سیستم‌های عامل

نیم‌سال اول ۰۳-۰۲

دانشکده‌ی مهندسی کامپیوتر

شماره دانشجویی: ۹۹۱۰۵۵۶۱

شایان صالحی

تمرین دوم

پاسخ مسئله‌ی ۱.

آ

نادرست، Context switch ممکن است به دلایل مختلفی از جمله تغییر در اولویت‌های برنامه‌ریزی وقتی که یک پردازه (process) در انتظار منابع است، یا برنامه‌ریزی زمانی برای اطمینان از توزیع منصفانه زمان پردازنده بین پردازه‌ها، رخ دهد. این فرآیند همیشه به صورت ناخواسته و در زمان‌های نامشخص رخ نمی‌دهد.

ب

نادرست، در حالت عمومی، هر ریسه (thread) دارای استک (stack) اختصاصی خود است و به طور مستقیم نمی‌تواند به استک سایر ریسه‌ها دسترسی داشته باشد. دسترسی به داده‌های استک یک ریسه توسط ریسه‌ای دیگر نیازمند مکانیزم‌های خاصی مانند اشتراک‌گذاری حافظه است.

ج

نادرست، تابع execv در لینوکس برای جایگزینی فعلی پردازه با یک پردازه جدید استفاده می‌شود، نه ایجاد یک پردازه جدید. این تابع، کد موجود در یک پردازه را با یک برنامه جدید جایگزین می‌کند. برای ایجاد یک پردازه جدید، معمولاً از تابع fork استفاده می‌شود، که یک کپی از پردازه فعلی را ایجاد می‌کند.

د

درست، در سیستم‌های عاملی مانند UNIX و لینوکس، یک پردازه می‌تواند چندین file descriptor داشته باشد که به یک file description واحد اشاره می‌کنند. این امر به ویژه در مواردی مانند استفاده از توابعی چون fork یا دوباره باز کردن یک فایل مشاهده می‌شود.

ه

نادرست، اگر تردها فقط برای خواندن داده‌ها از آرایه استفاده کنند. در مواردی که چندین ریسه (thread) فقط برای خواندن داده‌ها از یک منبع مشترک استفاده می‌کنند و هیچ نوشتنی در آن صورت نمی‌گیرد، race condition به وجود نمی‌آید و نیازی به استفاده از مکانیزم‌های سینکرونایزیشن نیست. Race condition زمانی رخ می‌دهد که چندین ریسه به طور همزمان داده‌ها را می‌خوانند و می‌نویسند.

و

درست، در سیستم‌های عامل مدرن، فضای آدرس هر پردازه از دیگر پردازه‌ها مجزا و محافظت شده است. یک پردازه نمی‌تواند مستقیماً فضای حافظه یا آدرس‌دهی یک پردازه دیگر را تغییر دهد یا به آن دسترسی پیدا کند.

ز

نادرست، زمان انتظار در الگوریتم Round Robin نسبت به FCFS بستگی به عوامل متعددی دارد، از جمله طول دوره زمانی (time slice) و خصوصیات ویژه بار کاری. در برخی موارد، زمان انتظار در Round Robin می‌تواند کمتر از FCFS باشد، به خصوص اگر دوره زمانی به خوبی انتخاب شود و کارها زمان پردازش کوتاهی داشته باشند. اما در سایر موارد، ممکن است زمان انتظار در Round Robin بیشتر از FCFS باشد.

ح

نادرست، FCFS (First-Come, First-Served) یک الگوریتم غیر preemptive است، یعنی وقتی یک پردازه شروع به اجرا می‌کند، تا پایان اجرای آن ادامه می‌یابد و نیازی به preemption ندارد. اما Round Robin یک الگوریتم preemptive است. در Round Robin، هر پردازه برای مدت زمان مشخصی (time slice) اجرا می‌شود و پس از آن، اگر همچنان کاری برای انجام دادن داشته باشد، به انتهای صف انتظار منتقل می‌شود و پردازه بعدی شروع به اجرا می‌کند. این فرآیند نیازمند preemption است.

