کتاب کوچک سمافورها

آلن بی.دونی

مترجمین:
سیّدمحمّدجواد رضویان، سیّدعلی آل‌طه و محمّدمهدی قاسمی‌نیا

نسخه ۲٫۲٫۱

کتاب کوچک سمافورها

ویرایش دوم

نسخه ۲٫۲٫۱

فرم اصلی این کتاب، سورس کد لاتک است. کامپایل این سورس لاتک سبب تولید یک نمایش کتاب بدون وابستگی به دستگاه خواهد شد که می‌تواند به دیگر فرمت‌ها تبدیل و چاپ گردد.

این کتاب توسط نویسنده با استفاده از لاتک، dvips و ps2pdf حروفچینی شده است که همگی برنامه‌هایی کدباز هستند. سورس لاتک این کتاب در آدرس http://greenteapress.com/semaphores موجود است.[۱]

[۱] در حروف‌چینی ترجمه این کتاب از زی‌لاتک و بسته زی‌پرشین استفاده شده است.

پیشگفتار

غالب کتاب‌های درسی سیستم‌های عامل در دوره کارشناسی مبحثی در همگام‌سازی دارند که مجموعه‌ای از ابزارهای بدوی (میوتکس، سمافور، ناظر و متغیرهای شرطی) و مسائل کلاسیکی مثل خواننده–نویسنده و تولیدکننده–مصرف‌کننده را به طور معمول معرفی می‌نمایند.

وقتی که من در برکلی کلاس سیستم‌عامل را برداشتم، و در کالج کالبی[۲] آن را تدریس کردم، این عقیده را یافتم که بیشتر دانشجویان قادر به درک راه‌حل ارائه شده برای اینگونه مسائل بودند، اما تنها برخی از این دانشجویان قادر بودند که خودشان آن راه‌حل را تولید کرده و یا مسائل مشابهی را حل نمایند.

یکی از دلایلی که دانشجویان نمی توانند به طور عمیق این قبیل مسایل را بفهمند، این است که این مسائل وقت و تلاش بیشتری نسبت به آنچه در غالب کلاس‌ها صرف می‌شود، نیاز دارند. همگام‌سازی تنها یکی از مباحث زمان‌بر در کلاس سیستم‌عامل است، و گمان نمی‌کنم بتوانم استدلال نمایم که مهمترین آن مباحث است. امّا فکر می‌کنم که همگام‌سازی یکی از چالشی‌ترین، جالب‌ترین و سرگرم کننده‌ترین بخش‌های سیستم‌عامل است.

با هدف شناساندن اصطلاحات والگوهای همگام‌سازی به گونه‌ای که به صورت مستقل قابل درک باشد و بتوان از آنها برای حل مسائل پیچیده استفاده نمود، اولین ویرایش این کتاب را نوشتم. نوشتن کدهمگام‌سازی چالش‌های مختص به خود را دارد زیرا که با افزایش تعداد اجزا و تعداد تعاملات به طور غیر قابل کنترلی افزایش می‌یابد.

با این وجود در بین راه حل‌هایی که دیدم، الگوهایی را یافتم و حداقل به برخی رهیافت‌های روشمند درست برای ترکیب راه حل‌ها رسیدم.

در زمانی که سیستم‌عامل را در کالج ولزلی[۳] تدریس کردم این شانس را داشتم تا این رهیافت را بیازمایم. اولین نسخهٔ **کتاب کوچک سمافورها** را در کنار یکی از کتب درسی استاندارد بکار بردم و همگام‌سازی را به صورت موازی با درس اصلی تدریس می کردم. هر هفته به دانشجویان چند صفحه از کتاب را می دادم که با یک معمّا تمام می‌شد و گاهی اوقات یه راهنمایی مختصر نیز داشت. به آن‌ها می‌گفتم که به راهنمایی نگاه نکنند مگر

[2]Colby College [3]Wellesley College

اینکه گیر افتاده باشند.

همچنین به آن‌ها ابزارهایی برای تست راه حل‌هایشان دادم: یه تخته مغناطیسی کوچک که می توانستند کدهایشان را بنویسند و یک بسته آهنربا برای نمایش تردهایی که کد را اجرا می‌کنند.

نتیجه بسیار چشمگیر بود. هر چه زمان بیشتری در اختیار دانشجویان می‌گذاشتم، عمق فهمشان بیشتر می‌شد که تا قبل این ندیده بودم. مهمتر اینکه غالبشان قادر به حل بیشتر معمّاها بودند. در برخی موارد، همان راه‌حل‌های کلاسیک را می‌یافتند و در مواردی به رهیافت‌های خلاقانه جدیدی می‌رسیدند.

وقتی که به کالج اُلین[۴] منتقل شدم، گام بعدی را با ایجاد کلاس فوق‌برنامه همگام‌سازی برداشتم، که در آن کلاس، کتاب **کتاب کوچک سمافورها** و همچنین پیاده‌سازی ابزارهای بدوی همگام‌سازی در زبان اسمبلی x86 و پازیکس و پایتون تدریس شد.

دانشجویانی که این درس را گرفتند در یافتن خطاهای ویرایش نخست کمک کردند و چند نفر از آنان راه‌حل‌هایی بهتر از راه‌حل‌های من ارائه دادند. در پایان ترم، از هر کدام آنان خواستم که یک مسأله جدید (ترجیحاً با یک راه‌حل) بنویسند. این مشارکت‌ها را به ویرایش دوم اضافه نمودم.

همچنین، پس از عرضهٔ ویرایش نخست، کنث ریک [۵] مقالهٔ «الگوهای طراحی سمافورها»[۶] را در گروه ویژهٔ ACM علاقمند به تعلیم در علوم کامپیوترارائه داد. او در این مقاله مسأله‌ای را که من به آن "مسأله سوشی بار"[۷] می گویم معرفی نموده و دو راه‌حل برای اثبات آن ارائه داد و الگوهایی که وی آن‌ها را "باتوم را منتقل کنید"[۸] و "بخاطر تو انجام این کار را خواهم داد"[۹] نامید را نشان می‌داد. هنگامی که با این الگوها آشنا شدم، توانستم آن‌ها را در مسائل ویرایش نخست کتاب به کار برم و راه‌حل‌هایی تولید کنم که به نظرم بهتر هستند.

تغییر دیگر در ویرایش دوم، نحو[۱۰] آن است. بعد از آنکه ویرایش اول را نوشتم، زبان برنامه نویسی پایتون را آموختم که نه تنها یکی از عالیترین زبان‌های برنامه‌نویسی است بلکه یک زبان شبه‌کد[۱۱] عالی را فراهم می‌آورد. در نتیجه از یک نحو شبیه به C در ویرایش نخست به نحوی که کاملاً نزدیک به زبان پایتون بود سوئیچ کردم.[۱۲] در حقیقت، شبیه‌سازی نوشته‌ام که بسیاری از راه‌حل‌های ارائه شده در این کتاب را می‌تواند اجرا کند.

خواننده‌هایی هم که با زبان پایتون آشنا نیستند نیز (امیدارم) آن را کاملاً واضح بیابند. در مواردی که از ویژگی‌های مشخص زبان پایتون استفاده می‌نمایم، نحو زبان و معنای آن را شرح می‌دهم. امیدوارم این تغییرات کتاب را خواناتر نماید.

صفحه بندی این کتاب ممکن است کمی عجیب به نظر آید، امّا برای صفحات خالی نیز متدی وجود دارد. بعد از هر معمّا، فضای خالی کافی برای راهنمایی که در صفحه بعد می‌آید و نیز راه‌حلی که در صفحه بعد از آن می‌آید را قرار داده‌ام. زمانی که این کتاب را در کلاسم بکار می‌برم، صفحاتی چند را در یک زمان به ایشان می‌دهم

[۱۲] تفاوت اساسی در این است گاهی اوقات از حاشیه به منظور نمایش کدهایی که توسط یک میوتکس محافظت می‌شوند استفاده می‌کنم و این می‌تواند سبب یک خطای نحوی در پایتون شود.

[4] Olin College [5] Kenneth Reek [6] Design Patterns for Semaphores [7] Sushi Bar Problem [8] Pass the baton [9] I'll do it for you [10] syntax [11] pseudocode

و آنان نیز آن صفحات را در کلاسور جمع‌آوری می‌نمایند. سیستم صفحه‌بندی، این امکان را فراهم می‌سازد تا مسائل را بدون صفحات راهنمایی و راه‌حل توزیع نمایم. گاهی اوقات صفحات راهنمایی را تا کرده و منگنه می‌زنم و همراه مسأله به دانشجویان می‌دهم تا خودشان تصمیم بگیرند که آیا به آن راهنمایی نگاهی بیندازند و چه زمانی چنین کنند. اگر شما کتاب را روی یک طرف صفحه چاپ می‌کنید، می‌توانید از صفحات سفید صرف نظر نمایید و این سیستم هنوز کار خواهد کرد.

این یک کتاب آزاده است، به این معنی که هر شخصی می‌تواند آن را بخواند، رونوشت بردارد، اصلاح کند و حتی بازپخش نماید با توجه به محدودیت‌های لایسنس کتاب. امیدوارم که افراد، این کتاب را سودمند بیابند، امّا همچنین امید دارم که با ارسال اصلاحات، پیشنهادات، و موادی اضافی توسعهٔ آن را ادامه دهند. با تشکر آلن

دونی
نیدهام، ماساچوست
اول ژوئن ۲۰۰۵

لیست مشارکت‌کنندگان

در ادامه لیست برخی افرادی که در این کتاب مشارکت داشته‌اند آمده است:

- بسیاری از مسائل این کتاب، گونه‌های دیگری از مسائل کلاسیک هستند که برای اولین بار در مقالات تخصصی و سپس در کتب مرجع مطرح شدند. هرجایی که بدانم یک مسأله یا راه حلی از کجا آمده است، در متن کتاب به آن اشاره می‌کنم.
- همچنین از دانشجویان کالج ولزلی که با ویرایش اول این کتاب و نیز دانشجویان کالج اُلین که با ویرایش دوم کتاب کار کردند، تشکر می‌نمایم.
- Se Won تصحیحی کوچک -لکن مهم- را در ارائه راه‌حل Tanenbaum نسبت به مسالهٔ فیلسوف‌های در حال غذا خوردن ارسال نموده‌اند.
- Daniel Zingaro در مسالهٔ Dancer نکته‌ای را متذکر گردید که سبب بازنویسی مجدد آن بخش گردید. امیدوارم اکنون با معنی‌تر شده باشد. علاوه بر این Daniel یک خطا را در نسخهٔ قبلی راه‌حل مسأله H_2O نشان داده‌اند و سال بعد از آن، تعدادی خطاهای تایپی را متذکر شده‌اند.
- Thomas Hansen یک خطای تایپی را در مسأله Cigarette smokers یافته است.
- Pascal Rütten به چندین اشکال تایپی از جمله تلفظ نادرست Edsger Dijkstra اشاره نموده است.

- Marcelo Johann خطایی را در راه‌حل من نسبت به مسألهٔ Dining Savages یافته و آن را اصلاح کرده است.
- Roger Shipman یک مجموعه اصلاحات به علاوهٔ یک گونهٔ جذاب از مسأله Barrier را ارسال نموده است.
- Jon Cass مطلبی را مشخص نمود که در مسأله فیلسوف‌های در حال غذا خوردن از از قلم افتاده بود.
- Krzysztof Kościuszkiewicz چندین اصلاح از جمله، جا افتادن خطی در تعریف کلاس Fifo را فرستاده است.
- Fritz Vaandrager از دانشگاه Radboud هلند و دانشجویانش Marc Schoolderman، Manuel Stampe و Lars Lockefeer ابزاری بنام UPPAAL را به منظور بررسی چندین راه‌حل این کتاب بکار برده و خطاهایی را در راه حل‌های ارائه شده برای مسأله‌های Room Party و Modus Hall یافته‌اند.
- Eric Gorr درست نبودن یک توضیح در فصل سوم را مشخص نموده است.
- Jouni Leppäjärvi در واضح نمودن مبدأ سمافورها کمک نموده است.
- Christoph Bartoschek خطایی را در راه‌حل مسأله رقص انحصاری یافته است.
- Eus یک خطای تایپی در فصل سوم را پیدا کرده است.
- Tak-Shing Chan یک خطای خارج از محدوده[۱۳] را در `counter_mutex.c` یافته است.
- Roman V. Kiseliov چند پیشنهاد برای بهبود ظاهر کتاب ارائه داده و با چند نکته در LaTeX مرا راهنمایی نموده است.
- Alejandro Céspedes در حال کار روی ترجمه اسپانیایی این کتاب است و چندین غلط تایپی را در آن یافته است.
- Erich Nahum مشکلی را در تطبیق راه‌حل Kenneth Reek نسبت به مسأله Sushi Bar یافته است.
- Martin Storsjö تصحیحی را در مسأله generalized smokers ارسال نموده است.
- Cris Hawkins به یک متغیر بدون استفاده اشاره نموده است.
- Adolfo Di Mare یک "and" از جا افتاده را یافته است.

