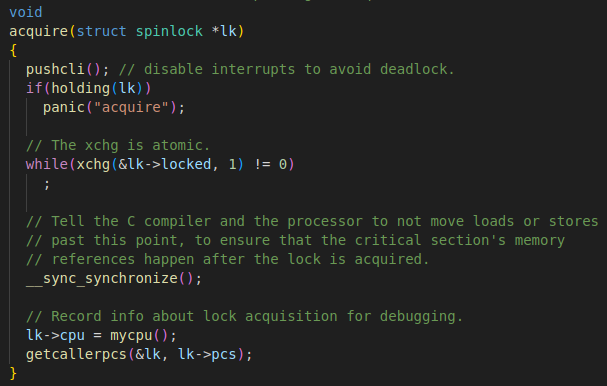
به نام خدا

گزارش آزمایشگاه آزمایش چهارم درس سیستم عامل

آیدین کاظمی: 810101561 علی زیلوچی: 810101560 بابک حسینی محتشم: 810101408

چگونگی همگام‌سازی در سیستم‌عامل XV6:

1. در تابع acquire داخل فایل spinlock.c که در تصویر زیر مشخص است، در ابتدای کار با صدا زدن pushcli وقفه‌های این CPU غیرفعال می‌شوند. دلیل این است که اگر وقفه‌ها غیرفعال نشوند و یا پس از دستور xchg که lock را یک کرده فعال شوند ممکن است باعث رخداد deadlock بشویم. برای مثال پس از اینکه CPU متغیر lock را یک کرد اگر وقفه‌ای بیاید و کد ISR نیاز به گرفتن قفل یکسانی داشته باشد، در حلقه whileگرفتن قفل تا ابد منتظر می‌ماند چون کد متن وقفه همواره باید تا اتمام آن اجرا شود در حالیکه در این کد در حلقه‌ بی‌نهایت افتاده‌ایم چون قبلا قفل آن گرفته شده.



تابع pushcli مشابه و popcli هم مانند cli و sti باعث فعال / غیرفعال کردن وقفه می‌شوند. تفاوت اصلی این دو با cli/sti در این است که این دو همدیگر را به نوعی خنثی می‌کنند یعنی برخلاف cli/sti که تنها وقفه را فعال و غیرفعال می‌کند، با اجرای pushcli یکی به تعداد cliهای CPU اضافه می‌شود وبا هر popcli یکی کم می‌شود تا وقتی که این مقدار صفر شود که در آن صورت فعال بودن یا نبودن وقفه به حالتی که پیش از این داشت برمی‌گردد.

1. پردازه‌ها در XV6 شش حالت مختلف می‌توانند داشته باشند:

|  |  |  |
| --- | --- | --- |
| توصیف در xv6 | وضعیت xv6 | وضعیت معادل |
| وضعیت خانه‌هایی از جدول پردازه‌ها که پردازه‌ای در آن‌ها قرار ندارد. | UNUSED | new |
| پردازه تازه ساخته شده که هنوز آماده اجرا نیست و مقادیر PCB والد در PCB آن کپی نشده است. | EMBRYO | new |
| پردازه‌ای که آماده اجرا است و متنظر انتخاب شدن توسط زمان‌بند است. | RUNNABLE | ready |
| پردازه‌ای که در حال اجرا شدن است. | RUNNING | running |
| پردازه منتظر رویدادی خاص است. | SLEEPING | waiting |
| پردازه‌ای که اجرایش تمام شده ولی هنوز در جدول پردازه‌ها وجود دارد چون والد مقدار برگشتی آن را دریافت نکرده است. | ZOMBIE | terminated |

