سوالات پروژه ۴ آزمایشگاه سیستم عامل

پرهام بدو:۸۱۰۱۰۰۲۴۴
علیرضا کامکار:۸۱۰۱۰۰۲۰۲
علی علم طلب:۸۱۰۱۰۰۱۸۹

بخش اول:تحلیل هسته و قفل های پیشرفته

۱) مفهوم مسیر کنترلی (Control Path) را تعریف کنید. تفاوت بین مسیرهای کنترلی ایجاد شده توسط «فراخوانی سیستمی» (Syscall)، «وقفه» (Interrupt) و «استثنا» (Exception) چیست؟

پاسخ:
مسیر کنترلی (Control Path) یعنی «مسیر اجرایی‌ای که CPU برای انجام یک رویداد یا درخواست طی می‌کند»، از لحظه‌ی تغییر جریان اجرای برنامه تا زمانی که دوباره به حالت عادی برگردد. به بیان ساده، وقتی اتفاقی می‌افتد (مثل درخواست سرویس از کرنل یا وقوع یک رویداد سخت‌افزاری)، CPU از مسیر معمول اجرای دستورها خارج می‌شود، وارد کدهای خاصی (معمولاً در کرنل) می‌شود و بعد از انجام کار، دوباره به همان نقطه یا نقطه‌ای مناسب بازمی‌گردد.
تفاوت اصلی این سه نوع مسیر کنترلی در «منشأ ایجاد»، «زمان وقوع»، و «نحوه‌ی انتقال کنترل» است. در **Syscall** منشأ رویداد نرم‌افزاری و «ارادی» است؛ یعنی خود برنامه‌ی کاربر عمداً با یک دستور/فراخوانی مشخص (مثل int/trap) از کرنل خدمت می‌خواهد، پس انتقال به کرنل کنترل‌شده و قابل پیش‌بینی‌تر است. در **Interrupt** منشأ سخت‌افزاری و «غیرارادی از دید برنامه» است؛ یعنی یک دستگاه (مثل تایمر یا دیسک) هر لحظه ممکن است سیگنال بدهد و CPU اجرای برنامه را موقتاً قطع کند تا هندلر وقفه اجرا شود، بنابراین وقفه ناهمگام (asynchronous) است.
در **Exception** منشأ نشان‌دهنده‌ی خطا/شرط ویژه در حین اجرای همان دستورهای جاری است (مثل page fault یا تقسیم بر صفر) و معمولاً «همگام» (synchronous) با جریان اجرای برنامه رخ می‌دهد؛ یعنی نتیجه‌ی مستقیم اجرای یک دستور است و کرنل برای رسیدگی به خطا یا فراهم‌کردن قابلیت‌هایی مثل demand paging وارد عمل می‌شود.

۲) اصطلاح Reentrant Kernel به چه معناست؟ سناریویی را در xv6 شرح دهید که یک پروسه در حال اجرای یک Syscall است، ناگهان وقفه دیسک رخ می‌دهد و CPU مجبور می‌شود مجدداً وارد کد هسته شود. در این حالت اگر همگام‌سازی نباشد، چه خطری داده‌های کرنل را تهدید می‌کند؟

پاسخ:
کرنل **Reentrant** یعنی هسته بتواند «در حالی که قبلاً در حال اجرای کد کرنل است، دوباره (به‌صورت تودرتو) وارد کرنل شود» بدون اینکه وضعیت داخلی‌اش خراب شود؛ این ورود مجدد معمولاً به دلیل وقفه‌ها یا رخدادهای همزمان اتفاق می‌افتد. سناریوی کلاسیک در xv6 این است که یک پردازه وارد یک syscall مثل `read` یا عملیات فایل می‌شود و کرنل در حال کار با ساختارهای مشترک (مثل جدول فایل‌ها، بافر کش دیسک، inodeها یا صف‌های I/O) است.
ناگهان **وقفه دیسک** رخ می‌دهد و CPU مجبور می‌شود اجرای مسیر فعلی کرنل را قطع کند و وارد هندلر وقفه دیسک شود؛ این هندلر هم ممکن است همان ساختارهای کرنلی را لمس کند (مثلاً وضعیت بافر را تغییر دهد، یک درخواست را کامل‌شده علامت بزند، یا پردازه‌ای را بیدار کند). اگر همگام‌سازی درست نباشد، خطر اصلی این است که **دو مسیر کنترلی کرنل همزمان روی داده‌های مشترک بنویسند** و کرنل دچار **Race Condition** شود؛ نتیجه می‌تواند فساد داده (Data Corruption)، ناسازگاری وضعیت (مثلاً بافر نیمه‌به‌روز شده)، از دست رفتن wakeup (گم شدن بیدار کردن)، یا حتی کرش/پانیک باشد. به همین دلیل در کرنل‌های واقعی و همین xv6، برای محافظت از داده‌های مشترک از قفل‌ها و (در برخی نقاط) غیرفعال‌سازی وقفه استفاده می‌شود تا ورود مجدد کنترل‌نشده باعث تخریب وضعیت داخلی نشود.