ط

نادرست، تأثیر الگوریتم‌های برنامه‌ریزی بر کارایی حافظه نهان به شدت به نوع برنامه‌ها و مدل دسترسی‌های حافظه آن‌ها بستگی دارد. در مواردی که برنامه‌ها به طور متناوب و با فاصله زمانی کوتاه اجرا شوند، ممکن است Round Robin باعث بهبود دسترسی به داده‌های حافظه نهان شود، زیرا این الگوریتم از ایجاد وقفه‌های طولانی در اجرای هر پردازه جلوگیری می‌کند. اما در سایر موارد، به خصوص زمانی که پردازه‌ها دارای دسترسی‌های حافظه متمرکز و طولانی هستند، FCFS ممکن است باعث کارایی بیشتر حافظه نهان شود.

ی

نادرست، وقتی یک ریسه درون یک پردازه file descriptor را می‌بندد، تنها برای آن پردازه بسته می‌شود، نه برای کل سیستم یا سایر پردازه‌ها. File descriptor ها در سطح پردازه مدیریت می‌شوند، نه در سطح سیستم. بنابراین، بسته شدن یک file descriptor توسط یک ریسه تنها برای همان پردازه‌ای که ریسه به آن تعلق دارد، تأثیر دارد.

ک

درست، در پردازه‌های چند ریسه‌ای، هر ریسه دارای پشته (stack) مخصوص به خود است. متغیرهایی که در پشته یک ریسه ذخیره می‌شوند، تنها توسط همان ریسه قابل دسترسی هستند و از دیگر ریسه‌ها مجزا هستند. بنابراین، دسترسی به این متغیرها thread-safe است، زیرا هیچ تداخلی بین ریسه‌ها در دسترسی به این متغیرها وجود ندارد.

ل

نادرست، اگرچه این الگوریتم می‌تواند به لحاظ نظری کارایی بالایی در کاهش زمان انتظار داشته باشد، اما بهینه‌ترین الگوریتم برای همه سناریوها و محیط‌های سیستم عامل نیست. عملکرد و کارایی یک الگوریتم زمان‌بندی به شدت به ماهیت بار کاری و خصوصیات سیستم وابسته است. همچنین، SRTF می‌تواند منجر به مشکلاتی مانند starvation برای پردازه‌های با زمان پردازش بلند شود.

پاسخ مسئله‌ی ۲.

آ

اتفاقاتی که باید ذخیره شوند:

مقادیر رجیسترها: شامل رجیسترهای محلی ریسه، مانند مقادیر پایینتر و بالاتر stack pointer، program counter و سایر رجیسترهای مربوط به وضعیت فعلی ریسه.

مقادیر مربوط به اجرای ریسه: این شامل مواردی مانند اولویت ریسه و وضعیت آماده به اجرا یا در انتظار است.

از آنجایی که هر دو ریسه به یک پردازه تعلق دارند، داده‌های مربوط به فضای کاربر پردازه (مانند فضای آدرس حافظه) مشترک هستند و نیازی به ذخیره و بازیابی مجدد آن‌ها در یک context switch داخلی نیست.

ب

مقادیر رجیسترها: مشابه حالت الف، مقادیر رجیسترهای هر ریسه باید ذخیره و بازیابی شوند.

فضای آدرس حافظه: از آنجایی که ریسه‌ها به پردازه‌های مختلف تعلق دارند، فضای آدرس حافظه هر پردازه (شامل بخش‌هایی مانند داده‌ها، کد و پشته) باید ذخیره و هنگام بازگشت به آن پردازه بازیابی شود.

اطلاعات مربوط به حافظه نهان و سایر منابع سیستم: این شامل اطلاعاتی است که برای حفظ وضعیت پردازه و بهینه‌سازی دسترسی‌های حافظه لازم است.

پاسخ مسئله‌ی ۳.

آ

در اینجا نرخ سرویس دهی برابر است با تعداد درخواست‌هایی که سرور می‌تواند در واحد زمانی سرویس دهد. با توجه به اینکه زمان سرویس دهی هر درخواست ۲۰ میلی ثانیه است، سرور می‌تواند در هر ثانیه ۵۰ درخواست را پردازش کند ($\frac{۱}{۰/۰۲} = ۵۰$).

با استفاده از این دو نرخ، می‌توانیم میزان استفاده از سرور (ρ) را محاسبه کنیم، که برابر است با نسبت نرخ رسیدن به نرخ سرویس دهی ($\rho = \frac{\lambda}{\mu}$). در این حالت، ρ برابر است با $\frac{۶۰}{۵۰} = ۱/۲$. از آنجایی که ρ بیشتر از ۱ است، این نشان دهنده این است که سرور قادر به پردازش تمام درخواست‌های ورودی در زمان واقعی نیست و صف درخواست‌ها به مرور زمان افزایش می‌یابد. در این حالت، تاخیر صف به بینهایت میل می‌کند.