[13] out-of-bounds

- Simon Ellis یک خطای تایپی را یافته است.
- Benjamin Nash یک خطای تایپی، خطایی در یک راه‌حل و یک ایراد در راه‌حل دیگری را یافته است.
- Alejandro Pulver مشکلی را در راه‌حل مسأله Barbershop یافته است.

فهرست مطالب

فصل ۱

معرفی

۱.۱ همگام‌سازی

اصطلاحاً "همگام‌سازی" به معنی وقوع همزمان دو چیز است. در سیستم‌های کامپیوتری همگام‌سازی کمی کلی‌تر است؛ این اصطلاح به رابطه‌های بین رویدادها -هر تعداد رویداد و هر نوع رابطه (قبل، حین، بعد)- اشاره دارد.

غالباً برنامه‌نویسان کامپیوتر با محدودیت‌های همگام‌سازی مواجه‌اند، که این محدودیت‌ها الزاماتی در ارتباط با ترتیب این رویدادها می‌باشد. مثال‌هایی نظیر:

تسلسل: رخداد الف باید که پیش از رخداد ب اتفاق بیفتد.

انحصار متقابل: رخداد الف و ب نباید در یک زمان رخ دهند.

در زندگی واقعی غالباً محدودیت‌های همگام‌سازی را با کمک یک ساعت بررسی و اعمال می‌کنیم. چگونه می‌فهمیم که رخداد الف قبل از رخداد ب روی داده است؟ اگر زمان وقوع هر دو رویداد را بدانیم، می‌توانیم زمان‌ها را با هم مقایسه کنیم.

در سیستم‌های کامپیوتری غالباً می‌بایست که محدودیت‌های همگام‌سازی را بدون بهره بردن از ساعت برآورده نماییم، زیر را که یا هیچ ساعت جهانی وجود ندارد یا اینکه با یک دقت به اندازه کافی خوب، زمان دقیق وقوع رویدادها را نمی‌دانیم.

این کتاب درباره تکنیک‌های نرم‌افزاری برای اعمال‌های محدودیت‌های همگام‌سازی در کامپیوتر است.

۲.۱ مدل اجرا[1]

به منظور درک همگام‌سازی نرم‌افزاری، باید مدلی از چگونگی اجرای برنامه‌های کامپیوتری داشته باشید. در ساده‌ترین مدل، کامپیوترها دستورات را به ترتیب، یکی پس از دیگری اجرا می‌نمایند. در این مدل، همگام‌سازی بدیهی است؛ ترتیب وقایع را با نگاه به برنامه می‌توان بیان نمود. اگر دستور A قبل از دستور B آمده باشد، اوّل اجرا می‌گردد.

در دو صورت، همگام‌سازی پیچیده‌تر خواهد شد. ممکن است کامپیوتر موازی باشد، بدین معنی که چندین پردازنده در یک زمان در حال اجرا باشد. در این حالت نمی‌توان به سادگی فهمید که دستوری در یک پردازنده قبل از دستور دیگری در پردازندهٔ دیگر اجرا شده است.

و یا ممکن است یک پردازنده چندین نخ[2] اجرایی داشته باشد. نخ، دنباله‌ای از دستورات است که به ترتیب اجرا می‌شوند. اگر چندین نخ وجود داشته باشد آنگاه پردازنده می‌تواند برای مدتی بر روی یکی از نخ‌ها کار کند و سپس به نخی دیگر منتقل شود و به همین ترتیب ادامه دهد.

به طور کلی برنامه‌نویس هیچ کنترلی روی زمان اجرای نخ‌ها ندارد؛ در واقع سیستم‌عامل (بخصوص زمان‌بند) در این باره تصمیم می‌گیرد. در نتیجه برنامه‌نویس نمی‌تواند بگوید که دستورات چه زمانی در نخ‌های مختلف اجرا خواهد شد.

برای مقاصد همگام‌سازی، تفاوتی بین مدل موازی و مدل چند نخی وجود ندارد. مسأله یکی است-در یک پردازنده (یا یک نخ) ترتیب اجرا مشخص است امّا بین پردازنده‌ها (یا نخ‌ها) بیان این ترتیب غیر ممکن است.

یک مثال واقعی این مسأله را روشن‌تر می‌نمایند. تصور کنید که شما و دوستتان Bob در شهرهای متفاوتی زندگی می‌کنید. یک روز نزدیک وقت ناهار، شما به این فکر می‌افتید که چه کسی امروز زودتر ناهار خواهد خورد، شما یا Bob. چگونه این را در می‌یابید؟

به سادگی می‌توانید به او زنگ بزنید و بپرسید که چه زمانی ناهار خورده است. امّا اگر شما با ساعت خودتان در ۵۹/۱۱ غذا را شروع نموده باشید و Bob با ساعت خودش در ۰۱/۱۲، آن وقت چه؟ آیا می‌توانید مطمئن باشید که چه کسی زودتر شروع نموده است؟ تنها در صورتی ممکن است که هر دوی شما نسبت به دقیق بودن ساعت‌هایتان حساس بوده باشید.

سیستم‌های کامپیوتری با مشکل مشابهی مواجه هستند زیرا با وجود اینکه معمولاً ساعت‌هایشان دقیق است امّا همیشه در میزان دقت ساعت‌ها محدودیت وجود دارد. به علاوه، اکثر اوقات کامپیوتر زمان وقوع رخدادها را دنبال نمی‌نماید. چرا که تعداد بسیار زیادی رخداد آن هم با سرعتی بسیار در حال وقوع است که ذخیره زمان دقیق همه آن‌ها ممکن نیست.

معمّا: با فرض اینکه Bob می‌خواهد دستورات ساده‌ای را دنبال نماید آیا راهی وجود دارد که **تضمین** نمایید فردا شما زودتر از او ناهار خواهید خورد؟

[1] Execution model [2] thread

۳.۱ تسلل به کمک تبادل پیام

یک راه آن است که به Bob بگویید تا شما به او زنگ نزده‌اید ناهار نخورد. شما نیز اطمینان دهید پس از ناهار زنگ می‌زنید. اگر چه این راهکار بدیهی به نظر می‌رسد لکن ایدهٔ پایهٔ آن، تبادل پیام[۳]، راه حلّی واقعی برای بسیاری از مسائل همگام‌سازی می‌باشد. جدول زمانی زیر را در نظر بگیرید.

نخ A (شما)

```
1 Eat breakfast
2 Work
3 Eat lunch
4 Call Bob
```

نخ B (Bob)

```
1 Eat breakfast
2 Wait for a call
3 Eat lunch
```

اولین ستون لیست اعمالی است که شما انجام می‌دهید؛ به عبارت دیگر نخ اجرای شما. ستون دوم نیز نخ اجرای Bob است. درون یک نخ همیشه می‌توانیم ترتیب اجرای وقایع را بگوییم. ترتیب وقایع را به این صورت می‌توانیم نشان دهیم

$$a۱ < a۲ < a۳ < a۴$$

$$b۱ < b۲ < b۳$$

که در آن، رابطه $a۱ < a۲$ به معنای وقوع $a۱$ پیش از $a۲$ است.

اگرچه به طور کلی، هیچ راهی برای مقایسه رخدادهای نخ‌های مختلف نداریم؛ برای مثال ایده‌ای از اینکه چه کسی ابتدا صبحانه می‌خورد نداریم (آیا $a۱ < b۱$ است؟).

امّا با کمک تبادل پیام (تماس تلفنی) **می‌توانیم** بگوییم چه کسی زودتر ناهار خورده است ($a۳ < b۳$). با فرض اینکه باب هیچ دوست دیگری نداشته باشد هیچ تماسی جز از شما دریافت نخواهد کرد بنابراین ($b۲ < a۴$). با ترکیب تمامی روابط، داریم

$$b۳ > b۲ > a۴ > a۳$$

که ثابت می‌کند شما قبل از باب ناهار خورده‌اید.

در این حالت، می‌گوییم شما و باب به صورت **متوالی**[۴] ناهار خورده‌اید زیرا ترتیب وقایع را می‌دانیم. از طرف دیگر صبحانه را به صورت **همروند**[۵] خورده‌اید زیرا که ترتیب مشخص نیست.

مواقعی که دربارهٔ رخدادهای همروند صحبت می‌کنیم، اغوا کننده است که بگوییم آن‌ها به صورت همزمان اتفاق افتاده‌اند. تعبیر فوق تا زمانی که تعریف دقیق زیر را در خاطر دارید بلامانع است:

دو واقعه، همروند هستند اگر با نگاه به برنامه نتوانیم بگوییم کدامیک زودتر رخ می‌دهد.

[3] message passing [4] sequential [5] concurrent

گاهی اوقات پس از اجرای برنامه می‌توانیم بگوییم که کدامیک ابتدا رخ داده است، امّا غالباً ممکن نیست، و حتی اگر هم بتوانیم، باز هم تضمینی نیست که مرتبه بعد نتیجه‌ای یکسان بگیریم.

۴.۱ عدم قطعیت

برنامه‌های همروند اغلب **غیر قطعی**[۶] هستند به این معنی که با نگاه به برنامه نمی‌توانیم بگوییم آن زمانی که برنامه اجرا می‌شود چه اتفاقی خواهد افتاد. در ادامه یک برنامه سادهٔ غیر قطعی آمده است:

نخ A

```
1 print "yes"
```

نخ B

```
1 print "no"
```

از آنجایی که دو نخ به صورت همروند اجرا می‌شوند، ترتیب اجرا بستگی به زمان‌بند دارد. در هر اجرای این برنامه، خروجی ممکن است "yes no" یا "no yes" باشد.

عدم قطعیت یکی از مواردی است که اشکال‌زدایی برنامه‌های همروند را مشکل می‌سازد. برنامه‌ای ممکن است ۱۰۰۰ بار بر روی یک سطر به درستی کار کرده و سپس در اجرای ۱۰۰۱ام بسته به تصمیمات خاص زمان‌بند با مشکل مواجه شده و اجرای برنامه متوقف شود.

پیدا کردن این نوع خطاها با بررسی کد تقریباً ناممکن است؛ این نوع خطاها تنها از طریق برنامه‌نویسی دقیق قابل اجتناب هستند.

۵.۱ متغیرهای اشتراکی

بیشتر مواقع، غالب متغیرها در اکثر نخ‌ها **محلی**[۷] هستند، بدین معنی که تنها به یک نخ تعلق دارند و سایر نخ‌ها نمی‌توانند به آن‌ها دسترسی داشته باشند. تا زمانیکه این نکته برقرار است، مشکلات همگام‌سازی کمی وجود دارد زیرا که نخ‌ها دخالتی در آن متغیرها ندارند.

امّا گاهی اوقات برخی متغیرها بین دو یا چند نخ به صورت **اشتراکی**[۸] در دسترس هستند؛ این یکی از شیوه‌های تعامل نخ‌ها با یکدیگر است. برای مثال، یک راهِ تبادل اطلاعات بین نخ‌ها، این است که نخی مقداری را بخواند و نخ دیگر آن را بنویسد.

اگر نخ‌ها ناهمگام باشند آنگاه با نگاه کردن به کد نمی‌توانیم بگوییم که آیا نخ خواننده، مقداری را که نویسنده نوشته است می‌بیند یا همان مقدار قبلی را خواهد دید. لذا بسیاری از برنامه‌ها محدودیت‌هایی را بر روی خواننده‌ها اعمال می‌نمایند؛ تا زمانیکه نویسنده مقدار را ننوشته است چیزی را نخواند. این دقیقاً همان مسأله تسلسل است که در بخش ۳.۱ آمده است. نوشتن همروند (دو یا بیشتر نویسنده) و بروزرسانی همروند (دو یا بیشتر نخ که خواندنی پس از نوشتن دارند)، شیوه‌های دیگری از تعامل نخ‌ها با یکدیگر است. دو بخش بعدی با این تعاملات سر و کار خواهد داشت. خواندن همروند متغیرهای اشتراکی که گونهٔ دیگری از این تعامل است عموماً مشکل همگام‌سازی تولید نمی‌نماید.

