پروژه دوم درس سیستم های عامل

دکتر نستوه طاهری جوان

نویسندگان:
غزل صادقیان
محمدمهدی ناصری

دانشکده مهندسی کامپیوتر، دانشگاه صنعتی امیرکبیر

۹ دی ۱۳۹۸

۱ سوال اول

۱.۱ پیاده‌سازی قفل بلیت (Ticket lock)

قفل گذاری در هسته xv۶ توسط دو سری تابع صورت می‌گیرد. دسته اول شامل توابع acquire و release می‌شود که یک پیاده سازی ساده از spin locks هستند. این قفل‌ها منجر به انتظار مشغول [۱] شده و در حین اجرای ناحیه بحرانی [۲] وقفه را نیز غیرفعال می‌کنند. دسته دوم شامل توابع acquiresleep و releasesleep بوده که مشکل انتظار مشغول را حل نموده و امکان تعامل میان پردازه ها را نیز فراهم میکنند. تفاوت اصلی توابع این دسته نسبت به دسته قبل این است که در صورت عدم امکان در اختیار گرفتن قفل، از تلاش دست کشیده و پردازنده را رها میکنند. قفل بلیت نوعی قفل است که در آن به هر پردازه‌ای که می‌خواهد قفل را بگیرد , یک بلیت اختصاص داده می‌شود. مقدار هر بلیت جدید یکی بیشتر از مقدار آخرین بلیت داده‌شده به پردازه‌ها است. قفل در هر زمان دارای مقدار بلیتی است که باید به آن سرویس دهد. پردازه‌ای که دارای بلیتی با آن مقدار است، می‌تواند قفل را بگیرد و پس از رها کردن قفل، نوبت بلیت بعدی می‌شود. (این نوبت‌ها به ترتیب خواهند بود) در واقع قفل بلیت نوعی قفل عادلانه است و مزیت اصلی استفاده از قفل بلیت در مقایسه با spin lock ، این است که به صورت FIFO بوده و در نتیجه، از starvation جلوگیری می‌کند. توابع initTicketlock ، acquireTicketlock و releaseTicketlock مربوط به قفل بلیت را پیاده سازی کنید. با مراجعه به فایل‌های spinlock.h و spinlock.c می‌توانید پیاده‌سازی مربوط به spinlock را ببینید و با کمک گرفتن از آن، قفل بلیت را پیاده‌سازی کنید. در پیاده‌سازی برای اضافه کردن یک واحد به مقدار بلیت باید از inline assembly ، استفاده کنید تا به شکل اتمیک اجرا شود. (کد مربوطه باید در فایل x86.h اضافه شود.)

```
147 static inline int
148 fetch_and_add (int *var, int value)
149 {
150   asm volatile("lock; xaddl %0, %1"
151     : "+r" (value), "+m" (*var)
152     :
153     : "memory"
154   );
155   return value;
156 }
```

دو فراخوانی سیستمی ticketlockInit و ticketlockTest را به سیستم عامل اضافه کنید. فراخوانی سیستمی نخست، قفل را مقداردهی اولیه نموده و دومی توسط پردازه‌هایی به طور موازی از آن استفاده می‌نماید تا از صحت عملکرد آن اطمینان حاصل شود. برنامه سطح کاربر باید چیزی مشابه برنامه زیر باشد.

---

[۱] Busy waiting

[۲] Critical Section

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

#define NCHILD 10

int main ()
{
        int pid;
        ticketlockinit();

        pid = fork();
        for (int i = 1; i < NCHILD; i++)
                if (pid < 0)
                {
                        printf(1, "fork failed\n");
                        exit();
                }
                else if (pid > 0)
                        pid = fork();

        if (pid < 0)
        {
                printf(1, "fork failed\n");
                exit();
        }
        else if (pid == 0)
        {
                printf(1, "child adding to shared counter\n");
                ticketlocktest();
        }
        else
        {
                for (int i = 0; i < NCHILD; i++)
                        wait();
                printf(1, "user program finished\n");
                printf(2, "ticket lock value: %d\n", ticketlocktest() - 1);
        }

        exit();

