بخش اول

سوال۱:

مسیر کنترلی در سیستم عامل مسیری هست که پردازنده برای انتقال کنترل از حالت کاربر به حالت کرنل طی می‌کند و زمانی اتفاق میوفتد که نیاز به اجرا کد های هسته برای دسترسی به منابع سیستمی و یا مدیریت سخت افزار داریم

در فراخوانی سیستمی برنامه کاربر یک system call را صدا میرند و سپس cpu دستور syscall را اجرا می‌کند و سپس حالت خود را از حالت کاربر به حالت کرنل تغییر می‌دهد. پردازنده به handler فراخوانی سیستمی می‌رود عملیات مورد نظر در کرنل انجام می شود و در آخر cpu مقدار برگشتی را بر می گرداند و به حالت کاربر می رود و اجرای برنامه کاربر ادامه پیدا می‌کند.

اما در وقفه سخت افزاری یک رویداد سخت افزاری اتفاق میوفتد و cpu اجرای برنامه را متوقف می‌کند وضعیت پردازنده ها ذخیره می شود و cpu به حالت کرنل می‌رود پس از پایان وضعیت ذخیره شده بازیابی می شود و اجرای برنامه قبلی ادامه پیدا می‌کند.

در استثنا cpu در هنگام اجرای یک دستور خاص با خطا مواجه می‌شود cpu اجرای برنامه جاری را متوقف می‌کند وضعیت پردازنده ذخیره می شود و cpu به handler در کرنل می رود و handler مشکل را بررسی می‌کند و در صورت امکان حل می‌کند.

سوال۲:

کرنل reentrant به کرنلی گفته می شود که امکان ورود هم‌زمان چند مسیر اجرایی مختلف به کدهای هسته را داشته باشد بدون اینکه اجرای هم‌زمان این مسیرها باعث تخریب داده‌های مشترک یا رفتار نادرست سیستم شود در یک کرنل reentrant ممکن است در حالی که یک پردازنده یا یک مسیر کنترلی در حال اجرای کدهای کرنل است مسیر کنترلی دیگری نیز وارد کرنل شود.

جدول فایل‌ها یک ساختار داده سراسری است که توسط تمام پردازه‌ها و همچنین برخی هندلرهای وقفه مورد استفاده قرار می‌گیرد

اگر یک پردازه در حال اجرای یک فراخوانی سیستمی مانند read یا open باشد این فراخوانی سیستمی در نهایت وارد کرنل می شود و به توابع filealloc یا fileread می رسد در این توابع برای جلوگیری از دسترسی همزمان چند مسیر به جدول فایل ها قفل ftable.lock گرفته می شود.

بنابراین اگر همزمان وقفه سخت افزاری رخ بدهد cpu وارد هندلر دیسک می شود و این هندلر نیز ممکن است برای به‌روزرسانی وضعیت فایل یا بافرها، به داده‌های مشترک فایل‌سیستم دسترسی پیدا کند. در این حالت، بدون وجود قفل، هر دو مسیر کنترلی می‌توانند به‌طور هم‌زمان داده‌های فایل‌سیستم را تغییر دهند

اگر spinlock وجود نداشته باشد، این دسترسی هم‌زمان می‌تواند باعث بروز race condition شود؛ برای مثال، یک مسیر ممکن است در حال تخصیص یک ساختار فایل باشد، در حالی که مسیر دیگر همان ساختار را تغییر داده یا آزاد می‌کند. نتیجه چنین وضعیتی می‌تواند خراب شدن ساختارهای داخلی فایل‌سیستم یا از کار افتادن کل سیستم‌عامل باشد.

سوال ۳:

در سیستم‌عامل، کدهای کرنل می‌توانند در دو بستر اجرایی متفاوت اجرا شوند که به آن‌ها Process Context و Interrupt Context گفته می‌شودProcess Context به حالتی گفته می‌شود که کرنل در نتیجه اجرای یک پردازه مشخص وارد شده است. این حالت معمولاً پس از یک فراخوانی سیستمی رخ می‌دهد. در این بستر، کرنل به یک پردازه مشخص تعلق دارد و اطلاعات مربوط به آن پردازه مانند شناسه پردازه، جدول فایل‌ها و فضای آدرس‌دهی مشخص است. در Process Context، کرنل می‌تواند پردازه جاری را مسدود کند یا به حالت خواب ببرد، زیرا زمان‌بند سیستم‌عامل می‌تواند پردازه را کنار گذاشته و پردازه دیگری را برای اجرا انتخاب کند. به همین دلیل، عملیات‌هایی که ممکن است زمان‌بر باشند معمولاً در این بستر انجام می‌شوند.
در مقابل، Interrupt Context به حالتی گفته می‌شود که کرنل در پاسخ به یک وقفه سخت‌افزاری وارد اجرا می‌شود. این ورود مستقل از اجرای پردازه‌ها است و ممکن است در هر لحظه، حتی در حین اجرای کد کرنل، اتفاق بیفتد. در این بستر، کد اجراشده به پردازه خاصی تعلق ندارد و معمولاً نمی‌توان فرض کرد که پردازه جاری همان پردازه‌ای است که قبل از وقوع وقفه در حال اجرا بوده است. به همین دلیل، در Interrupt Context امکان مسدود شدن یا رفتن به حالت خواب وجود ندارد، زیرا وقفه باید در کوتاه‌ترین زمان ممکن پردازش شود و CPU به مسیر قبلی خود بازگردد.

سوال ۴:

کدی که در Interrupt Context اجرا می‌شود، مستقیماً در پاسخ به یک وقفه سخت‌افزاری وارد کرنل می‌شود و به هیچ پردازه مشخصی تعلق ندارد. هدف اصلی این کد، رسیدگی سریع به رویداد سخت‌افزاری و بازگرداندن کنترل پردازنده به مسیر اجرایی قبلی است. به همین دلیل، کدهای اجراشده در این بستر باید بسیار کوتاه، سریع و غیرمسدودکننده باشند.
علاوه بر این، اگر کد وقفه مسدود شود، پردازنده در همان مسیر باقی می‌ماند و قادر به رسیدگی به سایر وقفه‌ها نخواهد بود. این موضوع می‌تواند باعث از دست رفتن وقفه‌های بعدی، افزایش شدید تأخیر سیستم و حتی قفل شدن کامل سیستم‌عامل شود. به همین دلیل، در طراحی سیستم‌عامل‌ها، هندلرهای وقفه تنها وظایف حداقلی مانند به‌روزرسانی وضعیت‌ها، ثبت رویداد و بیدار کردن پردازه‌های منتظر را انجام می‌دهند و ادامه‌ی پردازش‌های سنگین به Process Context واگذار می‌شود.