[6] non-deterministic [7] local [8] shared

۱.۵.۱ نوستن‌های همروند

در مثال بعد، x یک متغیر اشتراکی است که دو خواننده به آن دسترسی دارند.

نخ A

```
1 x = 5
2 print x
```

نخ B

```
1 x = 7
```

کدام مقدار x چاپ خواهد شد؟ در پایان اجرای تمام این دستورات، مقدار x چیست؟ این بستگی به ترتیب اجرای هر یک از دستورات دارد که به آن **مسیر اجرا**[۹] گفته می‌شود. $a۱ < a۲ < b۱$ یکی از مسیرهای ممکن است که در آن خروجی برنامه ۵ است، درحالی‌که مقدار نهایی ۷ خواهد بود.

معمّا: چه مسیری منجر به خروجی ۵ و مقدار نهایی ۵ می‌شود؟

معمّا: چه مسیری منجر به خروجی ۷ و مقدار نهایی ۷ می‌شود؟

معمّا: آیا مسیری وجود دارد که منجر به خروجی ۷ و مقدار نهایی ۵ شود؟ می توانید جواب خود را ثابت کنید؟

پاسخ به چنین سوالاتی یکی از بخش‌های مهم برنامه‌نویسی همروند است: مسیرهای ممکن کدام‌ها هستند و هر یک از این مسیرها چه تاثیراتی دارند؟ آیا می‌توان ثابت نمود که اثری (خواسته) ضروری است و یا اینکه اثری (ناخواسته) غیر ممکن است؟

۲.۵.۱ بروزرسانی‌های همروند

بروزرسانی عملی است که مقدار متغیری را خوانده، یک مقدار جدید را بر مبنای مقدار قبلی محاسبه نموده و سپس مقدار جدید را می‌نویسد. رایج‌ترین نوع بروزرسانی، یک افزایش[۱۰] است که مقدار جدید برابر مقدار قبلی به اضافه یک واحد است. مثال بعد متغیر اشتراکی `count` را نشان می‌دهد که بوسیله دو نخ به صورت همزمان بروزرسانی می‌گردد.

نخ A

```
1 count = count + 1
```

نخ B

```
1 count = count + 1
```

در نگاه اول، اینکه یک مشکل همگام‌سازی در اینجا وجود دارد اینقدر واضح نیست. تنها دو مسیر اجرا وجود دارد و هر دو، نتیجه‌ای یکسان تولید می‌نمایند.

مشکل این است که این دستورات قبل از اجرا به زبان ماشین ترجمه می‌شوند و در زبان ماشین، یک بروزرسانی شامل دو گام است: یک خواندن و یک نوشتن. اگر کد را، با یک متغیر موقتی `temp` بازنویسی نماییم، این مشکل واضح‌تر خواهد شد.

[۹] execution path [۱۰] increment

نخ A

```
1 temp = count
2 count = temp + 1
```

نخ B

```
1 temp = count
2 count = temp + 1
```

اکنون مسیر اجرای زیر را در نظر بگیرید

$$a۱ < b۱ < b۲ < a۲$$

اگر مقدار اولیه x برابر ۰ باشد، مقدار نهایی چند است؟ از آنجایی که هر دو نخ مقدار اولیه یکسانی را می‌خوانند، هر دو مقدار یکسانی را می‌نویسند. این متغیر تنها یک مرتبه افزایش می‌یابد که احتمالاً آن چیزی نیست که برنامه‌نویس در ذهن خود داشته است.

چنین مسائلی از ظرافت بالایی برخودار هستند زیرا که همیشه این امکان وجود ندارد که با نگاه کردن به یک برنامهٔ سطح‌بالا بگوییم کدام عملیات در یک گام انجام شده و کدام‌ها وقفه‌پذیر هستند. در واقع، برخی کامپیوترها نوعی دستور افزایشی را فراهم می‌آورند که به صورت سخت‌افزاری پیاده‌سازی شده است و وقفه‌پذیر نیست. عملی که وقفه‌پذیر نباشید **اتمی**[11] گفته می‌شود.

خوب اگر ندانیم چه اعمالی اتمی هستند چگونه می توانیم برنامه‌هایی همروند بنویسیم؟ یک روش، جمع‌آوری اطلاعات مشخصی درباره هر عمل بر روی هر سکوی سخت‌افزاری است. ایرادات این رهیافت واضح است.

رایج‌ترین جایگزین این است که محتاطانه فرض کنیم تمامی بروزرسانی‌ها و نوشتن‌ها اتمی نیستند و از محدودیت‌های همگام‌سازی به منظور کنترل دسترسی همروند به متغیرهای اشتراکی استفاده نماییم.

معمول‌ترین محدودیت انحصار متقابل[12] یا mutex است که در بخش ۱.۱ اشاره شد. انحصار متقابل تضمین می‌نماید در یک زمان خاص فقط یک نخ به متغیر اشتراکی دسترسی دارد که موجب برطرف شدن این نوع خطاهای همگام‌سازی مطرح شده در این بخش می‌گردد.

معمّا: فرض کنید ۱۰۰ تا نخ، برنامه زیر را به صورت همزمان اجرا می‌نمایند (اگر با زبان پایتون آشنا نیستند حلقهٔ for یکصد مرتبه بروزرسانی انجام می‌دهد.):

```
1 for i in range(100):
2     temp = count
3     count = temp + 1
```

بزرگترین مقدار ممکن count پس از اجرای تمام نخ‌ها چقدر است؟ کوچکترین مقدار ممکن چقدر است؟ راهنمایی: سوال اول ساده ولی دومی به آن سادگی نیست.

[11] atomic [12] mutual exclusion

۳.۵.۱ انحصار متقابل با تبادل پیام

همانند تسلل، انحصار متقابل می‌تواند با استفاده از تبادل پیام پیاده‌سازی شود. برای مثال، فرض کنید شما و باب با یک راکتور هسته‌ای سر و کار دارد و آن را از راه دور تحت نظر دارید. غالب زمان‌ها، هر دوی شما چراغ‌های اخطار را زیر نظر دارید امّا هر دو اجازه دارید برای ناهار دست از کار بکشید. اینکه چه کسی اول ناهار می‌خورد اهمیتی ندارد امّا مهم است که ناهار هر دوی شما همزمان نباشد تا راکتور بدون نظارت باقی نماند!

معمّا: فرض کنید از یک سیستم تبادل پیام (تماس‌های تلفنی) برای اعمال این محدودیت‌ها استفاده می‌کنید. هیچ ساعتی وجود ندارد و شما نمی‌توانید زمان شروع ناهار یا مدت زمان صرف ناهار را پیش‌بینی کنید. حداقل تعداد تماس‌های لازم چقدر است؟

فصل ۲

سمافورها

در دنیای واقعی، سمافور یک سیستم از سیگنال‌هایی است که به منظور ارتباط بصری بکار می‌رود این سیگنال‌ها معمولاً پرچم، نور یا مکانیزم دیگری هستند. در نرم‌افزار، سمافور ساختمان داده‌ای است که برای حل انوع گوناگونی از مسائل همگام‌سازی مفید است.

سمافورها توسط Edsger Dijkstra -دانشمند مشهور و اعجوبه کامپیوتر- ابداع گردیده است. از زمان طراحی اولیه، برخی از جزئیات تغییر کرده است ولی ایدهٔ اصلی یکسان است.

۱.۲ تعریف

سمافور شبیه یک عدد صحیح است امّا با سه تفاوت:

۱. زمانیکه سمافوری را ایجاد می‌نمایید مقدار اولیه آن را می‌توانید هر عدد صحیحی قرار دهید، امّا پس از آن تنها اعمال مجاز، افزایش و کاهش، آن هم به اندازه یک واحد است و مقدار جاری سمافور را نمی‌توانید بخوانید.

۲. زمانیکه یک نخ، سمافور را کاهش می‌دهد اگر نتیجه مقداری منفی باشد، نخ خودش را مسدود[1] نموده و تا زمانیکه نخ دیگری آن سمافور را افزایش ندهد نمی‌تواند ادامه دهد.

۳. زمانیکه یک نخ مقدار سمافور را افزایش می‌دهد، اگر نخ‌های دیگری در انتظار باشند آنگاه یکی از نخ‌های در حال انتظار، از حالت انسداد خارج می‌شود.

[1] block

وقتی می‌گوییم یک نخ خود را مسدود کرده است منظور این است که به زمان‌بند اعلام می‌نماید که دیگر نمی‌تواند ادامه دهد. تا زمانیکه رخدادی سبب رفع انسداد نخ نگردد زمان‌بند مانع اجرای نخ می‌گردد. در استعارات رایج علم کامپیوتر رفع انسداد اغلب بیدار شدن[۲] گفته می‌شود.

تمام تعریف همین است، امّا این تعریف پی‌آمدهایی در پی دارد که ممکن است بخواهید درباره آن‌ها بیندیشید.

- به طور کلی، نمی‌توان از قبل گفت که کاهش سمافور توسط یک نخ منجر به مسدود شدن آن نخ می‌شود یا نه (در حالت‌های خاص ممکن است بتوانید ثابت کنید که مسدود می‌شود یا خیر).
- پس از اینکه نخی یک سمافور را افزایش داد و نخ دیگری بیدار شد، هر دو نخ به صورت همروند اجرای خود را ادامه می‌دهند. هیچ راهی وجود ندارد که بدانیم کدامیک از این دو نخ، بلافاصله ادامه می‌یابد.
- زمانیکه به یک سمافور سیگنال می‌دهید ضرورتاً نمی‌دانید که آیا نخی در حالت انتظار هست یا نه؟ لذا تعداد نخ‌های رفع انسداد شده ممکن است صفر یا یک باشد.

نهایتاً، ممکن است فکر کنید مقدار سمافور چه معنایی دارد؟ اگر مقدار مثبت باشد نشانگر تعداد نخ‌هایی است که می‌توانند بدون مسدود شدن، آن کاهش را دهند. اگر مقدار منفی باشد نشانگر تعداد نخ‌هایی است که مسدود شده و در حالت انتظار هستند. اگر مقدار صفر باشد به این معنی است که هیچ نخی در حالت انتظار نیست امّا اگر نخی سمافور را کاهش دهد، مسدود خواهد شد.

۲.۲ نحو

در بیشتر محیط‌های برنامه‌نویسی، پیاده‌سازی سمافور به صورت بخشی از زبان برنامه‌نویسی یا سیستم‌عامل موجود است. گاهی اوقات پیاده‌سازی‌های مختلف، اندک توانایی‌های متفاوتی را فراهم آورده و نحو متفاوتی را نیاز دارند.

در این کتاب از یک شبه زبان[۳] ساده به منظور نمایش شیوهٔ عملکرد سمافورها استفاده می‌کنیم. نحوِ ایجاد یک سمافور و مقداردهی اولیه آن در ادامه آمده است.

نحو مقداردهی اولیه سمافور

```
1 fred = Semaphore(1)
```

تابع `Semaphore` سازنده‌ای[۴] است که یک سمافور را ایجاد نموده و آن را بر می‌گرداند. مقدار اولیه سمافور به عنوان یک پارامتر به سازنده ارسال می‌گردد.

[2] waking [3] pseudo-language [4] constructor

اعمال سمافور در محیط‌های مختلف نام‌های گوناگونی دارند. رایج‌ترین آن‌ها عبارت است از:

اعمال سمافور

```
fred.increment()
fred.decrement()
```

و

اعمال سمافور

```
fred.signal()
fred.wait()
```

و

اعمال سمافور

```
fred.V()
fred.P()
```

وجود این همه نام ممکن است تعجب بر انگیز باشد امّا بی‌دلیل نیست. `increment` و `decrement` بیان می‌نمایند که این اعمال چه انجام می‌دهند. `signal` و `wait` شرح می‌دهند که اغلب به چه هدفی به کار می‌روند. و V و P نام‌های پیشنهادی Dijkstra هستند و او به خوبی می‌دانست که یک نام بی‌معنی بهتر از نامی گمراه کننده است[۵].

بقیهٔ اسامی را گمراه کننده می‌دانم زیرا که `increment` و `decrement` هیچ اشاره‌ای به امکان انسداد و رفع انسداد ندارند و سمافورها اغلب به گونه‌ای استفاده می‌شوند که کاری با `signal` و `wait` ندارند.