۳) تفاوت Process Context و Interrupt Context را شرح دهید.

پاسخ:
Process Context زمانی است که CPU در حال اجرای کد «در چارچوب یک پردازه مشخص» است؛ یعنی یک پردازه‌ی جاری (current process) داریم، حالت پردازه (رجیسترها، کرنل استک همان پردازه، ساختار `proc`) معتبر است و کرنل می‌تواند عملیات‌های مسدودکننده انجام دهد (مثل `sleep`) چون مفهوم زمان‌بندی و تغییر حالت پردازه وجود دارد. در مقابل، **Interrupt Context** زمانی است که CPU به علت یک وقفه سخت‌افزاری، اجرای فعلی را قطع کرده و وارد هندلر وقفه شده است؛ این مسیر باید سریع و محدود باشد، معمولاً نباید کار طولانی انجام دهد و مهم‌تر اینکه وابستگی‌اش به «اراده و جریان طبیعی پردازه» کمتر است. در interrupt context، شما در یک مسیر ناهمگام قرار دارید که آمده تا یک رویداد فوری را رسیدگی کند و بعد سریع برگردد؛ بنابراین قواعد همگام‌سازی و محدودیت‌ها سخت‌تر می‌شود، چون ممکن است هر لحظه وسط اجرای هر کدی رخ دهد.

۴) چرا کدی که در Interrupt Context اجرا می‌شود (مثلاً هندلر تایمر)، به هیچ وجه نباید مسدود (Block) شود یا به خواب (Sleep) برود؟

پاسخ:
چون هندلر وقفه باید **سریع** اجرا شود و **کنترل CPU را به جریان اصلی برگرداند**؛ اگر در interrupt context کد block یا sleep کند، عملاً CPU در یک مسیر تودرتو گیر می‌افتد و سیستم ممکن است دچار بن‌بست یا توقف عملی شود. از طرف دیگر، خوابیدن معمولاً به این معناست که پردازه‌ای باید کنار گذاشته شود تا بعداً بیدار شود، اما در interrupt context شما در حال اجرای «پردازه‌ی عادی» نیستید که بتوانید آن را مثل یک پردازه زمان‌بندی کنید؛ ضمن اینکه بسیاری از سیستم‌ها هنگام اجرای هندلر وقفه، وقفه‌های دیگر را هم محدود می‌کنند، پس اگر هندلر بخوابد، ممکن است هیچ‌وقت رویدادی که باید آن را بیدار کند اجرا نشود. نتیجه می‌تواند از دست رفتن وقفه‌ها، قفل شدن سیستم، یا ایجاد شرایطی باشد که زمان‌بند و دستگاه‌ها نتوانند درست کار کنند. به همین دلیل، هندلرها معمولاً فقط کارهای ضروری (ack کردن دستگاه، ثبت رویداد، بیدار کردن منتظرها، و defer کردن کار سنگین به بخش‌های مناسب‌تر) را انجام می‌دهند و وارد مسیرهای مسدودکننده نمی‌شوند.

