جلسه‌ی هفتم — کتابخانه‌ی PThreads

یکی از راه‌های اجرای همروند دستورات در سیستم‌های عامل استفاده از ریسمان[1] می‌باشد (اگر چه ریسمان یا نخ ترجمه‌ی مناسبی برای «Thread» به نظر نمی‌رسند، به علت رواج آنها، در این مستند از ریسمان استفاده می‌نماییم). در هر پردازه، تعدادی ریسمان می‌توانند به صورت همروند اجرا شوند. چون همه‌ی ریسمان‌های یک پردازه در فضای آدرس پردازه اجرا می‌گردند، می‌توانند با استفاده از حافظه‌ی مشترک (برای مثال متغیرهای Global) با هم ارتباط داشته باشند. اما برای جلوگیری از مشکل «Race condition»، دسترسی همزمان ریسمان‌های مشترک باید مدیریت شود.

استاندارد «POSIX» رابط[2] یک کتابخانه‌ی ریسمان را معرفی کرده است که «POSIX Threads» یا «PThreads» نامیده شده است. این کتابخانه در بسیاری از سیستم‌های عامل رایج از جمله لینوکس پیاده‌سازی شده است. در این جلسه با این کتابخانه آشنا می‌شوید. برای استفاده از کتابخانه‌ی PThreads در لینوکس باید پارامتر lpthread- را به Linker فرستاد (در برخی از محیط‌ها باید پارامتر pthread- را هم به Linker و هم به مترجم فرستاد).

آشنایی با کتابخانه‌ی PThreads

با تابع pthread_create() می‌توان یک ریسمان ایجاد نمود. ورودی اول این تابع یک اشاره‌گر به یک متغیر برای ذخیره‌سازی شناسه‌ی[3] ریسمان جدید است و ورودی سوم این تابع، تابعی است که ریسمان جدید باید از آن اجرای خود را شروع کند. برای اطلاع بیشتر در مورد ورودی‌های این تابع، به صفحه‌ی راهنمای آن مراجعه نمایید. در صورتی که این تابع با موفقیت اجرا شود، ریسمان جدید اجرای خود را از تابع مشخص شده در ورودی سوم شروع خواهد نمود.

```
#include <pthread.h>

pthread_t tid;
pthread_create(&tid, NULL, func, NULL);      /* create a thread */
```

۱ Thread
۲ Interface
۳ Identifier

تابعی که به pthread_create() فرستاده می‌شود یک اشاره‌گر دریافت می‌کند (ورودی چهارم pthread_create()) و یک اشاره‌گر بر می‌گرداند:

```
void *func(void *dat)
{
        printf("Thread started!\n");
        return NULL;
}
```

برای انتظار برای اتمام یک ریسمان می‌توان تابع pthread_join() را فراخوانی کرد (مشابه تابع wait() برای پردازه‌ها). مقدار برگشت داده شده توسط تابع اصلی یک ریسمان (ورودی سوم pthread_create()) در ورودی دوم این تابع (در صورتی که به یک متغیر * void اشاره کند) قرار می‌گیرد.

```
pthread_join(tid, NULL);            /* wait for thread tid */
```

مدیریت دسترسی‌های همزمان

کتابخانه‌ی PThreads امکاناتی را برای مدیریت دسترسی‌های همزمان ارائه می‌دهد. یکی از آنها، قفل‌های Mutex می‌باشند. برای استفاده از این قفل، باید یک متغیر با نوع pthread_mutex_t ساخت و با استفاده از تابع pthread_mutex_init() به آن مقدار اولیه داد. با تابع pthread_mutex_lock() این قفل بسته و با تابع pthread_mutex_unlock() باز می‌شود. در هر لحظه فقط یک ریسمان می‌تواند یک Mutex را قفل کند و بقیه‌ی ریسمان‌هایی که با فراخوانی pthread_mutex_lock() قصد قفل کردن آن را داشته باشند در حالت Blocking قرار می‌گیرند. پس از استفاده از یک قفل، می‌توان منابع اختصاص یافته به آن را با فراخوانی pthread_mutex_destroy() آزاد کرد.

```
pthread_mutex_t lock;

pthread_mutex_init(&lock, NULL);    /* initialize a mutex */

pthread_mutex_lock(&lock);          /* lock the mutex */
pthread_mutex_unlock(&lock);        /* unlock the mutex */
```

```
pthread_mutex_destroy(&lock);        /* free a mutex */
```

