گزارش آزمایشگاه سیستم عامل: پروژه شماره ۴

گروه ۲۲

پردیس زندکریمی - محمد امانلو - شهزاد ممیز

۸۱۰۱۰۱۰۸۱ - ۸۱۰۱۰۰۰۸۴ - ۸۱۰۱۰۰۲۷۲

Last commit code:

ce614d9b0771fd7bb7dd60dd7e2d31505cb02e60

Github repo:

https://github.com/MohammadAmanlou/OS_lab

1) علت غیرفعال کردن وقفه در هنگام استفاده از این نوع قفل چیست؟ چرا ممکن است CPU با مشکل deadlock روبرو شود؟

وقفه در متن وقفه اجرا می شود. به طور کلی در سیستم عامل ها کد های وقفه قابل مسدود کردن نیستند. ماهیت این کد های اجرایی بدین صورت است که باید در اسرع وقت انجام شوند و لذا قابل زمانبندی نیز نیستند. به همین دلیل همگام سازی آنها نباید باعث مسدود شدن آنها شود. مثلا از قفل های چرخشی(spin lock) استفاده شود یا در پردازنده های تک هسته ای وقفه غیرفعال شود.

سمافورها باعث می‌شوند که کارها بر اساس اختلاف بخوابند، که برای کنترل‌کنندگان وقفه غیرقابل قبول است. اساساً، برای چنین کار کوتاه و سریعی (دست زدن به وقفه) کار انجام شده توسط سمافور بیش از حد است. همچنین، اسپین‌لاک‌ها را نمی‌توان با بیش از یک کار نگه داشت.

Spinlock قفلی است که هرگز تسلیم نمی شود.

مشابه mutex، دو عملیات قفل و باز کردن دارد.

اگر قفل موجود باشد، فرآیند آن را بدست می‌آورد و در بخش بحرانی ادامه می‌یابد و پس از اتمام آن قفل را باز می‌کند. این شبیه به mutex است. اما، اگر قفل در دسترس نباشد چطور ؟ تفاوت جالب اینجاست. با mutex، تا زمانی که قفل در دسترس باشد، روند به حالت خواب می‌رود. ولی، در صورت قفل چرخشی، به حلقه محکم می رود، جایی که به طور مداوم وجود قفل را بررسی می کند تا زمانی که در دسترس قرار گیرد.

در سطح بالا، یک قفل خواب دارای یک locked field است که توسط یک قفل چرخشی محافظت می‌شود و فراخوانی اسلیپ به خواب به طور اتمی CPU را تولید می‌کند و قفل چرخشی را آزاد می‌کند. نتیجه این است

که رشته‌های دیگر می‌توانند در حالی که accesleep منتظر است اجرا شوند. از آنجایی که قفل‌های خواب وقفه‌ها را فعال می‌کنند، نمی‌توان از آن‌ها در کنترل‌کننده‌های وقفه استفاده کرد. از آنجایی که ممکن است با اسلیپ، پردازشگر تولید شود، قفل‌های خواب نمی‌توانند در بخش‌های حیاتی قفل چرخشی استفاده شوند (اگرچه قفل‌های اسپین را می‌توان در داخل قفل خواب حیاتی استفاده کرد.)

Xv6 در اکثر مواقع از قفل های چرخشی استفاده می کند، زیرا سربار آنها پایین است. از قفل خواب فقط در سیستم فایل استفاده می‌کند، جایی که می‌توان قفل‌ها را در عملیات طولانی دیسک نگه داشت.

تلاش برای به دست آوردن یک قفل چرخشی به صورت بازگشتی تضمین شده است که باعث بن بست می شود: نمونه نگهدارنده یک روال بازگشتی نمی تواند قفل چرخشی را آزاد کند در حالی که نمونه دوم می چرخد و سعی می کند همان قفل چرخشی را بدست آورد.

دستورالعمل‌های زیر نحوه استفاده از قفل‌های چرخشی را با روال‌های بازگشتی توضیح می‌دهند:

روال بازگشتی نباید خود را در حین نگه داشتن یک قفل چرخشی فراخوانی کند، یا نباید سعی کند در تماس های بعدی همان قفل چرخشی را بدست آورد.

در حالی که روال بازگشتی دارای یک قفل چرخشی است، اگر بازگشت مجدد ممکن است باعث بن بست شود یا باعث شود تماس گیرنده قفل چرخش را برای بیش از 25 میکروثانیه نگه دارد، روال دیگر درایور نباید روال بازگشتی را فراخوانی کند.

2) **توابع pushcli و popcli به چه منظوری استفاده شده اند و چه تفاوتی با cli و sti دارند؟**

در توابع pushcli و popcli به ترتیب cli و sti فراخوانی شده است ولی تفاوت این توابع با cli و sti این است که در این توابع، قابلیت شمارش وجود دارد و مشخص است که هر برنامه چقدر اجرا شده است و در مدیریت آن کمککننده است. در واقع میتوان با استفده از pushcli به یک تعداد مشخص interrup ها را غیرفعال کنیم و)گویا به آن تعداد عدد در استک فرضی پوش شده است و باید برای فعال کردن interrup ها به تعداد همان مقدار عدد از استک فرضی پاپ کنیم.(اگر عددncli از 0 بزرگتر باشد، interrupt ها غیر فعال و اگر برابر با 0 باشد، interrupt ها فعال است؛ در موقع استفاده از spinlock از pushcli استفاده میکنیم

pushcli و popcli دستورالعمل ها یا توابع استاندارد در اسمبلی x86 یا هر زبان برنامه نویسی سطح بالا نیستند و ممکن است به مفاهیم سطح بالاتر از یک هسته یا انتزاع سیستم عامل خاص اشاره داشته باشید.