اگر اصرار به نام‌هایی با معنی دارید، نام‌های زیر را به شما پیشنهاد می‌کنم:

اعمال سمافور

```
fred.increment_and_wake_a_waiting_process_if_any()
fred.decrement_and_block_if_the_result_is_negative()
```

گمان نمی‌کنم کسی به این زودی این اسامی را بپذیرد. فعلاً تا آن زمان، `signal` و `wait` را برای استفاده انتخاب می‌کنم.

۳.۲ چرا سمافورها؟

با نگاه به تعریف سمافورها، اینکه چرا سمافورها مفید هستند اصلاً واضح نیست. درست است که برای حل مسائل همگام‌سازی به سمافورها **نیازی** نداریم، امّا استفاده از آن‌ها مزایایی دارد:

[۵] اگر زبان شما هلندی باشد V و P آنقدرها هم بی‌معنی نیستند.

- سمافورها محدودیت‌هایی تعمّدی تحمیل می‌نمایند که به برنامه‌نویس‌ها کمک می‌کند تا از خطاها بدور باشند.
- راه حل‌هایی که از سمافورها استفاده می‌نمایند اغلب تمیز و سازمان‌یافته هستند به گونه‌ای که اثبات درستی آن‌ها را ساده می‌سازد.
- سمافورها می‌توانند به صورتی کارا در بسیاری از سیستم‌ها پیاده‌سازی شوند، لذا راه حل‌هایی که از سمافورها استفاده نموده‌اند معمولاً قابل حمل و کارا هستند.

فصل ۳

الگوهای همگام سازی پایه

در این فصل تعدادی از مسائل همگام سازی پایه ارائه شده و نشان داده می‌شود چگونه با استفاده از سمافورها آن‌ها را حل کنیم. این مسائل شامل موضوعات مختلفی می‌شود از جمله تسلسل و انحصار متقابل -که قبلاً با آن‌ها آشنا شده ایم-.

۱.۳ علامت‌دهی

ساده‌ترین شکل استفاده از یک سمافور احتمالاً مکانیزم **علامت‌دهی**[1] است، به این معنی که یک نخ، پیامی به نخ دیگر می فرستد تا وقوع رخدادی را اعلام کند.

علامت‌دهی این امکان را فراهم می‌آورد تا مطمئن شویم که یک قطعه کد در یک نخ، حتماً قبل از قطعه کدی دیگر در نخ دیگری اجرا خواهد شد؛ به عبارت دیگر، مسئلهٔ تسلسل را حل می‌کند.

فرض کنید یک سمافور با نام `sem` و مقدار اولیهٔ ۰ داریم، و نخ‌های A و B هر دو به آن دسترسی دارند.

نخ A

```
1 statement a1
2 sem.signal()
```

نخ B

```
1 sem.wait()
2 statement b1
```

کلمهٔ `statement` نشان دهندهٔ یک عبارت دلخواه در برنامه است. برای اینکه مثال مشخص‌تر شود، فرض کنید `a1` یک خط از یک فایل را می‌خواند، و `b1` آن خط را در صفحه نمایش نشان می دهد. سمافور در این برنامه تضمین می‌کند که نخ A عملیات `a1` را، قبل از آنکه نخ B عملیات `b1` را شروع کند، به طور کامل انجام داده

[1] signaling

است.

روش کار به این صورت است: اگر اول نخ B به عبارت `wait` برسد، با مقدار اولیه، یعنی صفر، مواجه می‌شود و مسدود خواهد شد. سپس هر زمان نخ A علامت دهد، نخ B ادامه خواهد داد.

به طور مشابه، اگر اول نخ A به عبارت `signal` برسد، مقدار سمافور افزایش می یابد، و هنگامی که نخ B به `wait` برسد، بدون وقفه ادامه می‌یابد. در هر صورت، ترتیب `a1` و `b1` تضمین می‌شود.

این شیوهٔ استفاده از سمافورها، پایه و اساس نام‌های `signal` و `wait` است، و در این مورد، این اسامی به راحتی به خاطر سپرده می‌شوند. اما متأسفانه، موارد دیگری را خواهیم دید که این اسم‌ها کمتر به ما کمک می‌کنند.

حالا که از اسامی بامعنی صحبت به میان آمد، باید بدانیم که `sem` با مسمّی نیست. اگر امکان داشته باشد، ایدهٔ خوب این است که به یک سمافور نامی دهیم که مشخص کند به چه دلالت دارد. در این مثال نام `a1done` می تواند خوب باشد، چرا که `a1done.signal()` به این معنی است که «علامت بده `a1` انجام شده است»، و `a1done.wait()` به این معنی است که «صبر کن تا اینکه `a1` انجام شود».

۲.۳ Sync.py

تمرین: دربارهٔ استفاده از sync بنویسید، از signal.py شروع کنید.

چرا نخ B به `initComplete` علامت می دهد؟

۳.۳ قرار ملاقات

معمّا: الگوی علامت دهی را طوری تعمیم دهید که بتواند در دو جهت کار کند. نخ A باید منتظر نخ B بماند و بالعکس. به عبارت دیگر اگر کد زیر را داشته باشیم

نخ A

```
1 statement a1
2 statement a2
```

نخ B

```
1 statement b1
2 statement b2
```

می‌خواهیم مطمئن شویم که `a1` پیش از `b2` رخ می‌دهد و نیز `b1` قبل از `a2` اتفاق می‌افتد. هنگام نوشتن راه‌حل خود، نام و مقدار اولیهٔ سمافورها را حتماً مشخص نمایید.

راه حل شما نباید قید و بندهای زیادی داشته باشد. مثلاً، ترتیب `a1` و `b1` برای ما اهمیتی ندارد. در راه‌حل شما، باید امکان هر ترتیبی وجود داشته باشد.

این مسئلهٔ همگام سازی یک نام دارد و آن، قرار ملاقات است. ایدهٔ آن به این صورت است که دو نخ در یک نقطه از اجرا با یکدیگر قرار ملاقات می‌گذارند، و تا زمانی که هر دو نرسیده باشند، دیگری حق ادامه ندارد.

۱.۳.۳ راهنمای قرار ملاقات

اگر خوش شانس باشید می‌توانید به یک راه‌حل برسید، ولی اگر هم نرسیدید، در ادامه یک راهنمایی آمده است. دو سمافور به نام‌های `aArrived` و `bArrived` ایجاد کنید، و به هر دو مقدار اولیه صفر بدهید.

همان طور که از نام‌ها مشخص است، `aArrived` نشان می دهد که آیا نخ A به محل ملاقات رسیده است، و به همین صورت `bArrived` نیز در مورد نخ B می‌باشد.

۲.۳.۳ راه حل قرار ملاقات

راه حلی برای مبنای راهنمایی قبل آمده است:

نخ A

```
1 statement a1
2 aArrived.signal()
3 bArrived.wait()
4 statement a2
```

نخ B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```

هنگام کار بر روی مسأله قبلی، ممکن است کدی مانند زیرا را امتحان کرده باشید:

نخ A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

نخ B

```
1 statement b1
2 bArrived.signal()
3 aArrived.wait()
4 statement b2
```

اگرچه این راه‌حل کار می‌کند امّا احتمالاً بهینگی کمتری دارد زیرا که ممکن است بین A و B یک بار بیش از آن چیزی که لازم است جابجایی داشته باشد.

اگر اول A برسد باید منتظر B بماند. زمانیکه B رسید A را بیدار نموده و ممکن است بلافاصله ادامه یافته و به `wait` برسد و مسدود شود تا اجازه دهد که A به `signal` برسد که پس از آن هر دو نخ بتواند ادامه یابد.

درباره سایر مسیرهای ممکن از طریق این کد بیندیشید و متقاعد شوید که در تمامی حالات هیچ نخی نمی‌تواند ادامه یابد مگر اینکه هر دو رسیده باشند.

۳.۳.۳ بن‌بست #۱

دوباره هنگام کار بر روی مسأله قبلی، ممکن است کدی مانند زیرا را امتحان کرده باشید:

نخ A

```
1 statement a1
2 bArrived.wait()
3 aArrived.signal()
4 statement a2
```

نخ B

```
1 statement b1
2 aArrived.wait()
3 bArrived.signal()
4 statement b2
```

اگر چنین است، امیدوارم خیلی سریع آن را رد کرده باشید، زیرا که این طراحی مشکلی جدی دارد. با فرض اینکه ابتدا A برسد در خط `wait` مسدود می‌شود. زمانیکه B برسد، آن نیز مسدود خواهد شد زیرا که A نمی‌تواند به `aArrived` پیام دهد. در این نقطه، هیچکدام از نخ‌ها ادامه نیافته و هرگز نیز ادامه نمی‌یابند.

این وضعیت بن‌بست[۲] نامیده می‌شود و بوضوح یک راه‌حل ناموفق برای مسالهٔ همگام‌سازی است. در این حالت، خطا واضح است امّا درک امکان رخداد بن‌بست اغلب همیشه به این روشنی نیست. مثال‌های بیشتری را بعدا خواهیم دید.

۴.۳ Mutex

دومین کاربرد رایج سمافورها إعمال انحصار متقابل است. پیش از این یکی از کاربردهای انحصار متقابل در کنترل دسترسی همروند به متغیرهای اشتراکی را دیده‌ایم. میوتکس[۳] تضمین می‌نماید که در هر زمان تنها یک نخ به متغیر اشتراکی دسترسی دارد.

میوتکس شبیه یک توکن[۴] است که از نخی به نخ دیگر منتقل می‌شود و در هر زمان به یک نخ اجازه ادامه فعالیت می‌دهد. برای نمونه، در رمان **سالار مگس‌ها**[۵] یک گروه از بچه‌ها از یک صدف به عنوان میوتکس بهره می‌برند. برای صحبت کردن شما باید صدف را در اختیار داشته باشید. از آنجایی که تنها یک کودک صدف را در اختیار دارد، لذا تنها یک نفر می‌تواند صحبت کند. [۶]

به طور مشابه، به منظور دسترسی یک نخ به متغیری اشتراکی، باید میوتکس را "گرفته" و زمانی که کارش تمام شد آن را "رها" کند. در هر زمان تنها یک نخ می‌تواند میوتکس را در اختیار داشته باشد.

معمّا: سمافورهایی به مثال زیر بیفزایید تا انحصار متقابل متغیر اشتراکی `count` را إعمال نمایند.

نخ A

```
1 count = count + 1
```

نخ B

```
1 count = count + 1
```

[۶] اگرچه این تعبیر در اینجا مفید است ولی می‌تواند گمراه‌کننده نیز باشد، همانطوری که در بخش ۶.۵ خواهید دید.

[2] deadlock [3] mutex [4] token [5] The Lord of the Flies

۱.۴.۳ راهنمای انحصار متقابل

سمافوری با نام `mutex` و مقدار اولیه 1 ایجاد نمایید. مقدار یک به این معنی است که یک نخ می‌تواند ادامه یافته و به متغیر اشتراکی دسترسی داشته باشد؛ مقدار صفر به معنی این است که باید منتظر نخ دیگری بماند تا میوتکس را آزاد نماید.

۲.۴.۳ راه‌حل انحصار متقابل

راه حلی را در ادامه می‌بینیم:

نخ A

```
1 mutex.wait()
2     # critical section
3     count = count + 1
4 mutex.signal()
```

نخ B

```
1 mutex.wait()
2     # critical section
3     count = count + 1
4 mutex.signal()
```

از آنجایی که `mutex` مقدار اولیه 1 را دارد،اولین نخی که در کد خود به `wait` می‌رسد می‌تواند بلافاصله ادامه یابد. البته عمل wait روی سمافور موجب کاهش مقدار آن می‌گردد لذا دومین نخی که به `wait` می‌رسد باید منتظر پیام‌دهی نخ اول بماند.

عملیات بروزرسانی متغیر حاشیه‌دار شده است تا نشان دهد که درون یک میوتکس قرار دارد.

در این مثال، هر دو نخ کد یکسانی را اجرا می‌نمایند. گاهی اوقات این نوع راه‌حل را **متقارن**[۷] می‌نامیم. اگر نخ‌ها کدهای مختلفی را اجرا می‌نمایند این راه‌حل **نامتقارن**[۸] گفته می‌شود. راه حل‌های متقارن اغلب راحت‌تر تعمیم‌داده می‌شوند. در این حالت، راه‌حل میوتکس می‌تواند هر تعداد نخ همروند را بدون نیاز به هیچگونه تغییری مدیریت نماید. تا زمانی که هر نخ پیش از بروزرسانی متغیر، wait و پس از آن نیز signal را فرا می‌خواند هیچ دو نخی به صورت همزمان به متغیر `count` دسترسی نخواهند داشت.