سوال۵:

از آنجایی در Interrupt Context امکان مسدود شدن یا رفتن به خواب وجود ندارد نمی توان از قفل هایی که باعث به خواب رفتن می شود استفاده کرد

به همین دلیل از spinlock استفاده میکنیم spinlock به‌جای خواب بردن پردازه، باعث می‌شود پردازنده به‌صورت مشغول منتظر آزاد شدن قفل بماند.

اما استفاده از spinlock به‌تنهایی کافی نیست برای مثال یک پردازنده ممکن است spinlock را بگیرد و سپس یک وقفه سخت‌افزاری رخ دهد. در این حالت، CPU وارد هندلر وقفه می‌شود و اگر هندلر وقفه نیز تلاش کند همان spinlock را بگیرد، پردازنده وارد یک بن‌بست می‌شود؛ زیرا قفل در اختیار همان CPU است و هرگز آزاد نخواهد شد پس برای جلوگیری از این اتفاق 6xv هنگام گرفتن spinlock، وقفه‌ها را روی همان پردازنده غیرفعال می‌کند. این کار تضمین می‌کند که در بازه‌ای که قفل در اختیار پردازنده است، هیچ وقفه‌ای نتواند اجرا شود و مسیر کنترلی دیگری نتواند مجدداً تلاش کند همان قفل را بگیرد.

سوال۶:

Spinlock

یکی از ساده‌ترین و سریع‌ترین مکانیزم‌های همگام‌سازی در سیستم‌عامل است که در آن، اگر قفل در اختیار مسیر اجرایی دیگری باشد، پردازنده به‌صورت مشغول در یک حلقه باقی می‌ماند تا قفل آزاد شود. این نوع قفل معمولاً در سطح کرنل و به‌ویژه در Interrupt Context استفاده می‌شود، زیرا در این بستر امکان مسدود شدن یا رفتن به حالت خواب وجود ندارد. Spinlockها برای نواحی بحرانی بسیار کوتاه طراحی شده‌اند و در صورت استفاده صحیح، سربار کمی دارند.

مزایا:

سربار کمی دارد و برای critical section هایی که کوتاه مدت هستند سریع است و می توان از آن در Interrupt Context استفاده کرد

معایب:

اگر مدت زمان انتظار طولانی شود باعث هدر رفتن cpu می شود و برای عملیات های زمان بر مناسب نیست

Sleeplock

مکانیزمی است که در آن، اگر قفل در دسترس نباشد، پردازه به حالت خواب می‌رود و CPU به پردازه دیگری واگذار می‌شود این رویکرد برای critical section طولانی‌تر، به‌خصوص در عملیات‌های io مناسب است. Sleeplock تنها در Process Context قابل استفاده است، زیرا نیازمند تعامل با زمان‌بند سیستم‌عامل برای خواب و بیدار کردن پردازه‌ها است.

مزایا:

زمان انتظار باعث هدر رفتن cpu نمی شود و برای عملیات های طولانی و io مناسب است

معایب:

دارای سربار است که ناشی از context switch می باشد.

Lock-Free Programming

به‌جای استفاده از قفل‌های سنتی، از دستورات اتمیک و الگوریتم‌های خاص برای مدیریت هم‌زمانی استفاده می‌شود. در این روش، سیستم به‌گونه‌ای طراحی می‌شود که حتی در صورت رقابت چند مسیر اجرایی، حداقل یکی از آن‌ها همیشه پیشرفت کند. این رویکرد بیشتر در سیستم‌های با مقیاس‌پذیری بالا و پردازنده‌های چندهسته‌ای استفاده می‌شود، اما پیاده‌سازی آن بسیار پیچیده است

مزایا:

مقایس پذیری بالایی دارد و خطر deadlock هم ندارد

معایب:

طراحی و پیاده سازی آن بسیار پیچیده است و دیباگ و اثبات درستی آن کار دشوار است.

سوال۷:

فرضا پردازنده در حال اجرای کدی در کرنل است و با فراخوانی تابع acquire، یک spinlock را با موفقیت می‌گیرد. در این حالت، قفل در اختیار CPU قرار دارد. حال اگر وقفه‌ها فعال باشند، ممکن است بلافاصله پس از گرفتن قفل، یک وقفه سخت‌افزاری مانند وقفه تایمر رخ دهد.

در نتیجه این وقفه، CPU اجرای کد فعلی را متوقف می‌کند و وارد هندلر وقفه می‌شود. اگر هندلر وقفه برای انجام وظیفه خود نیاز داشته باشد همان spinlock را بگیرد تلاش می‌کند قفل را با فراخوانی acquire بگیرد. اما از آنجا که قفل قبلاً توسط همان CPU گرفته شده است، هندلر وقفه وارد حالت انتظار مشغول می‌شود.

در این وضعیت، پردازنده در داخل هندلر وقفه در حال spin کردن است و هرگز نمی‌تواند به کد قبلی بازگردد تا قفل را آزاد کند. از سوی دیگر، قفل نیز هرگز آزاد نمی‌شود، زیرا تنها پردازنده موجود در سیستم در یک حلقه بی‌پایان گرفتار شده است. این وضعیت یک Deadlock قطعی در سیستم تک‌هسته‌ای محسوب می‌شود.

سوال۸:

دستورات cli و sti به‌ترتیب برای غیرفعال و فعال‌سازی وقفه‌ها در پردازنده استفاده می‌شوند. استفاده مستقیم از این دستورات در کرنل می‌تواند در شرایط ساده مفید باشد، اما در حالت‌های پیچیده‌تر، به‌ویژه زمانی که نواحی بحرانی به‌صورت تودرتو (Nested) در داخل یکدیگر قرار می‌گیرند، منجر به بروز خطاهای جدی می‌شود. به همین دلیل، 6xv به‌جای استفاده مستقیم از cli و sti، از توابع سطح بالاتر pushcli و popcli استفاده می‌کند.