        return 0;
}
```

مقدار ticket lock value در برنامه بالا باید ۱۰ باشد.

۲.۱ پیاده‌سازی قفل خوانندگان و نویسندگان

با استفاده از قفل بلیت، قفل خوانندگان_نویسندگان را برای حالت تقدم خوانندگان پیاده‌سازی نمایید. برای این کار لازم است دو فراخوانی سیستمی rwinit و rwtest(uint pattern) را اضافه نمایید. ساختار توابع، مشابه توابع پیاده‌سازی شده برای قفل بلیت است. با این تفاوت که در rwtest باید الگوی دسترسی به داده مشترک به صورت یک پارامتر صحیح مثبت به نام pattern به هسته داده شود. به این شکل که سمت چپ‌ترین بیت همواره یک بوده و بیت‌های بعدی ترتیب زمانی خواندن یا نوشتن داده مشترک را مشخص کنند. صفر معادل خواندن و یک معادل نوشتن خواهد بود. مثلاً عدد ۱۹ در فرم دودویی به صورت زیر نوشته میشود : ۱۰۰۱۱ با صرف‌نظر کردن از پرارزش‌ترین بیت، به ترتیب خواندن، خواندن، نوشتن و نوشتن رخ خواهد داد. به این ترتیب هر پردازه در سطح هسته، متناسب با دسترسی مورد نظر، قفل خواندن یا نوشتن را درخواست می‌کند. در اینجا نیز می‌توان فرض نمود که متغیر مشترک یک عدد بوده که با هر بار نوشتن یک واحد به مقدار آن افزوده میشود .

برنامه سطح کاربر باید چیزی مشابه برنامه زیر باشد:

```c
#include "types.h"
#include "user.h"
#define NCHILD 10

void testReadersWriters(int* pattern, int pattern_size);
int main (){
    char argv[100];