اغلب کدی که به حفاظت نیاز دارد **ناحیه بحرانی**[۹] نامیده می‌شود، زیرا که جلوگیری از دسترسی همزمان، اهمیتی حیاتی دارد.

در استعارات رایج علم کامپیوتر، گاهی اوقات به طرق دیگری درباره میوتکس‌ها صحبت می‌شود. در تعبیری که تاکنون استفاده نمودیم، میوتکس توکنی است که از یک نخ به نخ دیگر انتقال داده می‌شود.

در تعبیر دیگر، از ناحیه بحرانی به عنوان اتاقی یاد می‌شود که در هر زمان تنها یک نخ اجازه دارد داخل آن باشد. در این تعبیر، میوتکس‌ها قفل[۱۰] نامیده می‌شوند و گفته می‌شود یک نخ پیش از ورود، میوتکس را قفل نموده و هنگام خروج آن را باز می‌نماید. گرچه گهگاهی، کاربران تعابیر را خلط نموده و صحبت از گرفتن[۱۱] و رهانمودن[۱۲] یک قفل می‌نمایند که این تعبیر، آن قدر هم با معنی نیست.

هر دو تعبیر، بالقوه مفید و بالقوه گمراه کننده هستند. هنگام کار بر روی مسأله بعدی، هر دو شیوه تفکر را امتحان نموده و ببینید کدام یک شماره به راه‌حل می‌رساند.

---

[7] symmetric [8] asymmetric [9] critical section [10] lock [11] getting [12] releasing

Multiplex ۵.۳

معمّا: راه‌حل قبل را چنان تعمیم دهید که به چند نخ اجازه دهد به صورت همزمان در ناحیه بحرانی اجرا شوند اما یک حدّ بالا روی تعداد نخ‌های همروند اعمال شود. به عبارت دیگر، بیش از n نخ به صورت همزمان در ناحیه بحرانی اجرا نشود.

این الگو یک مالتی‌پلکس[۱۳] نامیده می‌شود. در دنیای واقعی، مسأله مالتی‌پلکس در یک کلوپ شبانهٔ شلوغ رخ می‌دهد زمانی که برای تعداد افرادی که مجاز به حضور در ساختمان در یک زمان هستند یک حداکثر وجود دارد؛ خواه به منظور تامین ایمنی آتش‌سوزی یا به منظور ایجاد یک انحصار.

معمولاً در چنین اماکنی یک مأمور، با نگاه‌داشتن تعداد افرادی که داخل هستند و جلوگیری از ورود افراد جدید زمانی که اتاق به ظرفیت خود برسد محدودیت همزمانی را تضمین می‌کند. سپس، هر زمان که یک فرد خارج شود فرد دیگری اجازه ورود می‌یابد.

تضمین این محدودیت با سمافورها ممکن است سخت به نظر آید امّا تقریباً بدیهی است.

[13] multiplex

۱.۵.۳ راه‌حل مالتی‌پلکس

به منظور اینکه اجرای چند نخ را در ناحیه بحرانی ممکن سازیم، کافی است مقدار اولیه سمافور را n-حداکثر تعداد نخ‌های مجاز- قرار دهیم.

در هر زمان، مقدار سمافور نشانگر تعداد نخ‌هایی است که می‌توانند داخل شوند. اگر این مقدار صفر باشد آنگاه نخ بعدی تا زمانی که یکی از نخ‌های درونی خارج شده و پیام‌دهی نماید مسدود می‌گردد. هنگامی که تمام نخ‌ها از ناحیه بحرانی خارج شوند مقدار سمافور دوباره n می‌شود.

از آنجایی که این راه‌حل متقارن است، به طور قراردادی تنها یک کپی از کد نمایش داده می‌شود امّا شما باید چندین کپی از کد را که به صورت همروند در چندین نخ اجرا می‌شود در نظر بگیرید.

راه‌حل مالتی‌پلکس

```
multiplex.wait()
    critical section
multiplex.signal()
```

اگر ناحیه بحرانی پر شده باشد و بیش از یک نخ سر برسند چه اتفاقی می‌افتد؟ البته چیزی که می‌خواهیم این است که تمامی آن‌ها منتظر بمانند. این راه‌حل دقیقاً همین کار را انجام می‌دهد. هر زمان که یک نخ تازه به صف ملحق شود، سمافور کاهش می‌یابد لذا مقدار سمافور که منفی است نشانگر تعداد نخ‌هایی است که در صف هستند.

زمانی که یک نخ ناحیه بحرانی را ترک می‌کند، به سمافور پیام‌داده و مقدار آن را افزایش می‌دهد که این کار اجازه می‌دهد یکی از نخ‌های در حال انتظار ادامه یابد.

با در نظر گرفتن تعابیر گذشته، در این حالت بهتر است سمافورها به صورت مجموعه‌ای از توکن‌ها دیده شود (تا یک قفل). هر نخ که `wait` را فراخوانی نماید، یکی از توکن‌ها را در اختیار می گیرد؛ زمانی که `signal` را فراخواند آن توکن را رها می‌نماید. فقط نخی که توکنی در اختیار دارد می‌تواند به اتاق وارد شود. زمانی که یک نخ می‌رسد اگر هیچ توکنی موجود نباشد باید تا زمانی که نخ دیگری توکنی را رها نماید، منتظر بماند.

در دنیای واقعی، گاهی اوقات باجه‌های بلیط فروشی سیستمی مشابه را بکار می‌برند. به مشتری‌هایی که در صف هستند توکن‌هایی داده می‌شود. هر توکن به دارنده آن اجازه خرید یک بلیط را می‌دهد.

۶.۳ حصار[۱۴]

دوباره مسأله قرار ملاقات را از بخش ۳.۳ در نظر بگیرید. راه‌حلی که ارائه دادیم یک محدودیت دارد و آن این است که برای بیشتر از دو نخ کار نمی‌کند.

[14] Barrier

معمّا: راه‌حل قرار ملاقات را تعمیم دهید. هر نخ باید کد زیر را اجرا نماید:

کد حصار

```
rendezvous
critical point
```

لازمه همگام‌سازی این است که هیچ نخی `critical point` را اجرا ننماید مگر اینکه تمامی نخ‌ها `rendezvous` را اجرا نموده باشند.

فرض کنید که n نخ وجود دارد و این مقدار در یک متغیر با نام `n` ذخیره شده است و تمامی نخ‌ها به آن دسترسی دارند.

زمانی که $n-1$ نخِ اول می‌رسند باید تا زمان رسیدن nامین نخ، مسدود شوند و پس از آن، همگی می‌توانند ادامه یابند.

۱.۶.۳ راهنمای حصار

برای بسیاری از مسائل این کتاب، با ارائه متغیرهایی که در راه‌حل‌هایم بکار برده‌ام و توضیح قوانین آن‌ها، راهنمایی‌هایی را فراهم خواهم آورد.

راهنمای حصار

```
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

`count` تعداد نخ‌های رسیده را نگاه می‌دارد. `mutex` دسترسی انحصاری به `count` را به گونه‌ای فراهم می‌آورد که نخ‌ها بتوانند به صورت ایمن آن را افزایش دهند.

`barrier` تا زمانی که تمامی نخ‌ها برسند قفل شده است (صفر یا منفی)؛ سپس باید باز شود (یک یا بیشتر).

۲.۶.۳ ناراه‌حل حصار

ابتدا راه‌حلی را ارائه می‌دهیم که به طور کامل درست نمی‌باشد، چرا که بررسی راه‌حل‌های ناصحیح برای فهم اینکه چه چیزی نادرست است مفید است.

ناراه‌حل حصار

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()

critical point
```

از آنجایی که `count` به وسیله یک mutex محافظت می‌شود، تعداد نخ‌هایی که رد می‌شود را می‌شمارد. $n - ۱$ نخ اول زمانی که به حصار می‌رسند منتظر می‌مانند، چرا که حصار در ابتدا قفل است. زمانی که nامین نخ می‌رسد حصار را باز می‌نماید.

معمّا: مشکل این راه‌حل چیست؟

۳.۶.۳ بن‌بست #۲

مشکل راه‌حل قبلی، بن‌بست است. به عنوان مثال، تصور کنید $n = ۵$ و چهار نخ پشت حصار منتظر هستند. مقدار سمافور، منفیِ تعداد نخ‌های در صف است که در اینجا -۴ می‌باشد.

زمانی که پنجمین نخ به حصار پیام می‌دهد، یکی از نخ‌های در حال انتظار اجازه ادامه کار یافته و سمافور به -۳ افزایش می‌یابد.

امّا پس از آن، دیگر هیچ نخی به سمافور پیام نداده و هیچ کدام از سایر نخ‌ها نمی‌تواند از حصار عبور نماید. این دومین مثال از بن‌بست است.

معمّا: آیا این کد همیشه یک بن‌بست تولید می‌نماید؟ آیا می‌توانید یک مسیر اجرایی از طریق این کد بیاید که منجر به بن‌بست نشود؟

معمّا: این مشکل را حل کنید.

۴.۶.۳ راه‌حل حصار

بالاخره، کد راه‌حل صحیح مسأله حصار در ادامه آمده است:

راه حل حصار

```
rendezvous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
```

تنها تغییر، یک signal دیگر پس از انتظار برای حصار است. اکنون، پس از اینکه هر نخ عبور کرد، به سمافور پیام می‌دهد که نخ بعدی می‌تواند عبور کند.

این الگو (یک wait و یک signal بلافاصله پشت سر هم) آنقدر رایج است که یک نام دارد: ترن‌استایل[۱۵]، زیرا که در یک زمان به تنها یک نخ اجازه عبور می‌دهد و می‌تواند قفل گردد تا تمام نخ‌ها را نگه دارد.

در وضعیت آغازین (صفر)، ترن‌استایل قفل است. نخ nام آن را بار نموده و سپس تمام n نخ از آن عبور می‌کنند.

ممکن است خواندن مقدار count بیرون از mutex مخاطره آمیز به نظر رسد؛ اگر چه در اینجا مشکلی وجود ندارد، امّا به طور کلی احتمالاً ایدهٔ خوبی هم نیست. چند صفحه بعد کد تمیزتری ارائه می‌دهیم امّا فعلاً ممکن است بخواهید این سوالات را در نظر بگیرید: پس از nامین نخ، ترن‌استایل در چه وضعیتی است؟ آیا راهی وجود دارد که ممکن باشد حصار بیش از یکبار پیام دهی شود؟

[۱۵] turnstile: تیرکی که معمولاً سه بازوی گردنده دارد و هرکس می‌خواهد از آن بگذرد کوپن خود را در سوراخ آن انداخته و آن را چرخانده وارد می‌شود. معمولاً در ورودی ایستگاه‌های مترو و یا فروشگاه‌ها وجود دارد.

۵.۶.۳ بن‌بست #۳

از آنجایی که در هر زمان تنها یک نخ می‌تواند از میوتکس عبور کند و نیز در هر زمان تنها یک نخ می‌تواند از ترن‌استایل عبور کند، ممکن است قرار دادن ترن‌استایل درون میوتکس منطقی به نظر آید، مانند زیر:

راه حل بدِ حصار

```
rendezvous

mutex.wait()
    count = count + 1
    if count == n: barrier.signal()

    barrier.wait()
    barrier.signal()
mutex.signal()

critical point
```

این ایدهٔ بدی است زیرا که می‌تواند سبب بن‌بست گردد.

تصور کنید اولین نخ وارد `mutex` شده و پس از رسیدن به ترن‌استایل مسدود می‌گردد. از آنجایی که `mutex` قفل شده است، هیچ نخ دیگری نمی‌تواند وارد گردد، لذا شرط `condition==n` هرگز درست نبوده و هیچ نخی ترن‌استایل را باز نخواهد کرد.

در اینجا، بن‌بست کاملاً واضح است، امّا یک منشأ رایجِ بن‌بست‌ها را نشان می‌دهد: مسدود شدن روی یک سمافور در حالیکه یک `mutex` در اختیار دارد.

۷.۳ حصار با قابلیت استفاده مجدد

اغلب، مجموعهٔ نخ‌های همکار، یک سری از گام‌های یک حلقه را اجرا نموده و پس از هر گام نزدِ یک حصار همگام‌سازی می‌شوند. برای این کاربرد، به یک حصار با قابلیت استفاده مجدد نیاز داریم که پس از عبور تمامی نخ‌ها از آن، خود را قفل نماید.

معمّا: راه‌حل حصار را چنان بازنویسی کنید که پس از عبور تمام نخ‌ها، ترن‌استایل دوباره قفل شود.

