**به نام خدا**

آزمایشگاه سیستم عامل

# پروژه چهارم: همگام سازی

طراحان: امیرحسین عباسکوهی، سجاد علی‌زاده

---

## مقدمه

در این پروژه با سازوکارهای همگام‌سازی[1] سیستم‌عامل‌ها آشنا خواهید شد. با توجه به این که سیستم عامل xv6 از ریسه‌های سطح کاربر پشتیبانی نمی‌کند همگام‌سازی در سطح پردازه‌ها مطرح خواهد بود. همچنین به علت عدم پشتیبانی از حافظه مشترک در این سیستم عامل، همگام سازی در سطح هسته صورت خواهد گرفت. به همین سبب مختصری راجع به این قسم از همگام سازی توضیح داده خواهد شد.

[1] Synchronization Mechanisms

## ضرورت همگام سازی در هسته سیستم عامل ها

هسته سیستم‌عامل‌ها دارای مسیرهای کنترلی[۲] مختلفی می‌باشد. به طور کلی، دنباله دستورالعمل‌های اجرا شده توسط هسته جهت مدیریت فراخوانی سیستمی، وقفه یا استثنا این مسیرها را تشکیل می‌دهند. در این میان برخی از سیستم‌عامل‌ها دارای هسته با ورود مجدد[۳] می باشند. بدین معنی که مسیرهای کنترلی این هسته‌ها قابلیت اجرای همروند[۴] دارند. تمامی سیستم عامل‌های مدرن کنونی این قابلیت را دارند. مثلا ممکن است برنامه سطح کاربر در میانه اجرای فراخوانی سیستمی در هسته باشد که وقفه‌هایی رخ دهد. به این ترتیب در حین اجرای یک مسیر کنترلی در هسته (اجرای کد فراخوانی سیستمی)، مسیر کنترلی دیگری در هسته (اجرای کد مدیریت وقفه) شروع به اجرا نموده و به نوعی دوباره ورود به هسته صورت می‌پذیرد. وجود همزمان چند مسیر کنترلی در هسته می‌تواند منجر به وجود شرایط مسابقه برای دسترسی به حالت مشترک هسته گردد. به این ترتیب، اجرای صحیح کد هسته مستلزم همگام‌سازی مناسب است. در این همگام سازی باید ماهیت‌های مختلف کدهای اجرایی هسته لحاظ گردد.

هر مسیر کنترلی هسته در یک متن خاص اجرا می‌گردد. اگر کد هسته به طور مستقیم یا غیرمستقیم توسط برنامه سطح کاربر اجرا گردد، در متن پردازه[۵] اجرا می‌گردد. در حالی که کدی که در نتیجه وقفه اجرا می گردد در متن وقفه[۶] است. به این ترتیب فراخوانی سیستمی و استثناها در متن پردازه فراخواننده هستند. در حالی که وقفه در متن وقفه اجرا می‌گردد. به طور کلی در سیستم‌عامل‌ها کدهای وقفه قابل مسدود شدن نیستند. ماهیت این کدهای اجرایی به این صورت است که باید در اسرع وقت اجرا شده و لذا قابل زمانبندی توسط زمانبند نیز نیستند. به این ترتیب سازوکار

[2] Control Path
[3] Reentrant Kernel
[4] Concurrent
[5] Process Context
[6] Interrupt Context

همگام‌سازی آنها نباید منجر به مسدود شدن آنها گردد، مثلا از قفلهای چرخشی[۷] استفاده گردد یا در پردازنده های تک هسته ای وقفه غیر فعال گردد.

## همگام‌سازی در xv6

قفل گذاری در هسته xv6 توسط دو سری تابع صورت می‌گیرد. دسته اول شامل توابع acquire() (خط ۱۵۷۳) و release() (خط ۱۶۰۱) می‌شود که یک پیاده‌سازی ساده از قفل‌های چرخشی هستند. این قفل‌ها منجر به انتظار مشغول[۸] شده و در حین اجرای ناحیه بحرانی وقفه را نیز غیرفعال می‌کنند.

۱) علت غیرفعال کردن وقفه چیست؟ توابع pushcli() و popcli() به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

۲) چرا قفل مذکور در سیستم‌های تک‌هسته‌ای مناسب نیست؟ روی کد توضیح دهید.

دسته دوم شامل توابع acquiresleep() (خط ۴۶۲۱) و releasesleep() (خط ۴۶۳۳) بوده که مشکل انتظار مشغول را حل نموده و امکان تعامل میان پردازه‌ها را نیز فراهم می‌کنند. تفاوت اصلی توابع این دسته نسبت به دسته قبل این است که در صورت عدم امکان در اختیار گرفتن قفل، از تلاش دست کشیده و پردازنده را رها می‌کنند.