در 6xv، ممکن است یک مسیر اجرایی وارد یک ناحیه بحرانی شود و وقفه‌ها را غیرفعال کند، و سپس در داخل همان ناحیه بحرانی، وارد ناحیه بحرانی دیگری شود که آن هم نیاز به غیرفعال‌سازی وقفه‌ها دارد. اگر در این شرایط از cli و sti به‌صورت مستقیم استفاده شود، ممکن است با خروج از ناحیه بحرانی داخلی، وقفه‌ها مجدداً فعال

شوند، در حالی که هنوز در ناحیه بحرانی بیرونی قرار داریم. این وضعیت می‌تواند باعث اجرای وقفه در زمانی شود که داده‌های حساس کرنل هنوز محافظت نشده‌اند.
توابع pushcli و popcli این مشکل را با استفاده از یک شمارنده تودرتویی حل می‌کنند. تابع pushcli در هنگام فراخوانی، وقفه‌ها را غیرفعال کرده و یک شمارنده را افزایش می‌دهد که نشان‌دهنده تعداد دفعات ورود به ناحیه بحرانی است. در مقابل، تابع popcli این شمارنده را کاهش می‌دهد و تنها زمانی وقفه‌ها را مجدداً فعال می‌کند که شمارنده به صفر برسد، یعنی تمام نواحی بحرانی تودرتو به‌طور کامل خاتمه یافته باشند.

بخش دوم

سوال ۱:

تSpinlock مکانیزمی از همگام‌سازی است که بر اساس Busy Waiting عمل می‌کند و فرض می‌کند پردازنده‌ای که قفل را در اختیار دارد، در مدت زمان کوتاهی critical section را ترک کرده و قفل را آزاد خواهد کرد. در چنین شرایطی، فعال بودن وقفه‌ها روی پردازنده‌ای که spinlock را گرفته است می‌تواند منجر به Deadlock شود.اگر وقفه‌ها فعال باشند، ممکن است پردازنده پس از گرفتن spinlock با یک وقفه سخت‌افزاری مواجه شود. در این حالت، CPU اجرای کد فعلی را متوقف کرده و وارد هندلر وقفه می‌شود. اگر هندلر وقفه نیز برای انجام وظیفه خود نیازمند همان spinlock باشد، تلاش می‌کند قفل را بگیرد. اما از آنجا که قفل قبلاً توسط همان پردازنده گرفته شده است، هندلر وقفه وارد حالت انتظار مشغول می‌شود.
در این وضعیت، پردازنده در داخل هندلر وقفه در حال spin کردن باقی می‌ماند و هرگز نمی‌تواند به کد قبلی بازگردد تا قفل را آزاد کند. این شرایط یک Self-Deadlock ایجاد می‌کند که به‌ویژه در سیستم‌های تک‌هسته‌ای بن‌بست قطعی و در سیستم‌های چندهسته‌ای نیز منبع جدی اختلال محسوب می‌شود.
به همین دلیل، در 6xv و سایر سیستم‌عامل‌ها، هنگام گرفتن spinlock، وقفه‌ها روی همان پردازنده غیرفعال می‌شوند. این کار تضمین می‌کند که تا زمانی که قفل در اختیار پردازنده است، هیچ وقفه‌ای نتواند اجرا شود و مسیر کنترلی دیگری تلاش نکند همان قفل را مجدداً بگیرد. در نتیجه، یکپارچگی داده‌های کرنل حفظ شده و از وقوع بن‌بست جلوگیری می‌شود.

سوال۲:

در صورتی که وقفه‌ها هنگام در اختیار داشتن یک spinlock غیرفعال نشوند ممکن است نوع خاصی از بن‌بست به نام Self-Deadlock یا Deadlock بازگشتی رخ دهد. این نوع بن‌بست زمانی اتفاق می‌افتد که یک پردازنده، خودش مانع پیشرفت خود شود. فرضا پردازنده در حال اجرای کدی در کرنل است و با فراخوانی تابع acquire، یک spinlock را در اختیار می‌گیرد. در این لحظه، قفل به‌طور کامل توسط همان CPU گرفته شده است. قبل از آنکه پردازنده به اجرای ناحیه بحرانی پایان دهد و قفل را آزاد کند، یک وقفه سخت‌افزاری مانند وقفه تایمر رخ می‌دهد. در اثر این وقفه، CPU اجرای کد جاری را متوقف کرده و وارد هندلر وقفه می‌شود. اگر هندلر وقفه برای انجام وظیفه

خود نیاز به دسترسی به همان داده مشترک داشته باشد، تلاش می‌کند همان spinlock را بگیرد. از آنجا که قفل در اختیار همان پردازنده است، هندلر وقفه وارد حالت انتظار مشغول spin می‌شود.در این حالت، پردازنده درون هندلر وقفه در یک حلقه بی‌پایان باقی می‌ماند و هرگز نمی‌تواند به مسیر اجرایی قبلی بازگردد تا قفل را آزاد کند. از سوی دیگر، قفل نیز هرگز آزاد نخواهد شد، زیرا تنها پردازنده موجود در سیستم درگیر این انتظار مشغول است. این وضعیت یک بن‌بست قطعی ایجاد می‌کند که به آن Self-Deadlock گفته می‌شود.

سوال۳:

در سیستم‌های چندهسته‌ای، هر هسته پردازنده دارای حافظه نهان (Cache) اختصاصی خود است. زمانی که چندین هسته به‌طور هم‌زمان و مکرر یک متغیر مشترک سراسری مانند GlobalCounter را تغییر می‌دهند، پدیده‌ای به نام Cache Line Bouncing یا Cache Thrashing رخ می‌دهد که عامل اصلی کند شدن سیستم است. این پدیده به این صورت اتفاق می‌افتد که متغیر سراسری در یک Cache Line قرار دارد. هر بار که یکی از هسته‌ها مقدار این متغیر را تغییر می‌دهد، باید مالکیت آن Cache Line را به‌صورت انحصاری در اختیار بگیرد. در نتیجه، پروتکل‌های همگام‌سازی حافظه نهان مانند MESI وارد عمل می‌شوند و خطوط کش مربوط به سایر هسته‌ها را نامعتبر می‌کنند. پس از آن، اگر هسته دیگری بخواهد همان متغیر را تغییر دهد، باید مجدداً آن خط کش را از حافظه اصلی یا از کش هسته دیگر دریافت کند.این رفت‌وآمد مداوم یک خط کش بین هسته‌ها باعث افزایش شدید ترافیک روی Memory Bus و کاهش کارایی کل سیستم می‌شود. در چنین شرایطی، حتی اگر عملیات انجام‌شده روی متغیر بسیار ساده باشد، هزینه‌ی هماهنگ‌سازی کش‌ها بسیار بیشتر از خود عملیات محاسباتی خواهد بود.پروتکل MESI با تعریف حالت‌های مختلف برای Cache Lineتلاش می‌کند سازگاری داده‌ها بین کش‌های مختلف را حفظ کند. با این حال، در حالتی که چندین هسته به‌طور مداوم یک متغیر مشترک را می‌نویسند، MESI ناچار است بارها خطوط کش را نامعتبر کرده و مالکیت را جابه‌جا کند، که همین موضوع منشأ اصلی افت کارایی است.