با این حال، ایده ای که توسط pushcli و popcli ارائه می شود، ترکیب عملیات فشار یا پشته پاپ با غیرفعال کردن و فعال کردن وقفه ها است. این ترکیب فرضی به صورت زیر عمل می کند:

pushcli ابتدا وضعیت فعلی پرچم وقفه (IF) را روی پشته فشار می دهد و سپس cli را برای پاک کردن پرچم وقفه اجرا می کند و در نتیجه وقفه های بعدی را غیرفعال می کند.

سپس popcli حالت قبلی پرچم وقفه (IF) را از پشته بیرون می‌آورد و به طور موثر آن را بازیابی می‌کند، که می‌تواند وقفه‌ها را دوباره فعال کند اگر قبل از عملیات pushcli فعال شده باشند.

تفاوت اساسی بین این عملیات با cli مستقل (Clear Interrupt Flag) و sti (Set Interrupt Flag) این است که cli و sti به سادگی وقفه ها را بدون ذخیره کردن حالت قبلی پرچم وقفه غیرفعال و فعال می کنند. اگر قرار بود از cli و بعداً sti استفاده کنید، وقفه‌ها را فعال می‌کنید بدون اینکه لزوماً بدانید که آیا پرچم وقفه قبل از استفاده از دستورالعمل cli تنظیم یا پاک شده است یا خیر.

cli پرچم IF را روی 0 تنظیم می کند و وقفه ها را غیرفعال می کند.

sti پرچم IF را روی 1 تنظیم می کند و وقفه ها را فعال می کند.

pushcli و popcli فرضی در شرایطی که نمی‌خواهید وضعیت جهانی وقفه‌ها را فراتر از اجرای بخش مهم کد خود تغییر دهید، ایمن‌تر خواهند بود. ذخیره وضعیت پرچم وقفه قبل از غیرفعال کردن وقفه ها (و بازیابی آن بعدا) از فعال کردن مجدد وقفه هایی که قرار بود توسط قسمت های قبلی کد غیرفعال باقی بمانند بالقوه جلوگیری می کند.

3) **چرا قفل مذکور در سیستم های تک هسته ای مناسب نیست؟ روی کد ها توضیح دهید.**

در تابع acquire، تابع holding فراخوانی میشود در این تابع چک میشود که آیا cpu قفل را نگه داشته است یا خیر؛ یعنی در واقع پیداسازی قفلها در سیستمعامل 6xv به صورت waiting busy است و اگر پردازنده تک هستهای باشد، میتوان یا مشغول انجام عملیات و یا نگهداری قفل باشد و این دو کار باهم ممکن نیست.

برای سیستم‌های تک پردازنده، هسته مقدار تعداد چرخش را نادیده می‌گیرد و آن را به عنوان صفر در نظر می‌گیرد - اساساً باعث می‌شود یک قفل اسپین یک بدون عملیات باشد.

اسپین‌لاک‌ها فقط در صورتی کارآمد هستند که رشته‌ای که قفل را به دست می‌آورد احتمالاً برای مدت کوتاهی مسدود می‌شود، به‌عنوان مثال به این دلیل که قفل توسط رشته‌ای که روی پردازنده دیگری اجرا می‌شود به دست آمده است تا از هزینه تعویض متن جلوگیری شود. در یک سیستم تک پردازنده، رشته ای که اسپینلاک را به دست می آورد، تا زمانی که (1) توسط زمانبند قطع شود، (2) رشته ای که در حال حاضر صاحب قفل است برنامه ریزی شود، و (3) این رشته، قفل را آزاد کند، مسدود می شود. در این مورد، برای موضوع اکتسابی کارآمدتر است که در صف انتظار مسدود شود تا زمان‌بندی‌کننده بتواند بلافاصله رشته دیگری را برنامه‌ریزی کند. اگر بخش بحرانی کوچک است و قفل در سطح هسته است، می توانید به جای چرخش، وقفه ها را غیرفعال کنید.

4) در مجموعه دستورات RISC-V دستوری با نام amoswap وجود دارد. دلیل تعریف و نحوه ی کار آن را توضیح دهید.

```
amoswap.w rd,rs2,(rs1)
```

مقدار داده امضا شده 32 بیتی را به صورت اتمی از آدرس در rs1 بارگیری کنید، مقدار را در ثبات rd قرار دهید، مقدار بارگذاری شده و مقدار امضا شده 32 بیتی اصلی را در rs2 عوض کنید، سپس نتیجه را به آدرس در rs1 ذخیره کنید.

دلیل تعریف چنین دستورالعمل‌های اتمی این است که وسیله‌ای برای پردازنده‌ها یا هسته‌های متعدد فراهم کند تا دسترسی به حافظه مشترک را به گونه‌ای هماهنگ کنند که ثبات و صحت را تضمین کند. بدون عملیات اتمی، تضمین نظم و تکمیل توالی‌های خواندن، اصلاح و نوشتن، زمانی که رشته‌های متعدد درگیر هستند، دشوار خواهد بود، که به طور بالقوه منجر به داده‌های خراب می‌شود.

