دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

به نام خدا

پاسخ تمرین سری پنجم درس سیستم های عامل

پاییز 1403

استاد درس: دکتر زرندی

سوال اول)

به سوالات زیر پاسخ دهید.

الف) مزایای استفاده از ریسمان ها در مقابل فرایند ها چیست؟

- **پاسخگویی(Responsiveness): امکان اجرای ادامه برنامه حتی زمانی که اجرای بخشی از آن بلاک شده و یا طولانی است.**
- **اشتراک منابع: حافظه اصلی، I/O، داده**
- **اقتصادی تر است: سرعت بیشتر دارد، در نتیجه با یک سخت افزار مشخص کارایی بهتری دریافت میکنیم**
- **مقیاس‌پذیری: ریسمان‌ها میتوانند به صورت موازی روی هسته‌های پردازشی اجرا شوند**

ب) taskها می توانند به دو صورت موازی و هم روند اجرا شوند.تفاوت این دو روش را توضیح دهید.

- **موازی سازی به این معنی است که یک سیستم می تواند بیش از یک وظیفه را به طور دقیقا همزمان انجام دهد(با یک واحد پردازشی امکان پذیر نیست).**
- **Concurrency به توانایی سیستم برای مدیریت و اجرای چندین وظیفه و پردازش مجزا به صورت همروند گفته می‌شود(با یک واحد پردازشی هم امکان پذیر است). هر واحد مقداری از یک وظیفه را انجام داده و سپس مقداری از وظیفه های دیگر را انجام میدهد و دوباره به پردازش ادامه وظیفه های قبلی میپردازد.**

ج) انواع مدل های چندریسمانی را نام برده و توضیح دهید.

- **مدل چند به یک: چندین ریسمان سطح کاربر به یک ریسمان سطح کرنل مپ میشود. در صورتی که یکی از ریسمان های کاربر بلاک شود، تمامی بقیه ریسمان های سطح کاربری که به همان ریسمان سطح کرنل مپ شده بودند نیز بلاک میشوند. چند ریسمان سطح کاربر نمیتوانند در پردازنده های چند هسته ای به صورت موازی اجرا شوند، زیرا تنها یکی از آنها در هر لحظه میتواند در کرنل باشد.**
- **مدل یک به یک: هر ریسمان سطح کاربر به یک ریسمان سطح کرنل مپ میشود. با ساخت هر ریسمان سطح کاربر، یک ریسمان کرنل نیز ساخته میشود. به نسبت مدل چند به یک، همروندی بیشتری دارد. تعداد ریسمان‌های هر پردازه ممکن است به علت سربار محدود شوند.**
- **مدل چند به چند: چندین ریسمان سطح کاربر میتوانند به چندین ریسمان سطح کرنل مپ شوند. سیستم عامل را قادر میسازد تا به تعداد بهینه ای ریسمان سطح کرنل بسازد.**

- **مدل دو سطحی: همانند مدل چند به چند است با این تفاوت که امکان محدود کردن یک ریسمان سطح کاربر به یک ریسمان کرنل را نیز دارد.**

د) انواع حالت وضعیت ریسمان‌ها را نام برده و هرکدام را توضیح دهید.

- **تولید(Spawn): هنگامی که یک فرایند جدید ایجاد می شود، یک ریسمان برای آن فرآیند نیز ایجاد می شود.**
- **مسدود(Block): زمانی که یک رشته باید منتظر یک رویداد باشد، مسدود می شود.**
- **نامسدود(Unblock): هنگامی که رویدادی رخ می دهد که رشته برای آن رویداد مسدود شده بود، رشته به صف آماده(Ready Queue) منتقل می شود.**
- **پایان(Finish): هنگامی که کار یک رشته تمام می شود، رجیسترهای زمینه(Context) و فضای پشته آن آزاد(Deallocate) میشود.**

ه) Thread-local storage (TLS) چیست و چه مواقعی کاربرد دارد؟ تفاوت آن با متغیرهای داخلی را شرح دهید.