هر پردازه که بخواهد CPU را رها کند، باید قفل ptable را بگیرد، هر قفل دیگری را رها کند و وضعیت خود را به وضعیتی به جز RUNNING تغییر دهد. تمام این شرایط در توابعی که پیش از تغییر پردازه فراخوان می‌شوند یعنی Yield/sleep/exit به درستی صورت می‌گیرد با این حال دوباره تمام این شرایط در تابع sched نیز بررسی می‌شوند. سپس متغیر intena که نشان دهنده فعال یا غیرفعال بودن وقفه‌ها پیش از pushcli است ذخیره شده و سپس context switch صورت میگیرد و تابع scheduler صدا زده می‌شود تا پردازه بعدی را برای اجرا انتخاب کند. پس از اتمام اجرای پردازه، مقدار intena برای CPU به مقدار آن پیش از context switch بازمی‌گردد. دلیل این کار این است که این متغیر مخصوص یک kernel thread است و نه خود CPU پس باید پیش از اجرای kernel thread ذخیره و پس از اجرای یک پردازه و موقع context switch به مقدار قبلی آن که پردازه مورد نظر پیش از رها کردن CPU داشت، برگردانیم.

cache coherency در سیستم‌عامل XV6:

1. در این روش دو بیت به حافظه cache اضافه می‌کنیم که نشان‌دهنده استیت مقدار آن آدرس در حافظه cache است:

Invalid: بدین معنی است که مقدار آدرس مورد نظر در cache صحیح نیست یعنی با حافظه اصلی متفاوت است.

Shared: بدین معنی است که مقدار آدرس مورد نظر در cache صحیح است یعنی با مقدار در حافظه اصلی یکسان است.

Modified: بدین معنی است که مقدار آدرس مورد نظر تنها در این CPU با حافظه اصلی یکسان است.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| write | read | hear write | hear read | from state |
| Modified | Shared | Invalid | Invalid | Invalid |
| Modified | Shared | Shared | Shared | Shared |
| Modified | Modified | Invalid | Shared | Modified |

جدول بالا نشان می‌دهد که یک Cache از هر استیت با هر تغییری در خود یا بقیه Cacheها، از چه استیتی به چه استیتی می‌رود و همچنین اگر تغییر استیتی آبی باشد یعنی Cache باید استیت جدید خود را از طریق shared bus به بقیه Cacheها اطلاع دهد.

بدین ترتیب در این روش با هر تغییر در یک cache block نیازی به تغییر در مقدار بقیه Cacheها تا وقتی که درخواست داشتن آن داده را ندارند، نیست و تنها با تغییر استیت اطلاع رسانی صورت می‌گیرد که بسیار سریع تر است و در صورت نیاز به داده، آن Cache که داده را دارد ابتدا در حافظه اصلی می‌نویسد و سپس Cache دیگر می‌خواند.

1. قفل بلیت بدین صورت کار می‌کند که دو متغیر global مشترک بین CPUها وجود دارد که یکی نشان‌دهنده شماره بلیت در حال اجرا است و دیگری نشان دهنده شماره آخرین بلیت داده شده به CPUها است. هر CPU که درخواست گرفتن قفل را دارد، به صورت atomic شماره‌ای به اندازه شماره آخرین بلیت دریافت می‌کند و آن را یکی افزایش می‌دهد. سپس تا وقتی که شماره بلیت در حال اجرا برابر شماره بلیت آن CPU نباشد، منتظر می‌ماند. موقع رها کردن CPU هم، شماره بلیت در حال اجرا را یکی افزایش می‌دهیم.

نقاط قوت این روش:

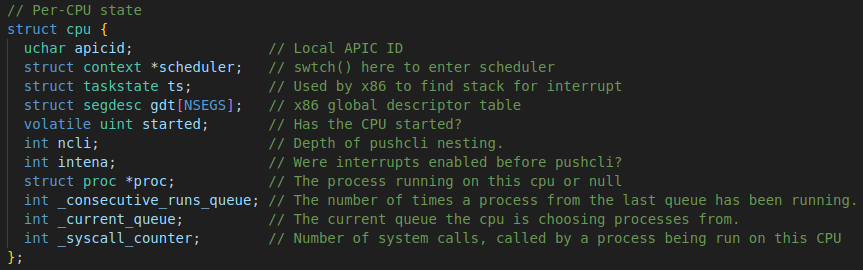
* در این روش اولین کسی که متقاضی دریافت قفل است، آن را دریافت می‌کند و بدین صورت CPUها به صورت FCFS طور قفل را دریافت می‌کنند و starvation رخ نمی‌دهد.
* از لحاظ هزینه حافظه مناسب است زیرا تمام CPUها در حال بررسی یک متغیر global مشترک هستند.
* رفتار مشخص و قابل پیشبینی در این مدل همگام‌سازی وجود دارد.