۱.۷.۳ ناراه‌حل حصار با قابلیت استفاده مجدد #۱

دوباره با یک تلاش ساده برای رسیدن به راه‌حل، شروع نموده و به تدریج آن را بهبود می‌دهیم:

ناراه‌حل حصار با قابلیت استفاده مجدد

```

rendezvous

mutex.wait()
    count += 1
mutex.signal()

if count == n: turnstile.signal()

turnstile.wait()
turnstile.signal()

critical point

mutex.wait()
    count -= 1
mutex.signal()

if count == 0: turnstile.wait()
```

توجه کنید که کدِ پس از ترن‌استایل بسیار مشابه کدِ قبل از آن است. دوباره لازم است که دسترسی به متغیر اشتراکی `count` را با استفاده از میوتکس محافظت کنیم. با این‌حال متاسفانه این کد به طور کامل صحیح نیست.

معمّا: مشکل چیست؟

۲.۷.۳ مسأله حصار با قابلیت استفاده مجدد #۱

مشکلی در خط ۸ کد قبل وجود دارد.

اگر $n - ۱$امین نخ در این نقطه با یک وقفه مواجه شود، و سپس nامین نخ وارد میوتکس شود، هر دو نخ `count=n` را صحیح یافته و هر دو به ترن‌استایل پیام خواهند داد. در حقیقت حتی این امکان وجود دارد که تمام نخ‌ها به ترن‌استایل پیام دهند.

به طور مشابه در خط ۱۹ ممکن است چندین نخ `wait` را اجرا نماید که منجر به بن‌بست خواهد شد.

معمّا: مشکل را حل نمایید.

۳.۷.۳ ناراه‌حل حصار با قابلیت استفاده مجدد #۲

تلاش ذیل، خطای قبل را حل نموده ولی یک مشکل ظریف باقی می‌ماند.

ناراه‌حل حصار با قابلیت استفاده مجدد

```
rendezvous

mutex.wait()
    count += 1
    if count == n: turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

critical point

mutex.wait()
    count -= 1
    if count == 0: turnstile.wait()
mutex.signal()
```

در هر دو حالت بررسی درون میوتکس صورت می‌گیرد لذا یک نخ نمی‌تواند بعد از تغییر شمارنده و قبل از بررسی آن با وقفه مواجه شود.

متاسفانه این کد هنوز صحیح نیست. فراموش نکنید که این کد حصار درون یک حلقه قرار دارد. لذا پس از اجرای آخرین خط هر نخ به rendezvous باز می‌گردد.

معمّا: مشکل را بیابید و آن را رفع نمایید.

۴.۷.۳ راهنمای حصار با قابلیت استفاده مجدد

همانطوری که نوشته شده است، این کد به نخی که زودتر از ناحیه بحرانی خود خارج می‌شود اجازه عبور از میوتکس دوم را داده و سپس در حلقه کد خود از میوتکس اول و ترن‌استایل عبور می‌نماید، که در عمل همین نکته سبب می‌شود آن نخ، یک دور جلوتر از دیگر نخ‌ها قرار گیرد.

برای رفع این مسأله می‌توانیم از دو ترن‌استایل استفاده نماییم.

راهنمای حصار با قابلیت استفاده مجدد

```
1 turnstile = Semaphore(0)
2 turnstile2 = Semaphore(1)
3 mutex = Semaphore(1)
```

در ابتدا ترن‌استایل اوّلی قفل و دوّمی باز است. زمانی که تمامی نخ‌ها به ترن‌استایل اوّل برسند، آن دوّمی را قفل و اوّلی را باز می‌نماییم. آن هنگام که تمامی نخ‌ها به ترن‌استایل دوّم برسند اوّلی را دوباره قفل کرده -که این کار از بازگشت نخ‌ها به ابتدای حلقه جلوگیری می‌نماید- و سپس دوّمی را باز می‌نماییم.

۵.۷.۳ راه‌حل حصار با قابلیت استفاده‌مجدد

راهنمای حصار با قابلیت استفادهٔ مجدد

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()       # lock the second
        turnstile.signal()      # unlock the first
mutex.signal()

turnstile.wait()                # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()         # lock the first
        turnstile2.signal()      # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
turnstile2.signal()
```

گاهی اوقات به این راه حل، **حصار دو-فاز** گفته می‌شود زیرا که تمامی نخ‌ها را دوبار مجبور به انتظار می‌کند: یکبار برای اینکه تمام نخ‌ها برسند و بار دیگر برای آنکه تمامی نخ‌ها ناحیه بحرانی را اجرا نمایند.

متاسفانه، این راه‌حل از نابدیهی‌ترین انواع کدهای همگام‌سازی است؛ زیرا که اطمینان یافتن از صحت این راه‌حل سخت است. اغلب یافتن اینکه یک مسیر خاص از طریق برنامه سبب خطایی گردد، آنچنان واضح نیست.

بدتر اینکه، بررسی پیاده‌سازی یک راه‌حل آنچنان کمکی نمی‌کند. خطا ممکن است خیلی به ندرت اتفاق افتد، زیرا مسیر بخصوصی که سبب خطا می‌گردد ممکن است نیازمند ترکیبی فوق‌العاده بی‌بدیل از شرایط باشد. تولید و اشکال‌زدایی چنین خطاهایی با روش‌های مرسوم تقریبا ناممکن است.

تنها جایگزین، بررسی دقیق کد و "اثبات" صحت آن است. "اثبات" را بین گیومه قرار دادیم زیرا که ضرورتاً منظور این نیست که یک اثبات رسمی برای آن بنویسید (اگرچه متعصب‌هایی وجود دارند که به چنین حماقتی ترغیب می‌نمایند).

نوع اثبات مد نظر، بسیار غیررسمی است. می‌توانیم از مزیت ساختار کد و اصطلاحاتی که توسعه داده‌ایم برای اثبات و سپس نشان دادن یک تعدادی از ادعاهای سطح متوسط دربارهٔ برنامه استفاده کنیم. برای نمونه:

۱. فقط nامین نخ می‌تواند ترن‌استایل‌ها را قفل یا باز نماید.

۲. قبل از اینکه یک نخ بتواند اولین ترن‌استایل را باز نماید، باید دوّمی را بسته باشد و بالعکس؛ بنابراین برای یک نخ، عبور از دیگر نخ‌ها به اندازهٔ بیش از یک ترن‌استایل ناممکن است.

با یافتن انواع صحیح عبارات به منظور اعلام و اثبات، گاهی اوقات می‌توانید یک راه موجز برای متقاعد کردن خودتان (یا یک همکار شکاک) بیابید که کدتان ضدّ گلوگه است.

۶.۷.۳ ترن‌استایل از پیش بارگذاری‌شده

یک چیز جالب در رابطه با ترن‌استایل این است که یک جزء همه منظوره است که می‌توانید آن را در راه حل‌های مختلفی بکار برید. امّا یک اشکال آن این است که نخ‌ها را مجبور می‌نماید که به صورت ترتیبی عبور نمایند و این ممکن است سبب تعویض بسترهای[۱۶] بیشتر از آنچه نیاز است گردد.

در مسالهٔ حصار با قابلیت استفادهٔ مجدد، اگر نخی که ترن‌استایل را از حالت قفل خارج می‌نماید آن را با تعداد سیگنال کافی به منظور عبور نخ‌های لازم، از پیش بارگذاری نماید، راه‌حل ساده‌تر می‌شود[۱۷].

نحو مورد استفاده در اینجا فرض می‌کند که `signal` می‌تواند پارامتری دریافت دارد که تعداد سیگنال‌ها را مشخص می‌نماید. این، یک ویژگی استاندارد نیست، امّا پیاده‌سازی آن با یک حلقه، آسان خواهد بود. تنها چیزی که باید در ذهن داشت این است که سیگنال‌های چندگانه اتمی نیستند؛ بدین معنی که نخ علامت دهنده ممکن است در میانه حلقه با وقفه روبرو شود. لکن در اینجا، این مسأله مشکلی محسوب نمی‌شود.

راه حل حصار با قابلیت استفادهٔ مجدد

```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile.signal(n)      # unlock the first
mutex.signal()

turnstile.wait()                 # first turnstile

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile2.signal(n)     # unlock the second
mutex.signal()

turnstile2.wait()                # second turnstile
```

زمانیکه nامین نخ می‌رسد، ترن‌استایل اوّل را با یک سیگنال بازای هر نخ از پیش بارگذاری می‌نماید. زمانیکه nامین نخ از ترن‌استایل عبور می‌نماید "آخرین توکن را در اختیار گرفته" و ترن‌استایل را دوباره قفل می‌نماید.

اتفاق مشابهی در ترن‌استایل دوّم رخ می‌دهد، یعنی آن زمان که آخرین نخ از mutex عبور نمود آن را باز می‌نماید.

[۱۷] با تشکر از Matt Tesch بابت این راه‌حل!

[16] context switch

۷.۷.۳ اشیاء حصار

طبیعی است که یک حصار را درون یک شیء محصور[۱۸] نماییم. برای تعریف یک کلاس در اینجا، نحو پایتون را بکار می‌بریم.

کلاس Barrier

```
class Barrier:
    def __init__(self, n):
        self.n = n
        self.count = 0
        self.mutex = Semaphore(1)
        self.turnstile = Semaphore(0)
        self.turnstile2 = Semaphore(0)

    def phase1(self):
        self.mutex.wait()
            self.count += 1
            if self.count == self.n:
                self.turnstile.signal(self.n)
        self.mutex.signal()
        self.turnstile.wait()

    def phase2(self):
        self.mutex.wait()
            self.count -= 1
            if self.count == 0:
                self.turnstile2.signal(self.n)
        self.mutex.signal()
        self.turnstile2.wait()

    def wait(self):
        self.phase1()
        self.phase2()
```

زمانیکه یک شیء Barrier ایجاد می‌گردد متد __init__ اجرا می‌شود و متغیرهای نمونه را مقداردهی اولیه می‌نماید. پارامتر n تعداد نخ‌هایی است که باید پیش از اینکه حصار باز شود wait را فراخوانند.

متغیر self به شیئی اشاره دارد که متدْ روی آن عمل می‌نماید. از آنجایی که هر شیء حصار، میوتکس و ترن‌استایل‌های خود را دارد، self.mutex به میوتکس شیء جاری اشاره می‌نماید.

در اینجا مثالی مشاهده می‌نمایید که یک شیء Barrier را ایجاد نموده و روی آن منتظر می‌ماند:

واسط Barrier

```
barrier = Barrier(n)        # initialize a new barrier
```

[18] encapsulate

```
barrier.wait()                # wait at a barrier
```

کدی که حصار را بکار می‌برد می‌تواند phase1 و phase2 را اگر چیزی وجود داشته باشد که می‌بایست در آن بین اجرا شود، به صورت مجزا فراخواند.

۸.۳ صف

سمافورها می توانند برای نمایاندن یک صف نیز استفاده شوند. در این حالت، مقدار اولیه 0 است و معمولاً کد به گونه‌ای نوشته می‌شود که سیگنال‌دهی ممکن نباشد مگر اینکه یک نخ در حال انتظار وجود داشته باشد؛ لذا مقدار سمافور هیچگاه مثبت نیست.

به عنوان مثال، تصور کنید که نخ‌ها، نمایانگر رقصنده‌هایی باشند و دو نوع رقاص -جلودار و دنباله‌رو- در دو صف، پیش از ورود به اتاق رقص منتظر هستند. هنگامی که که یک جلودار می‌رسد، بررسی می‌کند که آیا هیچ دنباله‌روئی منتظر هست یا خیر. اگر چنین باشد، هر دو می‌توانند داخل شوند در غیر اینصورت می بایست منتظر بماند.

به طور مشابه، زمانیکه یک دنباله‌رو می‌رسد، چک می‌کند که آیا جلوداری وجود دارد یا خیر و بر طبق آن، وارد شده یا صبر می‌نماید.

معمّا: کدی بنویسید برای جلودارها و دنباله‌روها که این شرایط در آن صدق کند.

۱.۸.۳ راهنمای صف

متغیرهایی که در راه‌حل خود بکار برده‌ام، در اینجا آمده است:

راهنمای صف

```
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
```

leaderQueue صف حاوی جلودارهای منتظر و followerQueue صف حاوی دنباله‌روهای منتظر می‌باشد.

۲.۸.۳ راه‌حل صف

کد مربوط به جلودارها در ادامه آمده است:

راه‌حل صف (جلودارها)