سوال۴:

در رویکرد CPU-per، به‌جای استفاده از یک متغیر سراسری مشترک، برای هر هسته پردازنده یک نسخه مستقل از متغیر نگه‌داری می‌شود. به این ترتیب، هر CPU فقط متغیر محلی خودش را تغییر می‌دهد و نیازی به هماهنگی مداوم با کش سایر هسته‌ها وجود ندارد. از آنجا که هر متغیر CPU-per معمولاً فقط توسط یک هسته نوشته می‌شود، خط کش مربوط به آن در حالت انحصاری باقی می‌ماند و پروتکل‌های همگام‌سازی کش مانند MESI مجبور به انجام عملیات invalidate روی کش‌های دیگر نمی‌شوند.در این حالت، تعداد invalidationهای کش به‌شدت کاهش پیدا می‌کند، زیرا هیچ هسته‌ای به‌طور مستقیم روی داده‌ای که توسط هسته دیگر تغییر داده می‌شود، عملیات نوشتن انجام نمی‌دهد. در نتیجه، ترافیک گذرگاه حافظه کاهش یافته و کارایی سیستم به‌ویژه در بارهای کاری با هم‌زمانی بالا به‌طور قابل توجهی افزایش می‌یابد.در نهایت، زمانی که نیاز به مقدار نهایی متغیر

وجود داشته باشد، می‌توان مقادیر محلی هر CPU را با یک عملیات تجمیع (Aggregation) ساده با هم جمع کرد. این کار به‌مراتب کم‌هزینه‌تر از هماهنگ‌سازی مداوم روی یک متغیر سراسری در طول اجرای سیستم است.

سوال ۵:

تابع Spinlock بر اساس Busy Waiting عمل می‌کند. در این نوع قفل، اگر قفل در اختیار مسیر اجرایی دیگری باشد، پردازنده به‌طور فعال در یک حلقه باقی می‌ماند و به‌صورت مداوم وضعیت قفل را بررسی می‌کند تا آزاد شود. در این حالت، پردازنده مشغول باقی می‌ماند و به پردازه یا هسته دیگری واگذار نمی‌شود. به همین دلیل، spinlock برای نواحی بحرانی کوتاه و عملیات‌های سریع مناسب است و معمولاً در سطح کرنل و Interrupt Context مورد استفاده قرار می‌گیرد.
تابع Sleeplock به‌گونه‌ای طراحی شده است که در صورت اشغال بودن قفل، پردازه منتظر به حالت خواب می‌رود و CPU به پردازه دیگری واگذار می‌شود. این نوع قفل باعث می‌شود از هدررفت پردازنده جلوگیری شود، اما در عوض سربار context switch را به سیستم تحمیل می‌کند. به همین دلیل، sleeplock برای نواحی بحرانی طولانی‌تر، به‌ویژه عملیات‌های I/O، مناسب است و تنها در Process Context قابل استفاده می‌باشد.

بخش سوم

سوال ۱:

گرسنگی زمانی رخ می‌دهد که یک پردازه با وجود آن‌که بارها درخواست دسترسی به منبع را مطرح می‌کند، به دلیل سیاست تخصیص قفل هرگز موفق به دریافت آن نمی‌شود.
در plock، سیاست اعطای قفل بر اساس اولویت پردازه‌ها انجام می‌شود، به این معنا که هر بار که قفل آزاد می‌شود، پردازه‌ای که بالاترین اولویت را دارد انتخاب می‌شود، بدون توجه به مدت زمانی که سایر پردازه‌ها در صف انتظار بوده‌اند. این سیاست اگرچه برای پردازه‌های مهم‌تر مناسب است، اما می‌تواند باعث نادیده گرفته شدن پردازه‌های با اولویت پایین شود. بنابراین در طراحی قفل الویت دار امکان بروز گرسنگی وجود دارد.

سوال ۲:

قفل بلیطی یک مکانیزم همگام‌سازی است که با الهام از سیستم نوبت‌دهی نانوایی طراحی شده و هدف اصلی آن تضمین انصاف در اعطای قفل است. در این روش، هر مسیر اجرایی که قصد گرفتن قفل را دارد، ابتدا یک بلیط با شماره‌ای یکتا دریافت می‌کند و سپس منتظر می‌ماند تا نوبت آن بلیط فرا برسد.
در Ticket Lock دو شمارنده سراسری نگه‌داری می‌شود یکی برای تخصیص بلیط جدید و دیگری برای مشخص کردن بلیطی که در حال حاضر اجازه ورود به ناحیه بحرانی را دارد. زمانی که یک پردازه یا پردازنده درخواست قفل می‌دهد، مقدار next_ticket به‌صورت اتمیک افزایش می‌یابد و مقدار قبلی آن به‌عنوان بلیط پردازه ثبت می‌شود. سپس پردازه در یک حلقه منتظر می‌ماند تا مقدار now_serving برابر با شماره بلیط آن شود.وقتی صاحب فعلی

قفل کار خود را تمام می‌کند و قفل را آزاد می‌سازد، مقدار now_serving افزایش می‌یابد. این کار باعث می‌شود پردازه‌ای که بلیط بعدی را دارد، وارد ناحیه بحرانی شود. به این ترتیب، قفل دقیقاً به ترتیب ورود درخواست‌ها و بدون هیچ‌گونه رقابت یا شانس تصادفی بین پردازه‌ها واگذار می‌شود.در Ticket Lockهیچ پردازه‌ای از دریافت قفل محروم نمی‌شود و هر درخواست در نهایت و پس از طی شدن نوبت‌های قبلی، به قفل دسترسی پیدا می‌کند. به همین دلیل، این نوع قفل به‌طور طبیعی در برابر گرسنگی مقاوم است، هرچند ممکن است نسبت به قفل‌های ساده‌تر سربار بیشتری داشته باشد.