۵) چرا برای محافظت از داده‌ها در Interrupt Context، معمولاً از ترکیب Spinlock و Disable Interrupts استفاده می‌شود؟

پاسخ:
در Interrupt Context هر لحظه ممکن است یک وقفه وسط اجرای کد فعلی رخ دهد و کنترل را به هندلر وقفه منتقل کند؛ اگر هندلر قرار باشد به همان داده‌ی مشترکی دست بزند که کدِ در حال اجرا نیز از آن استفاده می‌کند، بدون محافظت مناسب شرایط Race ایجاد می‌شود. استفاده از **spinlock** برای این است که در محیط کرنل و برای نواحی بحرانی کوتاه، دسترسی همزمان چند CPU/مسیر کنترلی به داده محدود شود. اما تنها spinlock کافی نیست، چون روی یک CPU ممکن است پردازه قفل را گرفته باشد و سپس وقفه رخ دهد و همان CPU وارد هندلری شود که بخواهد همان قفل را بگیرد؛ در این حالت هندلر شروع به spin کردن می‌کند، ولی چون قفل دست همان CPU است و آن CPU هم دیگر نمی‌تواند به مسیر قبلی برگردد تا قفل را آزاد کند، سیستم در عمل گیر می‌کند. بنابراین معمولاً **قبل از گرفتن spinlock، وقفه‌ها غیرفعال می‌شوند** تا در مدت نگه‌داشتن قفل، همان CPU دوباره به‌واسطه وقفه وارد کرنل نشود و تداخل تودرتو روی همان داده رخ ندهد. نتیجه این ترکیب، هم جلوگیری از رقابت بین CPUها (با spinlock) و هم جلوگیری از بن‌بست ناشی از ورود مجدد وقفه روی همان CPU (با disable interrupts) است.

۶) سه رویکرد کلی برای همگام‌سازی وجود دارد. هر یک را در یک پاراگراف تعریف کرده و دو مزیت/معایب برای آن ذکر کنید: Spinlock، Sleep lock، Lock-free Programming.

پاسخ (Spinlock):
Spinlock قفلی است که وقتی در اختیار دیگری باشد، نخ/CPU منتظر به‌جای خوابیدن، در یک حلقه‌ی کوتاه «چرخ می‌زند» تا قفل آزاد شود؛ این روش مناسب نواحی بحرانی کوتاه در کرنل است که انتظار می‌رود زمان نگه‌داشتن قفل کم باشد. مزیت اصلی آن **سریع بودن** در شرایط انتظار کوتاه و **سادگی پیاده‌سازی** در سطح پایین کرنل است (به‌خصوص در بخش‌هایی که خوابیدن مجاز نیست). در مقابل، دو عیب مهم دارد: اول اینکه باعث **مصرف بیهوده CPU** در انتظارهای طولانی می‌شود (busy-waiting) و دوم اینکه در رقابت زیاد می‌تواند **گلوگاه کارایی** و افزایش ترافیک کش/باس ایجاد کند، چون CPUها مرتب روی یک متغیر قفل می‌کوبند.

(Sleep lock):
Sleep lock قفلی است که اگر در اختیار دیگری باشد، پردازه منتظر «می‌خوابد» و CPU را آزاد می‌کند تا پردازه‌های دیگر اجرا شوند؛ بنابراین برای انتظارهای طولانی مثل I/O بسیار مناسب‌تر از spinlock است. مزیت مهم آن این است که **CPU هدر نمی‌رود** و سیستم می‌تواند بهره‌وری بالاتری داشته باشد، و همچنین در بارهای سنگین معمولاً **رفتار پایدارتر** از busy-wait نشان می‌دهد. اما از طرف دیگر، دو عیب کلیدی دارد: اول اینکه گرفتن/آزاد کردن آن معمولاً **سربار بیشتری** دارد چون به scheduler و sleep/wakeup وابسته است، و دوم اینکه در برخی contextها (مثل interrupt context) **اصلاً قابل استفاده نیست** چون خوابیدن در هندلر وقفه ممنوع است.

(Lock-free Programming):
Lock-free programming به مجموعه تکنیک‌هایی گفته می‌شود که بدون قفل‌های سنتی، با استفاده از عملیات اتمیک (مثل CAS) و طراحی الگوریتمی، پیشروی سیستم را تضمین می‌کنند؛ در این رویکرد تلاش می‌شود با حذف قفل، هم مشکل بن‌بست و هم گلوگاه‌های رقابت کاهش یابد. مزیت آن معمولاً **مقیاس‌پذیری بهتر** در سیستم‌های چند هسته‌ای و **کاهش ریسک deadlock** است، چون قفل مرکزی وجود ندارد. در عوض دو عیب اصلی دارد: اول اینکه طراحی و اثبات درستی این الگوریتم‌ها **بسیار پیچیده** است و خطاهای ظریف همزمانی زیاد رخ می‌دهد، و دوم اینکه ممکن است در عمل با پدیده‌هایی مثل **livelock/بازآزمایی‌های زیاد** (retry) یا مشکلات حافظه‌ای (مثل ABA) روبه‌رو شود که دیباگ و نگهداری را سخت می‌کند.

