گزارش پروژه چهارم آزمایشگاه درس سیستم‌عامل

پریا پاسه‌ورز 810101393 کوثر شیری جعفرزاده 810101456 پریسا یحیی‌پور فتیده 810101551

Hash کامیت آخر: f9945eb4a58ae1a2d72858ffb12e6a237a193822

---

1) علت غیرفعال شدن وقفه در حین اجرای ناحیه بحرانی، توضیح توابع pushcli و popcli و تفاوت آنها با cli وsti ؟

تابع cli برای غیرفعال کردن وقفه‌ها و تابع sti برای فعال کردن وقفه‌ها استفاده می‌شود.

توابع pushcli و popcli به ترتیب، به نوعی یک wrapper برای توابع cli و sti هستند با این تفاوت که می‌توانیم فرض کنیم یک stack مدیریت فعال کردن و یا غیرفعال کردن وقفه‌ها را به عهده می‌گیرد.

تابع pushcli به ازای هربار فراخوانی، تابع cli را صدا می‌زند و چیزی را برروی استک می گذارد و وقفه‌ها را غیرفعال می‌کند اما تابع popcli تنها زمانی با استفاده از تابع sti وقفه‌ها را فعال می‌کند که استک کاملا خالی باشد لازم به ذکر است در زمان خالی بودن استک وقفه ها فعال هستند.

پیاده سازی استک مربوطه به این نحو است که تعداد فراخوانی‌های هر یک از توابع در متغیری به نام ncli در هر پردازنده ذخیره می‌شود (به ازای فراخوانی تابع pushcli، مقدار این متغیر یک واحد افزایش پیدا می‌کند و به ازای فراخوانی تابع popcli، مقدار آن یک واحد کاهش می‌یابد) و زمانی که این متغیر برابر با 0 شود یعنی استک ما خالی است، وقفه‌ها فعال می‌شوند و هر موقع مقدار این متغیر بیشتر از 0 شود، وقفه‌ها غیرفعال می‌شوند. حال در این قسمت اگر چندبار pushcli صدا زده شده است باید به همان میزان popcli صدازده شود تا اندازه متغیر (سایز استک)به صفر برسد.

کاربرد این توابع این است که اگر برای مثال به طور همزمان از دو قفل استفاده می‌کردیم، آزاد کردن یکی از قفل‌ها سبب فعال شدن وقفه‌ها نشود و این مورد فقط زمانی انجام شود که هر دو قفل آزاد شده باشند.

2) توضیح حالات مختلف پردازه‌ها در xv6 و وظیفه تابع()sched

حالت یک پردازه در متغیر state آن که از جنس enum procstate است نگه داشته می‌شود:

این حالات عبارت است از :

enum procstate {UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE};

- UNUSED
  از آنجا که پردازه‌ها در یک لیست 64 تایی نگه داشته می‌شوند، خانه‌هایی که درشان پردازه حقیقی‌ای نیست با این stateمشخص شده‌اند و به معنی نبود پردازه است.
- EMBRYO
  وقتی که پردازه جدیدی ساخته می‌شود مثلا با fork در ابتدا حالت پردازه این است. یعنی تابع allocproc از بین پردازه‌های UNUSED یکی را انتخاب و آن را EMBRYO می‌کند.

- SLEEPING

در این وضعیت، پردازه در بین انتخاب‌های scheduler برای تخصیص پردازنده به آن قرار نمی‌گیرد و بدون هیچ فعالیتی می‌ماند. پردازه می‌تواند به صورت داوطلبانه یا توسط کرنل به این حالت برود و در انتظار دسترسی به یک منبع بماند.

- RUNNABLE

وقتی پردازه در این حالت است، یعنی در صف اجرای scheduler قرار دارد و در یکی از راندهای زمان‌بندی بعدی CPUبه آن می‌رسد و RUNNING می‌شود. چند حالت که منجر به ورود به این وضعیت می‌شوند:

پردازه تازه تشکیل شده و از EMBRYO به RUNNABLE می‌آید.

پردازه در حال اجرا و RUNNING بوده و با اتمام time slice یا yield توسط کرنل پردازنده از آن گرفته می‌شود.

پردازه در SLEEPING بوده و با wakeup قابل اجرا می‌شود.