سوال۳:

از نظر انصاف Ticket Lock رفتاری کاملاً منصفانه دارد. در این قفل، درخواست‌ها به ترتیب ورود سرویس داده می‌شوند و هیچ پردازه‌ای از نوبت خود عبور داده نمی‌شود. در مقابل، plock منصفانه به معنای کلاسیک نیست، زیرا ترتیب اعطای قفل بر اساس اولویت پردازه‌ها تعیین می‌شود و نه زمان درخواست. در نتیجه، پردازه‌های با اولویت پایین ممکن است بارها نادیده گرفته شوند، حتی اگر زودتر از دیگران درخواست قفل داده باشند.
از نظر پیچیدگی پیاده‌سازی Ticket Lock ساختار ساده‌تری دارد. این قفل تنها به دو شمارنده اتمیک نیاز دارد و منطق آن مستقیم و قابل پیش‌بینی است. در مقابل، plock به ساختارهای داده پیچیده‌تری مانند صف انتظار، نگه‌داری اطلاعات پردازه‌ها و جست‌وجوی پردازه با بالاترین اولویت نیاز دارد. این موضوع باعث می‌شود پیاده‌سازی plock پیچیده‌تر و مستعد خطاهای طراحی باشد.
از نظر احتمال گرسنگی Ticket Lock به‌طور ذاتی در برابر گرسنگی مقاوم است، زیرا هر پردازه پس از دریافت بلیط، قطعاً و پس از طی شدن نوبت‌های قبلی به قفل دسترسی پیدا می‌کند. در مقابل، plock در صورت ورود مداوم پردازه‌های با اولویت بالا می‌تواند باعث گرسنگی پردازه‌های با اولویت پایین شود، مگر آن‌که مکانیزم‌های تکمیلی مانند افزایش تدریجی اولویت در آن پیاده‌سازی شود. بنابراین Ticket Lock برای سیستم‌هایی که انصاف و پیش‌بینی‌پذیری اهمیت بیشتری دارد بهتر است،در حالی که plock برای سیستم‌های حساس یا بلادرنگ که تقدم پردازه‌های مهم‌تر اولویت دارد، گزینه بهتری است.

سوال 1 - عملی)

در فایل `sleeplock.h`، یک فیلد `int pid` به ساختار `struct sleeplock` اضافه کردیم.

هدف: ذخیره شناسه پروسه‌ای که هم‌اکنون قفل را در اختیار دارد.

سوال ۲: ثبت مالک در زمان گرفتن قفل

- در تابع `acquiresleep` (فایل `sleeplock.c`)، پس از اینکه قفل با موفقیت گرفته شد (`lk->locked = 1`)، خط `lk->pid = myproc()->pid` را اضافه کردیم.

هدف: مشخص شدن صاحب قفل در لحظه تصاحب.

سوال ۳: ایمن‌سازی آزادسازی قفل

- در تابع `releasesleep`، شرطی اضافه کردیم که چک می‌کند آیا `lk->pid` با `myproc()->pid` برابر است یا خیر. اگر برابر نبود، سیستم `panic` می‌کند.
- هدف: جلوگیری از اینکه یک پروسه بیگانه (غیر مالک) قفل پروسه دیگری را آزاد کند.

سوال ۴ (و بخش امتیازی): طراحی قفل Reader-Writer

- تغییر: ساختار جدید `struct rwlock` را با فیلدهای `read_count` و `write_locked` پیاده‌سازی کردیم. توابع `acquire_read` (اجازه ورود چندگانه) و `acquire_write` (ورود انحصاری) را با استفاده از `sleep/wakeup` نوشتیم.
- هدف: افزایش کارایی در سناریوهایی که تعداد خوانندگان زیاد است (مثل لیست فایل‌ها).

بخش دوم

هدف این بخش، اندازه‌گیری میزان رقابت (Contention) روی قفل‌های سیستم‌عامل با استفاده از داده‌های اختصاصی هر پردازنده (Per-CPU) بود. ما قفل `ptable.lock` را به عنوان قفل هدف برای بررسی انتخاب کردیم.

صورت سوال: اضافه کردن آرایه‌هایی برای ذخیره آمار هر هسته پردازنده به صورت جداگانه. کاری که انجام دادیم: در فایل `spinlock.h`، به ساختار `struct spinlock` دو آرایه اضافه کردیم. این کار باعث می‌شود هر CPU مکان حافظه مخصوص به خود را برای ثبت آمار داشته باشد و از "تداخل حافظه نهان" (Cache Thrashing) جلوگیری شود.

- **acq_count[NCPU]: تعداد دفعاتی که هر CPU موفق به گرفتن این قفل شده است.**
- **total_spins[NCPU]: مجموع دفعاتی که هر CPU در حلقه while منتظر مانده (Spin کرده) تا قفل آزاد شود.**

سوال ۴: مقداردهی اولیه (initlock در spinlock.c)

کاری که انجام دادیم: در تابع `initlock`، یک حلقه `for` اضافه کردیم که به ازای تمام هسته‌ها (از ۰ تا NCPU-1)، مقادیر `acq_count` و `total_spins` را برابر صفر قرار می‌دهد تا آمار از لحظه شروع سیستم تمیز باشد.

سوال ۵: منطق جمع‌آوری آمار (acquire در spinlock.c)

کاری که انجام دادیم: این مهم‌ترین تغییر در هسته بود. در تابع acquire:

1. **یک متغیر محلی loop_count تعریف کردیم.**
2. **در یک حلقه که پردازنده منتظر قفل است، این متغیر را زیاد کردیم.**
3. **پس از گرفتن قفل (و خروج از حلقه)، با استفاده از تابع cpuid() شناسه پردازنده جاری را گرفتیم.**
4. **مقدار acq_count آن پردازنده را یکی زیاد کردیم و مقدار loop_count را به total_spins آن اضافه کردیم.**

سوال ۶: پیاده‌سازی سیستم‌کال در کرنل (sys_getlockstat در sysproc.c)

یک فراخوانی سیستمی جدید به نام `sys_getlockstat` نوشتیم. این تابع:

1. به قفل سراسری `ptable.lock` (که در `proc.c` تعریف شده) دسترسی پیدا می‌کند.
2. یک آرایه از سمت کاربر دریافت می‌کند.
3. آمار `total_spins` (یا نسبت `total/acq`) مربوط به هر CPU را در یک آرایه موقت کپی می‌کند.
4. با استفاده از تابع `copyout`، این داده‌ها را به فضای آدرس پروسه کاربر منتقل می‌کند.