ب

در اینجا فاصله بین درخواست‌ها ۳۰ میلی ثانیه بوده که یعنی در هر ثانیه $\frac{۱۰۰۰}{۳۰} = ۳۳$ درخواست داریم.

در این حالت، نسبت استفاده از سرور (ρ) برابر است با $\frac{۳۳}{۵۰} = ۰/۶۶$. این نشان می‌دهد که سرور قادر به پردازش تمام درخواست‌های ورودی است و صف در نهایت به حالت پایدار می‌رسد. تاخیر صف در حالت پایدار به عواملی مانند تعداد اولیه درخواست‌ها در صف و نرخ سرویس دهی بستگی دارد، اما با توجه به اینکه ρ کمتر از ۱ است، صف به طور مداوم از بین نخواهد رفت و در نهایت به یک تعادل خواهد رسید.

ج

میزان بهره‌وری سرور در زمان طولانی به نسبت استفاده از سرور (ρ) بستگی دارد. در این مثال، از آنجایی که ρ برابر با ۰/۶۶ است، می‌توان گفت که سرور در حدود ٪۶۶/۶ زمان خود را صرف پردازش درخواست‌ها می‌کند و ٪۳۳/۳ زمان در حالت آماده‌به‌کار بدون پردازش درخواست است. این نشان می‌دهد که سرور به طور موثری استفاده می‌شود و فضای کافی برای پردازش درخواست‌های اضافی وجود دارد.

پاسخ مسئله‌ی ۴.

زمانبندی Round-Robin بدون در نظر گرفتن اولویت‌ها:

در این حالت، تمام پردازه‌ها بدون توجه به اولویت‌شان و بر اساس زمان ورود و کوانتوم زمانی ۶ میلی‌ثانیه‌ای، اجرا می‌شوند. این شامل تغییرات زیر است:

- هر پردازه به مدت ۶ میلی‌ثانیه اجرا شده و سپس جای خود را به پردازه بعدی می‌دهد.
- در صورتی که پردازه‌ای io-bound باشد مانند D، پس از ۱ میلی‌ثانیه به صف io منتقل می‌شود و پس از ۲ میلی‌ثانیه به صف ready بازمی‌گردد.
- تغییرات و وضعیت هر پردازه در هر زمان در جدول ثبت می‌شود.

زمانبندی Round-Robin با در نظر گرفتن Preemption و اولویت وظایف در لحظه ورود:

در این حالت، وظایف بر اساس اولویت‌شان و با در نظر گرفتن preemption اجرا می‌شوند:

- هر وظیفه بر اساس اولویت و زمان ورود خود در صف ready قرار می‌گیرد.
- پردازه با اولویت بالاتر در هر لحظه جای خود را به پردازه با اولویت پایین‌تر می‌دهد.
- مانند حالت قبل، پردازه‌های io-bound به صف io منتقل می‌شوند و پس از آن به صف ready بازمی‌گردند.

زمانبندی FCFS با در نظر گرفتن اولویت و بدون Preemption:

در این حالت، وظایف بر اساس زمان ورود و اولویت اجرا می‌شوند، اما بدون preemption:

- وظایف به ترتیب زمان ورود و اولویت در صف ready قرار می‌گیرند.
- هر پردازه به طور کامل اجرا می‌شود قبل از اینکه پردازه بعدی شروع به کار کند.
- پردازه‌های io-bound به همان شکل قبل به صف io و سپس به صف ready منتقل می‌شوند.

برای محاسبه turnaround time هر وظیفه، زمان پایان هر وظیفه منهای زمان ورود آن محاسبه می‌شود. سپس، میانگین این زمان‌ها برای کل وظایف به دست می‌آید. این محاسبات نیاز به توجه دقیق به جزئیات و دنبال کردن هر پردازه در طول زمان دارد.

پاسخ مسئله‌ی ۵.

در اینجا ابتدا در ترمینال Starting main چاپ شده و پس از آن به خاطر وجود خط

```
dup2(file_fd, STDOUT_FILENO);
```

مابقی خروجی‌ها در این فایل نوشته خواهند شد.

در خط ۵ متود `fork()` صدا زده شده و در پروسه والد در شرط `if` رفته و در حالت `wait` می‌مانیم.