نقاط ضعف این روش:

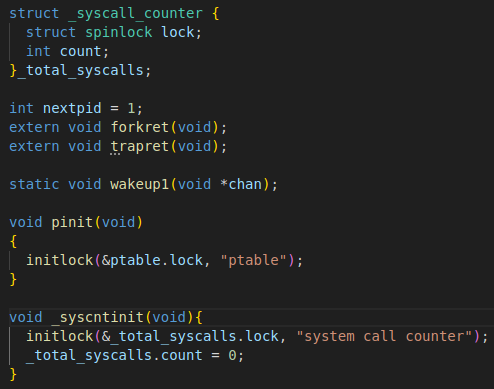
* با افزایش تعداد پردازه‌ها کارایی این روش به صورت نمایی افت پیدا می‌کند.
* همچنین **مشکل مذکور نیز برای این قفل وجود دارد**. بدین صورت که تمام CPUها در cache خود متغیر مشترک بلیت در حال اجرا را دارند و وقتی که قفل رها می‌شود، مقدار این متغیر در تمام CPUها نادرست می‌شود و بدین ترتیب تمام CPUها باید مقدار جدید متغیر را از حافظه بخوانند.

گزارش بخش سیستم‌کال جدید:

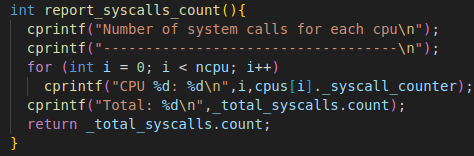
ابتدا به ساختار داده cpu در فایل proc.h، متغیر \_syscall\_counter را اضافه می‌کنیم که نشان‌دهنده تعداد سیستم‌کال‌هایی است که توسط پردازه‌هایی که روی این پردازنده اجرا شده‌اند صورت گرفتند. همچنین این متغیر را برای هر پردازنده در ابتدای کار در تابع mpmain برابر صفر قرار می‌دهیم.



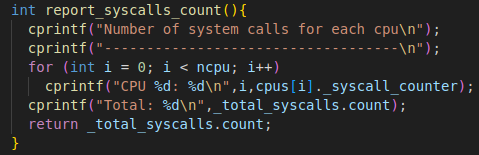
ساختار داده \_syscall\_counter را برای متغیر global ایجاد می‌کنیم که شامل قفل و عدد صحیحی برای شمارش تعداد فراخوانی‌های سیستمی است. در تابع\_syscntinit متغیر گلوبال \_total\_syscalls رو مقداردهی اولیه می‌کنیم و این تابع را در تابع main در فایل main.c صدا می‌زنیم تا ابتدای کار، مقداردهی اولیه صورت گیرد.