دستورالعمل AMOSWAP (تغییر عملیات حافظه اتمی)، به طور خاص، یک مقدار را در یک ثبات با یک مقدار در حافظه به صورت اتمی تعویض می کند. در اینجا نحوه کار آن آمده است:

دستورالعمل مقدار اصلی را از یک آدرس حافظه مشخص شده می خواند.

سپس یک مقدار جدید از یک ثبات در آن آدرس حافظه می نویسد.

در نهایت، مقدار اصلی را در یک ثبات مقصد مشخص می‌نویسد.

5) مختصری راجع به تعامل میان پردازه ها توسط دو تابع مذکور توضیح دهید.

آدرس قفل به تابع acquiresleep پاس داده می شود و پراسس تا زمانی که فرصت برای در دست گرفتن قفل به آن داده نشده است sleep میکند. در تابع releasesleep، پردازهای که قفل را نگه داشته بود، تابع wakeup را فراخوانی میکند که در آن 1wakeup فراخوانی میشود و در این تابع، تمام پردازههایی که روی آن قفل خاص، sleep کردهاند را بیدار میکند و در واقع وضعیت آنها را از SLEEPING به RUNNABLE تغییر میدهد.

در «acquiresleep»، اگر منبع در دسترس نباشد، & lock می‌شود

■ وضعیت فرآیند را روی SLEEPING تنظیم می کند: تا زمانی که بیدار نشود برنامه ریزی نمی شود

○ در «releasesleep»، کد با قفل خواب، تمام فرآیندهای انتظار در & lock را بیدار می کند.

■ همه فرآیندهای در حالت خواب و قفل را روی RUNNABLE تنظیم می کند: می توان آنها را برنامه ریزی کرد

■ همه فرآیندهای انتظار بیدار می شوند. یکی از آنها قفل را می گیرد. بقیه دوباره خواهند خوابید.

برای جلوگیری از بن بست، sleep –lock به یک روتین نیاز دارد که به آن acquiresleep می گویند. پردازنده در حین انتظار، و وقفه ها را غیرفعال نمی کند.

acquisleep در سطح بالا، sleep-lock دارای یک locked field است که توسط یک قفل اسپین‌لاک محافظت می‌شود و فراخوانی accesleep به خواب به طور اتمی CPU را تولید می‌کند و قفل چرخشی را آزاد می‌کند. نتیجه این است که رشته‌های دیگر می‌توانند در حالی که accesleep منتظر است اجرا شوند.

از آنجایی که sleep-lock وقفه‌ها را فعال می‌کنند، نمی‌توان از آن‌ها در کنترل‌کننده‌های وقفه استفاده کرد. از آنجایی که ممکن است با اسلیپ، پردازشگر تولید شود، قفل‌های خواب نمی‌توانند در بخش‌های حیاتی قفل چرخشی استفاده شوند (اگرچه قفل‌های اسپین را می‌توان در داخل sleep-lock critical استفاده کرد.

6) **حالات مختلف پردازه ها در xv6 را توضیح دهید. تابع sched چه وظیفه ای دارد؟**

xv6 از اسم های زیر برای نشان دادن استیت های یک پراسس استفاده می کند:

UNUSED: در این حالت از پردازه استفادهای نمیشود.

EMBRYO: زمانی که پردازهای از وضعیت UNUSED خارج میشود به وضعیت EMBRYO میرود.

SLEEPING: زمانی که پردازه به منبعی نیاز دارد که آماده نیست)مثال عملیات O/I)، پردازه در حالت SLEEPING قرار میگیرد و دیگر در cpu نیست.

RUNNABLE: وقتی پردازهای در این حالت قرار میگیرد یعنی آماده است و منتظر است تا cpu ،scheduler را در اختیار آن قرار دهد.

RUNNNING: وقتی پردازهای در این حالت قرار دارد یعنی در حال اجرا است و cpu در حال حاضر به آن اختصاص داده شده است.

ZOMBIE: زمانی که کار پردازهای تمام میشود و پایان مییابد، به حالت ZOMBIE در میآید و تا زمانی که والدش wait را فراخوانی نکرده است در این حالت میماند زیرا با وجود پایان این پردازه، اطالعات آن هنوز در ptable وجود دارد.

تابع sched به سادگی شرایط مختلف را بررسی می‌کند و Swtch را برای تغییر به رشته زمان‌بندی فراخوانی می‌کند. هر تابعی که sched را فراخوانی می کند باید این کار را با ptable انجام دهد.

ptable.lock، هر قفل دیگری را که نگه می دارد آزاد می کند، وضعیت خود را به روز می کند (proc->state)، و سپس sched را فراخوانی می کند.

بازده از این قرارداد پیروی می کند، مانند sleep و exit که بعداً بررسی خواهیم کرد. Sched آن شرایط را دوبار بررسی می کند و سپس مفهومی از آن شرایط را بررسی می کند: از آنجایی که یک قفل نگه داشته می شود، CPU باید با وقفه های غیرفعال در حال اجرا باشد. در نهایت، Swtch را فراخوانی می‌کند تا زمینه فعلی را در proc->context ذخیره کند و در cpu->scheduler به زمینه زمان‌بندی سوئیچ کند.

7) تغییری در توابع دسته دوم داده تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی کنید.

قفل: برای حذف متقابل استفاده می شود. وقتی یک مدعی قفل را نگه می‌دارد، هیچ رقیب دیگری نمی‌تواند آن را نگه دارد (دیگران مستثنی هستند). شناخته‌شده‌ترین قفل‌های اولیه در هسته، اسپین‌لاک‌ها و موتکس‌ها هستند.