    printf(1, "enter pattern for readers/writers test\n");
    read_size = read(0, argv, sizeof(argv));
    argv[read_size - 1] = '\0';
    int pattern[100], i;
    for (i = 0; argv[i + 1] != '\0'; i++){
        if (argv[i + 1] == '0')
            pattern[i] = 0;
        else if (argv[i + 1] == '1')
            pattern[i] = 1;
        else{
            printf(1, "pattern must consist of only 0 and 1");
            exit();
        }
    }
    testReadersWriters(pattern, i);
    exit();
    return 0;
}
void testReadersWriters(int* pattern, int pattern_size){
    int pid, i;
    rwinit();
    pid = fork();
    for (i = 1; i < pattern_size; i++){
        if (pid < 0){
            printf(1, "fork failed\n");
            exit();
        }
        else if (pid > 0)
            pid = fork();
        else
            break;
    }
    if (pid < 0){
        printf(1, "fork failed\n");
        exit();
    }
    else if (pid == 0){
        printf(1, "child adding to shared counter\n");
        int res = rwtest(pattern[i - 1]);
        if (pattern[i - 1] == 0)
            printf(2, "reader read from shared counter : %d\n", res);
        else
            printf(1, "writer added to shared counter\n");
    }
    else{
        for (i = 0; i < pattern_size; i++)
            wait();
        printf(1, "user program finished\n");
        int res = rwtest(0);
        printf(1, "last value of shared counter : %d\n", res);
    }
}
```

برای هر قسمت از مراحل انجام کار و خروجی‌ها گزارش تهیه کنید.

۲ سوال دوم: پیاده سازی ترد (Thread) در سطح کرنل

۱.۲ مقدمات

این نوع ترد ها در سطح کرنل مدیریت می شوند و در برابر ترد های در سطح کاربر دارای مزایا و معایبی می باشند؛ این موارد را در گزارش خود ذکر کنید. هدف این سوال پیاده سازی این نوع ترد هاست. برای این منظور باید یک struct مشابه با struct proc به نام struct thread تعریف کنید و تصمیم بگیرید با توجه به این که هر پراسس [۳] می تواند چندین ترد داشته باشد کدام یک از فیلد های struct proc در proc باقی بمانند و کدام ها به thread بروند و در آنجا تعریف شوند؛ به عنوان مثال احتیاجی به ساختن فضای مموری جدید برای هر ترد نخواهد بود.

برای شروع باید در تمامی کدها بخش هایی که از فیلدی از struct proc استفاده شده است را بررسی کنید و جاهایی که لازم است آن فیلدها را با متناظرشان از struct thread جایگزین کنید. پس از انجام این کار هر پراسس در ابتدای شروع به کار خود دارای یک ترد اصلی خواهد بود که در حال اجرا شدن می باشد. همچنین در فایل proc.h یک ثابت به نام MAX_THREADS با مقداری مثلا برابر با ۱۶ انتخاب کنید و به واسطه ی آن به هر پراسس اجازه ندهید بیشتر از این تعداد ترد داشته باشد.

در proc.h باید struct cpu را طوری تغییر بدهید که بتواند از ترد هم در کنار پراسس پشتیبانی کند. همچنین تابعی مشابه با myproc() به نام mythread() برای دسترسی به ترد فعلی تعریف کنید و در هر جایی که از myproc استفاده شده است تصمیم بگیرید که آیا لازم است به جای آن از mythread استفاده بشود. همینطور لازم است scheduler را طوری تغییر دهید تا به جای اختصاص پردازنده به پراسس ها، به ترد ها پردازنده اختصاص بدهد. به این معنا که به هر ترد از یک پراسس، پردازنده را اختصاص داده و سپس به سراغ پراسس بعدی برود.

توجه کنید که سیستم کال فعلی fork باید به گونه ای اصلاح شود که در صورت اجرا شدن فقط ترد فراخوانی کننده ی آن در پراسس جدید ایجاد شده وجود داشته باشد، نه ترد های دیگر آن پراسس. همینطور سیستم کال exit باید پراسس و تمامی ترد های آن را خاتمه دهد؛ حتی اگر در زمان فراخوانی توسط یک ترد، ترد هایی دیگر از پراسس صاحب، در حال اجرا باشند.

۲.۲ پیاده سازی سیستم کال ها

سیستم کال int createThread(void (*function)() , void* stack) را پیاده سازی کنید که وظیفه ی ایجاد یک ترد برای پراسسی که فراخوانی را انجام داده است بر عهده دارد. ورودی های آن stack یک پوینتر به user stack برای ترد ساخته شده است که باید توسط پراسس فراخوانی کننده allocate بشود و با kernel stack که توسط خود createThread ساخته می شود تفاوت دارد. همچنین ورودی دیگر function یک پوینتر به تابعی است که ترد ایجاد شده وظیفه ی اجرای آن را برعهده دارد. خروجی آن در صورت موفقیت، id مرتبط با ترد ساخته شده است و در غیر این صورت -1 می باشد.

سیستم کال int getThreadID() باید id مرتبط با ترد فراخوانی کننده را برگرداند که در حالت کلی با id پراسس صاحب آن متفاوت است.

سیستم کال void exitThread() باید ترد فراخوانی کننده آن را خاتمه دهد. توجه کنید که ممکن است ترد های دیگری

[۳] process

از پراسس صاحب ترد فراخوانی کننده هم وجود داشته باشند که نباید خاتمه پیدا کنند. تنها درصورتی که این ترد آخرین ترد از پراسس صاحب آن باشد، پراسس اصلی نیز خاتمه می یابد.

سیستم کال int joinThread(int threadID) اجرای ترد فراخوانی کننده را تا خاتمه یافتن ترد با id برابر با threadID متوقف می کند. در صورتی که از قبل ترد متناظر با threadID خاتمه یافته باشد، ترد فراخوانی کننده متوقف نمی شود. پس از اتمام توقف باید مقدار ۰ و یا اگر مشکلی در اجرا به وجود آمد 1− بازگردانده شود.

پس از پیاده سازی ویژگی های بالا، سیستم کال های مذکور باید در برنامه ای به نام testThreadSystemCalls امتحان شود و نتایج خروجی به همراه توضیحات نحوه ی تست کردن موارد مذکور در این برنامه در گزارش شما آورده شود. همچنین توجه کنید که برای بهره مندی از ویژگی چند پردازنده لازم است در Makefile تغییرات لازم را اعمال کنید.

بخش امتیازی: با توجه به شرایط کنونی که چندین پردازنده به طور همزمان در حال اجرا هستند ممکن است بخش هایی از کد مثل تابع growproc() در proc.c که مسئولیت گسترش حافظه ی مورد نیاز برای پراسس ها را بر عهده دارد، در صورتی که دو یا چند ترد از یک پراسس به طور همزمان در حال اجرا هستند، دچار اشکال شود؛ به همین دلیل می توانید با راهکاری از اجرای همزمان چنین بخش هایی توسط چندین پردازنده به طور همزمان جلوگیری کنید.

توجه: فایل گزارش را به نام OS-[StNum]-P2-Report.pdf به همراه کدهای خود و پوشه git. در حالت زیپ به نام OS-[StNum1]-[StNum2]-P2.zip آپلود کنید. دقت کنید با توجه به صحبت هایی که دکتر طاهری در کلاس کردند ترکیب گروه ها باید مطابق با ترکیب پروژه قبلی باشد. همچنین در صورت مشکل در رابطه با سوال ۱ با Sadeghian.ghazal77@gmail.com و در رابطه با سوال ۲ با m.n4seri@gmail.com تماس بگیرید.