۷) در تابع acquire، قبل از گرفتن قفل، وقفه‌ها غیرفعال می‌شوند (cli)، چرا؟ فرض کنید وقفه‌ها باز باشند و یک پردازنده قفلی را بگیرد، سپس یک وقفه رخ دهد و هندلر وقفه بخواهد همان قفل را بگیرد. چه اتفاقی می‌افتد؟ (شرح سناریوی Deadlock در سیستم تک‌هسته‌ای).

پاسخ:
در سیستم تک‌هسته‌ای اگر وقفه‌ها فعال باشند، ممکن است CPU یک **spinlock** را در مسیر عادی (process context) بگیرد و هنوز آن را آزاد نکرده باشد که ناگهان یک وقفه (مثلاً تایمر یا دیسک) رخ دهد. در این لحظه CPU مجبور است اجرای مسیر فعلی را قطع کند و وارد هندلر وقفه شود. حال اگر هندلر برای ادامه کار نیاز داشته باشد همان قفل را بگیرد، تلاش می‌کند `acquire` را اجرا کند و چون قفل قبلاً گرفته شده، وارد حلقه spin می‌شود. مشکل اینجاست که قفل دست **همان CPU** است، اما CPU دیگر نمی‌تواند به مسیر قبلی برگردد تا قفل را آزاد کند، چون در هندلر گیر کرده و هندلر هم تا گرفتن قفل جلو نمی‌رود؛ بنابراین سیستم وارد یک **بن‌بست کلاسیک** می‌شود: هندلر منتظر آزاد شدن قفل است و آزاد شدن قفل هم تنها با برگشتن از هندلر ممکن است. به همین دلیل در xv6 قبل از گرفتن

spinlock، وقفه‌ها با `cli` (یا در عمل pushcli/popcli) غیرفعال می‌شوند تا در مدت نگه‌داشتن قفل، همان CPU نتواند با یک وقفه وارد مسیر دیگری شود که دوباره همان قفل را درخواست کند و deadlock بسازد.

۸) چرا xv6 به جای استفاده مستقیم از `cli` و `sti`، از توابع `pushcli` و `popcli` استفاده می‌کند؟ این توابع چگونه مشکل «تودرتو بودن نواحی بحرانی» (Nested Critical Sections) را حل می‌کنند؟

پاسخ:
در xv6 خاموش و روشن‌کردن وقفه‌ها اگر با `cli` و `sti` به‌صورت مستقیم و "خام" انجام شود، در نواحی بحرانی تودرتو (Nested) خیلی راحت خطا ایجاد می‌کند؛ چون ممکن است یک تابع برای ورود به بخش بحرانی وقفه‌ها را با `cli` ببندد، بعد داخل همان مسیر تابع دیگری هم (به دلایل خودش) دوباره `cli` بزند، و وقتی از تابع داخلی برمی‌گردد با یک `sti` وقفه‌ها را روشن کند؛ در حالی که هنوز در بخش بحرانی تابع بیرونی هستیم و وقفه‌ها نباید روشن شوند. این دقیقاً همان مشکل Nested Critical Sections است: **بازگردانی اشتباه وضعیت وقفه‌ها** باعث می‌شود وسط ناحیه بحرانی وقفه فعال شود و احتمال race یا deadlock بالا برود.

برای حل این مشکل، xv6 از `pushcli` و `popcli` استفاده می‌کند که دو کار مهم انجام می‌دهند: اول اینکه وضعیت قبلی وقفه‌ها را «به‌صورت امن و قابل بازگشت» ذخیره می‌کنند (اینکه قبل از ورود، وقفه‌ها روشن بوده یا خاموش)، و دوم اینکه یک شمارنده تودرتویی نگه می‌دارند (معمولاً در سطح CPU مثل `ncli`) تا مشخص شود چند بار پشت سر هم وارد ناحیه بحرانی شده‌ایم. `pushcli` هر بار که فراخوانی می‌شود، وقفه‌ها را خاموش می‌کند و شمارنده را افزایش می‌دهد؛ اما `popcli` فقط وقتی واقعاً وقفه‌ها را روشن می‌کند که شمارنده به صفر برگردد، یعنی همه لایه‌های تودرتوی بخش بحرانی تمام شده باشد. علاوه بر این، اگر قبل از اولین `pushcli` وقفه‌ها خاموش بوده باشند، هنگام خروج هم همان وضعیت حفظ می‌شود و به اشتباه وقفه‌ها روشن نمی‌گردند. در نتیجه، این سازوکار تضمین می‌کند که **وقفه‌ها دقیقاً به وضعیت اولیه برگردند** و تودرتویی نواحی بحرانی باعث روشن شدن زودهنگام وقفه‌ها نشود.