پردازه‌ای که SLEEPING بوده kill شده و پس از 1 کردن فیلد killed آن پردازه، به حالت RUNNABLEمی‌آید تا وقتی که دوباره اجرا شد، همان اول با توجه به کشته شدن، exit شود.

- RUNNING

این حالت یعنی پردازه درحال اجرا توسط پردازنده است. تعداد پردازه‌های RUNNING در یک زمان حداکثر معادل تعداد پردازنده‌ها است.

- ZOMBIE

وقتی پردازه کارش تمام می‌شود و می‌خواهد exit بکند، ابتدا ZOMBIE می‌شود. یعنی مستقیم به UNUSED نمی‌رود و در حالتی می‌ماند که پدرش بتواند با استفاده از تابع wait از اتمام کار فرزندش باخبر شود.

تابع sched برای context switch کردن به context زمان‌بند است. پردازه برای رها کردن CPU به این تابع می‌آید (که از قبل باید stateاش از RUNNING عوض شده باشد و قفل ptable را داشته باشد). در تابع فلگ interrupt enable ذخیره شده و پس از بازگشت برگردانده می‌شود.

این تابع با استفاده از swtch، context را تغییر می‌دهد و ادامه تابع scheduler اجرا می‌شود که به context پردازه RUNNABLE دیگری تعویض می‌کند.

3) یکی از روش های سینک کردن این حافظه های نهان با یکدیگر روش Modified-Shared-Invalid است آن را به اختصار توضیح دهید

روش MSI یکی از روش های cache coherence protocol است که برای سیستم هایی به کار می رود که multiprocessor هستند و consistancy بین داده هایی که در کش پردازه های مختلف هستند دارای اهمیت بالایی هستند. این پروتکل دارای سه وضعیت برای هماهنگی بین حافظه های نهان است:

1. وضعیت M یا Modified:
   a. وضعیت cache line ولید است و جدیدترین مقدار در آن قرار دارد.
   b. داده با داده ای که در main memory هست consistent نیست بنابراین داده موجود در main memory قدیمی است.
   c. در این line تنها منبعی که داده valid دارد cache است که به این ویژگی exclusive ownership می گویند.

2. وضعیت S یا Shared
   a. وضعیت cache line ولید است و این مقدار داده با مقدار داده موجود در main memory یکسان است
   b. ممکن است این داده در cache های مختلفی وجود داشته باشد.
3. وضعیت I یا Invalid
   a. وضعیت cache line ولید نیست و نمی توان از آن استفاده کرد و داده داخل آن Invalid است.

نحوه جابجایی بین وضعیت ها

گذار	نام گذار	توضیحات
I->S	Read Miss	اگر پردازه بخواهد از خطی از کش بخواند که ولید نیست، این خط از حافظه اصلی یا کشی دیگر fetch میشود که این خط در آن Modified باشد
I->M	Write Miss	اگر پردازه بخواهد در خطی از کش بنویسد که ولید نیست، این خط از سایر کش ها fetch شده و invalid میشود
S->M	Write Hit	اگر پردازه بخواهد در خطی از کش بنویسد که در وضعیت اشتراک قرار دارد، سایر کش هایی که این خط اشتراکی را دارند invalid میشوند
S or M->S	Read Hit	اگر پردازه از خطی بخواند که در وضعیت shared یا modified هست
M or S->I	Invalidate	اگر پردازه دیگری در خط کش بنویسد، این خط در کش پردازه فعلی invalid میشود
M->S	Flush	اگر خطی از کش که در وضعیت modified قرار دارد و نیاز داریم آن را replace کنیم باید در حافظه اصلی نوشته شود

4) یکی از روش های همگام سازی استفاده از قفل هایی معروف به قفل بلیت است. این قفل ها را از منظر مشکل مذکور بررسی کنید.

ابتدا توضیح مختصری از قفل بلیت می دهیم. این نوع قفل در سیستم های multi threaded کاربرد دارد که با استفاده از آن بتوان منابع اشتراکی را به شکل عادلانه و FCFS مدیریت کرد. روش کار این قفل به این صورت است که انگار تعدادی بلیت می دهد تا نوبت کسی شود و خدمت را دریافت کند. برای پیاده سازی دوتا counter داریم:

1. شمارنده ticket: نشان دهنده شماره بلیت بعدی است که به کسی می دهیم.
2. شمارنده serving: نشان دهنده شماره بلیت فعلی است که دارد از آن منبع استفاده می کند.