کتابخانه‌ی PThreads امکان استفاده از Semaphore را نیز ارائه می‌دهد. برای یک Semaphore متغیری با نوع sem_t باید معرفی گردد. فراخوانی sem_init() آن را مقدار دهی می‌کند (ورودی سوم، مقدار اولیه‌ی آن را مشخص می‌کند) و پس از استفاده از آن، فراخوانی sem_destroy() منابع اختصاص یافته به آن را آزاد می‌نماید. تابع sem_wait() مقدار یک Semaphore را کاهش می‌دهد (و در صورت نیاز منتظر می‌ماند) و تابع sem_post() مقدار آن را افزایش می‌دهد.

```
sem_t sem;

sem_init(&sem, 0, 1);                /* initialize a semaphore */
sem_post(&sem);                      /* increment a semaphore */
sem_wait(&sem);                      /* decrement a semaphore */
sem_destroy(&sem);                   /* free a semaphore */
```

مسئله‌ی تولید کننده و مصرف کننده

در درس سیستم عامل مسئله‌ی تولید کننده و مصرف کننده معرفی شده است. با استفاده از راه‌کارهای ارائه شده در کتابخانه‌ی PThreads می‌توان این مسئله را با در نظر گرفتن دسترسی‌های همزمان حل کرد. هنگامی که اندازه‌ی بافر[1] (که برای انتقال داده‌های تولید شده به مصرف کننده استفاده می‌شود) یک باشد، می‌توان با استفاده از دو Semaphore داده‌های تولید شده توسط تولید کننده را به مصرف کننده انتقال داد.

```
sem_t sem1, sem2;
sem_init(&sem1, 0, 1);
sem_init(&sem2, 0, 0);
```

در ادامه قسمت مربوط به تولید کننده نشان داده شده است.

```
sem_wait(&sem1);
...                                  /* Append an item to the buffer */
sem_post(&sem2);
```

برای اندازه‌ی بافر بزرگ‌تر، لازم است دسترسی‌های همزمان به آن نیز مدیریت گردد.

---

۱ Buffer

تمرین هفتم

در این تمرین نیز مشابه تمرین ششم، باید برنامه‌ای را به دو قسمت تقسیم نمایید اما در این تمرین به جای دو پردازه، باید از دو ریسمان استفاده کنید. پس از دریافت فایل ex6.c، نام آن را به ex7.c تغییر دهید و در شاخه‌ی ~/ex7 قرار دهید. همان طور که در تمرین گذشته دیدید، در این برنامه، قسمت اول پردازش در تابع ()prod و قسمت دوم در تابع ()cons انجام می‌شود (در حلقه‌ی تابع ()main، هر خروجی ()prod به تابع ()cons فرستاده می‌شود). در این تمرین باید یک ریسمان بسازید: تابع ()prod در ریسمان اصلی و تابع ()cons در ریسمان جدید باید فراخوانی شوند. برای انتقال خروجی تابع ()prod به ریسمان جدید، باید از تعدادی متغیر مشترک استفاده کنید. با قفل‌های Mutex و Semaphore از مشکلات مربوط به دسترسی‌های همزمان جلوگیری نمایید.

گام‌های پیشنهادی برای انجام این تمرین:

۱ دریافت و ترجمه‌ی فایل ex7.c

۲ ایجاد یک ریسمان جدید

۳ انتظار برای اتمام ریسمان جدید در ریسمان اصلی

۴ انتقال فراخوانی تابع ()cons به ریسمان جدید

۵ معرفی یک متغیر Global برای انتقال داده بین ریسمان‌ها

۶ مشاهده‌ی نیاز به مدیریت دسترسی‌های همزمان

۷ مدیریت دسترسی‌های همزمان توسط دو Semaphore