تمرین دوم

|  |  |
| --- | --- |
| علی علی‌محمّدی | 9613027 |

|  |  |
| --- | --- |
| 1) | بهتر است از سیستم چندفرایندی استفاده شود. سیستم کنترل هواپیما (Avionics) از اجزای بسیاری تشکیل شده است که برخی از آن‌ها عبارتند از: سنسورها، سطوح کنترلی، ارتباطات بی‌سیم و غیره. سیستم‌های هواپیما از دسته‌ی سیستم‌های بلادرنگ و safety-critical است و باید چند پردازنده داشته باشد تا در صورت از کار افتادن یکی از آن‌ها، عملکرد هواپیما و سیستم‌های دیگر آن، تحت تأثیر قرار نگیرند. (Redundancy) در واقع برای هر یک از سیستم‌ها پردازنده‌ی جدایی تعبیه شده است. از طرف دیگر، سیستم باید بلادرنگ عمل کند و نباید تأخیر داشته باشد و در نتیجه، استفاده از ساختار چندنخی ممکن است باعث ایجاد تأخیر در عملکرد بشود؛ زیرا زمان انجام یک کار در ساختار چندنخی بیشتر از ساختار چندفرایندی است که به دلیل تخصیص یافتن منابع یک پردازنده‌ی واحد به چند نخ متفاوت است. پس ساختار چندفرایندی سریع‌تر است که در سیستم‌های بلادرنگ اهمیت بسیاری دارد. پس با توجه به داشتن چند پردازنده‌ی مختلف برای سیستم‌های متفاوت هواپیما و اهمیت Redundancy و همچنین ماهیت بلادرنگ این نرم‌افزار، استفاده از ساختار چندفرایندی توصیه می‌شود. |
| 2) | Thread Switching نوعی تغییر زمینه از یک نخ به یک نخ دیگر در همان فرایند است. تغییر نخ بسیار کارآمد و بسیار ارزان‌تر است؛ زیرا تنها شامل تغییر شناسه‌ها و منابعی مانند شمارنده‌ی برنامه، ثبات‌ها و نشانگرهای پشته است. هزینه‌ی تغییر نخ‌به‌نخ تقریباً برابر همان هزینه‌ی ورود و خروج به هسته‌ی سیستم‌عامل است.  Process Switching نوعی تغییر زمینه است که در آن ما یک فرایند را با فرایند دیگری جایگزین (تعویض) می‌کنیم. این شامل جایگزین کردن تمام منابع فرآیند قبلی با منابع مورد نیاز فرایند جدید است. این به معنای تغییر فضای آدرس حافظه است که خود شامل آدرس‌های حافظه، جدول‌های صفحه (Page Tables) و منابع هسته و حافظه‌ی پنهان درون پردازنده (cache) است. |
| 3) | fork یک فرایند فرزند جدید ایجاد می‌کند، که یک کپی کامل از فرایند پدر است. فرایندهای فرزند و پدر از فضاهای آدرس مجازی مختلفی استفاده می‌کنند که در ابتدا با همان صفحات حافظه پر شده‌اند. سپس با اجرای هر دو فرایند، فضاهای آدرس مجازی آن‌ها بیشتر و بیشتر متفاوت می‌شوند؛ زیرا سیستم‌عامل یک کپی از صفحات حافظه را که توسط هر یک از این دو فرایند در حال نوشته شدن است، انجام می‌دهد و نسخه‌هایی مستقل از صفحات حافظه‌ی اصلاح شده را به هر فرایند اختصاص می‌دهد. به این روش، Copy-On-Write (COW) گفته می‌شود.  clone یک فرایند فرزند جدید ایجاد می‌کند. پارامترهای مختلف این فراخوانی سیستمی، مشخص می‌کند که کدام قسمت از فرایند پدر باید در فرایند فرزند کپی شود و کدام قسمت‌ها باید بین آن‌ها به اشتراک گذاشته شود. در نتیجه، از این فراخوان سیستمی می‌توان برای ایجاد انواع موجودیت‌های اجرایی، از نخ تا فرایندهای کاملاً مستقل استفاده کرد. در واقع فراخوانی سیستمیِ clone پایه و بنیانی است که برای پیاده‌سازی فراخوانی‌های سیستمیِ pthread\_create و تمام خانواده‌ی فرایند‌های سیستمیِ fork استفاده می‌شود.  