تمرینات عملی از مبحث atomic variables and operations

در سیستم‌های چندنخی، چندین پردازه یا ترد به‌صورت هم‌زمان به منابع مشترک دسترسی دارند. اگر این دسترسی‌ها به‌درستی کنترل نشود، *Race Condition* به‌وجود می‌آید و داده‌ها به‌صورت پیش‌بینی‌ناپذیری تغییر می‌کنند. یکی از راهکارهای اساسی برای جلوگیری از این وضعیت، استفاده از سازوکارهای هم‌زمانی (*Synchronization Mechanisms*) است.

در زبان *C* از نسخه‌ی *C11* به بعد، کتابخانه‌ای به نام *stdatomic.h* معرفی شده است که امکان استفاده از عملیات اتمیک (*Atomic Operations*) را فراهم می‌کند.
این عملیات باعث می‌شوند تغییرات روی حافظه در یک مرحله‌ی غیرقابل‌تقسیم انجام شود و در نتیجه، چند ترد نتوانند به‌طور هم‌زمان در اجرای آن دخالت کنند.

تمرین عملی : پیاده‌سازی چند ابزار هم زمانی با استفاده از عملیات های atomic

در این تمرین، با استفاده از دو عملگر مهم اتمیک یعنی *Test-And-Set* و *Compare-And-Swap* (*CAS*)، سازوکارهایی مانند *Spinlock* و *Semaphore* را پیاده‌سازی خواهید کرد تا یک آشنایی اولیه با برنامه نویسی بدون قفل (*lock-free programming*) داشته باشید.

هدف تمرین

هدف از این تمرین آشنایی شما با عملیات های *atomic* و پیاده‌سازی چند ابزار همزمانی با استفاده از آن‌هاست.
در این تمرین دو دستور کلیدی *CAS* و *test and set* به صورت عملی آشنا می‌شوید.

پیش نیاز ها

- آشنایی اولیه با برنامه نویسی چندخطی و کتابخانه *pthread*
- داشتن C۱۱ به بالا
- آشنایی با کتابخانه *stdatomic*

مفاهیم کلیدی استفاده شده

- **Test-And-Set (TAS)**: دستور اتمیکی که مقدار متغیر را بررسی کرده و در یک گام غیرقابل‌تقسیم آن را به مقدار جدید تغییر می‌دهد.
- **Compare-And-Swap (CAS)**: دستور اتمیکی که تنها در صورت برابر بودن مقدار فعلی با مقدار مورد انتظار، مقدار جدید را در حافظه می‌نویسد.

شرح وظیفه

1. پیاده‌سازی یک *spinlock* با استفاده از عملیات اتمیک *test_and_set* :

قفل شما باید استاندارد های زیر را رعایت کند:

- توابع لیست شده در فایل *tas_spinlock.h* را مطابق با نام و تعریف آن در فایل **C** مربوطه پیاده‌سازی کنید.
- از توابع مربوط به عملیات اتمیک *test_and_set* موجود در کتابخانه *stdatomic* برای پیاده‌سازی استفاده کنید.
- توجه داشته باشید که قفل طراحی توسط شما باید محافظت شده باشد. یه این معنی که دیگر ترد ها نتوانند قفل ترد های دیگر را رها کنند. (هر قفل دارای صاحب باشد).
- میتوانید عمل‌کرد قفل خود را با اجرای دستور *make run_test1* تست کنید. مقدار به دست آمده باید برابر 0 باشد.

.

2. پیاده‌سازی یک *spinlock* با استفاده از *CAS* :

یک قفل دیگر با استفاده از *atomic_compare_exchange* بسازید. قفل شما باید استاندارد های زیر را رعایت کند:

- توابع لیست شده در فایل *cas_spinlock.h* را مطابق با نام و تعریف آن در فایل **C** مربوطه پیاده‌سازی کنید.
- از توابع مربوط به عملیات اتمیک *CAS* موجود در کتابخانه *stdatomic* برای پیاده‌سازی استفاده کنید.
- توجه داشته باشید که قفل طراحی توسط شما باید محافظت شده باشد. یه این معنی که دیگر ترد ها نتوانند قفل ترد های دیگر را رها کنند. (هر قفل دارای صاحب باشد)

- میتوانید عمل‌کرد قفل خود را با اجرای دستور *make run_test2* تست کنید. مقدار به دست آمده باید برابر 0 باشد.

