آزمایشگاه سیستم عامل

تمرین کامپیوتری 4

اعضای گروه :

محمد سوری – 810100249

صفورا علوی پناه – 810100254

ریحانه حاجبی – 810100116

همگام‌سازی در xv6

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

برای جلوگیری از مشکل deadlock باید قبل از استفاده از این قفل وقفه ها را غیر فعال کرد. همچنین غیرفعال کردن وقفه ها این اطمینان را به ما میدهند که دستورات مربوط به قفل به صورت اتمیک اجرا میشوند و چیزی بینشان قرار نمی گیرد و ترتیبشان عوض نمی شود. اگر در وقفه نیاز به قفلی داشته باشیم که در اختیار پردازهای که متوقف شده باشد، آنگاه ددالک رخ می دهد زیرا آن وقفه منتظر می ماند تا قفل آزاد شود اما پردازه ی دیگر نیز منتظر وقفه است تا وقفه تمام شود. با غیرفعال کردن وقفه ها این مشکل دیگر پیش نمیاید.

2)حالات مختلف پردازه ها در xv6 را توضیح دهید. تابع () sched چه وظیفه‌ای دارد؟

پردازه ها در سیستم عامل 6xv در یکی از حالات زیر هستند:

1. UNUSED: اگر در یک خانه از جدول پردازه ها (ptable)، واقعا یک پردازه وجود نداشته باشد. (برای مثال در آن خانه پردازه ساخته نشده یا پردازه ی مربوط به آن خانه کارش تمام شده و به اصطلاح terminate شده ،) حالت متغیر پردازه ی مربوط به آن

خانه UNUSED میشود. این به این معناست که اگر بخواهیم پردازه ای جدید بسازیم می توانیم از این خانه در جدول پردازه ها استفاده کنیم.

2. EMBRYO: زمانی که یک پردازه در مرحله ی ساخته شدن هست اما هنوز کامل آماده ی اجرا شدن نیست، در این حالت قرار میگیرد.

3. SLEEPING: زمانی که یک پردازه منتظر اتفاقی برای رخ دادن است در این حالت قرار میگیرد. برای مثال اگر منتظر پاسخ O/I باشد یا منتظر سیگنال یک تایمر باشد در حالت SLEEPING قرار می گیرد و در این حالت پردازه زمانبندی نمیشود و تا زمانی که آن رخدادی که منتظرش است رخ ندهد، اجرا نمی شود.

4. RUNNABLE: زمانی که پردازه آماده ی اجرا شدن است و منتظر پردازنده است که آن را برای اجرا زمانبندی کند، در این حالت قرار میگیرد.

5. RUNNING: زمانی که پردازه در یک پردازنده در حال اجرا است، در این حالت قرار می گیرد. در هر لحظه در هر پردازنده در سیستم عامل xv6 یک پردازه این حالت را داراست.

6. ZOMBIE: زمانی که کار یک پردازه تمام میشود و به اصطلاح terminate می شود، به حالت ZOMBIE میرود و در این حالت منتظر پردازه ی پدرش می ماند تا تمام شدن آن را با سیستم کال wait متوجه شود و بعد از آن کامل از سیستم عامل پاک شود و به حالت UNUSED برود.

هر پردازه زمانی که قرار است از حالت RUNNING خارج شود)که به 3 دلیل ممکن است رخ دهد: تمام شدن پردازه، تمام شدن تایمر یا یک اینتراپت دیگر(، تابع sched را صدا می زند. تابع switch-context sched را انجام میدهد تا به همان پردازه ای که عملیات scheduling را انجام میداد برگردیم. به همین دلیل بعد از context switch- در تابع sched، ادامه ی تابع scheduler، و از آنجایی که به یک پردازه سوییچ کرده بودیم، اجرا می شود.