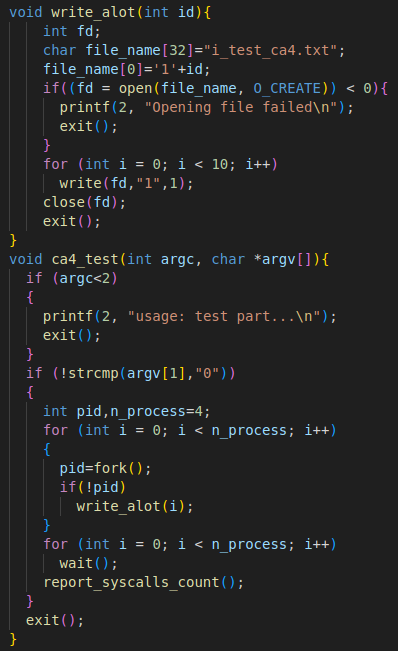
برای پروژه دوم، تابع \_log\_syscall رو ایجاد کردیم و آن را در تابع syscall فراخوانی می‌کنیم. حال این تابع را تغییر می‌دهیم تا با انجام هر سیستم‌کال، به تعداد وزن آن سیستم‌کال به متغیر \_syscall\_counter در cpu فعلی و همچنین به متغیر گلوبال \_total\_syscalls نیز اضافه می‌کنیم. البته این متغیر برای همگام‌سازی مناسب ایمن شده و برای آپدیت آن باید قفل آن را بگیریم. همچنین چون برای داشتن cpu فعلی باید وقفه‌ها غیرفعال باشند، mycpu را نیز پس از گرفتن قفل فراخوانی می‌کنیم.



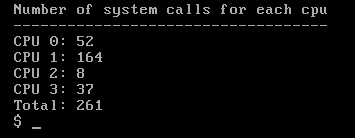
همچنین در فایل proc.c تابع report\_syscalls\_count را اضافه کردیم که توسط همین سیستم‌کال صدا زده می‌شود و این تابع، متغیر \_syscall\_counter را برای هر cpu چاپ می‌کند و همچنین تعداد کل سیستم‌کال‌ها را نیز که در متغیر global مشترک \_total\_syscalls ذخیره شده است را نیز چاپ می‌کند.



حال برنامه تست را از پروژه‌های قبل بدین صورت تغییر می‌دهیم که تعدادی پردازه جدید fork کند و هر پردازه در فایلی جداگانه بنویسد.



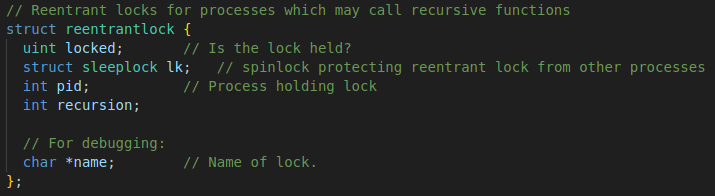
حال برنامه را اجرا می‌کنیم و در خروجی مشخص است که جمع تعداد سیستم‌کال‌های تمام پردازنده‌ها با حاصل جمع حساب شده در متغیر gloal یکسان شده است.



میوتکس پردازه مالک (Reentrant Mutex):

گزارش بخش قفل با امکان ورود مجدد:

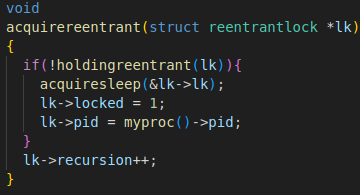
برای پیاده‌سازی قفل جدید، مشابه دو نوع قفل موجود عمل می‌کنیم. ابتدا فایل reentrantlock.h را تشکیل داده و در این فایل، ساختار داده قفل را تعریف می‌کنیم. ساختارداده این قفل مشابه دو قفل دیگر است با این تفاوت که متغیری برای ذخیره تعداد acquire توسط پردازه مالک دارد.



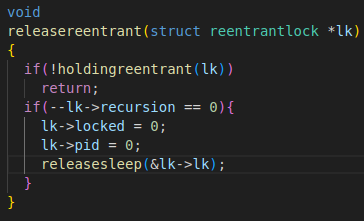
در فایل reentrantlock.c، توابع مربوط به قفل را پیاده‌سازی کردیم. تابع اول تابع initreentrantlock است که پوینتری به قفل و نام قفل را دریافت نموده و آن را مقداردهی اولیه می‌کند. در این تابع دو قفل موجود در ساختار داده و نام و باقی متغیرها مقداردهی اولیه می‌شوند.