```
followerQueue.signal()
leaderQueue.wait()
dance()
```

و در این قسمت کد مربوط به دنباله‌روها را مشاهده می‌نمایید:

راه‌حل صف (دنباله‌روها)

```
leaderQueue.signal()
followerQueue.wait()
dance()
```

این راه‌حل به همان اندازه که نشان می‌دهد ساده است؛ زیرا که تنها یک قرار ملاقات است. هر جلودار دقیقاً به یک دنباله‌رو سیگنال می‌دهد و هر دنباله‌رو به یک جلودار سیگنال می‌دهد، لذا تضمین می‌کند که جلودارها و دنباله‌روها تنها به صورت جفت جفت اجازه اجرا داشته باشند. امّا اینکه آیا آن‌ها واقعاً به صورت جفت جفت وارد عمل می‌شوند یا خیر واضح نیست. تجمّع هر تعدادی از نخ‌ها قبل از اجرای `dance` ممکن است، لذا هر تعداد از جلودارها می‌توانند قبل از دنباله‌روها `dance` را اجرا نمایند. بسته به معنای `dance` این رفتار ممکن است مشکل‌ساز باشد و یا نباشد.

برای اینکه کمی جالب‌تر شود بگذارید محدودیتی بیفزاییم که در آن، هر جلودار تنها با یک دنباله‌رو به صورت همزمان بتواند `dance` را فراخواند و بالعکس. به عبارت دیگر، باید با کسی برقصید که شما را آورده است.[۱۹]

معمّا: یک راه‌حل برای این مسأله «صف انحصاری» ارائه کنید.

[۱۹] "Dance with the one that brought you": متن آهنگ اجرا شده توسط Shania Twain

۳.۸.۳ راهنمای صف انحصاری

متغیرهایی را که در راه‌حل خود بکار برده‌ام در ادامه مشاهده می‌نمایید:

راهنمای صف

```
leaders = followers = 0
mutex = Semaphore(1)
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
rendezvous = Semaphore(0)
```

leaders و followers شمارنده‌هایی هستند که تعداد هر یک از دو نوع رقصنده‌های در حال انتظار را در خود نگه می دارند. میوتکس دسترسی انحصاری به شمارنده‌ها را ضمانت می‌کند.

leaderQueue و followerQueue صف‌هایی هستند که رقصنده‌ها در آن منتظر می‌مانند. از rendezvous به منطور بررسی پایان کار هر دو نخ استفاده می شود.

۴.۸.۳ راه حل صف انحصاری

قطعه کد مربوط به جلودارها:

راه حل صف (جلودارها)

```
mutex.wait()
if followers > 0:
    followers--
    followerQueue.signal()
else:
    leaders++
    mutex.signal()
    leaderQueue.wait()

dance()
rendezvous.wait()
mutex.signal()
```

زمانی که یک جلودار می‌رسد، میوتکسی را می‌گیرد که از leaders و followers محافظت می‌کند. اگر یک دنباله‌رو منتظر باشد، جلودار مقدار followers را کاهش می‌دهد، به یک جلودار سیگنال می‌دهد و سپس dance را فرا می‌خواند، تمام این کارها قبل از آزادسازی mutex انجام می‌شود. این عمل تضمین می‌کند که تنها یک نخ دنباله‌رو وجود دارد که dance را به طور همزمان (با جلودار) اجرا نماید.

اگر هیچ دنباله‌روی در حالت انتظار نباشد، جلودار باید میوتکس را قبل از انتظار روی leaderQueue رها کند.

کدمربوط به دنباله‌روها نیز مشابه است:

راه حل صف (دنباله‌روها)

```
mutex.wait()
if leaders > 0:
    leaders--
    leaderQueue.signal()
else:
    followers++
    mutex.signal()
    followerQueue.wait()

dance()
rendezvous.signal()
```

زمانی که یک دنباله‌رو می‌رسد، وجود یک جلودار در حال انتظار را بررسی می‌نماید. اگر یکی وجود داشته باشد، دنباله‌رو مقدار leaders را کاهش می‌دهد، به جلودار سیگنال می‌دهد، و dance را اجرا می‌کند، تمامی این کارها بدون آزادسازی mutex انجام می‌شود. در حقیقت، در این مورد دنباله‌رو **هرگز** mutex را رها نمی‌کند؛

جلودار این کار را انجام می‌دهد. لازم نیست نخی که میوتکس را دارد رد گیری کنیم زیرا که می‌دانیم یکی از آن‌ها میوتکس را دارد و هر کدام از آن دو نخ می‌تواند آن را آزاد کند. در راه‌حل من همیشه جلودار است که این کار را انجام می‌دهد.

زمانی که یک سمافور به عنوان یک صف استفاده می‌شود، [۲۰] به نظرم بهتر است که "انتظار" را "انتظار برای این صف" بخوانیم و "سیگنال" را "بگذار شخصی از این صف برود".

در این کد، ما هرگز به یک صف سیگنال نمی‌دهیم، مگر اینکه کسی در حال انتظار باشد، در نتیجه مقدار سمافورهای صف به ندرت مثبت می‌باشد. اگرچه این امکان نیز وجود دارد. ببینید آیا می‌توانید بفهمید چنین چیزی چگونه ممکن است.

[۲۰] سمافوری که به عنوان یک صف بکار می‌رود بسیار شبیه به یک متغیر شرطی است. تفاوت اصلی در این است که نخ‌ها باید پیش از انتظار میوتکس را به طور صریح رها نموده و پس از آن به صورت صریح، آن میوتکس را دوباره در اختیار گیرند (البته اگر آن را نیاز داشته باشند).

فصل ۴

مسائل همگام‌سازی کلاسیک

در این فصل مسائل کلاسیک همگام‌سازی را که تقریباً در هر کتاب درسی سیستم‌عامل وجود دارد بررسی می‌نماییم. آن‌ها معمولاً در غالب مسائل جهان واقع ارائه می‌گردد، لذا بیان مسأله واضح است و بنابراین دانشجویان می‌توانند شهود خود را بکار بندند.

گرچه در اکثر موارد، این مسائل در دنیای واقعی رخ نمی‌دهد یا اگر هم رخ دهد راه حل‌های دنیای واقع چندان شبیه کد همگام‌سازی که با آن کار می‌کنیم نیست.

دلیل علاقه‌مندی ما به اینگونه مسائل این است که شبیه مسائل رایجی هستند که سیستم‌عامل‌ها (و برخی برنامه‌ها) نیاز به حل آن‌ها دارند. برای هر مسأله کلاسیک، یک فرمول‌بندی کلاسیک ارائه می‌دهیم و همچنین شباهت آن را به مسأله متناظر در سیستم‌عامل بیان می‌نماییم.

۱.۴ مسئله تولیدکننده-مصرف‌کننده

در برنامه‌های چندنخی اغلب یک تقسیم‌بندی از کار، بین نخ‌ها وجود دارد. در یک الگوی رایج، برخی نخ‌ها تولیدکننده و برخی مصرف‌کننده هستند. تولیدکننده‌ها عناصری از برخی نوع‌ها را ایجاد نموده و آن‌ها را به یک ساختمان داده می‌افزایند؛ مصرف‌کننده‌ها عناصر را حذف نموده و پردازش می‌کنند.

برنامه‌های رویداد-محور مثال‌های خوبی هستند. یک "رویداد"[1]، رخدادی است که به پاسخ برنامه نیاز دارد: کاربر کلیدی را فشار می‌دهد یا ماوس را جابجا می‌کند، یک بلوک داده از دیسک می‌رسد، یک بسته از شبکه می‌رسد، یک عملِ درحال انتظار تکمیل می‌شود.

[1] event

هر زمان که یک رویداد رخ می‌دهد، یک نخ تولیدکننده یک شیئ رویداد را ایجاد نموده و آن را به بافر رویداد می‌افزاید. به طور همزمان، نخ مصرف‌کننده، رویدادها را از بافر خارج کرده و آن‌ها را پردازش می‌کند. در این حالت، مصرف‌کننده ها "مدیریت‌کنندهٔ رویداد"[۲] نامیده می‌شوند.

چندین محدودیت همگام‌سازی وجود دارد که می‌بایست آن‌ها را اعمال کرد تا سیستم به درستی کار کند.

- زمانی که یک عنصر به بافر افزوده شده یا از آن حذف می‌گردد، بافر در وضعیتی ناپایدار است. بنابراین نخ‌ها باید دسترسی انحصاری به بافر داشته باشند.
- اگر یک نخ مصرف‌کننده در زمانی که بافر خالی است سر برسد، تا زمانی که یک تولید کننده عنصر جدیدی را بیفزاید مسدود می‌گردد.

فرض کنید که تولیدکننده‌ها عملیات زیر را بارها و بارها انجام می‌دهند:

کد پایهٔ تولیدکننده

```
1 event = waitForEvent()
2 buffer.add(event)
```

همچنین فرض کنید مصرف کننده‌ها عملیات زیر را اجرا می‌کنند:

کد پایهٔ مصرف‌کننده

```
1 event = buffer.get()
2 event.process()
```

همانطوری که در بالا مشخص شده، دسترسی به بافر باید انحصاری باشد، اما `waitForEvent` و `event.process` می‌توانند به طور همزمان اجرا شوند.

معمّا: دستورات همگام‌سازی را به کد تولیدکننده و مصرف‌کننده بیفزایید تا محدودیت‌های همگام‌سازی اعمال شود.

[2] event handler

۱.۱.۴ راهنمایی تولیدکننده-مصرف‌کننده

در اینجا متغیرهایی آمده است که ممکن است بکار بندید:

مقداردهی اولیه تولیدکننده-مصرف‌کننده

```
1 mutex = Semaphore(1)
2 items = Semaphore(0)
3 local event
```

mutex دسترسی انحصاری را به بافر فراهم می‌آورد. هنگامی که items مثبت است، تعداد عناصر موجود در بافر را نشان می‌دهد. زمانی که منفی است، نشان‌دهندهٔ تعداد نخ‌های مصرف‌کنندهٔ در صف می‌باشد.

در اینجا event یک **متغیر محلی**[۳] است بدین معنی که هر نخ نسخهٔ خود را دارد. تاکنون فرض کرده‌ایم که تمام نخ‌ها به تمام متغیرها دسترسی دارند، امّا گاهی اوقات بهتر است که به هر نخ یک متغیر الحاق شود.

راه‌های مختلفی برای پیاده‌سازی این نکته در محیط‌های گوناگون وجود دارد:

- اگر هر نخ پشتهٔ زمان اجرای خودش را داشته باشد، آنگاه هر متغیر تخصیص داده شده در پشته، مختصّ به نخ[۴] است.
- اگر نخ‌ها به صورت اشیائی نشان داده شوند، می‌توانیم به هر شیئ نخ یک خصیصه اضافه نماییم.
- اگر نخ ها IDهای یکتا داشته باشند، می‌توانیم ID را به عنوان اندیس یک آرایه یا جدول درهم سازی به کار بریم، و داده‌های هر نخ را در آن ذخیره کنیم.

در بیشتر برنامه‌ها، اغلب متغیرها محلی هستند مگر اینکه به صورت دیگری اعلان شوند، اما در این کتاب بیشتر متغیرها اشتراکی‌اند، لذا پیش فرض متغیرها اشتراکی هستند مگر اینکه به صراحت به صورت local اعلان شوند.

[3] local variable [4] thread-specific

۲.۱.۴ راه‌حل تولیدکننده-مصرف‌کننده

کد تولیدکننده راه‌حل من در ادامه آمده است.

راه‌حل تولیدکننده

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
    items.signal()
mutex.signal()
```

تولیدکننده تا زمان دریافت یک رویداد، نباید دسترسی انحصاری به بافر داشته باشد. چندین نخ به صورت همزمان می‌توانند `waitForEvent` را اجرا نمایند.

سمافور `items` تعداد عناصر موجود در بافر را در خود نگاه می‌دارد. هر زمان که تولیدکننده یک عنصر بیفزاید، با سیگنال دادن به `items` یک واحد آن را افزایش می‌دهد.

کد مصرف‌کننده نیز مشابه است.

راه‌حل مصرف‌کننده

```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
event.process()
```

دوباره، عملیات بافر توسط یک میوتکس حفاظت می‌شود امّا پیش از اینکه مصرف‌کننده به آن برسد باید `items` را کاهش دهد. اگر `items` مقدار صفر یا منفی داشته باشد، مصرف‌کننده تا زمانی که یک تولیدکننده سیگنال دهد مسدود می‌گردد.