در پیاده‌سازی fork، از clone استفاده می‌شود و این حقیقت که در clone، لزوماً تمام اطلاعات مربوط به فرایندها کپی نمی‌شوند و بخشی از آن‌ها می‌توانند بین دو فرایند به اشتراک گذاشته شوند؛ در نتیجه، clone اندکی سریع‌تر از fork است که البته قابل چشم‌پوشی است.  طبق توضیحات فوق، برای ایجاد یک نخ جدید باید از clone استفاده کرد. در مقابل، fork یک فرایند کاملاً جدا ایجاد می‌کند. |
| 4) | در سیستم‌عامل‌هایی که بر پایه‌ی Unix هستند، init (کوتاه‌شده‌ی initialization) اولین فرایندی است که در هنگام بوت شدن سیستم‌عامل اجرا می‌شود. init یک فرایندِ daemon است که تا زمان خاموش شدن سیستم به کار خود ادامه می‌دهد. این فرایند، جدّ مستقیم یا غیرمستقیم تمام فرایندهای دیگر است و به طور خودکار تمام فرایندهای یتیم را به فرزندی قبول می‌کند. در هنگام بوت شدن سیستم، init توسط هسته به اجرا در می‌آید. اگر هسته قادر به شروع آن نباشد، Kernel Panic اتفاق می‌افتد. معمولاً شناسه‌‌ی فرایند 1 (PID 1) به فرایندِ init اختصاص داده می‌شود. |
| 5) | ارتباط بین فرایندی به سه دلیل ضروری است:   * اشتراک‌گذاری اطلاعات: از آنجا که ممکن است چندین برنامه به اطلاعات یکسانی علاقه‌مند باشند (به عنوان مثال copy کردن و paste کردن) ، بنابراین ما باید محیطی را فراهم کنیم که امکان دسترسی همزمان به این اطلاعات را فراهم سازد. * افزایش سرعت محاسبه: اگر بخواهیم عملیات خاصی سریع‌تر انجام شود، باید آن را به وظایف فرعی تقسیم کنیم که هرکدام موازی با کارهای دیگر اجرا می‌شوند. توجه داشته باشید که چنین افزایشی تنها در صورتی امکان‌پذیر است که رایانه چندین هسته‌ی پردازشی داشته باشد. * ماژولار بودن: ممکن است بخواهیم سیستم را به صورت ماژولار بسازیم و عملکردهای سیستم را به فرایندها یا نخ‌های جداگانه تقسیم کنیم.   دو مدل اساسی ارتباط درون پردازشی وجود دارد: حافظه‌ی مشترک و ارسال پیام. در مدل حافظه‌ی مشترک، قسمتی از حافظه که توسط فرایندهای همکار به اشتراک گذاشته می‌شود، ایجاد می‌شود. سپس فرایندها می‌توانند با خواندن و نوشتن داده‌ها در منظقه‌ی مشترک، اطلاعات را تبادل کنند. در مدل ارسال پیام، ارتباط با استفاده از پیام‌هایی که بین فرایندهای همکار رد و بدل می‌شود، صورت می‌گیرد. ارسال پیام برای مبادله مقادیر کم از داده‌ها مناسب است؛ زیرا نیازی به جلوگیری از هیچ تداخلی (Conflict) نیست. علاوه بر آن، پیاده‌سازی مکانیزم ارسال پیام در سیستم‌های توزیع‌شده نسبت به مکانیزم حافظه‌ی مشترک آسان‌تر است. حافظه‌ی مشترک سریع‌تر از ارسال پیام است؛ زیرا سیستم‌های انتقال پیام معمولاً با استفاده از فراخوانی‌های سیستم پیاده‌سازی می‌شوند و بنابراین به دخالت بیشتر هسته نیاز دارند که باعث زمان‎بر شدن کل این مکانیزم گردد.  مثال‌هایی از روش‌های مختلف ارتباط بین فرایندی عبارتند از حافظه‌ی مشترکِ POSIX، ارسال پیامِ Mach، Windows و خطوط لوله (pipes). |