در استراکت mutex یک فیلد به نام owner وجود دارد، این فیلد در حین آزادسازی قفل چک میشود تا تنها صاحب قفل مجاز به این کار باشد.

توضیحات بیشتر:

در اینجا برخی از مکانیسم های قفل کلید مورد استفاده در هسته لینوکس آورده شده است:

اسپینلاک:

توضیحات: اسپین‌لاک‌ها قفل‌های سبک وزنی هستند که برای به دست آوردن قفل از حالت انتظار مشغول هستند. اگر یک نخ سعی کند یک قفل چرخشی را بدست آورد که قبلاً توسط نخ دیگری نگه داشته شده است، در یک حلقه می چرخد تا زمانی که قفل در دسترس قرار گیرد.

توابع API: spin_lock، spin_unlock، spin_lock_irq، spin_unlock_irq و غیره.

:Mutexes

توضیحات: موتکس ها (قفل های حذف متقابل) مکانیزم قفل سنگین تری در مقایسه با اسپینلاک ها هستند. آنها به فرآیندها اجازه می دهند در حالی که منتظر در دسترس قرار گرفتن قفل هستند، بخوابند و استفاده از CPU را در طول بحث کاهش می دهند.

توابع API: mutex_init، mutex_lock، mutex_unlock، mutex_trylock و غیره.

سمافورها:

توضیحات: سمافورها برای سیگنال دهی بین فرآیندها استفاده می شوند و اغلب برای کنترل دسترسی به یک منبع مشترک استفاده می شوند. آنها می توانند باینری (مانند mutex) یا سمافورهای شمارشی باشند.

توابع API: sema_init، پایین، بالا و غیره.

قفل های Reader-Writer:

توضیحات: قفل‌های Reader-Writer به چندین خواننده اجازه می‌دهند به طور همزمان به یک منبع دسترسی داشته باشند، اما دسترسی انحصاری را برای یک نویسنده تضمین می‌کنند. این می تواند کارآمدتر از استفاده از mutex در زمانی که منبع اشتراکی عمدتا خوانده می شود، باشد.

توابع API: rwlock_init، read_lock، read_unlock، write_lock، write_unlock و غیره.

:Completion

توضیحات: تکمیل مکانیزمی برای سیگنال دهی بین دو زمینه اجرایی است. یک زمینه می تواند منتظر دیگری برای تکمیل یک کار باشد. اغلب برای همگام سازی بین کنترل کننده های وقفه و کد هسته معمولی استفاده می شود.

توابع API: init_completion، wait_for_completion، تکمیل و غیره.

عملیات اتمی:

توضیحات: عملیات اتمی تضمین می کند که برخی عملیات به صورت اتمی و بدون وقفه انجام می شود. اینها اغلب در همگام سازی سطح پایین استفاده می شوند و در سناریوهایی که قفل ها خیلی سنگین هستند بسیار مهم هستند.

توابع API: atomic_t، atomic_add، atomic_sub، atomic_inc، atomic_dec و غیره.

8) روشی دیگر برای نوشتن برنامه ها استفاده از الگوریتم lock– free است. مختصری راجب به آن توضیح داده و از مزایا و معایب آن ها نسبت به برنامه نویسی با lock بگویید.

برای جلوگیری از هزینه‌های مرتبط با قفل‌ها، بسیاری از سیستم‌عامل‌ها از ساختارها و الگوریتم‌های داده lock– free استفاده می‌کنند. به عنوان مثال، می توان یک لیست پیوندی پیاده سازی کرد که در طول جستجوی لیست نیازی به قفل ندارد و برای درج یک مورد در لیست یک دستورالعمل اتمی وجود دارد. با این حال، برنامه نویسیlock– free پیچیده تر از برنامه نویسی قفل است. برای مثال، باید نگران ترتیب مجدد آموزش و حافظه بود. برنامه نویسی با قفل در حال حاضر سخت است، بنابراین xv6 از پیچیدگی اضافی برنامه نویسی بدون قفل جلوگیری می کند. (از سورس)

تفاوت مهم این است که الگوریتم‌های بدون قفل تضمین می‌شوند که به تنهایی پیشرفت کنند - بدون کمک رشته‌های دیگر. با یک قفل یا قفل چرخشی، هر thread ضعیفی که نمی تواند قفل را بدست آورد، کاملاً در اختیار thread است که قفل را در اختیار دارد. رشته ضعیفی که نمی تواند قفل را بدست بیاورد کاری جز انتظار نمی تواند انجام دهد (چه از طریق یک انتظار مشغول یا یک خواب با کمک سیستم عامل).
به طور کلی، الگوریتم‌های بدون قفل در هر رشته کارایی کمتری دارند برای پیاده‌سازی یک الگوریتم بدون قفل نسبت به یک قفل ساده، کار بیشتری انجام می‌دهید. (stackoverflow)
الگوریتم‌های lock– free به‌طور مؤثر «قفل‌های» خود را پیاده‌سازی می‌کنند، اما این کار را به‌گونه‌ای انجام می‌دهند که از تعداد سوئیچ‌های زمینه جلوگیری یا کاهش می‌دهد، به همین دلیل است که آنها تمایل دارند مشابه قفل خود را انجام دهند.
lock– free لزوماً سریع‌تر نیست، اما می‌تواند احتمال بن‌بست یا زنده بودن را از بین ببرد، بنابراین می‌توانید تضمین کنید که برنامه شما همیشه به سمت اتمام پیشرفت خواهد کرد. با قفل ها، ایجاد چنین تضمینی دشوار است -- از دست دادن برخی از دنباله های اجرایی ممکن که منجر به بن بست می شود بسیار آسان است.

