیاده سازی Thread در سطح کرنل

2.1– مقدمات

Thread در سطح کرنل نسبت به Thread در سطح کاربر:

مزایا:

1- چونکه کرنل از تمامی رشته ها اطلاع دارد، با اطلاعات دقیق خود scheduler را خیلی بهتر مدیریت می‌کند و مثلاً به پروسه ای که تعداد زیادی thread دارد، زمان بیشتری از cpu را اختصاص می‌دهد نسبت به پروسه‌ای که تعداد کمتری thread دارد.

2- این نوع thread به خصوص برای برنامه‌هایی که زیاد بلاک می‌شوند خیلی مناسب است.

معایب:

1- این نوع threadها کند و غیر بهینه هستند و صدها برابر نسبت به threadهای در سطح کاربر کندتر هستند.

2- چون کرنل باید این رشته‌ها را مدیریت و زماندهی کند در کنار پروسه ها، به یک بلوک کامل برای کنترل Thread (TCB) نیاز دارد که باعث افزایش پیچیدگی و overhead می‌شود.

در مقابل رشته‌های در سطح کاربر مدیریت ساده ای دارند و سریع هستند.

در این قسمت در استراکت هر پروسه، یک لیست از Threadها را هم برای آن پروسه با حداکثر اندازه‌ی مشخص شده اضافه کرده ام و وضعیت ها و فیلدهای مختلف استراکت proc را که برای اجرا نیاز بود (مثل استک کرنل و trap frame و ...) به استراکت thread منتقل کرده ام:

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

};

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int killed;                  // If non-zero, have been killed

  struct thread threads[MAX_THREADS]; //Threads for this process
  struct spinlock lock;        // A spinlock to sync threads
};
```

همچنین برای دسترسی به لیس threadهای هر پروسه به یک قفل مشابه قفل لیست پروسه‌ها نیاز است که آنرا هم اضافه کرده ام. برای cpu هم یک فیلد برای مشخص کردن اینکه کدام thread در حال اجرا است اضافه کرده ام.

تقریباً تمامی بخش‌های proc.c و حتی فایل های دیگر مثل trap.c و ... دچار تغییر شده اند و scheduler هم به طور کامل تغییر کرده تا به جای process، با thread هر پروسه کار کنند. توابعی هم لازم بوده که اضافه شده:

```c
struct thread*  mythread();
void            killSelf(void);
void            killSiblings(void);
```

۲.۲ پیاده سازی سیستم کال ها

تمامی سیستم کال ها با کامنتگذاری کامل پیاده سازی شده اند و در نهایت یک تست به صورت زیر در فایل testThreadSystemCalls.c نوشته شده تا از صحت عملکرد همه‌ی سیستم کال های این بخش اطمینان حاصل شود:

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

void runFunc(){
    printf(1, "Created thread id: %d\n", getThreadID());
    printf(1, "Created thread parent process id: %d\n", getpid());
    exitThread();
}
void runFunc2(){

}

void runFunc3(){

}

int main()
{
    printf(1, "Main process id: %d\n", getpid());
    printf(1, "Main thread id: %d\n", getThreadID());
    void* stack = (void*)malloc(4000);
    void(*func)();
    func = (void*) runFunc;
    int tid1 = createThread(func, stack);
//    int tid2 = createThread(func, stack);
    int ret = joinThread(tid1);
//    joinThread(tid2);
    printf(1, "tid: %d, joinThread(tid) returned: %d\n", tid1, ret);
    exit();
}
```

خروجی این تست به صورت زیر می‌باشد:

```
arash@ubuntu: ~/Desktop/FinalProject
SeaBIOS (version 1.12.0-1)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8D100+1FECD100 C980



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ testThreadSystemCalls
Main process id: 3
Main thread id: 3
Created thread id: 4
Created thread parent process id: 3
tid: 4, joinThread(tid) returned: 0
$
```

همینطور که می‌بینیم در ابتدا thread اصلی برای main اجرا شده و آیدی این رشته و پروسه‌ی پدر آن چاپ می‌شود، سپس رشته‌ای که ساختیم اجرا شده و این رشته آیدی خودش و پروسه‌ای که متعلق به آن است را چاپ کرده، رشته‌ی اصلی منتظر اتمام این رشته‌ی ایجاد شده می‌ماند و وقتی این رشته خارج شد، رشته‌ی اصلی مقدار برگردانده شده در threadJoin را که 0 است (موفقیت) چاپ کرده و برنامه پس از خروج thread اصلی اتمام می‌یابد.