اگر ترد بخواهد این lock را بگیرد:

1. بخش Acquire
   a. مقدار شمارنده ticket آپدیت می شود.
   b. می چرخد (spin) تا نوبتش شود که یعنی برابر بودن عدد شمارنده serving با شماره بلیتی که ترد دارد
2. بخش Release:
   a. مقدار شمارنده Serving را آپدیت می کند تا نوبت را به دیگری بدهد.

کمک به cache coherency

می دانیم که کش یک منبع اشتراکی محسوب می شود. اگر همزمان چند ترد بخواهند کش را آپدیت کنند، به race condition می خوریم. برای جلوگیری از این موضوع می توانیم از قفل بلیت استفاده کنیم. این قفل به شکل غیر مستقیم می تواند مشکل آپدیت شدن سایر کش ها و حافظه اصلی را تحت تاثیر قرار دهد. به این صورت که اگر نوبت ترد رسید (طبق فرایند استفاده از قفل بلیت) و این ترد تغییراتی روی کش انجام داد، همچنان که قفل را دارد، با توجه به سیاست کش آپدیت ها انجام شوند. بنابراین چون این قفل دسترسی به کش را کنترل می کند، آپدیت کردن حافظه اصلی و سایر کش ها هم به شکل مطمئن انجام خواهد شد.

5) دو مورد از معایب استفاده از قفل با امکان ورود مجدد را بیان نمایید.

1. از آنجایی که باید کدهایی برای بخش ساخت، lock و unlock این کد پیاده‌سازی شود، پس به کد بیشتری نسبت به حالت عادی نیاز داریم.
2. نسبت به خطاها حساس است، چون ممکن است به ترتیب اشتباهی قفل کنیم یا قفل‌ها را آزاد کنیم.
3. اگر به ترتیب درستی قفل کردن صورت نگیرد، با تعداد قفل‌های بالا ممکن است دچار deadlock یا livelock شویم.

6) یکی دیگر از ابزارهای همگام‌سازی قفل، Read-write lock است. نحوه کارکرد این قفل را توضیح دهید. در چه مواردی این قفل نسبت به قفل با امکان ورود مجدد برتری دارد؟

در بسیاری از شرایط، بیش از آنکه data نوشته شود یا تغییر پیدا کند، خوانده می‌شود. در این شرایط می‌توانیم به thread های مختلف اجازه دهیم که همزمان بخوانند ولی فقط یک thread، lock را در اختیار داشته باشد و بتواند بنویسد.

در واقع با کمک این lock، یکی از مسائل readers–writers problems حل می‌شود.

بعضی از این قفلها اجاره می‌دهند که قفل به صورت خودکار از read mode به write mode، upgrade شود یا اینکه از write mode به read mode، downgrade شود. در حالت اول ممکن است دچار بن‌بست شویم، چون ممکن است دو thread که هر دو reader lock را در اختیار دارند سعی کنند به write mode، upgrade شوند و دچار بن‌بست شوند. این بن‌بست تنها در صورتی شکسته می‌شود که یکی از threadها، read-lock اش را رها کند.

پیاده‌سازی:

Begin Read

- Lock *g*
- While *num_writers_waiting* > 0 or *writer_active*:
  - wait *cond*, *g*[a]
- Increment *num_readers_active*
- Unlock *g*.

End read

- Lock *g*
- Decrement *num_readers_active*
- If *num_readers_active* = 0:
    - Notify *cond* (broadcast)
- Unlock *g*.

Begin Write

- Lock *g*
- Increment *num_writers_waiting*
- While *num_readers_active* > 0 or *writer_active* is *true*:
    - wait *cond*, *g*
- Decrement *num_writers_waiting*
- Set *writer_active* to *true*
- Unlock *g*.

End Write

- Lock *g*
- Set *writer_active* to *false*
- Notify *cond* (broadcast)
- Unlock *g*.

زمانی که تعداد درخواست‌های read خیلی بیشتر از write است، و دسترسی read سبب تغییر resource مشترک نمی‌شود، بهتر است از Read-write lock استفاده کنیم. چون این قفل در شرایط ذکر شده concurrency بهتری ارائه می‌دهد و البته همانند reentrant lock، خاصیت چرخشی را داراست.