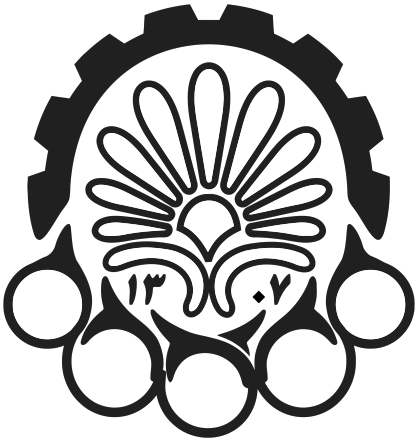
سیستم‌های عامل

دکتر زرندی



دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

رضا آدینه پور ۴۰۲۱۳۱۰۵۵

تمرین سری پنجم

۱۳ آبان ۱۴۰۳

سیستم‌های عامل

تمرین سری پنجم

رضا آدینه پور ۴۰۲۱۳۱۰۵۵

دانشکده مهندسی کامپیوتر

سوال اول

به سوالات زیر پاسخ دهید.

۱. مزایای استفاده از ریسمان‌ها در مقابل فرایندها چیست؟

پاسخ

(آ) **اشتراک منابع:** ریسمان‌ها در یک فرایند به حافظه و منابع مشترکی دسترسی دارند، مانند فضای آدرس، فایل‌ها و متغیرهای سراسری. این امر ارتباط و انتقال داده‌ها بین ریسمان‌ها را ساده‌تر و سریع‌تر از فرایندها می‌کند، چرا که نیازی به کپی کردن داده‌ها بین فضای آدرس‌های جداگانه نیست.

(ب) **کارایی بالاتر و سربار کمتر:** ایجاد و ازبین بردن ریسمان‌ها به‌مراتب کم‌هزینه‌تر از فرایندها است. به دلیل این که ریسمان‌ها فضای آدرس مشترک دارند، سیستم‌عامل برای سوئیچ کردن بین ریسمان‌ها نیازی به تغییر کامل فضای آدرس (Context Switching) ندارد. این امر سربار کمتری روی سیستم ایجاد کرده و کارایی را افزایش می‌دهد.

(ج) **بهبود عملکرد در سیستم‌های چندپردازنده‌ای:** ریسمان‌ها می‌توانند به‌طور هم‌زمان در پردازنده‌های مختلف اجرا شوند. با استفاده از ریسمان‌ها، یک برنامه می‌تواند از چندین پردازنده به‌طور مؤثرتری بهره ببرد و کارایی کلی افزایش یابد.

(د) **پشتیبانی بهتر از هم‌روندی (Concurrency):** ریسمان‌ها امکان اجرای موازی چندین تسک را در یک برنامه فراهم می‌کنند، که این موضوع به‌ویژه برای برنامه‌های بلادرنگ و برنامه‌هایی که به پاسخگویی سریع نیاز دارند، مفید است.

(ه) **صرفه‌جویی در حافظه:** ریسمان‌ها به دلیل اشتراک منابع، به فضای کمتری نسبت به فرایندها نیاز دارند. به این ترتیب، در استفاده از حافظه سیستم صرفه‌جویی می‌شود و امکان اجرای تعداد بیشتری از ریسمان‌ها وجود دارد.

۲. وظایف (tasks) می‌توانند به دو صورت موازی و هم‌روند اجرا شوند. تفاوت این دو روش را توضیح دهید.

پاسخ

(آ) **اجرای موازی:**

- در این روش، وظایف به‌طور واقعی و هم‌زمان روی چندین پردازنده یا هسته اجرا می‌شوند.
- برای انجام تسک‌ها به‌صورت موازی، نیاز به سخت‌افزاری داریم که از چندین پردازنده یا هسته پشتیبانی کند، مثل پردازنده‌های چندهسته‌ای یا سیستم‌های چندپردازنده‌ای.

ادامه پاسخ

- در موازی‌سازی، هر وظیفه در یک پردازنده یا هسته جداگانه اجرا شده و می‌تواند هم‌زمان پیش برود. این روش به‌ویژه برای پردازش‌های سنگین و وظایفی که به سرعت بالا نیاز دارند، مناسب است. برای مثال، یک سیستم پردازش تصویر می‌تواند از موازی‌سازی استفاده کند تا هر هسته پردازنده بخشی از تصویر را به‌صورت مستقل پردازش کند.