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

دستور amoswap یک دستور اتمیک در مجموعه دستورات V-RISC است و برای عملیات های حافظه اتمیک استفاده می شود که در برنامه نویسی چندنخی برای همگام سازی دسترسی به منابع مشترک بسیار حیاتی است.

```
amoswap.w rd, rs۲, (rs۱)
```

این دستور به صورت اتمیک یک مقدار داده 32 بیتی با علامت را از آدرس موجود در "rs1 "می خواند، مقدار را در رجیستر "rd "قرار می دهد، مقدار خوانده شده را با مقدار اولیه 32 بیتی با علامت موجود در "rs 2 "جابه جا (swap)میکند، سپس نتیجه را دوباره در آدرس موجود در "rs1 "ذخیره می کند.

مراحل این دستور به شرح زیر است:

1.بارگذاری (load): ابتدا دستور یک مقدار داده 32 بیتی با عالمت را از آدرس حافظه مشخص شده توسط "rs1 " میخواند.

2. جابه جایی(swap): سپس این مقدار خوانده شده را با مقدار موجود در "rs 2"جابه جا میکند.

3. ذخیره (store): در نهایت، مقدار جابه جا شده(که ابتدا در " rs 2"بوده است) را دوباره در آدرس حافظه مشخص شده توسط "rs 1"ذخیره میکند.

این عملیات به صورت اتمیک انجام می شود، به این معنا که در یک مرحله اجرا می شود. این باعث می شود که اگر چند نخ همزمان دستورات amoswap را در هسته های مختلف اجرا کنند، هر عملیات amoswap یا کامل اجرا میشود یا هیچ کدام، اجرا نمی شوند. اما هرگز در یک وضعیت نیمه کامل قرار نمی گیرد. دستور amoswap معمولا در پیاده سازی ابزارهای همگام سازی مانند قفل های چرخشی (spinlock)استفاده می شود. به عنوان مثال، می تواند برای بررسی اتمیک و به دست آوردن یک قفل در صورت آزاد بودن آن استفاده شود.

4)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 نمایید.

اگر به تعریف sleeplock مراجعه کنیم، میبینیم که شناسه ی پردازه ی صاحب قفل را نگه میدارد:

```
// Long-term locks for processes
struct sleeplock {
  uint locked;      // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;       // Name of lock.
  int pid;          // Process holding lock
};
```

پس کافیست در تابع releasesleep ابتدا بررسی کنیم که آیا پردازه ی فعلی همان پردازه ی صاحب قفل هست یا خیر:

5)روشی دیگر برای نوشتن برنامه‌ها استفاده از الگوریتم های lock-free است. مختصری راجع به آن ها توضیح داده و از مزایا و معایب آن‌ها نسبت به برنامه نویسی با lock بگویید.

در برنامه نویسی free-lock همانطور که از نام آن پیداست، از قفل ها در همگام سازی استفاده نمی کنیم و به جای آن از دستورات در سطح سخت افزار(مانند دستور swap_and_compare و یا استفاده از barriers_memory)و یا متغیرهای atomic برای همگام سازی و جلوگیری از condition race استفاده میکنیم. از مزایای این روش می توان به نداشتن هزینه ی عملیات های مربوط به قفل مثل گرفتن قفل و یا آزادسازی آن اشاره کرد. همچنین به دلیل عدم نیاز به قفلها، در توسعه ی نرم افزار

امکان مقیاس پذیری بهتری(scalability) وجود دارد. در این روش مشکل deadlock نیز رخ نمیدهد. از طرفی توسعه ی نرم افزار و تست آن با روش free-lock سخت تر و زمان برتر می شود. همچنین چون در این روش از دستور های سطح پایین تر و متغیرهای atomic استفاده میکنیم، برنامه نویس نیاز به درک عمیق تر و دانش بیشتری از موضوع دارد تا بتواند برنامه ای بدون مشکل، توسعه دهد. در کل در حالت هایی که بیشتر ممکن است پردازه ای قبل ازcritical section صبر کند و به اصطلاح contention بیشتری رخ دهد، استفاده از قفل ها توصیه می شود و در غیر این صورت استفاده از دستورات سخت افزاری و عدم استفاده از قفل، بهینه تر است زیرا دیگر هزینه ی قفل را متحمل نمیشویم.

5) روشی دیگر برای نوشتن برنامه ها استفاده از الگوریتم های free-lock است. مختصری راجع به آن ها توضیح داده و از مزایا و معایب آنها نسبت به برنامه نویسی با lock بگویید.