ذخیره‌سازی محلی (TLS) به هر رشته اجازه می‌دهد تا نسخه‌ای از داده‌های خود را داشته باشد

زمانی مفید است که کنترلی بر فرآیند ایجاد ریسمان ندارید (به عنوان مثال، هنگام استفاده از یک (Thread Pool

تفاوت آن با متغیرهای محلی:

- **متغیرهای محلی فقط در طول فراخوانی یک تابع قابل مشاهده هستند**
- **TLS در فراخوانی میان توابع مختلف قابل مشاهده است(مانند داده‌های static) و برای هر ریسمان منحصر به فرد است.**

و) انواع روش‌های thread termination را نام برده و هرکدام را مختصر توضیح دهید.

- **Asynchronous cancellation: می تواند در هر زمانی رخ دهد و ریسمان را terminate کند.**
- **Deferred cancellation: فقط در نقاط از پیش تعریف شده میتواند ریسمان را terminate کند.**

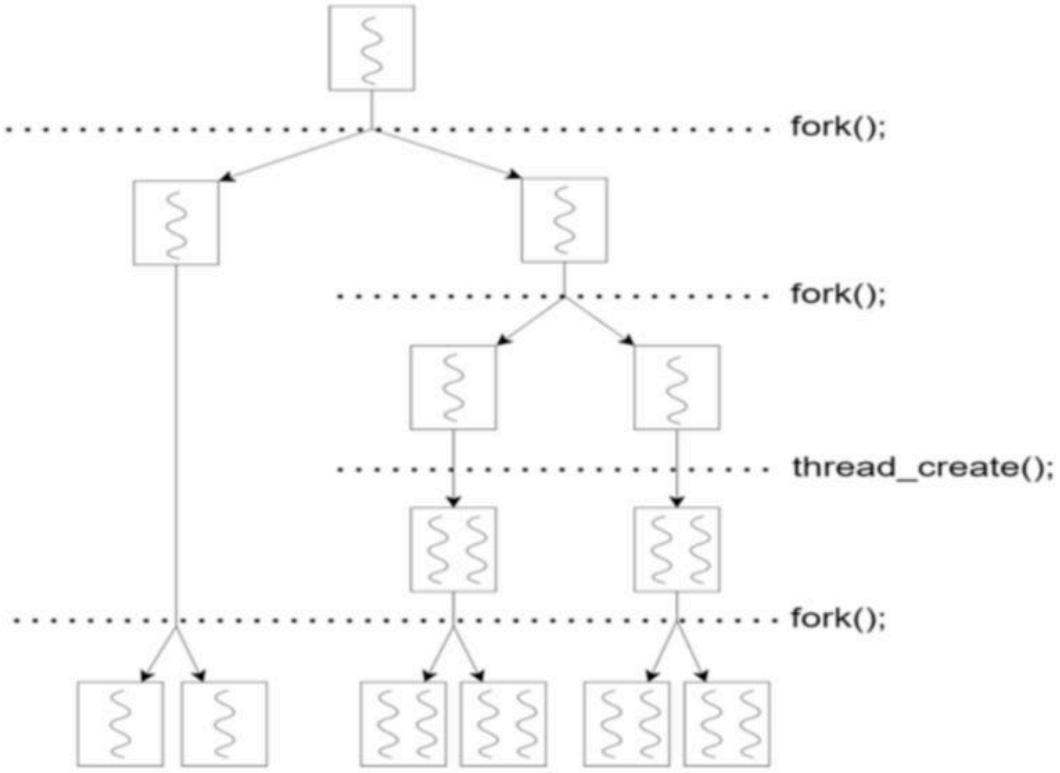
سوال دوم)

کد زیر را در نظر بگیرید. تابع create_thread() یک ریسمان جدیدی را در فرایند فراخوانی شروع می کند. چند فرآیند منحصر به فرد ایجاد می شود؟ چه تعداد رشته منحصر به فرد ایجاد می شود؟

```
pid_t pid;
pid = fork();
if (pid == 0) { // Child process
    fork();
    thread_create( . . . );
}
fork();
```

این قطعه کد ۶ فرآیند که شامل خود فرآیند اصلی نیز میباشد به همراه ۱۰ ریسمان تولید میکند(طبق فرض اینکه فراخوانی fork ریسمان های فرآیند صدازننده را نیز کپی میکند. بدون این فرض این کد همین تعداد فرآیند و ۸ ریسمان تولید میکند).