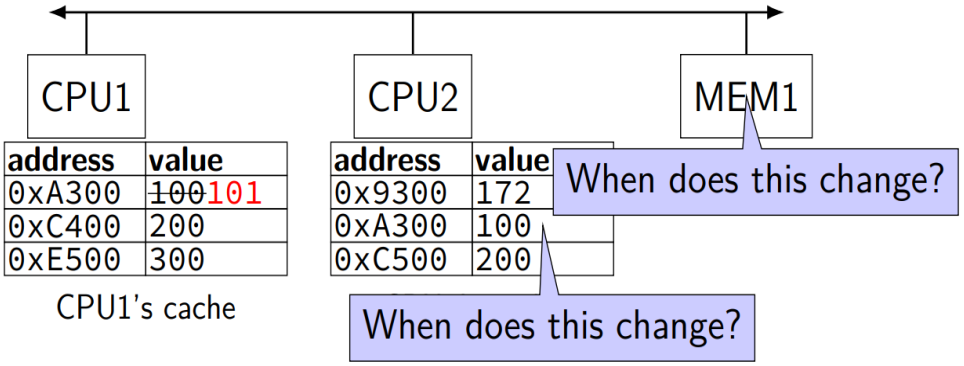
پیاده سازی متغیرهای مختص هر هسته پردازنده

الف) روشی جهت حل این مشکل در سطح سخت افزار وجود دارد. مختصرا آن را توضیح دهید.

Intel Core i7 Xeon 5500 at 2.4 GHz

Memory	Access time	Size
register	1 cycle	64 bytes
L1 cache	~4 cycles	64 kilobytes
L2 cache	~10 cycles	4 megabytes
L3 cache	~40-75 cycles	8 megabytes
remote L3	~100-300 cycles	
Local DRAM	~60 nsec	
Remote DRAM	~100 nsec	

the cache coherency problem



CPU1 writes 101 to 0xA300?

:Cache Coherence Protocols

مشکل انسجام حافظه نهان در سیستمی رخ می دهد که چندین هسته دارد که هر کدام کش محلی خود را دارند. مشکل انسجام حافظه پنهان اساساً با چالش‌های همگام‌سازی این حافظه‌های پنهان محلی متعدد سروکار دارد.

رویکردهای نرم افزاری و سخت افزاری برای دستیابی به انسجام کش وجود دارد. رویکرد مبتنی بر نرم‌افزار مکانیزم انسجام حافظه نهان مبتنی بر کامپایلر است که در آن ما کد را برای جلوگیری از مشکلات انسجام

حافظه پنهان با درمان متغیرهای مشترک احتمالی که ممکن است به طور جداگانه باعث مشکلات انسجام حافظه پنهان شوند، بهینه‌سازی می‌کنیم.

رویکرد مبتنی بر سخت‌افزار عمدتاً دارای پروتکل‌های پیوستگی حافظه پنهان مبتنی بر دایرکتوری و پروتکل‌های اسنوپی است.

در رویکرد انسجام کش مبتنی بر دایرکتوری(Directory-based cache coherence)، انسجام توسط یک کنترل کننده حافظه پنهان متمرکز حفظ می شود. برخلاف پروتکل اسنوپی، هر کنترل کننده کش مسئول حفظ انسجام حافظه نهان در بین تمام کنترل کننده های کش در یک سیستم چند پردازنده ای است.

Update-Based Protocol

یک پیغام جدید به دیگر CPU ها می دهیم. پیغام جدید حاوی مقدار آپدیت شده است ، و CPU هایی که پیغام رو دریافت می کنند ، کش خود را بر اساس اون پیام آپدیت می کنند ، اما این روش مرسوم نیست چون میتواند با توجه به مقدار اپدیتی و نوع متغیر اپدیتی ترافیک ها بیشتری برای ارسال پیغام ایجاد می نماید .

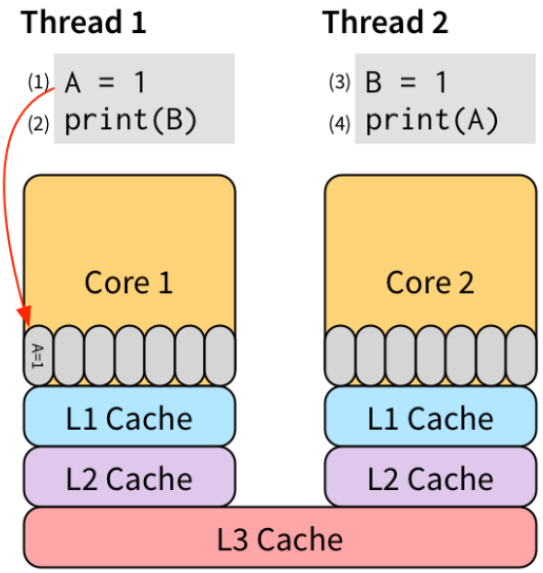
Memory Consistency Model:

:(Total store ordering)TSO

به جای اینکه منتظر بمانیم تا نوشتن (1) قابل مشاهده شود، می‌توانیم آن را در یک بافر فروشگاه قرار دهیم:

دو رشته در حال اجرا به صورت موازی

سپس (2) می تواند بلافاصله پس از قرار دادن (1) در بافر فروشگاه شروع شود، نه اینکه منتظر بماند تا به حافظه نهان L3 برسد. از آنجایی که بافر فروشگاه روی هسته است، دسترسی به آن بسیار سریع است. در زمانی در آینده، سلسله‌مراتب کش، نوشتن را از بافر ذخیره می‌کشد و آن را در کش‌ها منتشر می‌کند تا برای رشته‌های دیگر قابل مشاهده باشد. بافر ذخیره به ما اجازه می دهد تا تأخیر نوشتن را که معمولاً برای قابل مشاهده کردن نوشتن (1) برای همه رشته های دیگر لازم است، پنهان کنیم.