در برنامه نویسی free-lock همانطور که از نام آن پیداست، از قفل ها در همگام سازی استفاده نمی کنیم و به جای آن از دستورات در سطح سخت افزار(مانند دستور compare_and_swap و یا استفاده از memory_barriers) و یا متغیرهای atomic برای همگام سازی و جلوگیری از race condition استفاده می کنیم. از مزایای این

روش می توان به نداشتن هزینه ی عملیات های مربوط به قفل مثل گرفتن قفل و یا آزادسازی آن اشاره کرد. همچنین به دلیل عدم نیاز به قفل ها، در توسعه ی نرم افزار امکان مقیاس پذیری بهتری(scalability) وجود دارد. در این روش مشکل deadlock نیز رخ نمیدهد.

از طرفی توسعه ی نرم افزار و تست آن با روش lock- free سخت تر و زمان برتر می شود. همچنین چون در این روش از دستور های سطح پایین تر و متغیرهای atomic استفاده می کنیم، برنامه نویس نیاز به درک عمیق تر و دانش بیشتری از موضوع دارد تا بتواند برنامه ای بدون مشکل، توسعه دهد. در کل در حالت هایی که بیشتر ممکن است پردازه ای قبل از critical section صبر کند و به اصطلاح contention بیشتری رخ دهد، استفاده از قفل ها توصیه می شود و در غیر این صورت استفاده از دستورات سخت افزاری و عدم استفاده از قفل، بهینه تر است زیرا دیگر هزینه ی قفل را متحمل نمی شویم.

پیاده سازی سازوکارهای همگام سازی جدید

در سیستم‌های عامل، مدیریت منابع و جلوگیری از شرایط رقابتی (race conditions) بسیار مهم است. یکی از روش‌های رایج برای مدیریت منابع و همگام‌سازی، استفاده از میوتکس‌ها (mutex) است. هدف این پروژه، پیاده‌سازی میوتکس با قابلیت ورود مجدد است که به یک پردازه اجازه می‌دهد تا چندین بار میوتکس را اخذ و آزاد کند بدون

اینکه خود پردازه بلاک شود. این قابلیت در مواقعی که یک تابع بازگشتی نیاز به همگام‌سازی دارد بسیار مفید است.

تعریف ساختار میوتکس: در فایل‌های هدر و کد، ساختار میوتکس با ورود مجدد تعریف شده است. این ساختار شامل موارد زیر است:

- یک اسپین‌لاک برای عملیات اتمی
- اشاره‌گری به پردازه مالک میوتکس
- شناسه پردازه مالک
- تعداد دفعاتی که میوتکس به صورت بازگشتی اخذ شده است

پیاده‌سازی توابع اصلی میوتکس:

- `reentrant_mutex_init`: برای مقداردهی اولیه میوتکس
- `reentrant_mutex_lock`: برای اخذ میوتکس با قابلیت ورود مجدد
- `reentrant_mutex_unlock`: برای آزادسازی میوتکس

برنامه تست میوتکس:

- یک سیستم کال به نام `futex_sys` برای محاسبه عدد فیبوناچی nام به صورت هم‌زمان توسط دو پردازه طراحی شده است.

- هر پردازه در تکه کدهای بحرانی (critical sections) قفل را بر روی یک ساختار مشترک که حاوی اطلاعات فیبوناچی است، اخذ می‌کند. سپس عدد بعدی را محاسبه کرده و ساختار را به‌روزرسانی می‌کند و قفل را آزاد می‌کند.

پیاده‌سازی میوتکس با ورود مجدد می‌تواند در شرایطی که توابع بازگشتی نیاز به همگام‌سازی دارند، بسیار مفید باشد. این پروژه نشان داد که با استفاده از این میوتکس، می‌توان از بلاک شدن پردازه مالک جلوگیری کرد و عملکرد بهینه‌تری داشت. پیشنهاد می‌شود برای بهبود این پروژه، تست‌های بیشتری با تعداد پردازه‌های بیشتر و حالات مختلف انجام شود.

Last commit : link