(ب) **اجرای هم‌روند:**

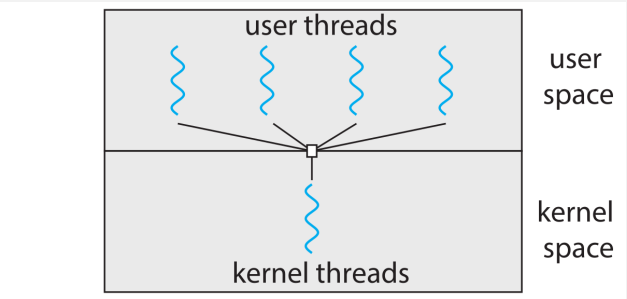
- در این روش، وظایف به‌صورت تکه‌تکه اجرا شده و به‌صورت متناوب به یکدیگر سوئیچ می‌کنند؛ یعنی هر وظیفه بخشی از زمان پردازنده را می‌گیرد، سپس وظیفه دیگری اجرا می‌شود.
- هم‌روندی نیاز به پردازنده‌های چندهسته‌ای ندارد و می‌تواند حتی در یک پردازنده‌ی تک‌هسته‌ای نیز اجرا شود. در این حالت، پردازنده بین وظایف مختلف جابجا می‌شود تا احساس هم‌زمانی به وجود بیاید.
- هم‌روندی معمولاً برای وظایفی که به‌طور هم‌زمان به منابع مختلف نیاز دارند، مانند ورودی/خروجی و محاسبات، کاربرد دارد. برای مثال، یک برنامه چت می‌تواند پیام‌های ورودی و خروجی را به‌طور هم‌روند مدیریت کند، حتی اگر تنها یک پردازنده در سیستم وجود داشته باشد.

۳. انواع مدل‌های چندریسمانی را نام برده و توضیح دهید.

پاسخ

(آ) **مدل Many-to-One**

در این مدل، چندین ریسمان سطح کاربر به یک ریسمان سطح هسته نگاشت می‌شوند. مدیریت ریسمان‌ها در سطح کتابخانه‌های ریسمان کاربر انجام می‌شود و سیستم‌عامل نیازی به دخالت مستقیم در مدیریت هر ریسمان ندارد. عیب این مدل در این است که اگر یکی از ریسمان‌ها نیاز به مسدود شدن داشته باشد (مثلاً در حال انتظار برای I/O)، تمام ریسمان‌ها مسدود می‌شوند زیرا همه به یک ریسمان هسته وابسته هستند. این مدل در برخی از سیستم‌های تک‌پردازنده‌ای به کار می‌رود زیرا پیاده‌سازی آن ساده‌تر است. تصویری از این مدل در ادامه آورده شده است:

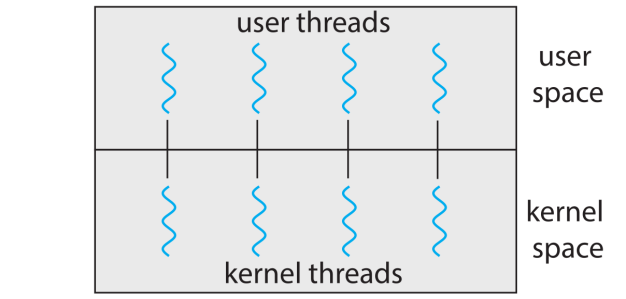


شکل ۱: مدل Many-to-One

ادامه پاسخ

(ب) **مدل One-to-One**

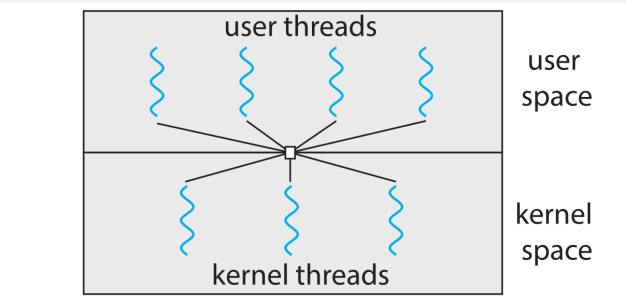
در این مدل، هر ریسمان کاربر به یک ریسمان هسته مرتبط می‌شود. هر ریسمان می‌تواند به‌صورت مستقل اجرا شود و از چندپردازنده‌ها نیز به خوبی پشتیبانی می‌کند؛ یعنی ریسمان‌ها می‌توانند در پردازنده‌های مختلف اجرا شوند. این مدل اجازه می‌دهد که ریسمان‌ها به‌صورت مستقل از یکدیگر به پردازنده‌ها اختصاص داده شوند، که بهبود عملکرد را در پی دارد. نقطه ضعف این مدل این است که ایجاد هر ریسمان کاربر به معنای تخصیص یک ریسمان هسته‌ای است، که ممکن است منابع زیادی را مصرف کند و در صورت افزایش تعداد ریسمان‌ها سربار زیادی ایجاد کند. در دامه تصویری از این مدل آورده شده است:



شکل ۲: مدل One-to-One

(ج) **مدل Many-to-Many**

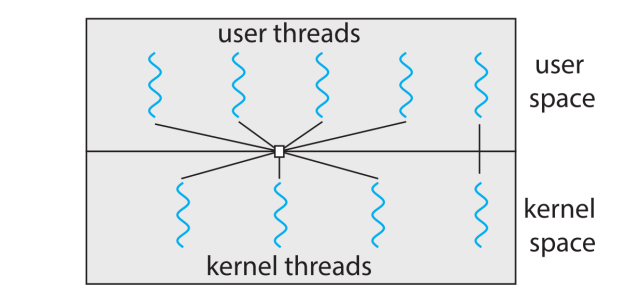
در این مدل، چندین ریسمان کاربر می‌توانند به چندین ریسمان هسته نگاشت شوند. این مدل انعطاف‌پذیری بیشتری دارد و به ریسمان‌ها این امکان را می‌دهد که به‌صورت موازی در چندین پردازنده اجرا شوند. در این مدل، کتابخانه‌های ریسمان می‌توانند تعداد زیادی ریسمان کاربر را ایجاد کنند بدون این که سربار زیادی برای سیستم‌عامل ایجاد شود، زیرا فقط تعداد محدودی از این ریسمان‌ها به ریسمان‌های هسته نگاشت می‌شوند. این مدل اجازه می‌دهد که اگر یکی از ریسمان‌ها مسدود شد، بقیه ریسمان‌ها بتوانند به اجرا ادامه دهند و کارایی سیستم کاهش نیابد. شکل زیر مثالی از این مدل است:



شکل ۳: مدل Many-to-Many

ادامه پاسخ

(د) **مدل two-level**
این مدل ترکیبی از مدل Many-to-Many و One-to-one است. برخی از ریسمان‌های کاربر می‌توانند به یک ریسمان هسته‌ای مستقل نگاشت شوند (مانند مدل یک به یک)، در حالی که سایر ریسمان‌ها به یک یا چند ریسمان هسته‌ای مشترک نگاشت می‌شوند. این مدل امکان انعطاف‌پذیری بیشتری در تخصیص و مدیریت ریسمان‌ها فراهم می‌کند و به برنامه‌های خاص اجازه می‌دهد که از هر دو روش استفاده کنند. این مدل در سیستم‌هایی به کار می‌رود که نیاز به ترکیب ویژگی‌های هر دو مدل برای بهینه‌سازی کارایی و استفاده از منابع دارند. شکل زیر مثالی از این مدل است:



شکل ۴: مدل Two-level

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

پاسخ

(آ) **حالت (Ready):**
در این حالت، ریسمان برای اجرا آماده است و تمامی منابع لازم (به جز پردازنده) را دارد. ریسمان در صف آماده قرار دارد و منتظر تخصیص پردازنده است تا اجرای آن آغاز شود. ریسمان می‌تواند پس از آزاد شدن پردازنده از حالت آماده به حالت اجرا منتقل شود.

(ب) **حالت (Running):**
در این حالت، ریسمان به پردازنده اختصاص داده شده و در حال اجرای دستورالعمل‌های خود است. یک ریسمان در هر لحظه فقط زمانی می‌تواند در حالت اجرا باشد که به پردازنده دسترسی داشته باشد. اگر اجرای ریسمان به دلیل اتمام زمان پردازنده یا درخواست منابع دیگر متوقف شود، ریسمان به حالت دیگری منتقل می‌شود.