| 6) | systemd یک سیستمِ init است که توسط برخی توزیع‌های لینوکس برای خودآماده‌سازی فضای کاربر و همچنین مدیریت همه‌ی فرایند‌های پس از خود است. این سیستم جایگزین Unix SystemV یا سیستمِ BSD init است. حرف d در systemd از کلمه‌ی daemon در سیستم‌های Unix گرفته شده است. این نرم‌افزار تحت پروانه‌ی عمومی همگانی GNU انتشار یافته است. یکی از هدف‌های systemd، یک‌پارچه‌سازی تنظیمات پایه‌ای لینوکس و رفتار سرویس‌ها در میان همه‌ی توزیع‌های این سیستم‌عامل است.  از مزایای systemd می‌توان ساده‌سازی اسکریپت‌های initialization، سازگاری بیشتر با SELinux و لاگ کردن فرایندهای پایه‌ای را نام برد اما به دلیل پیچیدگی بیشتر، رفع اشکالات آن سخت‌تر است که یک عیب محسوب می‌شود.  برخی از اجزای systemd عبارتند از:   |  |  | | --- | --- | | سیستم ساده‌ی مدیریت راه‌اندازِ UEFI (boot manager) | systemd-boot | | اعمال تنظیمات پایه‌ای سیستم پیش از اولین راه‌اندازی (بوت) | systemd-firstboot | | مدیریت حساب‌های کاربری انسانی | systemd-homed | | مدیریت جلسه | systemd-logind | | مدیریت تنظیمات شبکه | systemd-networkd | | محفظه‌ی فضای حافظه‌ی سبک | systemd-nspawn | | مدیریت سرویس Network Name Resolution | systemd-resolved | | ساخت کاربرها و گروه‌های سیستم و اضافه کردن کاربرها به گروه‌ها در هنگام نصبِ packageها یا هنگام بوت شدن | systemd-sysusers | | همگام‌سازی زمان سیستم در محیط شبکه | systemd-timesyncd | | لاگ کردن سیستم | systemd/Journal | | تایمرهای یکنواخت یا بلادرنگ برای کنترل فایل‌ها و اتفاق‌های .service و یک جایگزین منطقی برای cron | systemd/Timers | | نصب و برپایی (mounting) | systemd.mount | | تهیه‌ی نسخه‌ی binary فرایند سنتیِ init در لینوکس | systemd-sysvcompat | | مدیریت فایل‌های موقت | systemd-tmpfiles | |
| 7) | در کد زیر در مجموع هشت () فرایند تولید می‌شود. برای اطمینان از تولید این فرایندها کافی است که پس از سه بار فراخوانی تابعِ fork، یک عبارت ساده را چاپ کنیم. مشاهده می‌شود که آن عبارت، 8 بار چاپ می‌شود.  #include <unistd.h>  #include <stdio.h>  int main() {  fork(); fork(); fork();  printf("Hello\n");  return 0;  } |
| 8) | نتیجه‌ی چند اجرا:  #1:  #2:  #3:  #4:  #5:  خروجی هر اجرا عددی بین و است و در هر بار اجرا نتیجه‌ی متفاوتی به دست می‌آید؛ زیرا یک شرایط مسابقه‌ای ایجاد کرده‌ایم. در واقع، هر دو نخ با یکدیگر مسابقه می‌دهند و هر کدام تلاش می‌کند تا با حداکثر سرعت ممکن و حداکثر دفعات ممکن، به متغیر شمارنده‌ی مشترک دسترسی پیدا کند. خروجی نهایی با توجه تعداد و مقدار زمان‌هایی که هر نخ، در پردازنده اجرا می‌شود (scheduled)، وابسته است که این موارد نیز ناشی از عوامل متفاوتی است؛ مانند نحوه‌ی استفاده از سیستم و این که چه فرایندهای دیگری در آن زمان، در حال اجرا هستند. اگر هر دو نخ زمان کافی در اختیار داشته باشند، خروجی برابر خواهد بود. در غیر این صورت، با اجرای تابعِ join، اجرای یک نخ خاتمه می‌یابد و اجرای نخ دیگر قبل از این که بتواند زمان لازم برای اجرای کامل را در اختیار داشته باشد، خاتمه می‌یابد و عددی کمتر از حاصل می‌شود. |