اما در پردازه فرزند درون `else` رفته و مقدار درون آن در فایل ذخیره می‌گردد و همچنین به دلیل آنکه `child_pid` در پردازه فرزند صفر است خروجی صفر خواهیم داشت.

پس از اتمام کار پردازه فرزند، وارد پردازه والد شده و مقدار ۶۶۶۶ به عنوان `child_pid` شناخته و به این ترتیب در فایل خروجی به چنین حالتی برخواهیم خورد.

```
In child
Ending main: 0
In parent
Ending main: 6666
```

پاسخ مسئله‌ی ۶.

فقط `printf` سوم در اینجا مقدار متفاوتی را چاپ خواهد کرد. دلیل آن این است که در اینجا آدرس متغیرهای پاس داده شده به تابع در حال چاپ شدن است که هر دو در استک ریسه‌ها ذخیره می‌شوند. چونکه هر ریسه یک استک جدا دارد به این ترتیب مقادیر متفاوتی به خاطر مکان حافظه آن چاپ خواهد شد.

در میان خطوطی که چاپ شده‌اند، خطوط اول، دوم، و چهارم برای تمامی ریسه‌ها یکنواخت است. دلیل یکسان بودن خط اول این است که یک متغیر گلوبال است که مقدار آن تغییر نمی‌کند، بنابراین برای همه ریسه‌ها به یک شکل چاپ می‌گردد. خط دوم، که آدرس تابع `foo` در کد است، نیز برای همه یکسان است. خط آخر، که نشان‌دهنده مقدار ورودی است، هم برابر است، زیرا ما یک متغیر مشترک را به تمام ریسه‌ها داده‌ایم.

پاسخ مسئله‌ی ۷.

در این کد، ابتدا عدد ۱ چاپ می‌شود، سپس ۵ ترد ایجاد می‌کنیم که قرار است به ترتیب اعداد ۲ تا ۶ را چاپ کنند. پس از آن، منتظر تکمیل اجرای ۴ ترد اول می‌مانیم و در نهایت عدد ۷ چاپ می‌شود. برای محاسبه کل حالات ممکن، به تحلیل سناریوها می‌پردازیم.

اگر عدد ۶ توسط ترد آخر به موقع چاپ نشود و قبل از پایان برنامه ظاهر نگردد، فقط اعداد چاپ شده توسط ۴ ترد اول به هر ترتیبی ممکن است نمایان شوند و در نهایت عدد ۷ چاپ می‌شود. بنابراین، در این حالت $!۴$ حالت وجود دارد.

در صورتی که عدد ۶ نیز در خروجی چاپ شده باشد، ۴ عدد اول $!۴$ حالت مختلف دارند. همچنین، خود عدد ۶ می‌تواند قبل از اولین عدد چاپ شده در تردهای دیگر یا حتی بعد از عدد ۷ چاپ شود، پس خود آن ۶ حالت مختلف دارد. در نتیجه، در کل $۶ \times !۴$ حالت وجود دارد.

به این ترتیب برای تمامی رشته‌های متمایز خواهیم داشت:

$$۶ \times !۴ + !۴ = ۷ \times !۴$$

پاسخ مسئله‌ی ۸.

آ

این کد پس از ایجاد یک فرزند، هم در پردازه والد و هم در پردازه فرزند، فایل مورد نظر را باز کرده و در آن محتوایی را می‌نویسد. نکته قابل توجه این است که این دو پردازه ممکن است با سرعت‌های متفاوتی عمل کنند. اگر پردازه والد زودتر نوشته و سپس فرزند نوشتار خود را انجام دهد، نتیجه در فایل "a" خواهد بود. در صورتی که فرزند زودتر نویسندگی کند، نتیجه "b" خواهد بود، زیرا پردازه‌ای که دیرتر اقدام به نوشتن می‌کند، محتوای نوشته شده توسط پردازه قبلی را بازنویسی می‌کند. این اتفاق به دلیل اینکه فایل‌ها به طور جداگانه باز شده‌اند رخ می‌دهد و نوشتار یک پردازه تأثیری بر مکان‌نما (pointer) در توصیف‌گر فایل (file description) پردازه دیگر ندارد. بنابراین، پاسخ نهایی می‌تواند "a" یا "b" باشد.