سوال ۷: رابط کاربری سیستم‌کال (`user.h`, `usys.S`, `syscall.c`)

برای اینکه برنامه‌های ما بتوانند این سرویس کرنل را صدا بزنند، مراحل استاندارد ثبت سیستم‌کال را انجام دادیم:

- اضافه کردن شماره سیستم‌کال در `syscall.h`.
- تعریف امضای تابع `int getlockstat(uint *)` در `user.h`.
- اتصال آن در `usys.S` و `syscall.c`.

سوال ۸: برنامه تست فشار (`locktest.c`)

برنامه‌ای نوشتیم که

1. ابتدا آمار فعلی قفل را چاپ می‌کند (Initial Stats).
2. با استفاده از `fork` چندین پروسه فرزند ایجاد می‌کند.
3. هر فرزند در یک حلقه طولانی، توابع `fork` و `exit` و `wait` را صدا می‌زند. این توابع به شدت با `ptable.lock` درگیر می‌شوند و رقابت ایجاد می‌کنند.
4. در پایان، آمار نهایی (Final Stats) را چاپ می‌کند که نشان‌دهنده تغییرات چشمگیر در تعداد Spinها است.

تحلیل اعداد به دست آمده. نتیجه‌گیری ما: در خروجی مشاهده کردیم که اعداد `total_spins` برای هسته‌ها بزرگ شدند . این نشان داد که:

1. گلوگاه بودن `ptable.lock`: تمام پردازنده‌ها برای ایجاد یا حذف پروسه مجبورند از یک قفل واحد عبور کنند که باعث صف‌بندی و اتلاف وقت پردازنده (Spinning) می‌شود.
2. کارایی پروفایلینگ: مکانیزم ما به درستی توانست نقاط داغ (Hotspots) سیستم را شناسایی کند.

بخش 3

۱. طراحی ساختار داده (`plock.h`)

- تغییرات: یک ساختار `struct plock` تعریف کردیم که شامل یک صف انتظار (`head`) و وضعیت قفل است. همچنین برای جلوگیری از استفاده از `malloc` در کرنل، از یک آرایه ثابت (`node_pool`) برای مدیریت گره‌های صف استفاده کردیم. هر گره شامل اشاره‌گر به پروسه و اولویت (`priority`) آن است.

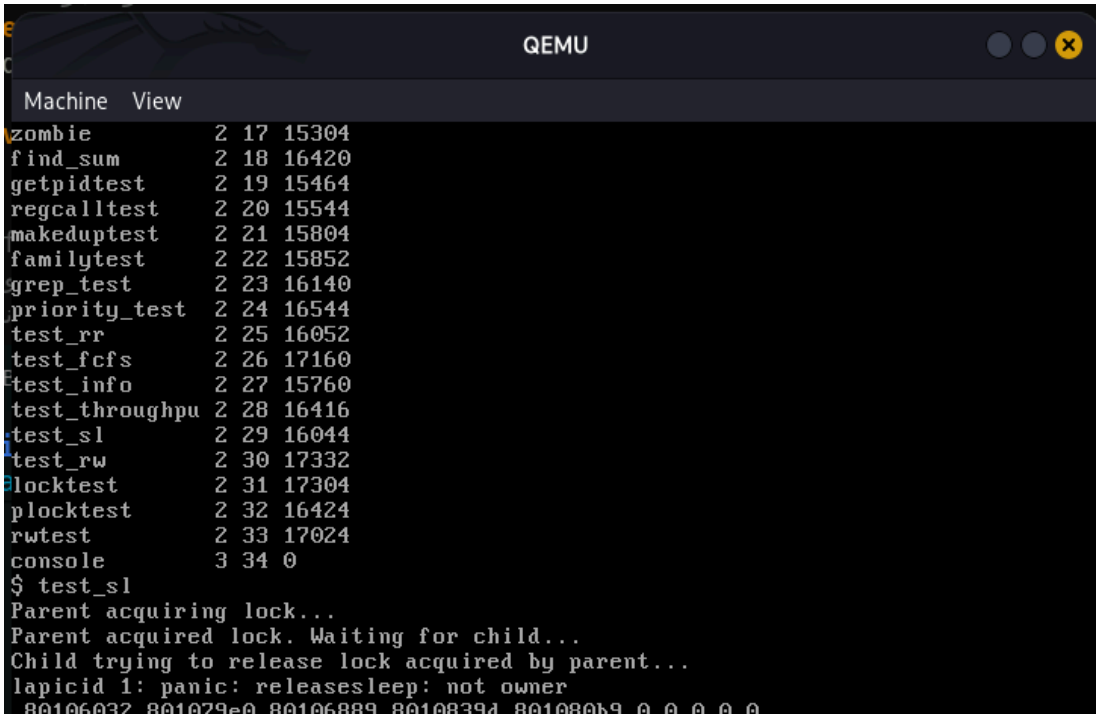
۲. پیاده‌سازی توابع `acquire` و `release`