- **صدا زدن fork اول یک فرآیند دیگر ایجاد میکند.**
- **صدا زدن fork دوم که داخل if قرار دارد تنها توسط فرآیند فرزند که از fork اول ساخته شده است صدا زده میشود.**
- **حال این دو فرآیند تابع thread_create را صدا میزنند و در این لحظه ما ۳ فرآیند و ۵ ریسمان خواهیم داشت.**
- **هر ۳ فرآیند، fork آخر را صدا میزنند و این یعنی ۳ فرآیند ما به ۶ فرآیند و ۵ ریسمانمان به ۱۰ ریسمان تبدیل خواهد شد.**



سوال سوم)

الف) race condition چه مواقعی پیش میآید و باعث چه مشکلی میشود؟ چطور میتوان از آن جلو گیری کرد؟

چندین فرآیند به طور همزمان به داده های یکسان دسترسی دارند و آنها را دستکاری می کنند. این باعث ناسازگاری داده میشود و نهایی داده ها به ترتیبی که دسترسی انجام می شود بستگی دارد.

راه حل جلوگیری از مشکل race condition اجرای ترتیبی پردازه هایی که با یک داده سروکار دارند است.

ب) در قطعه کد زیر توضیح دهید race condition در کدام قسمت ممکن است به وجود بیاید و یک سناریو که باعث ناسازگاری داده می شود مثال بزنید.

```
int shared_counter = 0;

void* increment_counter(void* arg) {
    for (int i = 0; i < 1000000; ++i) {
        shared_counter++;
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final value of shared_counter: %d\n",
shared_counter);
    return 0;
}
```

در کد فوق، متغیر مشترک shared_counter توسط دو thread به طور همزمان به روزرسانی می‌شود. این امر می‌تواند منجر به بروز race condition شود. مشکل از اینجا ناشی می‌شود که عملیات افزایش (shared_counter++) ذاتاً شامل چند مرحله است:

1 . خواندن مقدار فعلی از shared_counter.
2. افزایش مقدار خوانده شده.
3 . نوشتن مقدار جدید به shared_counter.

وقتی چند thread به طور همزمان این عملیات را انجام می‌دهند، به دلیل عدم هماهنگی (synchronization)، ممکن است یک thread مقدار قدیمی را بخواند و دیگری قبل از نوشتن مقدار جدید، مقدار را بازنویسی کند. این باعث ناسازگاری داده و نتیجه نهایی نادرست می‌شود.

سناریوی وقوع Race Condition

فرض کنید مقدار اولیه shared_counter برابر با 0 است. اگر هر دو thread به طور همزمان شروع به اجرا کنند، مراحل زیر ممکن است رخ دهد:

1 . Thread 1 مقدار shared_counter را می‌خواند (0).
2 . Thread 2 مقدار shared_counter را می‌خواند (0).
3. Thread 1 مقدار جدید (1) را محاسبه می‌کند اما هنوز آن را نمی‌نویسد.
4 . Thread 2 مقدار جدید (1) را محاسبه می‌کند و آن را در shared_counter می‌نویسد.
5 . Thread 1 مقدار 1 را که قبلاً محاسبه کرده بود در shared_counter می‌نویسد، که مقدار فعلی (1) را بازنویسی می‌کند.

در نتیجه، مقدار shared_counter به جای افزایش به 2، همچنان برابر 1 باقی می‌ماند.