ب

در این شرایط، نکته کلیدی این است که بر خلاف آنچه شاید انتظار برود، فقط یک حرف در نهایت در فایل چاپ خواهد شد. دلیل این موضوع این است که توصیف‌گرهای فایل file descriptors یا به اختصار fd بین این دو پردازه مشترک نیستند. هر پردازه دارای توصیف‌گر فایل منحصر به فرد خود است و هر یک به صورت مستقل به نوشتن (write) در فایل می‌پردازند.

بنابراین، اگر یکی از پردازه‌ها، چه پدر یا فرزند، دیرتر نسبت به دیگری اقدام به نوشتن در فایل کند، محتوای نوشته شده توسط پردازه قبلی توسط این پردازه بازنویسی خواهد شد. این بازنویسی به این معنی است که تنها آخرین نوشته (از پردازه‌ای که دیرتر نوشته است) در فایل باقی می‌ماند.

در نتیجه، اگر پردازه والد دیرتر نسبت به فرزند نوشته خود را انجام دهد، محتوای نوشته شده توسط فرزند بازنویسی شده و در نهایت "a" در فایل قرار خواهد گرفت. در صورتی که فرزند دیرتر اقدام به نوشتن کند، محتوای نوشته شده توسط والد جایگزین شده و نتیجه "b" خواهد بود. این بدان معناست که در هر دو حالت، امکان دارد که یکی از حروف "a" یا "b" به تنهایی در فایل باقی بماند.

ج

در این بخش از کد، در تضاد با بخش‌های قبلی، توصیف‌گر فایل (fd) به عنوان یک متغیر گلوبال تعریف شده و این امکان را فراهم می‌کند که بین یک پردازه و تردی که در آن ایجاد می‌شود به اشتراک گذاشته شود. در این کد، سه سناریو مختلف ممکن است رخ دهد:

- تردی که تازه ایجاد شده ممکن است قبل از اجرای خط ۹ کد اصلی فعال شود.
- ممکن است پس از اجرای خط ۹ و قبل از اتمام کامل اجرای کد، ترد اجرا شود.
- همچنین ممکن است ترد فرصت اجرا پیدا نکند تا قبل از اینکه برنامه به پایان برسد.

در این سه حالت، نتایج مختلفی در فایل حاصل می‌شوند: در حالت اول، مقادیر "ab" در فایل نوشته خواهند شد. در حالت دوم، مقادیر "ba" در فایل ظاهر می‌شوند. و در حالت سوم، تنها مقدار "b" در فایل باقی خواهد ماند. این سه سناریو نشان‌دهنده این واقعیت هستند که ترتیب اجرای تردها و پردازه‌ها می‌تواند تأثیر قابل توجهی بر نتیجه نهایی داشته باشد.

د

در حالت پیش‌فرض، با اتمام اجرای کد اصلی، تمام تردهایی که توسط آن کد ایجاد شده‌اند، خاتمه می‌یابند terminate می‌شوند. اما با افزودن `exit_pthread` در انتهای کد اصلی، امکان ادامه کار برای سایر تردها فراهم می‌شود و آن‌ها ترمینت نخواهند شد. این بدان معناست که در این حالت، برخلاف سناریویی که در آن ترد فرصت اجرا پیدا نمی‌کند، چنین موردی در کد رخ نمی‌دهد. بنابراین، در این حالت، هر دو نتیجه "ab" و "ba" می‌توانند در فایل نوشته شوند، چرا که هر دو ترد فرصت کافی برای اجرا قبل از پایان برنامه را خواهند داشت.

ه

در این قسمت، اگر هنگام تغییر بافر و نوشتن در بافر به فایل توسط یک پردازه یا ترد، در بخش موازی دیگر کد هیچ تغییری در بافر ایجاد نشود و چیزی در فایل نوشته نشود، رفتار کد کاملاً مشابه با حالت قبلی خواهد بود. در نتیجه، همه خروجی‌های احتمالی موجود در حالت قبلی در این حالت نیز ممکن هستند. اما، اگر یکی از بخش‌های کد مقداری را در بافر قرار دهد و قبل از نوشتن آن در فایل، بخش دیگری از کد بافر را دوباره تغییر دهد، احتمال دارد که یک حرف به طور دوبار در فایل چاپ شود. بنابراین، علاوه بر احتمال وجود مقادیر "ab" و "ba" در فایل، احتمال وجود مقادیر "aa" و "bb" نیز وجود دارد.