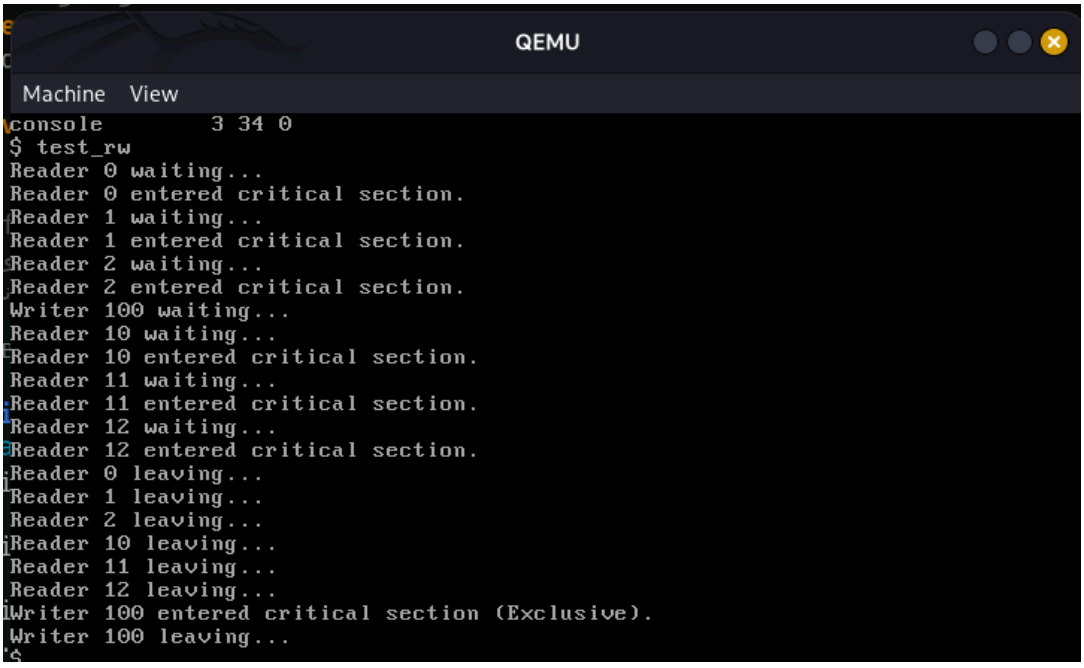
- تابع `plock_acquire`: اگر قفل اشغال باشد، به جای انتظار چرخشی (Spin)، یک گره جدید با اولویت پروسه جاری می‌سازیم و به صف اضافه می‌کنیم. سپس پروسه روی آدرس همان گره به خواب (`sleep`) می‌رود.
- تابع `plock_release` (مکانیزم Handoff): هنگام آزادسازی، لیست انتظار پیمایش می‌شود تا گره‌ای با بالاترین اولویت (Max Priority) پیدا شود. سپس قفل مستقیماً به آن پروسه منتقل می‌شود (بدون اینکه متغیر `locked` صفر شود تا رقابت مجدد رخ ندهد) و تنها همان پروسه با `wakeup` بیدار می‌شود.

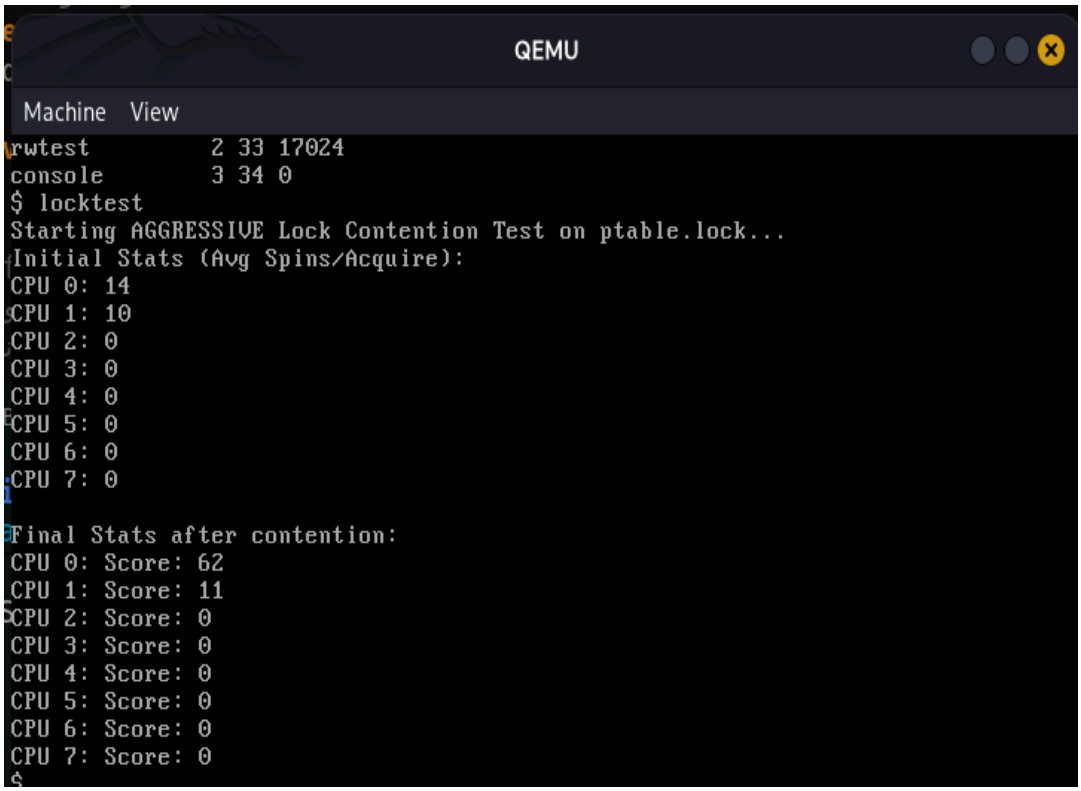
۳. سیستم‌کال‌ها و تست

- توابع `sys_plock_acquire` و `sys_plock_release` را به کرنل اضافه کردیم تا روی یک قفل سراسری (`global_plock`) عمل کنند.
- نتایج تست (`plocktest`): همانطور که در تصاویر ضمیمه مشخص است، سه پروسه با اولویت‌های ۱۰، ۵۰ و ۳۰ درخواست قفل دادند. با اینکه پروسه ۵۰ دیرتر از ۱۰ درخواست داد، اما پس از آزاد شدن قفل، پروسه ۵۰ زودتر قفل را گرفت. ترتیب دریافت قفل در خروجی (۵۰ -> ۳۰ -> ۱۰) نشان‌دهنده عملکرد صحیح اولویت‌دهی است.