۳) مختصری راجع به تعامل میان پردازه‌ها توسط دو تابع مذکور توضیح دهید. چرا در مثال تولیدکننده/مصرف‌کننده[۹] استفاده از قفل‌های چرخشی ممکن نیست.

۴) حالات مختلف پردازه‌ها در xv6 را توضیح دهید. تابع sched() چه وظیفه ای دارد؟

یک مشکل در توابع دسته دوم عدم وجود نگهدارنده[۱۰] قفل است. به این ترتیب حتی پردازه‌ای که قفل را در اختیار ندارد می‌تواند با فراخوانی تابع releasesleep() قفل را آزاد نماید.

[7] Spin Locks
[8] Busy Waiting
[9] Producer Consumer
[10] Owner

۵) 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.

۶) یکی از روش‌های افزایش کارایی در بارهای کاری چندریسه‌ای استفاده از حافظه تراکنشی[۱۱] بوده که در کتاب نیز به آن اشاره شده است. به عنوان مثال این فناوری در پردازنده‌های جدیدتر اینتل[۱۲] تحت عنوان افزونه‌های همگام‌سازی تراکنشی[۱۳] (TSX) پشتیبانی می‌شود.[۱۴] آن را مختصراً شرح داده و نقش حذف قفل[۱۵] را در آن بیان کنید؟

## مانع

مانع‌ها[۱۶] بسته به حوزه استفاده دارای انواع مختلفی هستند. هدف کلی مانع‌ها جلوگیری از اجرای خارج از ترتیب[۱۷] است. یک دسته از مانع‌ها موسوم به مانع‌های بهینه‌سازی[۱۸] بوده که در سطح کامپایلر کاربرد دارند. به این ترتیب که وجود یک دستور مانع در یک نقطه از کد برنامه، اجازه انتقال دستورهای پیش از این نقطه به پس از آن و بالعکس را (حین بهینه‌سازی) به کامپایلر نمی‌دهد.

۷) پیاده‌سازی ماکروی barrier() در لینوکس برای معماری x86 را فقط بنویسید.

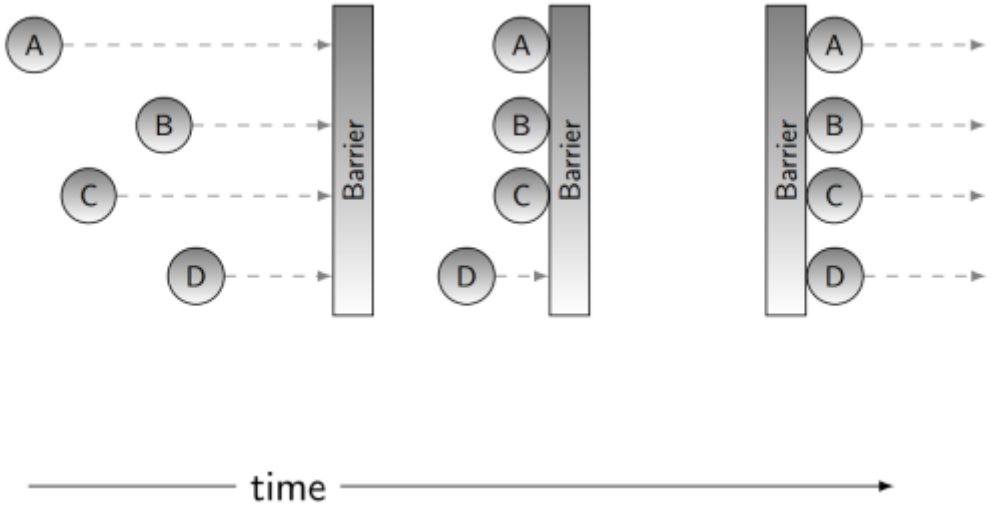
مانع بهینه‌سازی تنها از اجرای خارج از ترتیب در سطح کد جلوگیری می‌کند. جهت جلوگیری از تغییر ترتیب اجرا در سطح پردازنده از مانع‌های حافظه[۱۹] استفاده می‌شود. در این‌جا تمامی دستورالعمل‌های پیش از مانع حافظه باید پیش از هر دستورالعمل پس از آن اجرا شود.

۸) آیا یک دستور مانع حافظه باید مانع بهینه‌سازی هم باشد؟ نام ماکروی پیاده‌سازی سه نوع مانع حافظه در لینوکس در معماری x86 را به همراه دستورالعمل‌های ماشین پیاده‌سازی آن‌ها ذکر کنید.