(ج) **حالت (Blocked):**
زمانی که ریسمان نیاز به دسترسی به منبعی دارد که در حال حاضر در دسترس نیست (مثل یک عملیات ورودی/خروجی یا انتظار برای قفل)، به حالت مسدود می‌رود. در این حالت، ریسمان نمی‌تواند اجرا شود و باید منتظر بماند تا منبع مورد نیازش آزاد شود. هنگامی که منبع در دسترس قرار گرفت، ریسمان از حالت مسدود به حالت آماده باز می‌گردد.

(د) **حالت (Terminated):**
زمانی که ریسمان تمام وظایف خود را به پایان رسانده و دیگر نیازی به اجرا ندارد، به حالت پایان‌یافته می‌رود. در این حالت، منابعی که توسط ریسمان استفاده شده بودند آزاد می‌شوند و ریسمان از چرخه‌ی پردازش حذف می‌شود. این حالت نشان می‌دهد که چرخه عمر ریسمان به پایان رسیده است.

ادامه پاسخ

(ه) **حالت در انتظار (Waiting) یا خواب (Sleep):**
در برخی سیستم‌ها، حالت در انتظار یا خواب به عنوان یک حالت جداگانه برای زمانی که ریسمان منتظر یک رویداد خاص (مانند یک سیگنال) است، تعریف می‌شود. در این حالت، ریسمان تا زمانی که رویداد مورد نظر رخ ندهد، اجرا نخواهد شد. پس از وقوع رویداد، ریسمان به حالت آماده باز می‌گردد تا در صف اجرا قرار گیرد.

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

پاسخ

Thread-local storage (TLS) نوعی مکانیزم ذخیره‌سازی است که به هر نخ امکان می‌دهد متغیرهای خود را به صورت مجزا و مستقل از سایر نخ‌ها ذخیره کند. این ویژگی در مواقعی استفاده می‌شود که بخواهیم داده‌هایی را که تنها برای یک نخ خاص معتبر هستند، ذخیره کنیم تا نخ‌های دیگر به آن دسترسی نداشته باشند و تداخلی ایجاد نشود. این کار از اشتراک‌گذاری ناخواستهٔ داده‌ها میان نخ‌ها جلوگیری می‌کند.
از تفاوت‌های آن با متغیرهای داخلی می‌توان به موارد زیر اشاره نمود:

(آ) دامنهٔ دسترسی: متغیرهای داخلی یا محلی فقط در دامنهٔ توابعی که تعریف شده‌اند معتبر هستند و وقتی از آن تابع خارج شویم، این متغیرها دیگر وجود ندارند. در مقابل، متغیرهای TLS برای کل دورهٔ حیات نخ موجود و معتبر هستند.

(ب) محدودیت به نخ: متغیرهای TLS برای هر نخ به صورت جداگانه اختصاص داده می‌شوند، در حالی که متغیرهای داخلی توسط هر بار فراخوانی تابع در پشته ایجاد می‌شوند و فقط در آن دامنهٔ خاص معتبر هستند.

(ج) پایداری: متغیرهای TLS در طول اجرای نخ پایدار هستند و با پایان یافتن نخ از بین می‌روند، در حالی که متغیرهای محلی با خروج از بلوک کد یا تابعی که در آن تعریف شده‌اند، پاک می‌شوند.

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

پاسخ

(آ) **Voluntary Termination:**
در این روش، نخ به صورت خودخواسته و از طریق اجرای دستورات برنامه، به پایان می‌رسد. این حالت معمولاً زمانی اتفاق می‌افتد که نخ کار خود را به پایان رسانده و نیاز به ادامهٔ اجرا ندارد. برنامه‌نویس می‌تواند با فراخوانی تابعی مثل `pthread_exit` در `POSIX` یا `ExitThread` در ویندوز، نخ را به صورت ایمن خاتمه دهد.

(ب) **Forced Termination:**
در این روش، نخ بدون اطلاع و خواست خود توسط نخ دیگری خاتمه می‌یابد. برای مثال، نخ مادر می‌تواند یک نخ فرزند را به اجبار متوقف کند. این روش ممکن است موجب ناپایداری برنامه شود، زیرا ممکن است نخ در حال اجرای کدی مهم باشد و به طور ناگهانی متوقف شود. توابعی مانند `pthread_cancel` در `POSIX` برای این نوع خاتمه استفاده می‌شوند.

