Sharif-OS-Lab / session-7

Code Issues 1 Pull requests Actions Projects Security Insights

session-7 / session7.md

bhnum Fix second code sample for session7.md 1f50168 · 3 months ago

147 lines (96 loc) · 10.8 KB

Preview Code Blame Raw

آزمایش ۷ - آشنایی با ریسه‌ها

۷.۱ مقدمه

هدف اصلی این آزمایش، بررسی جنبه‌های مختلف ریسه‌ها و چند پردازشی (و چند ریسه‌ای) است. از اهداف اصلی این آزمایش، پیاده سازی توابع مدیریت ریسه‌ها است:

- ساخت ریسه‌ها
- پایان بخشیدن به اجرای ریسه
- پاس دادن متغیر به ریسه‌ها
- شناسه‌های ریسه‌ها
- متصل شدن ریسه‌ها

۷.۱.۱ پیش‌نیازها

انتظار می‌رود که دانشجویان با موارد زیر از پیش آشنا باشند:

- برنامه‌نویسی به زبان C++/C
- دستورات پوسته لینوکس که در جلسات قبل فرا گرفته شدند.

۷.۲ ریسه چیست؟

یک ریسه، شبه پردازه‌ای است که پشته‌ی خاص خود را در اختیار دارد و کد مربوط به خود را اجرا می‌کند. برخلاف پردازه، یک ریسه، معمولا حافظه‌ی خود را با دیگر ریسه‌ها به اشتراک می‌گذارد. یک گروه از ریسه‌ها، یک مجموعه از ریسه‌ها است که در یک پردازه‌ی یکسان اجرا می‌شوند. بنابراین آنها یک حافظه‌ی یکسان را به اشتراک می‌گذارند و می‌توانند به متغیرهای عمومی یکسان، حافظه‌ی heap یکسان و ... دسترسی داشته باشند. همه‌ی ریسه‌ها می‌توانند به صورت موازی (استفاده از برش زمانی، یا اگر چندین پردازه وجود داشته باشد، به معنای واقعی موازی) اجرا شوند.

۷.۳ pthread

بر اساس تاریخ، سازندگان سخت‌افزار نسخه‌ی مناسبی از ریسه‌ها را برای خود پیاده‌سازی کردند. از آنجا که این پیاده‌سازی‌ها با هم تفاوت می‌کرد، پس کار را برای برنامه نویسان، برای نگارش یک برنامه‌ی قابل حمل دشوار می‌کرد. بنابراین نیاز به داشتن یک واسط یکسان برای بهره بردن از فواید ریسه‌ها احساس می‌شد. برای سیستم های Unix این واسط با نام IEEE 1003.1c (POSIX) مشخص می‌شد و به پیاده‌سازی مرتبط با آن POSIX THREADS یا pthread گفته می‌شود. اکثر سازندگان سخت‌افزار، علاوه بر نسخه‌ی مناسب با خودشان، استفاده از pthread را نیز پیشنهاد می‌کنند. pthreadها در یک کتابخانه‌ی C تعریف شده‌اند که شما می‌توانید با برنامه‌ی خود link کنید.

توابع موجود در این کتابخانه به صورت غیررسمی به چند دسته تقسیم می‌شوند مانند:

- مدیریت ریسه‌ها: دسته‌ی اول از این توابع به صورت مستقیم با ریسه‌ها کار می‌کنند. همانند ایجاد، متصل کردن و ...
- Mutex: دسته‌ی دوم از این تابع برای کار با mutex ها ایجاد شده‌اند. توابع مربوط به mutex ابزار مناسب برای ایجاد، تخریب، قفل و بازکردن mutex ها را در اختیار قرار می‌دهند.
- متغیرهای شرطی (Condition Variables): این دسته از توابع، برای کار با متغیرهای شرطی و استفاده از مفهوم همزمانی در سطح بالاتر در اختیار قرار می‌گیرند. این دسته از توابع برای ایجاد، تخریب، wait و signal بر اساس مقادیر معین متغیرها استفاده می‌شوند.

نکته ۱ در این جلسه قصد آن را داریم تا با دسته‌ی اول از توابع آشنا شویم. توابع مربوط به این دسته به طور خلاصه در جدول زیر مشاهده می‌شود: برای آشنایی با جزییات می‌توانید از دستور `man` و یا اینترنت استفاده کنید.

جدول ۷.۱ :توابع مربوط به مدیریت ریسه ها

نام تابع	کاربرد
`pthread_create`	از کتابخانه‌ی pthread، درخواست ساخت یک ریسه‌ی جدید را می‌کند.
`pthread_exit`	این تابع توسط ریسه استفاده شده تا پایان بپذیرد.

نام تابع	کاربرد
`pthread_join`	این تابع، برای ریسه‌ی مشخص شده صبر می‌کند تا پایان بپذیرد.
`pthread_cancel`	درخواست کنسل شدن ریسه‌ی مشخص شده را ارسال می‌کند.
`pthread_attr_init`	مقدار attribute های پاس داده شده به خود را با مقادیر پیش‌فرض پر می‌کند.
`pthread_self`	شماره‌ی ریسه را بر می‌گرداند.