ب) همانطور که در اسلایدهای معرفی پروژه ذکر شده است، یکی از روشهای همگام سازی استفاده از قفلهایی مرسوم به قفل بلیت است. این قفلها را از منظر مشکل مذکور در بالا بررسی نمایید.

ticket locks:

قفل بلیت نوعی قفل است که با استفاده از یک صف که در آن پردازه ها به ترتیب درخواستشان برای دسترسی به آن قفل در آن قرار میگیرند. این قفل ها ممکن است به مشکل ناهمگامی حافظه نهان مواجه شود. این اتفاق زمانی می افتد که یک رشته مقدار قفل را تغییر میدهد و این مقدار باید در سایر حافظه های نهان هم لحاظ شود. این موضوع باعث میشود که bus به شدت درگیر شود و عملکرد سیستم کاهش یابد. برای حذف این مشکل میتوان پروتکل هایی را به کمک گرفت که در طی آن هر هسته به داده های update شده دسترسی داشته باشد و هنگامی که بخشی از اطلاعات تغییر پیدا کرد ورودی های متناظر با مقدار متفاوت در سایر حافظه های نهان invalid به حساب بیایند. پس تمام هسته ها میتوانند با داده های مشترک کار کنند.

ج) چگونه میتوان در لینوکس داده های مختص هر هسته را در زمان کامپایل تعریف نمود؟

per-core variables:

با استفاده از ماکروی زیر این قابلیت را داریم تا متغیر های مختص هر هسته را تعریف کنیم:

DEFINE_PER_CPU(int, per_cpu_n);

یک متغیر صحیح در data per cpu ایجاد شده که در هنگام پروسه kernel initialization تابع setup per cpu فراخوانی شده و در این تابع از data per cpu به دفعات (که بستگی به تعداد هسته های پردازنده دارد) استفاده میشود و متغیر های مربوط به هر هسته را ایجاد میکند.

اضافه کردن سیستم کال شمارش تعداد سیستم کال های هر هسته:

منطق کلی که برای پیاده سازی این بخش استفاده شده این است که ما به struct که برای cpu در proc.h داریم یک متغیر عددی به نام syscalls_count اضافه می کنیم. همچنین با توجه به صورت پروژه متغیر جهانی ای را هم برای نگهداری مجموع سیستم کال های فراخوانی شده استفاده میکنیم. هر بار در ابتدای تابع exec در فایل exec.c مقدار اولیه syscalls_count را صفر می کنیم.

در زیر نحوه قراردهی این متغیر ها قابل مشاهده است.

```
struct cpu {
  uchar apicid;                // Local APIC ID
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?
  struct proc *proc;           // The process running on this cpu or null
  int syscalls_count;
};

extern struct cpu cpus[NCPU];
extern int ncpu;
extern int sys_uptime(void);
extern int count_shared_syscalls;
```

هر بار که یک سیستم کال فراخوانی می شود، یکی به تعداد سیستم کال های آن هسته اضافه می شود، در ذیل در فایل syscall.c این امر اتفاق افتاده است:

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }

  pushcli();
  mycpu()->syscalls_count++;
  popcli();
  count_shared_syscalls++;
}
```

از سیستم کال زیر برای شمارش تعداد سیستم کال های فراخوانی شده در هر هسته استفاده می شود:

```c
int sys_syscalls_count(void) {
  int total_syscalls_count = 0;
  for(int i = 0 ; i < ncpu ; ++i) {
    int syscalls_count = 0;
    syscalls_count += cpus[i].syscalls_count;
    total_syscalls_count += syscalls_count;
    cprintf("cpus[%d].syscalls_count = %d\n", i, syscalls_count);
  }
  cprintf("total_syscalls_count = %d\n", total_syscalls_count);
  cprintf("Shared syscalls count = %d\n", count_shared_syscalls);
  return total_syscalls_count;
}
```

در ذیل نیز روش استفاده از این سیستم کال را در یک فایل تست در سطح کاربر مشاهده می کنید.

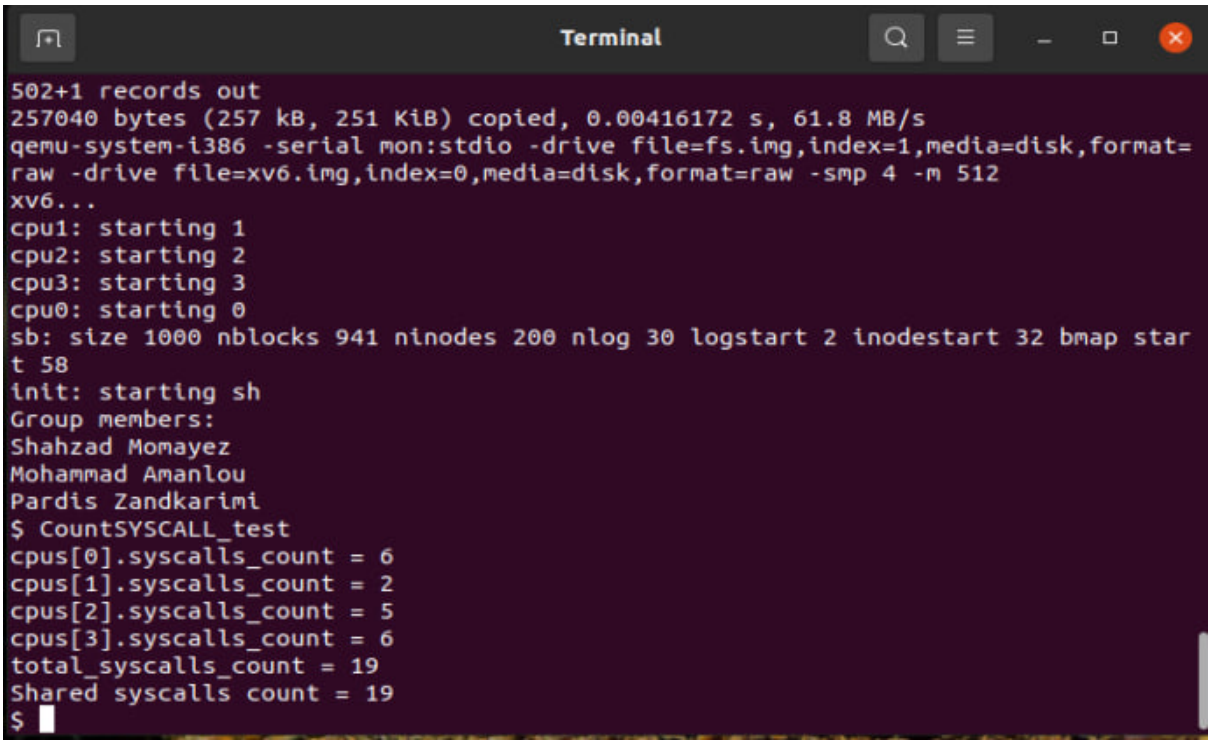
CountSYSCALL_test.c > main(int, char * [])

```c
#include "types.h"
#include "user.h"
#include "stat.h"
#include "fcntl.h"


int main(int argc, char* argv[]){
    char* write_data = "Hi everyone. This is MMD.\n";
    int fd=open("file.txt",O_CREATE|O_WRONLY);
    for (int i = 0; i < 3; i++){
        int pid = fork();
        if (pid == 0){
            volatile long long int temp = 0;
            while ((open("lockfile", O_CREATE  | O_WRONLY)) < 0){
                temp ++ ;
            } // acquiring user
            write(fd,write_data,strlen(write_data));
            unlink("lockfile"); // releasing user
            exit();

        }
    }
    while (wait() != -1);
    close(fd);
    syscalls_count();
    exit();
}
```

همچنین در ذیل نتیجه اجرا کردن این برنامه سطح کاربر قابل مشاهده است:



در نهایت قابل مشاهده است که هر یکی از core های cpu ما چه میزان فراخوانی سیستم کال داشته اند و مجموعا چند سیستم کال فراخوانی شده است.

پیاده سازی ساز و کار همگام سازی با قابلیت اولویت دادن

مشابه قفل های قبلی که از قبل در xv6 وجود داشت، یک قفل ایجاد می کنیم و آن را priority lock نامگذاری می کنیم. در یک فایل به نام priorityLock.c، به پیاده سازی 4 عمل اصلی در هر lock می پردازیم.

priorityLock.c > acquirePriorityLock(PriorityLock *)

```c
#include "spinlock.h"
#include "priorityLock.h"

void acquirePriorityLock(struct PriorityLock *lock)
{
    acquire(&lock->lock);
    while(lock->is_lock){
        sleep(lock, &lock->lock);
    }
    lock->is_lock = 1;
    lock->pid = myproc()->pid;
    release(&lock->lock);

}


void releasePriorityLock(struct PriorityLock *lock)
{
    acquire(&lock->lock);
    lock->is_lock = 0;
    lock->pid = 0;
    priority_wakeup(lock);
    release(&lock->lock);
}

void initPriorityLock(struct PriorityLock * lock){
    initlock(&lock->lock , "priority lock");
    lock->is_lock = 0;
    lock->pid = 0;

}

int
holdingPriorityLock(struct PriorityLock * lock)
{
    int ret = 0;
    acquire(&lock->lock);
    ret = (lock->pid == myproc()->pid)&& lock->is_lock;
    release(&lock->lock);
    return ret;
}
```

همچنین در هدر فایل مربوطه نیز استراکت مرتبط با این نوع قفل را ایجاد کرده ایم.

```c
struct PriorityLock{
    struct spinlock lock;
    uint is_lock;
    int pid;


};
```

به دلیل آنکه ما می خواهیم حتما پردازه هایی که pid بیشتری دارند زودتر اجرا شوند، در بخش release کردن به جای استفاده از wakeup معمول از یک wake up مخصوص این مورد استفاده می کنیم که پردازه صحیح را بیدار می کند.