ادامه پاسخ

(ج) **Termination Due to Error:**
زمانی که خطایی در نخ رخ دهد و ادامه اجرای نخ بی‌معنی شود، نخ خاتمه می‌یابد. برای مثال، اگر نخ به منابعی دسترسی نداشته باشد یا به یک وضعیت بحرانی برسد که نمی‌تواند از آن عبور کند، ممکن است خاتمه یابد.

(د) **System Termination:**
این نوع خاتمه زمانی اتفاق می‌افتد که سیستم یا پردازش اصلی که نخ در آن اجرا می‌شود، پایان یابد. با خاتمهٔ پردازش اصلی، تمام نخ‌های مربوط به آن پردازش نیز به پایان می‌رسند.

سوال دوم

کد زیر را در نظر بگیرید. تابع create_thread() یک ریسمان جدیدی را در فرایند فراخوانی شروع می‌کند. چند فرایند منحصر به فرد ایجاد می‌شود؟ چه تعداد رشته منحصر به فرد ایجاد می‌شود؟ توضیح دهید.

```
1 pid_t pid;
2 pid = fork();
3 if (pid == 0)
4 { /* Child process */
5     fork();
6     thread_create(...);
7 }
8 fork();
```

Listing 1: Code of Q2

پاسخ

این کد درمجموع، ۶ فرآیند و دو نخ ایجاد می‌کند که توضیحات آن را در ادامه می‌دهیم.
در اولین فراخوانی fork() که در خط ۲ انجام می‌شود، یک کپی از Process اصلی ایجاد می‌شود. تا اینجا ۲ Process داریم.
در فراخوانی دوم که در خط ۵ اتفاق می‌افتد، تنها توسط Process فرزند که حاصل از فراخوانی اول است اجرا می‌شود. تا اینجا ۳ Process داریم.
اکنون دو Process در حال اجرای کد داخل شرط if هستند، به این معنی که هر دو این فرآیندها thread_create() را فراخوانی می‌کنند (در این نقطه ۳ فرآیند و ۲ نخ داریم).
البته نکته‌ای که باید در صورت این مسئله واضح‌تر بیان می‌شد این است که هر نخ تازه ایجاد شده یک تابع متفاوت از تابعی که هم‌اکنون در حال اجرا است، شروع می‌کند. یکی از آرگومان‌های تابع thread_create() اشاره‌گری به تابعی است که باید اجرا شود. بنابراین نخ‌ها به فراخوانی fork() آخر نمی‌رسند.
هر سه Process فراخوانی نهایی به fork() را در خط ۸ اجرا می‌کنند، بنابراین هر Process در این نقطه خود را کپی می‌کند (در مجموع ۶ فرآیند و ۲ نخ داریم).
البته با توجه به اینکه هر Process به عنوان یک single-thread آغاز می‌شود، شاید بهتر باشد بگوییم این قطعه کد در مجموع شش Process و هشت نخ ایجاد می‌کند (دو نخ توسط فراخوانی‌های thread_create() و شش نخ که مربوط به شش فرآیند تک‌نخی هستند).

سوال سوم

۱. race condition چه مواقعی پیش می‌آید و باعث چه مشکلی می‌شود؟ چطور می‌توان از آن جلوگیری کرد؟

پاسخ

Race condition زمانی رخ می‌دهد که دو یا چند نخ یا فرآیند به طور همزمان به یک منبع مشترک (مانند یک متغیر یا فایل) دسترسی پیدا کرده و تلاش کنند عملیات خواندن یا نوشتن را روی آن انجام دهند. اگر دسترسی همزمان به منبع به درستی مدیریت نشود، ممکن است نتایج ناخواسته و غیرقابل پیش‌بینی به وجود آید.
از مشکلات ایجاد شده توسط Race Condition می‌توان به موارد زیر اشاره کرد:

(آ) **نتایج نادرست:** چون عملیات نخ‌ها به صورت غیرقابل پیش‌بینی انجام می‌شود، ممکن است نتیجه نهایی برنامه اشتباه باشد. مثلاً، اگر دو نخ همزمان مقدار یک متغیر را افزایش دهند، نتیجه ممکن است نادرست باشد، زیرا هر کدام مقدار قدیمی را می‌خوانند و تغییر می‌دهند، بدون توجه به تغییرات دیگری.