تابع بعدی acquirereentrant است که پوینتری به قفل دریافت می‌کند و در صورت امکان دسترسی به قفل را به پردازه فراخواننده می‌دهد. در این تابع ابتدا بررسی می‌کنیم که اگر فراخواننده در حال حاضر قفل را در دست دارد، تنها recursion آن را یکی افزایش می‌دهیم. اگر پردازه قفل را در دست ندارد، وارد شرط می‌شویم. ابتدا قفل sleeplock را در به پردازه می‌دهیم. اگر پردازه دیگری قفل را در دست داشته باشد، پردازه جدید در اینجا به حالت sleep می‌رود وگرنه متغیر locked به معنی قفل بودن را فعال می‌کند و شماره پردازه‌اش را در متغیر pid ذخیره می‌کند و متغیر recursion را یکی افزایش می‌دهد. دلیل این که قفل را از نوع sleeplock انتخاب کردیم این است که قفل بسته به عمق recursion می‌تواند زمان زیادی طول بکشد تا پردازه مالک قفل را رها کند.



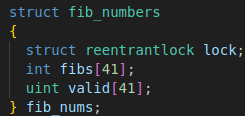
تابع بعدی releasereentrant است که پوینتری به قفل دریافت می‌کند و در صورتی که پردازه فراخواننده مالک قفل باشد، یکی از recursion کم می‌کند. اگر مقدار recursion صفر شد، می‌توان قفل را به پردازه دیگری داد پس در این صورت locked را صفر و pid را برابر صفر قرار می‌دهد و قفل را رها می‌کند تا پردازه‌های دیگر نیز بتوانند قفل را در دست بگیرند.



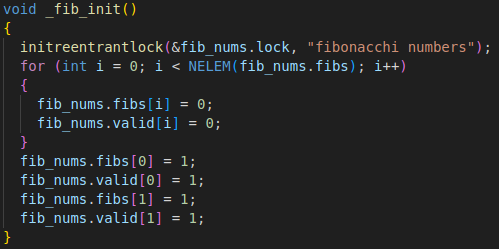
تابع آخر holdingreentrant است که دوباره پوینتری به قفل دریافت می‌کند و درصورتی که locked فعال باشد و شماره پردازه با pid قفل یکسان باشد یک وگرنه صفر برمی‌گرداند که نشان‌دهنده مالک این قفل بودن یا نبودن است.

برای تست قفل، سیستم‌کال جدیدی برای حساب اولین تا چهلمین عدد فیبوناچی تشکیل می‌دهیم و در تعریف سیستم‌کال از این نوع قفل استفاده می‌کنیم.

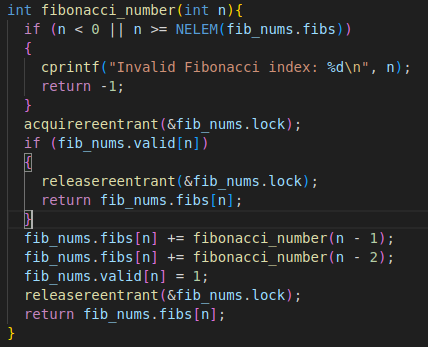
ابتدا ساختار داده fib\_numbers و متغیر global به نام fib\_nums برای ذخیره کردن اعداد فیبوناچی تشکیل می‌دهیم که باید مطمعن باشیم دسترسی به این متغیر توسط چند ریسه به درستی صورت بگیرد پس قفلی با امکان ورود مجدد در ساختارداده قرار می‌دهیم. همچنین در این تابع یک آرایه برای نگه داشتن مقدار عدد فیبوناچی iام به نام fibs و آرایه دیگری به نام valid برای نگه داشتن معتبر بودن یا نبودن مقدار در خانه متناظر در آرایه fibs نگه می‌داریم.