بخش دوم:مقیاس پذیری ، داده های Per-cpu و پروفایلینگ قفل ها در xv۶

۱) چرا زمانی که یک پردازنده یک spinlock را در اختیار دارد، حتماً باید وقفه‌ها روی آن پردازنده غیرفعال باشند؟

پاسخ:
وقتی یک پردازنده یک spinlock را در اختیار دارد، عملاً وارد یک ناحیه بحرانی کوتاه شده است که انتظار می‌رود بدون وقفه و به‌صورت پیوسته اجرا شود. اگر در این حالت وقفه‌ها فعال بمانند، ممکن است همان پردازنده به‌طور ناگهانی اجرای مسیر فعلی را رها کند و وارد هندلر وقفه شود. حال اگر هندلر وقفه برای انجام وظیفه خود نیاز داشته باشد همان spinlock را بگیرد، پردازنده وارد حلقه انتظار (spin) می‌شود، در حالی که قفل در اختیار خودِ همان پردازنده است و هیچ مسیر دیگری وجود ندارد که آن را آزاد کند. بنابراین غیرفعال‌سازی وقفه‌ها هنگام در اختیار داشتن spinlock

تضمین می‌کند که پردازنده تا آزاد کردن قفل، وارد مسیر کنترلی دیگری نشود و از بن‌بست و ناسازگاری جلوگیری گردد.

۲) اگر وقفه فعال بماند و هندلر وقفه (Interrupt Handler) بخواهد همان قفل را بگیرد، چه نوع بن‌بستی (Deadlock) رخ می‌دهد؟ (با مثال توضیح دهید)

پاسخ:
در این حالت یک **بن‌بست تک‌پردازنده‌ای ناشی از ورود مجدد وقفه** رخ می‌دهد. سناریو به این صورت است که CPU در حال اجرای کدی در کرنل است و یک spinlock را گرفته است. قبل از آزاد شدن قفل، وقفه‌ای (مثلاً تایمر یا دیسک) رخ می‌دهد و CPU وارد هندلر وقفه می‌شود. اگر هندلر نیز برای ادامه کار خود نیاز به همان قفل داشته باشد، تلاش می‌کند آن را بگیرد و چون قفل آزاد نیست، وارد حالت spin می‌شود. مشکل اینجاست که تنها پردازنده سیستم همان CPU است و قفل هم در اختیار همین CPU قرار دارد، اما CPU دیگر نمی‌تواند به مسیر قبلی برگردد تا قفل را آزاد کند، چون در هندلر گیر افتاده است. در نتیجه، سیستم در یک بن‌بست کامل قرار می‌گیرد که نه هندلر جلو می‌رود و نه مسیر اصلی آزاد می‌شود.

۳) اگر چندین هسته پردازنده به طور مداوم یک متغیر مشترک سراسری (مثلاً GlobalCounter) را تغییر دهند، چه پدیده‌ای رخ می‌دهد که باعث کندی سیستم می‌شود؟ پروتکل‌های همگام‌سازی حافظه نهان (مانند MESI) در این شرایط چه می‌کنند؟

پاسخ:
در این وضعیت پدیده‌ای به نام **رقابت روی حافظه نهان و کش‌لاین (Cache Line Contention)** رخ می‌دهد. وقتی چندین هسته یک متغیر سراسری مشترک را به‌طور مداوم تغییر می‌دهند، هر بار که یکی از هسته‌ها مقدار را می‌نویسد، کش‌لاین مربوط به آن متغیر باید در سایر هسته‌ها نامعتبر (invalidate) شود. پروتکل‌هایی مانند MESI دقیقاً مسئول مدیریت این وضعیت هستند و با ارسال پیام‌های invalidation بین هسته‌ها، تضمین می‌کنند که تنها یک نسخه معتبر از داده برای نوشتن وجود داشته باشد. اما این invalidationهای پی‌درپی باعث رفت‌وآمد زیاد پیام‌ها روی باس یا شبکه داخلی پردازنده می‌شود و در نهایت هزینه همگام‌سازی حافظه نهان آن‌قدر بالا می‌رود که کارایی کل سیستم به‌طور محسوسی کاهش پیدا می‌کند.