(ب) **ناپایداری در برنامه:** وجود condition race می‌تواند باعث رفتارهای ناپایدار و سخت برای پیش‌بینی در برنامه شود که دیباگ کردن و رفع مشکلات را بسیار دشوار می‌کند.

برای جلوگیری از این مشکل باید دسترسی نخ‌ها و فرآیندها به منابع مشترک را مدیریت کرد. روش‌های رایج عبارتند از:

(آ) **قفل‌ها (Locks):** استفاده از قفل‌ها (مثل mutex) مانع از دسترسی همزمان نخ‌ها به منابع مشترک می‌شود. هر نخ باید قبل از دسترسی به منبع قفل را بگیرد و پس از اتمام کار آن را آزاد کند. این کار تضمین می‌کند که فقط یک نخ در هر لحظه به منبع دسترسی داشته باشد.

(ب) **متغیرهای شرطی (Condition Variables):** متغیرهای شرطی به نخ‌ها اجازه می‌دهند منتظر شوند تا شرط خاصی برقرار شود. این روش به نخ‌ها کمک می‌کند تا فقط در زمان مناسب به منبع دسترسی پیدا کنند و تداخلات را کاهش می‌دهد.

(ج) **سمافور (Semaphores):** سمافورها مکانیزم دیگری برای هماهنگ کردن دسترسی به منابع مشترک هستند و به ویژه در برنامه‌هایی که به دسترسی همزمان محدود به تعداد خاصی از نخ‌ها نیاز دارند، کاربرد دارند.

(د) **بخش‌های بحرانی (Critical Sections):** استفاده از بخش‌های بحرانی به برنامه‌نویس اجازه می‌دهد تا کدی را که نیاز به دسترسی انحصاری دارد، به عنوان یک بخش بحرانی تعریف کند و فقط یک نخ در هر زمان اجازه ورود به این بخش را داشته باشد.

(ه) **استفاده از ساختارهای داده‌ای همگام‌سازی شده:** بسیاری از زبان‌های برنامه‌نویسی ساختارهای داده‌ای همگام‌سازی‌شده‌ای مثل صف و پشته را ارائه می‌دهند که به طور خودکار از تداخل‌های نخ‌ها جلوگیری می‌کنند.

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

```c
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;
}
```

Listing 2: Code of Q2

پاسخ

در قطعه کد ارائه‌شده Race Condition می‌تواند در بخش دسترسی و تغییر مقدار متغیر مشترک `shared_counter` در تابع `increment_counter` رخ دهد. دلیل این مشکل این است که دو Thread به صورت همزمان و بدون هماهنگی به متغیر مشترک دسترسی دارند و آن را افزایش می‌دهند.
در این برنامه، دو رشته (`thread1` و `thread2`) به صورت همزمان اجرا می‌شوند و هرکدام یک میلیون بار مقدار `shared_counter` را افزایش می‌دهند. هر رشته زمانی که به خط `shared_counter++` می‌رسد، باید مراحل زیر را انجام دهد:

(آ) مقدار فعلی `shared_counter` را بخواند.

(ب) یک واحد به مقدار خوانده‌شده اضافه کند.

(ج) مقدار جدید را در `shared_counter` ذخیره کند.

از آنجایی که این مراحل به صورت مجزا انجام می‌شوند، ممکن است دو رشته همزمان مقدار `shared_counter` را بخوانند، آن را افزایش دهند و مقدار جدید را در متغیر `shared_counter` ذخیره کنند. این امر باعث از دست رفتن بعضی از تغییرات می‌شود.
برای مثال می‌توان سناریوی زیر را درنظر گرفت:
فرض شود مقدار اولیه `shared_counter` برابر ۵ است و هر دو رشته به صورت همزمان مقدار فعلی `shared_counter` را می‌خوانند. هر دو مقدار ۵ را می‌خوانند. هر رشته یک واحد به مقدار خوانده‌شده اضافه می‌کند و مقدار ۶ را به `shared_counter` اختصاص می‌دهد. در نتیجه، به جای آنکه `shared_counter` به ۷ برسد، همچنان ۶ خواهد بود.
این ناسازگاری داده به دلیل عدم هماهنگی در دسترسی به متغیر مشترک و عدم استفاده از روش‌های همگام‌سازی رخ می‌دهد.