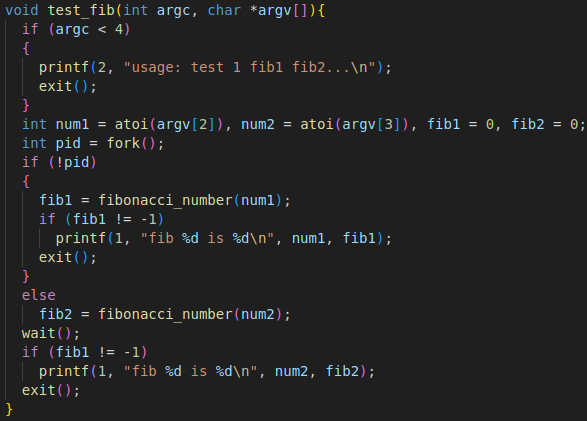
همچنین تابع \_fib\_init را تشکیل داده و در ابتدای کار در تابع main در فایل main.c فراخوانی می‌کنیم که در این تابع، قفل را مقداردهی اولیه کرده و سپس تمام خانه‌های fibs و valid را به جز دو خانه اول صفر می‌کنیم.



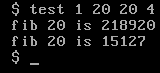
تابع Fibonacci\_number تابعی است که توسط سیستم‌کال صدا زده می‌شود. این تابع در صورتی که عددی که دریافت می‌کند در محدوده درستی باشد، قفل را دریافت کرده و سپس بررسی می‌کند اگر ایندکسی که دریافت کرده قبلا حساب شده و valid بود، قفل را رها کرده و مقدار را باز می‌گرداند وگرنه دو fib قبل را حساب کرده و مقدارشان را جمع کرده و در آرایه ذخیره می‌کند و پس از رها کردن قفل، عدد حساب شده را برمی‌گرداند.



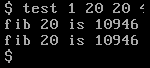
در برنامه سطح کاربر، تابعی نوشتیم یک بار fork انجام می‌دهد و هر دو پردازه سیستم‌کال Fibonacci\_number را صدا می‌زنند و مقدار Fibonacci حساب شده را چاپ می‌کنند.



حال برای تست، ابتدا تمام خطوط مربوط به قفل را کامنت می‌کنیم که اشتباه شدن پاسخ را مشاهده کنیم:



حال قسمت‌های قفل را اضافه می‌کنیم:



می‌توان دید که این سری عدد فیبوناچی بیستم به درستی حساب شده است در صورتی که در تصویر قبلی مشاهده کردیم به دلیل نداشتن قفل تغییر آرایه مشترک به نادرستی صورت می‌گرفت.

سوالات بخش قفل با امکان ورود مجدد:

1. قفل با امکان ورود مجدد مزایای دارد که مهم‌ترین آن‌ها قابلیت استفاده در توابع بازگشتی است ولی این قفل معایبی هم دارد از جمله:

* این قفل از لحاظ سربار زمانی کندتر از قفل mutex ساده است چون باید به ازای هر acquire و release، متغیر recursion آپدیت شود و همچنین مالک بودن بررسی شود.
* پیچیدگی این قفل نسبت به قفل معمولی بیشتر است چون کاربر برای استفاده باید اطمینان حاصل کند که به تعدادی که acquire می‌کند به همان تعداد هم release انجام دهد.

1. هدف این نوع قفل حس مسئله readers-writers است که هدف آن است که هر تعداد خواننده بتوانند به طور همزمان منبع مشترکی را بخوانند ولی وقتی که نویسنده می‌نویسد باید تنها نویسنده به فایل دسترسی داشته باشد.

برای پیاده‌سازی این قفل نیاز است تعداد خواننده‌هایی که قفل را در دست دارند و درخواست نویسنده برای نوشتن یا ننوشتن ذخیره شوند تا بتوان از آمدن خواننده‌های جدید در صورت درخواست برای نوشتن و همچنین دادن قفل به نویسنده تا وقتی که خواننده‌ای قفل را دارد جلوگیری کرد.

این نوع قفل برخلاف قفل با امکان ورود مجدد اجازه دسترسی به چندین ریسه خواننده را می‌دهد در صورتی که در قفل با امکان ورود مجدد در هر لحظه تنها یک نفر به منبع مشترک دسترسی دارد. به همین دلیل اگر تعداد خواننده‌ها نسبتا زیاد باشد به دلیل قابلیت موازی خواندن سربار این قفل می‌تواند کمتر باشد.