۴) چرا استفاده از متغیرهای محلی (Per-CPU) این مشکل را تا حد زیادی کاهش می‌دهد؟ به‌طور خلاصه توضیح دهید چگونه نگه‌داشتن شمارنده‌ها به‌صورت per-CPU می‌تواند تعداد invalidationهای کش را کم کند.

پاسخ:
در مدل Per-CPU، هر هسته نسخه محلی و اختصاصی خودش از متغیر (مثلاً شمارنده) را دارد و مستقیماً فقط همان نسخه را تغییر می‌دهد. چون این متغیر دیگر بین چند هسته مشترک نیست، هر نوشتن فقط روی کش‌لاین محلی همان CPU انجام می‌شود و نیازی به invalid کردن کش‌لاین سایر هسته‌ها وجود ندارد. در نتیجه، تعداد پیام‌های invalidation به‌شدت کاهش می‌یابد و ترافیک همگام‌سازی حافظه نهان کم می‌شود. تنها در زمان‌هایی خاص (مثلاً جمع‌زدن نهایی مقادیر) نیاز به دسترسی مشترک وجود دارد که آن هم به‌صورت محدود انجام می‌شود. به همین دلیل، استفاده از داده‌های per-CPU یکی از راهکارهای کلیدی برای افزایش مقیاس‌پذیری در سیستم‌های چندهسته‌ای است.

۵) تفاوت اصلی بین spinlock (دسته اول توابع در xv6) و sleeplock (دسته دوم توابع) در چیست؟ کدام یک باعث «انتظار مشغول» (Busy Waiting) می‌شود و کدام یک پردازنده را به پروسس دیگری واگذار می‌کند؟

پاسخ:
تفاوت اصلی spinlock و sleeplock در **نحوه انتظار پردازه هنگام در دسترس نبودن قفل** است. در spinlock، پردازه یا CPU منتظر به‌طور فعال در یک حلقه تکرار می‌چرخد و مرتب بررسی می‌کند که آیا قفل آزاد شده است یا نه؛ به همین دلیل spinlock باعث **انتظار مشغول (Busy Waiting)** می‌شود و CPU را درگیر نگه می‌دارد. در مقابل، sleeplock زمانی که قفل در اختیار دیگری است، پردازه منتظر را به حالت خواب می‌برد و CPU را آزاد می‌کند تا پردازه‌های دیگر اجرا شوند؛ بنابراین sleeplock باعث **واگذاری پردازنده به پردازه‌ای دیگر** می‌شود. به‌طور خلاصه، spinlock برای نواحی بحرانی کوتاه و contextهایی مثل interrupt مناسب است، اما sleeplock برای انتظارهای طولانی مانند عملیات I/O انتخاب منطقی‌تری محسوب می‌شود.

بخش سوم: قفل‌های اولویت‌دار و مسئله گرسنگی

سؤالات تئوری

۱) آیا در طراحی plock (قفلی که در ادامه پیاده‌سازی می‌کنید)، امکان گرسنگی برای پردازه‌های با اولویت پایین وجود دارد؟ یک سناریوی دقیق (Scenario) بنویسید که در آن یک پردازه با اولویت پایین (Low Priority) درخواست قفل می‌کند، اما با وجود اینکه قفل بارها توسط دیگران گرفته و آزاد می‌شود، این پردازه هرگز موفق به دریافت قفل نمی‌شود.