```
void priority_wakeup(void* chan){
  acquire(&ptable.lock);
  struct proc *p;
  struct proc * p_max_pid = 0 ;
  int first = 1 ;
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == SLEEPING && p->chan == chan){
      if(first){
        p_max_pid = p;
        first = 0;
      }
      else{
        if(p->pid > p_max_pid->pid){
          p_max_pid = p;
        }
      }
    }
  }
  if (p_max_pid)
  {
      p_max_pid->state = RUNNABLE;


  }

  release(&ptable.lock);
}
```

سپس باید در فایل proc.c یک قفل جهانی ساخته و از آن در مدیریت پردازه ها استفاده کنیم. همچنین مقدار دهی اولیه این قفل را در فایل main.c انجام می دهیم. و در نهایت سیستم کال مربوطه را اضافه می کنیم.

```
void priorityLock_test(){
  cprintf("Process pid:%d want access to critical section\n" , myproc()->pid);
  acquirePriorityLock(&lock);
  cprintf("Process pid:%d acquired access to critical section\n" , myproc()->pid);
  volatile long long temp = 0;
  for (long long l = 0; l < 200000000; l++){
    temp += 5 * 7 + 1;
  }

  make_priority_queue(&lock);
  releasePriorityLock(&lock);
  cprintf("Process pid: %d exited from critical section\n" , myproc()->pid);
}
```

و در نهایت در فایل سطح کاربر مربوط به تست عملکرد مطابق زیر چند پردازه ایجاد کرده و با مدیریت به این روش، فرآیند وارد و خارج شدن هر پردازه از ناحیه بحرانی را مشاهده می کنیم .

PriorityLock_test.c > main()

```c
#include "types.h"
#include "user.h"
#include "fcntl.h"
#include "stat.h"



int main(){
    for (int i = 0 ; i < 3 ; i ++){
        int pid = fork();
        if(pid == 0){
            priorityLock_test();
            exit();
        }
    }
    for(int i = 0 ; i < 3 ; i++){
        wait();
    }
    exit();
}
```

در نهایت پس از اجرای این برنامه سطح کاربر به نتایج زیر خواهیم رسید:

```
Group members:
Shahzad Momayez
Mohammad Amanlou
Pardis Zandkarimi
$ PriorityLock_test
Process pid:4 want access to critical section
Process pid:4 acquired access to critical section
Process pid:5 want access to critical section
Process pid:6 want access to critical section
Queue:
Pid 5 , Name: PriorityLock_te
Pid 6 , Name: PriorityLock_te
Now it is pid 6's turn
Process pid: 4 exited from critical section
Process pid:6 acquired access to critical section
Queue:
Pid 5 , Name: PriorityLock_te
Now it is pid 5's turn
Process pid: 6 exited from critical section
Process pid:5 acquired access to critical section
Queue:
Queue is empty
Process pid: 5 exited from critical section
$
```

آیا این پیاده سازی ممکن است که دچار گرسنگی شود؟ راه حلی برای برطرف کردن این مشکل ارائه دهید.

بله. امکان دارد در هر لحظه پردازه های جدیدی وارد شوند که به دنبال قفل است و از آنجا که هر پردازه جدید یک parent_id بزرگتر از parent_id پردازه های قبلی است، بنابراین از اولویت بیشتری نیز برخودار است و در صف بیشترین اولویت را دارد. در این شرایط، پردازه هایی که ابتدا در صف رفته اند ممکن است دچار گرسنگی شوند. برای حل این مشکل از مکانیزم aging استفاده می کنیم. در این مکانیزم مدت زمان مشخصی را معین کرده و هر پردازهای پس از گذشت این مدت زمان منتظر ماندن هربار اولویتش افزایش پیدا می کند و به موقعیت جلوتری در صف اولویت منتقل می شود.

اگر aging انجام دادیم و دیدیم که اولویت دو پردازه نیز با هم یکسان بود میتوانیم time waiting را معیار قرار دهیم و مدت زمان انتظار هر یک که بیشتر بود اولویت ان را بیشتر در نظر بگیریم.

در نهایت اگر همه چیز بین این دو پردازه یکسان بود، بر اساس parent_id تصمیم گیری می کنیم و هرچه parent_id کمتر میبود، آن را در الویت قرار می دادیم.

تفاوت های بین ticketLock و priorityLcok:

ticketLock مانندFIFO عمل می کند. به عنوان مثال می توان برای این نوع از لاک، تیکت ورود به بانک را مثال زد. که هر کس زودتر برسد زودتر به اون critical section خواهد رسید . . این شرایط در سیستم عامل با استفاده از قفل بلیت به این صورت است که هر Thread یا پردازه پس از درخواست برای کسب قفل مربوطه یک نوبت دریافت میکند و سپس براساس همان نوبت پردازه ها به قفل دسترسی پیدا خواهند کرد.
در priority Queue ممکن است یکی جز اولین نفر ها درخواست ورود به critical section را داشته باشد اما چون اولویت پایینی دارد به اواخر صف برود و مثل FIFO عمل نمی کند. از طرفی قفل اولویت نیز مکانیزم همگامسازی با الگوریتم قفل است که باعث میشود پردازه ها یا thread هایی که قصد ورود به ناحیه بحرانی دارند به ترتیب اولویت وارد ناحیه بحرانی شوند. ایده آن ها مانند هم است و همچنین هر دو عملیات خود را به صورت اتمی انجام میدهند ولی به دلیل اینکه در ticket lock نسبت به ترتیب ورود پردازه ها حساسیت وجود دارد دسترسی به قفل منصفانه تر است و starvation اتفاق نمی افتد. این موضوع برای قفل اولویت برقرار نمی باشد.