

## جلسه هشتم — کتابخانه‌ی PThreads

یکی از راه‌های اجرای همروند دستورات در سیستم‌های عامل استفاده از ریسمان<sup>۱</sup> می‌باشد (اگر چه ریسمان یا نخ ترجمه‌ی مناسبی برای «Thread» به نظر نمی‌رسند، به علت رواج آنها، در این مستند نیز از عبارت ریسمان استفاده می‌شود). در هر پردازش، تعدادی ریسمان می‌توانند به صورت همروند اجرا شوند. چون همه‌ی ریسمان‌های یک پردازش در فضای آدرس پردازش اجرا می‌گردند، می‌توانند با استفاده از حافظه‌ی مشترک (برای مثال متغیرهای سراسری<sup>۲</sup>) با هم ارتباط داشته باشند. اما برای جلوگیری از مشکل وضعیت رقابتی<sup>۳</sup>، دسترسی همزمان ریسمان‌ها به متغیرهای مشترک باید مدیریت شود.

استاندارد «POSIX»<sup>۴</sup> رابطه‌ی یک کتابخانه‌ی ریسمان را معرفی کرده است که «POSIX Threads» یا «PThreads» نامیده می‌شود. 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<sup>۵</sup> 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;
```

- 
- 1 Thread
  - 2 Global
  - 3 Race condition
  - 4 Interface
  - 5 Identifier

```
pthread_create(&tid, NULL, func, NULL); /* ایجاد یک ریسمان از تابع «func» */
```

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 پردازش‌ها). 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); /* انتظار برای فاتمه‌ی ریسمان «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`» سافت 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 انتظار<sup>۱</sup> 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); /* مقداردهی اولیه به یک «Mutex» */
pthread_mutex_lock(&lock); /* قفل کردن یک «Mutex» */
pthread_mutex_unlock(&lock); /* باز کردن یک «Mutex» */
pthread_mutex_destroy(&lock); /* آزاد کردن منابع اختصاص داده شده به یک «Mutex» */
```

---

1 Blocking

کتابخانه‌ی PThreads امکان استفاده از سمافور<sup>۱</sup> را نیز ارائه می‌دهد. برای یک سمافور 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 سمافور 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);          /* مقداردهی اولیه به یک سمافور */
sem_post(&sem);               /* افزایش مقدار یک سمافور */
sem_wait(&sem);               /* کاهش مقدار یک سمافور */
sem_destroy(&sem);           /* آزاد کردن منابع یک سمافور */
```

### 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<sup>۲</sup> (که برای انتقال داده‌های تولید شده به مصرف کننده استفاده می‌شود) یک باشد، می‌توان با استفاده از دو سمافور داده‌های تولید شده توسط تولید کننده را به مصرف کننده انتقال داد.

```
sem_t full;                   /* مقدار یک به معنای وجود یک عنصر در بافر است */
sem_t empty;                 /* مقدار یک به معنای خالی بودن بافر است */
sem_init(&full, 0, 0);
sem_init(&empty, 0, 1);
```

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

```
sem_wait(&empty);            /* انتظار برای خالی شدن بافر */
...                          /* اضافه کردن یک عنصر به بافر */
sem_post(&full);            /* تغییر مقدار سمافور «full» پس از پر شدن بافر */
```

همان طور که در درس مشاهده کرده‌اید، برای اندازه‌ی بافر بزرگ‌تر، لازم است دسترسی‌های همزمان به آن نیز

1 Semaphore  
2 Buffer

## تمرین هشتم

در این تمرین نیز مشابه تمرین هفتم، باید برنامه‌ای را به دو قسمت تقسیم نمایید اما در این تمرین به جای دو پردازش، باید از دو ریسمان استفاده کنید. پس از دریافت فایل «ex7.c»، نام آن را به «ex8.c» تغییر دهید و در شافهی «~/ex8» 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 ملقهی تابع «main()»، هر فروجی «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 فروجی تابع «prod()» به ریسمان جدید، باید از تعدادی متغیر مشترک استفاده کنید. با قفل‌های «Mutex» و سمافور از مشکلات مربوط به دسترسی‌های همزمان جلوگیری نمایید.

گام‌های پیشنهادی برای انجام این تمرین:

- ۱ دریافت و ترجمه‌ی فایل «ex8.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 سراسری برای انتقال داده بین ریسمان‌ها
- ۶ مشاهده‌ی نیاز به مدیریت دسترسی‌های همزمان
- ۷ مدیریت دسترسی‌های همزمان توسط دو سمافور