پاسخ:
بله، در طراحی قفل اولویت‌دار (plock) امکان **گرسنگی (Starvation)** برای پردازه‌های با اولویت پایین وجود دارد. سناریو به این صورت است که فرض کنید یک پردازه با اولویت پایین درخواست گرفتن قفل می‌دهد و وارد صف انتظار می‌شود. در همین زمان، چند پردازه با اولویت بالاتر نیز به‌صورت متناوب وارد سیستم می‌شوند و هر کدام درخواست قفل می‌کنند. چون سیاست plock این است که در زمان آزاد شدن قفل، همیشه پردازه‌ای با **بالاترین اولویت** انتخاب شود، هر بار که قفل آزاد می‌شود، یکی از پردازه‌های با اولویت بالاتر قفل را دریافت می‌کند. اگر این وضعیت ادامه پیدا کند و همواره پردازه‌های جدید با اولویت بالاتر وارد صف شوند، پردازه با اولویت پایین—even با وجود آزاد شدن مکرر قفل—هرگز انتخاب نمی‌شود. این دقیقاً نمونه‌ای از گرسنگی است که نتیجه مستقیم سیاست اولویت‌محور و ترجیح دائمی پردازه‌های مهم‌تر نسبت به پردازه‌های کم‌اهمیت‌تر محسوب می‌شود.

۲) یک مکانیزم دیگر برای مدیریت صف، استفاده از قفل بلیطی (Ticket Lock) است که شبیه سیستم نوبت‌دهی نانوایی عمل می‌کند (FIFO). نحوه کارکرد Ticket Lock را به صورت مفهومی توضیح دهید.

پاسخ:
قفل بلیطی (Ticket Lock) بر پایه یک ایده ساده و عادلانه کار می‌کند: هر پردازه‌ای که قصد گرفتن قفل را دارد، یک «شماره بلیط» دریافت می‌کند و سپس منتظر می‌ماند تا نوبتش برسد. معمولاً دو شمارنده وجود دارد: یکی برای صدور بلیط (ticket) و دیگری برای بلیطی که در حال سرویس‌دهی است (turn). هر پردازه هنگام درخواست قفل، مقدار ticket را افزایش می‌دهد و شماره قبلی را به‌عنوان نوبت خود نگه می‌دارد. سپس در یک حلقه منتظر می‌ماند تا مقدار turn برابر با شماره بلیط خودش شود. وقتی پردازه‌ای قفل را آزاد می‌کند، turn را یک واحد افزایش می‌دهد و به پردازه بعدی اجازه ورود می‌دهد. به این ترتیب، قفل دقیقاً به ترتیب ورود (FIFO) واگذار می‌شود و هیچ پردازه‌ای نمی‌تواند از نوبت دیگران جلو بزند؛ درست شبیه صف نانوایی که هر نفر با شماره بلیط، منتظر نوبت خودش می‌ماند.

۳) قفل اولویت‌دار (plock) را با قفل بلیطی (ticket lock) از نظر انصاف (Fairness)، پیچیدگی پیاده‌سازی (Complexity) و احتمال گرسنگی مقایسه کنید. کدام‌یک عادلانه‌تر رفتار می‌کند؟ کدام‌یک سربار محاسباتی کمتری دارد؟ کدام‌یک در برابر گرسنگی ایمن است؟

پاسخ:
از نظر **انصاف (Fairness)**، قفل بلیطی عادلانه‌تر از plock رفتار می‌کند، زیرا ترتیب دریافت قفل کاملاً FIFO است و همه پردازه‌ها دقیقاً به ترتیب ورود سرویس می‌گیرند؛ در حالی که plock به‌طور ذاتی عادلانه نیست، چون همواره پردازه‌های با اولویت بالاتر را ترجیح می‌دهد و ممکن است پردازه‌های کم‌اولویت را نادیده بگیرد. از نظر **پیچیدگی پیاده‌سازی و سربار محاسباتی**، ticket lock معمولاً ساده‌تر و کم‌هزینه‌تر است، چون فقط با دو شمارنده کار می‌کند و نیازی به مدیریت صف پیچیده یا جست‌وجوی پردازه با بیشترین اولویت ندارد؛ در مقابل، plock نیازمند نگهداری صف، ذخیره اولویت‌ها و پیمایش برای یافتن بیشترین اولویت است که سربار بیشتری ایجاد می‌کند. از نظر **احتمال گرسنگی**، ticket lock عملاً در برابر گرسنگی ایمن است، چون هر پردازه نهایتاً نوبت خود را دریافت می‌کند، اما plock مستعد گرسنگی است و در صورت ورود مداوم پردازه‌های با اولویت بالا، پردازه‌های با اولویت پایین ممکن است هرگز به قفل دسترسی پیدا نکنند.