3. پیاده‌سازی یک سمافور شمارنده (*Counting Semaphore*) با استفاده از *CAS* :

یک سمافور شمارنده با استفاده از *atomic_compare_exchange* بسازید. سمافور شما باید استاندارد های زیر را رعایت کند:

- توابع لیست شده در فایل *cas_semaphore.h* را مطابق با نام و تعریف آن در فایل c مربوطه پیاده‌سازی کنید.
- از توابع مربوط به عملیات اتمیک *CAS* موجود در کتابخانه *stdatomic* برای پیاده‌سازی استفاده کنید.
- توجه داشته باشید که سمافور شما باید شمارنده باشد. به این معنی که به تعداد مشخصی ترد دسترسی بدهد. سمافور طراحی شده توسط شما از هر دو جنبه شمارنده بودن و بلاک کردن ترد ها تست می‌شود.
- میتوانید عمل‌کرد سمافور خود را با اجرای دستور *make run_test3* تست کنید. مقدار اولیه به دست آمده باید برابر 0 باشد. مقادیر چاپ شده بعدی خاصیت شمارنده بودن سمافور را بررسی میکنند و تعداد ترد هایی که همزمان میتوانند وارد شوند را تست میکند. میتوانید با تغییر این مقدار در فایل *test3.c* خروجی این تست را تغییر دهید.

جزییات پیاده‌سازی

فایل‌ها

۱. ***tas_spinlock.h , cas_spinlock.h , cas_semaphore.h*** :

- شامل تعریف اولیه داده ساختار قفل و توابع مورد نیاز برای این قفل میباشد.

تغییر در این فایل ضروری نیست.

توضیح فایلهای(.h) Header :

فایلهایHeader در زبان C با پسوند $.h$ شناخته می‌شوند و نقش آنها تعریف واسط میان بخش‌های مختلف برنامه است. هدف اصلی استفاده از این فایلها جدا سازیDeclarations از Implementations است تا کد ساختاریافته تر و قابل نگهداری تر باشد.

مهم‌ترین کاربردهای فایلهای $.h$ عبارتند از:

اعلان توابع، تعریف ثوابت و ماکروها و تعریف ساختارهای داده (*Structs, Typedefs*)

به طور خلاصه، فایل $.h$ مشخص میکند چه چیزی وجود دارد (توابع، ثابتها، ساختارها) در حالیکه فایل $.c$ مشخص میکند چگونه کار میکند (منطق و پیاده‌سازی توابع)

۲. *tas_spinlock.c*:

- محل پیاده‌سازی توابع مربوط به قسمت الف تمرین میباشد.
- تابع *tas_init* برای مقدار دهی اولیه به قفل میباشد.
- تابع *tas_lock* برای به دست گرفتن قفل توسط یک ترد فراخوانی خواهد شد تا در صورت امکان صاحب قفل شود. توجه داشته باشید که این قفل باید blocking باشد. به این معنی که تا موفقیت در به دست گرفتن قفل ترد باید منتظر بماند و بلاک بشود.
- تابع *tas_trylock* دقیقاً عمل‌کرد تابع *tas_lock* را دارد با این تفاوت که blocking نیست.
- تابع *tas_unlock* قفل را آزاد میکند.

۳. *cas_spinlock.c*:

- محل پیاده‌سازی توابع مربوط به قسمت ب تمرین میباشد.
- تابع *cas_init* برای مقدار دهی اولیه به قفل میباشد.
- تابع *cas_lock* برای به دست گرفتن قفل توسط یک ترد فراخوانی خواهد شد تا در صورت امکان صاحب قفل شود. توجه داشته باشید که این قفل باید blocking باشد. به این معنی که تا موفقیت در به دست گرفتن قفل ترد باید منتظر بماند و بلاک بشود.
- تابع *cas_trylock* دقیقاً عمل‌کرد تابع *cas_lock* را دارد با این تفاوت که blocking نیست.
- تابع *cas_unlock* قفل را آزاد میکند.

۴. *cas_semaphore.c* :

- محل پیاده‌سازی توابع مربوط به قسمت الف تمرین میباشد.
- تابع *cas_sem_init* برای مقدار دهی اولیه به قفل میباشد. توجه داشته باشید که شروط اولیه مقدار دهی رعایت بشود.
- تابع *cas_sem_try_acquire* تلاش میکند تا در صورت امکان از سمافور استفاده کند. یعنی باید به نوعی پیاده‌سازی شود که با توجه به تعداد شمارنده موجود در سمافور ، تصمیم بگیرد که ترد نیز از آن استفاده کند و یا خیر. در صورت موفقیت در استفاده 1 و در غیر این صورت 0 را برگرداند. این تابع non-blocking میباشد.
- تابع *cas_sem_acquire* دقیقاً عمل‌کرد تابع بالا را دارد تنها با این تفاوت که blocking خواهد بود.
- تابع *cas_sem_release* مقدار شمارنده سمافور را به اندازه یک واحد افزایش خواهد داد. در صورت وجود امکان این افزایش و موفقیت مقدار 1 و در غیر این صورت مقدار 0 را برمیگرداند.

نحوه کامپایل

برای کامپایل و اجرا نیاز دارید که دو ابزار gcc و make را داشته باشید.

برای اجرای هر یک از تست های مربوط به هر بخش تمرین ، میتوانید دستورات زیر را اجرا کنید:

make run_test1

make run_test2

make run_test3

برای حذف فایل‌های اجرایی و فایلهای باینری نیز میتوانید دستور زیر را اجرا کنید:

make clean

برای اجرای تمامی تست ها به همراه یک دیگر و به ترتیب نیز میتوانید از دستور زیر استفاده کنید:

make check