۷.۴ شرح آزمایش

۷.۴.۱ آشنایی اولیه

1. وارد سیستم عامل مجازی ایجاد شده در جلسات قبل شوید.
2. با استفاده از تکه کد زیر، یک ریسه ایجاد کرده و خروجی را مشاهده کنید.

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *child(void *arg) {
    puts("Hello from child thread!");
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, child, NULL);
    sleep(1);
    return 0;
}
```

دقت کنید در هنگام کامپایل، `-lpthread` را به پرچم‌های linker اضافه کنید:

```
gcc thread.c -o threads -lpthread
```

3. برنامه را طوری تغییر دهید که به جای sleep با استفاده از تابع `pthread_join` منتظر به پایان رسیدن ریسه فرزند شود. علاوه بر این هم در ریسه فرزند و ریسه اصلی شماره‌ی پردازه (pid) را چاپ کنید. دقت کنید که پردازه‌ی اصلی باید بعد از پایان یافتن ریسه‌های فرزند تمام شود. آیا شماره پردازه‌های چاپ شده یکسان می‌باشند؟

4. برنامه‌ی بالا را در یک فایل جدید کپی کنید. حال، متغیر oslab را به این تکه کد به صورت سراسری اضافه کنید. حال، یک بار این متغیر را در ریسه‌ی اصلی و یک بار در ریسه‌ی فرزند تغییر دهید. بعد از تغییر در ریسه‌ی فرزند، بار دیگر در ریسه‌ی اصلی چاپ کنید. آیا ریسه‌ها کپی‌های جداگانه‌ای از متغیر را دارند؟

5. با استفاده از تابع `pthread_attr_init` و تنظیم کردن attributeهای ریسه به صورت پیش‌فرض، کدی بنویسید که در آن با گرفتن عدد n از ورودی، در ریسه‌ی فرزند حاصل جمع اعداد ۲ تا n را چاپ کند.

۷.۴.۲ ریسه‌های چندتایی

در این قسمت قصد آن را داریم تا در برنامه، چند ریسه داشته باشیم.

- با استفاده از تابع `pthread_create` تعدادی ریسه به تعداد دلخواه ایجاد کنید (حداقل پنج تا) و پیام Hello World را در آن چاپ کنید. سپس ریسه‌ها را با استفاده از تابع pthread_exit خاتمه دهید.

۷.۴.۳ تفاوت بین پردازه‌ها و ریسه‌ها

در این قسمت، قصد آن را داریم تا تفاوت میان پردازه‌ها و ریسه‌ها را بهتر متوجه شویم.

1. تکه کد زیر را به عنوان تابع ریسه در فایلی بنویسید:

دقت کنید در هنگام کامپایل، `-lpthread` را به پرچم های linker اضافه کنید.

```
void *child(void *arg) {
    int local_var;

    printf("Thread %ld, pid %d, addresses: &global: %p, &local: %p \n",
        pthread_self(), getpid(), &global_var, &local_var);

    global_var++;

    printf("Thread %ld, pid %d, incremented global var=%d\n",
        pthread_self(), getpid(), global_var);

    pthread_exit(0);
}
```

2. حال، در ریسه‌ی اصلی، یک متغیر عمومی به عنوان `global_var` تعریف کرده، مقداردهی کنید و دو ریسه‌ی فرزند با تابع child ایجاد و اجرا کنید. در پایان ریسه‌ها نیز مقدار متغیر `global_var` را چاپ کنید. آیا مفدار این متغیر تغییر کرده است؟

3. سپس، در تابع `main` مقدار متغیر `global_var` را بار دیگر تغییر داده و یک متغیر محلی جدید تعریف کرده و مقداردهی کنید. حال، با استفاده از تابع fork که در جلسات پیش یاد گرفته‌اید، یک پردازه‌ی فرزند ایجاد کرده و متغیرها را در آن دوباره مقداردهی کنید. در نهایت مقدار این متغیرها را با استفاده از تابع `printf` نمایش دهید. آیا مقادیر با هم متفاوت هستند؟

۷.۴.۴ پاس دادن متغیرها به ریسه‌ها

تابع `pthread_create` تنها اجازه می‌دهد که یک متغیر به عنوان ورودی به ریسه داده شود. برای حالاتی که چند پارامتر می‌بایست به ریسه داده شود، این محدودیت به راحتی با استفاده از ساختار (structure) حل می‌شود. تمامی متغیرها می‌بایست به وسیله‌ی reference و تبدیل به `void*` پاس داده شوند.

ساختار زیر را در فایلی قرار دهید:

```
typedef struct thdata {
    int thread_no;
    char message[100];
} stdata;
```

حال، در ریسه‌ی اصلی، دو متغیر از ساختار معرفی شده ایجاد کنید و مقادیر آن را به صورت دلخواه تنظیم کنید. سپس، متغیرها را به دو ریسه‌ی جداگانه پاس بدهید. در ریسه‌ها نیز عدد و پیام ذخیره شده در ساختار را نمایش بدهید.