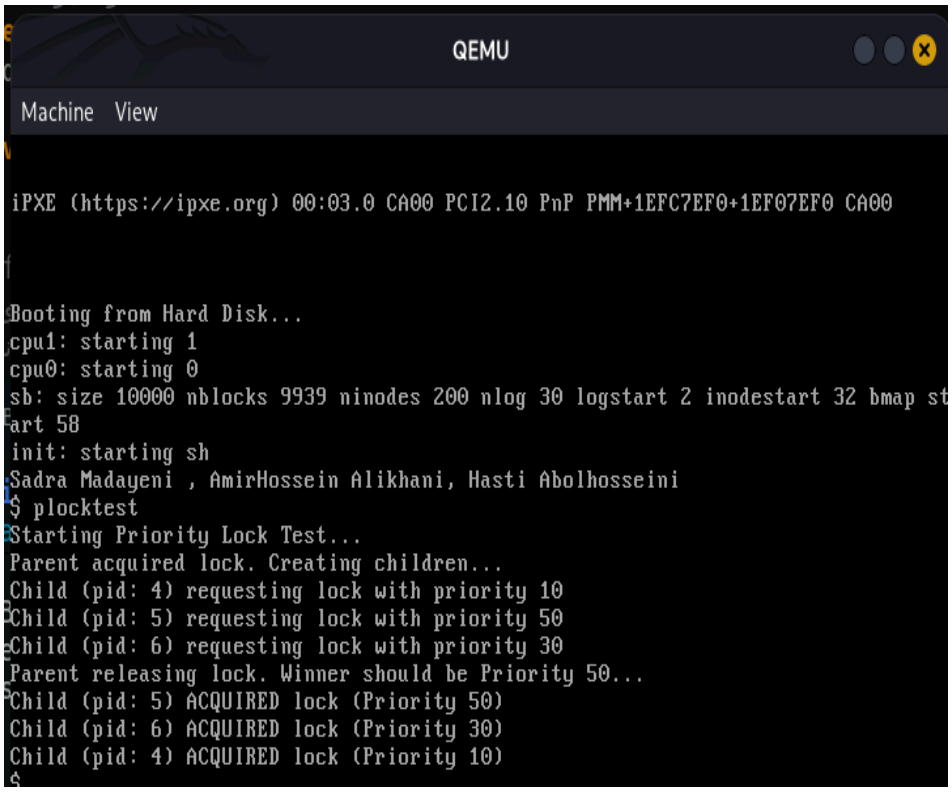
نتایج تست

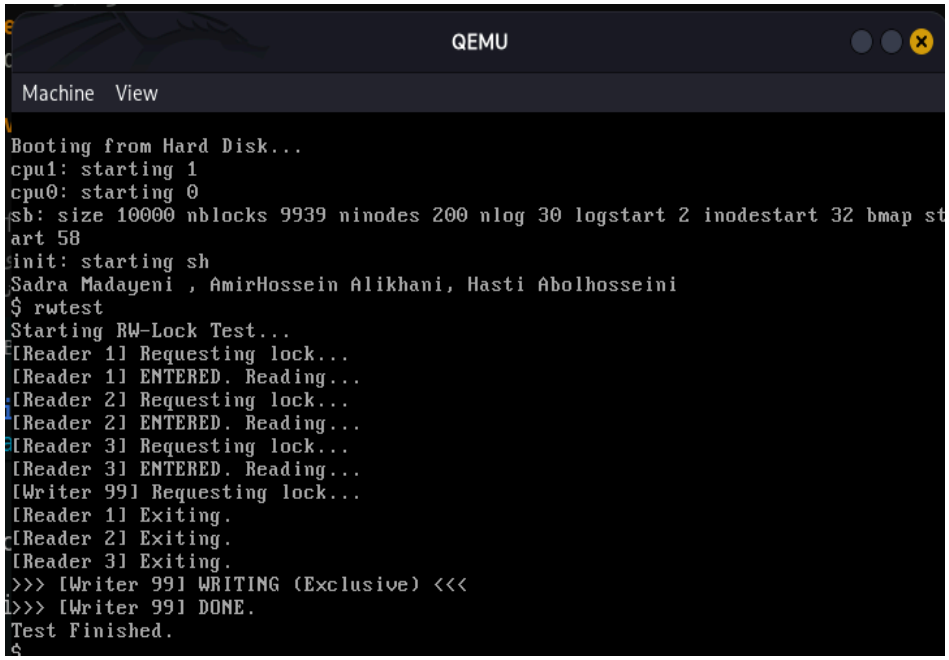

QEMU
Machine View
zombie 2 17 15304
find_sum 2 18 16420
getpidtest 2 19 15464
regcalltest 2 20 15544
makeduptest 2 21 15804
familytest 2 22 15852
grep_test 2 23 16140
priority_test 2 24 16544
test_rr 2 25 16052
test_fcfs 2 26 17160
test_info 2 27 15760
test_throughpu 2 28 16416
test_sl 2 29 16044
test_rw 2 30 17332
locktest 2 31 17304
plocktest 2 32 16424
rwtest 2 33 17024
console 3 34 0
$ test_sl
Parent acquiring lock...
Parent acquired lock. Waiting for child...
Child trying to release lock acquired by parent...
lapicid 1: panic: releasesleep: not owner


QEMU
Machine View
console 3 34 0
$ test_rw
Reader 0 waiting...
Reader 0 entered critical section.
Reader 1 waiting...
Reader 1 entered critical section.
Reader 2 waiting...
Reader 2 entered critical section.
Writer 100 waiting...
Reader 10 waiting...
Reader 10 entered critical section.
Reader 11 waiting...
Reader 11 entered critical section.
Reader 12 waiting...
Reader 12 entered critical section.
Reader 0 leaving...
Reader 1 leaving...
Reader 2 leaving...
Reader 10 leaving...
Reader 11 leaving...
Reader 12 leaving...
Writer 100 entered critical section (Exclusive).
Writer 100 leaving...



با استفاده از CPUS = 4


QEMU
Machine View
iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1EFC7EF0+1EF07EF0 CA00
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 10000 nblocks 9939 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st
art 58
init: starting sh
Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhosseini
$ plocktest
Starting Priority Lock Test...
Parent acquired lock. Creating children...
Child (pid: 4) requesting lock with priority 10
Child (pid: 5) requesting lock with priority 50
Child (pid: 6) requesting lock with priority 30
Parent releasing lock. Winner should be Priority 50...
Child (pid: 5) ACQUIRED lock (Priority 50)
Child (pid: 6) ACQUIRED lock (Priority 30)
Child (pid: 4) ACQUIRED lock (Priority 10)
$


QEMU
Machine View
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 10000 nblocks 9939 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st
art 58
init: starting sh
Sadra Madayeni , AmirHossein Alikhani, Hasti Abolhosseini
$ rwtest
Starting RW-Lock Test...
[Reader 1] Requesting lock...
[Reader 1] ENTERED. Reading...
[Reader 2] Requesting lock...
[Reader 2] ENTERED. Reading...
[Reader 3] Requesting lock...
[Reader 3] ENTERED. Reading...
[Writer 99] Requesting lock...
[Reader 1] Exiting.
[Reader 2] Exiting.
[Reader 3] Exiting.
>>> [Writer 99] WRITING (Exclusive) <<<
>>> [Writer 99] DONE.
Test Finished.
$