---

11 Transcational Memory
12 Intel
13 Transactional Synchronization Extensions
۱۴ به علت وجود اشکال‌های امنیتی، در اکثر ریزمعماری‌های کنونی، غیرفعال شده است. اما ظاهراً در آینده همچنان پشتیبانی خواهد شد.
15 Lock Elision
16 Barriers
17 Out-of-Order Execution
18 Optimization Barriers
19 Memory Barriers

در نهایت نوع سومی از مانع‌ها وجود دارد که در حوزه پردازش موازی کاربرد داشته که در شکل زیر نشان داده شده است:




در اینجا مانع برای یک دسته ریسه یا پردازه در سطح زبان برنامه‌نویسی تعریف شده و هر ریسه/پردازه باید تا رسیدن تمام ریسه‌ها/پردازه‌های دیگر در نقطع مانع، توقف کند.

۹) یک کاربرد از مانع در پردازش موازی ارائه دهید.

## شبیه‌سازی مسئله فلاسفه خورنده

در این بخش از پروژه، ابتدا به پیاده‌سازی ساختار هماهنگ‌سازی سمافور[۲۰] در سطح هسته خواهید پرداخت و سپس از آن در شبیه‌سازی اجرای مسئله فلاسفه خورنده[۲۱] استفاده خواهید کرد. برای این منظور از سمافور شمارشی[۲۲] استفاده خواهیم کرد که با استفاده از آن می‌توان اجازه حضور تعداد مشخصی پردازه را به صورت همزمان در ناحیه بحرانی داد و پس از آن تعداد، باقی پردازه‌ها باید پشت سمافور منتظر بمانند. در اینجا سمافور را به صورتی پیاده‌سازی می‌کنیم که در صورتی که پردازه‌ای اجازه ورود به آن را نیافت، به حالت خواب رفته و در صف قرار می‌گیرد. سپس بعد از

[20] Semaphore
[21] Dining Philosophers
[22] Counting Semaphore

این که یکی از پردازه‌ها از ناحیه بحرانی خارج شد، پردازه‌ها را به ترتیب زمان ورود از صف خارج کرده و اجازه ورود به ناحیه بحرانی را به آن‌ها می‌دهیم.
ابتدا یک آرایه پنج تایی از سمافور در سطح سیستم ایجاد کنید که برنامه‌های سطح کاربر، از طریق فراخوانی‌های سیستمی زیر می‌توانند به آنها دسترسی داشته باشند.

**sem_init(i, v)**: سمافور در خانه i ام آرایه را با تعداد v برای حداکثر پردازه‌های درون ناحیه بحرانی ایجاد می‌کند.
**sem_acquire(i)**: زمانی که یک پردازه بخواهد وارد ناحیه بحرانی شود، این فراخوانی سیستمی را صدا می‌زند.
**sem_release(i)**: زمانی که یک پردازه بخواهد از ناحیه بحرانی خارج شود، این فراخوانی سیستمی را صدا می‌زند.

حال باید مسئله فلاسفه خورنده را با پنج فیلسوف شبیه‌سازی کنید. برای این کار، می‌بایست در سطح کاربر برنامه فیلسوف‌ها را به همراه یک برنامه آزمون بنویسید. برای هر قاشق، از یک متغیر صحیح[۲۳] استفاده کنید که شماره فیلسوفی که در لحظه آن را در اختیار دارد را داشته باشد و اگر فیلسوفی آن را در اختیار ندارد، مقدارش برابر ۱- باشد. توجه کنید پیاده‌سازی شما **نباید مشکل بن‌بست[۲۴] را داشته باشد.** برای شبیه‌سازی این مسئله می‌توانید از روشی که در کتاب توضیح داده شده (پیاده سازی متغیر شرط با استفاده از سمافور و سپس استفاده از مانیتور) یا هر روش خلاقانه دیگری استفاده کنید.

[23] Integer
[24] Deadlock

سایر نکات:

- تمیزی کد و مدیریت حافظه مناسب در پروژه از نکات مهم پیاده‌سازی است.
- از لاگ‌های مناسب در پیاده سازی استفاده نمایید تا تست و اشکال‌زدایی کد ساده‌تر شود. واضح است که استفاده بیش از حد از آنها باعث سردرگمی خواهد شد.
- فقط فایل‌های تغییر یافته و یا افزوده شده را به صورت ZIP بارگذاری نمایید.
- پاسخ تمامی سوالات را در کوتاه‌ترین اندازه ممکن در گزارش خود بیاورید.
- همه افراد باید به پروژه مسلط باشند و نمره تمامی اعضای گروه لزوما یکسان نخواهد بود.
- در صورت تشخیص تقلب، نمره هر دو گروه صفر در نظر گرفته خواهد شد.
- فصل ۴ و انتهای فصل ۵ کتاب xv6 می‌تواند مفید باشد.
- هرگونه سوال در مورد پروژه را از طریق ایمیل‌های طراحان می‌توانید مطرح نمایید.

amirhossein.abaskohi@gmail.com
sajjadalizadeh2000@gmail.com

موفق باشید