اگرچه این راه‌حل درست است، این امکان وجود دارد که کارایی آن را قدری بهبود بخشید. تصور کنید زمانیکه یک تولیدکننده به `items` سیگنال می‌دهد حداقل یک مصرف‌کننده در صف قرار دارد. اگر زمان‌بند به مصرف‌کننده اجازه اجرا را بدهد، پس از آن چه اتفاقی می‌افتد؟ بلافاصله روی میوتکس که (هنوز) در اختیار تولیدکننده است مسدود می‌شود.

مسدودسازی و رفع انسداد، اعمال نسبتاً پر هزینه‌ای هستند: انجام غیرضروری آن‌ها می‌تواند به کارایی برنامه آسیب زند. لذا احتمالاً بازنویسی کد تولیدکننده به صورت زیر بهتر باشد.

راه‌حل بهبود یافتهٔ تولیدکننده

```
event = waitForEvent()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

اکنون تا زمانیکه بدانیم یک مصرف‌کننده می‌تواند به کار خود ادامه دهد برای رفع انسداد آن خود را بزحمت نمی‌اندازیم (بجز موردی نادر که تولیدکننده دیگری در گرفتن میوتکس، از مصرف‌کننده پیشی می‌گیرد).

یک چیز دیگری در رابطه با این راه‌حل وجود دارد که ممکن است یک فرد دقیق را آزار دهد. در بخش راهنمایی ادعا کردیم که سمافور `items` تعداد عناصر موجود در صف را در خود نگاه می‌دارد. امّا با نگاه به کد مصرف‌کننده، در می‌یابیم این امکان وجود دارد که چندین مصرف‌کننده بتوانند پیش از اینکه میوتکس را بگیرند و یک عنصر را از بافر بردارند، `items` را کاهش دهند. حداقل برای یک زمان کوتاه، `items` می‌تواند نادقیق باشد.

ممکن است بخواهیم این مشکل را بوسیله چک کردن بافر درون میوتکس، حل کنیم:

راه‌حل معیوب مصرف‌کننده

```
1 mutex.wait()
2     items.wait()
3     event = buffer.get()
4 mutex.signal()
5 event.process()
```

این ایدهٔ بدی است.

معمّا: چرا؟

۳.۱.۴ بن‌بست #۴

اگر مصرف‌کننده کد زیر را اجرا نماید

راه‌حل معیوب مصرف‌کننده

```
mutex.wait()
    items.wait()
    event = buffer.get()
mutex.signal()

event.process()
```

این کد می‌تواند سبب بروز بن‌بست گردد. تصور نمایید که بافر خالی است. یک مصرف‌کننده سر رسیده و میوتکس را گرفته و سپس روی `items` مسدود می‌گردد. زمانیکه تولیدکننده سر می‌رسد، روی `mutex` مسدود شده و سیستم وارد یک توقف تمام عیار گردد.

این مشکل، یک خطای رایج در کد همگام‌سازی است: هر زمان که برای یک سمافور منتظر می‌مانید در حالیکه یک میوتکس را در دست دارید، خطر بن‌بست وجود دارد. زمانیکه یک راه‌حل را نسبت به مسأله همگام‌سازی بررسی می‌نمایید، باید این نوع از بن‌بست را مدّ نظر داشته باشید .

۴.۱.۴ تولیدکننده-مصرف‌کننده با یک بافر متناهی

در این مثالی که بالا توضیح داده شد (نخ‌های کنترل‌کننده رویداد)، بافر اشتراکی معمولاً نامتناهی است (به طور دقیق‌تر، با منابع سیستم نظیر حافظه فیزیکی و فضای مبادله[5] محدود شده است).

گرچه در هستهٔ سیستم‌عامل، محدودیت‌هایی روی فضای موجود نیز وجود دارد. بافرهایی نظیر درخواست دیسک یا بسته‌های شبکه معمولاً سایز ثابتی دارند. در این گونه موارد، یک محدودیت همگام‌سازی دیگر نیز داریم:

- اگر زمانیکه بافر پر است یک تولیدکننده برسد، تا زمانیکه یک مصرف‌کننده عنصری را از بافر حذف کند مسدود می‌گردد.

فرض کنید که اندازه بافر را می‌دانیم. آن را `bufferSize` بنامید. از آنجاییکه سمافوری داریم که تعداد عناصر را نگاه می‌دارد، وسوسه می‌شویم کدی به صورت زیر بنویسیم.

راه‌حل معیوب بافر محدود

```
if items >= bufferSize:
    block()
```

[5] swap space

امّا نمی‌توانیم چنین کنیم. به یاد آورید مقدار جاری یک سمافور را نمی‌توان بررسی نمود؛ `wait` و `signal` تنها اعمال ممکن ما هستند.

معمّا: کد تولیدکننده–مصرف‌کننده‌ای بنویسید که محدودیت بافر–متناهی را مدیریت کند.

۵.۱.۴ راهنمایی بافر محدود تولیدکننده-مصرف‌کننده

سمافور دیگری را به منظور نگهداری تعداد فضای موجود در بافر بیفزایید.

مقداردهی اولیهٔ بافر محدود تولیدکننده-مصرف‌کننده

```
1 mutex = Semaphore(1)
2 items = Semaphore(0)
3 spaces = Semaphore(buffer.size())
```

زمانیکه مصرف‌کننده عنصری را حذف می‌کند باید به spaces سیگنال دهد. زمانیکه تولیدکننده می‌رسد باید spaces را کاهش دهد، که در این نقطه ممکن است تا آن هنگامیکه مصرف‌کنندهٔ بعدی سیگنال دهد مسدود گردد.

۶.۱.۴ راه‌حل بافر محدود تولیدکننده-مصرف‌کننده

و امّا راه حل.

راه‌حل بافر محدود مصرف‌کننده

```
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

کد تولیدکننده تا حدودی متقارن است:

راه‌حل بافر محدود تولیدکننده

```
event = waitForEvent()

spaces.wait()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```

به منظور اجتناب از بن‌بست، تولیدکننده‌ها و مصرف‌کننده‌ها پیش از گرفتن میوتکس، موجود بودنش را بررسی می‌نمایند. برای بهترین کارایی، آن‌ها میوتکس را پیش از سیگنال‌دهی آزاد می‌نمایند.

۲.۴ مسأله خوانندگان-نویسندگان

مسألهٔ کلاسیک بعدی، که مسألهٔ خواننده-نویسنده گفته می‌شود، مرتبط است با هر راه حلی که یک ساختمان‌داده، پایگاه‌داده یا سیستم فایل خوانده شده و توسط نخ‌های همروند ویرایش می‌گردند. زمانی‌که ساختمان‌داده نوشته شده یا ویرایش می‌گردد ضروری است که دیگر نخ‌ها از خواندن منع شده تا از دخالت در ویرایشی که در حال انجام است جلوگیری نموده و داده‌هایی نامعتبر یا ناپایدار خوانده نشود.

همانند مسألهٔ تولیدکننده-مصرف‌کننده، راه‌حل مسأله، متقارن است. خوانندگان و نویسندگان قبل از ورود به ناحیه بحرانی کد متفاوتی را اجرا می‌نمایند. محدودیت‌های همگام‌سازی عبارتند از:

۱. هر تعداد خواننده می‌تواند به صورت همزمان در ناحیه بحرانی باشد.

۲. نویسندگان باید دسترسی انحصاری به ناحیه بحرانی داشته باشند.

به عبارت دیگر، یک نویسنده تا زمانیکه نخ دیگری (اعم از خواننده یا نویسنده) در ناحیهٔ بحرانی وجود دارد، نمی‌تواند وارد شود و زمانیکه یک نویسنده وجود داشته باشد هیچ نخ دیگری نمی‌تواند وارد شود.

الگوی انحصاری اینجا ممکن است **انحصار متقابل دسته‌ای**[۶] نامیده شود. یک نخ در ناحیه بحرانی ضرورتاً دیگر نخ‌ها را بیرون نمی‌کند، امّا وجود یک دستهٔ خاص در ناحیه بحرانی دسته‌های دیگر را مانع می‌گردد.

معمّا: سمافورهایی برای اعمال این محدودیت‌ها به کار برید به گونه‌ای که به خوانندگان و نویسندگان اجازه دسترسی به ساختمان‌داده را بدهد و از وقوع بن‌بست جلوگیری نماید.

[6] categorical mutual exclusion

۱.۲.۴ راهنمایی خوانندگان-نویسندگان

مجموعه متغیرهایی که برای حل مسأله کافی هستند در ادامه آمده است.

مقداردهی اولیه خوانندگان-نویسندگان

```
1 int readers = 0
2 mutex = Semaphore(1)
3 roomEmpty = Semaphore(1)
```

شمارندهٔ readers تعداد خوانندگانی که در اتاق هستند را نگاه می‌دارد. mutex از شمارندهٔ اشتراکی محافظت می‌نماید.

اگر هیچ نخی (اعم از خواننده یا نویسنده) در ناحیه بحرانی نباشد roomEmpty مقدار 1 و الّا مقدار 0 دارد. نام roomEmpty بر اساس قاعده‌ای که قرار گذاشتیم انتخاب گردیده تا نام سمافور نشانگر شرط مورد نظر باشد. طبق این قرارداد، معمولاً "انتظار" به معنای "انتظار برای برقرار بودن شرط" و "سیگنال" به معنای "علامتی ده که شرط برقرار است" می‌باشد.

۲.۲.۴ راه‌حل خوانندگان-نویسندگان

کد نویسندگان ساده است. اگر ناحیه بحرانی خالی باشد، نویسنده می‌تواند وارد شود امّا اثر این ورود این است که از ورود تمامی دیگر نخ‌ها ممانعت به عمل می‌آید.

راه‌حل نویسندگان

```
roomEmpty.wait()
    critical section for writers
roomEmpty.signal()
```

آیا زمانی‌که نویسنده از اتاق خارج می‌شود، می‌تواند از خالی بودن آن مطمئن باشد؟ بله، زیرا که می‌داند هیچ نخی نمی‌تواند تا آن زمان که آنجا است وارد شود.

کد خوانندگان مشابه کد حصار است که در بخش قبل دیدیم. تعداد خواننده‌هایی که در اتاق هستند را نگاه می‌داریم لذا می‌توانیم به اولین خواننده‌ای که وارد می‌شود و آخرین خواننده‌ای که خارج می‌شود دستوراتی خاص را بدهیم.

اولین خواننده‌ای که می‌رسد باید منتظر `roomEmpty` بماند. اگر اتاق خالی باشد، خواننده می‌تواند وارد شده و در همان زمان مانع ورود نویسندگان شود. خوانندگان بعدی هنوز می‌توانند وارد شوند زیرا که هیچ کدام از آنان منتظر `roomEmpty` نخواهند ماند.

اگر زمانیکه یک نویسنده در اتاق است خواننده‌ای برسد، روی `roomEmpty` منتظر می‌ماند. از آنجایی که نویسنده میوتکس را در اختیار گرفته، خوانندگان بعدی روی `mutex` تشکیل صف می‌دهند.

راه‌حل خوانندگان

```
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()     # first in locks
mutex.signal()

# critical section for readers

mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()   # last out unlocks
mutex.signal()
```

کد پس از ناحیه بحرانی، مشابه است. آخرین خواننده‌ای که اتاق را ترک می‌نماید لامپ را خاموش کرده -بدین معنی که به `roomEmpty` سیگنال می‌دهد- تا ورود نویسندهٔ منتظر را ممکن سازد.

دوباره، برای اینکه نشان دهیم این کد درست است، خوب است تعدادی از ادعاهایی که درباره نحوهٔ رفتار برنامه داریم را شرح داده و ثابت کنیم. آیا می‌توانید خودتان را مجاب نمایید که آنچه در ادامه آمده صحیح است؟

- فقط یک خواننده می‌تواند در انتظار برای `roomEmpty` در صف قرار گیرد امّا چندین نویسنده ممکن است در این صف قرار گیرند.
- زمانیکه یک خواننده به `roomEmpty` سیگنال می‌دهد اتاق باید خالی باشد.

الگوهای مشابه این کد خواننده رایج هستند: اولین نخ وارد شده به بخش، سمافور (یا صف‌ها) را قفل نموده و آخرین نخ خروجی آن را باز می‌نماید. در واقع، این کد اینقدر رایج است که بهتر است به آن نامی دهیم و آن را به صورت یک شیء بسته‌بندی نماییم.

نام این الگو **Lightswitch**[۷] است، شبیه به الگویی که اولین نفری که وارد اتاق می‌شود لامپ را روشن می‌کند (میوتکس را قفل می‌نماید) و آخرین نفری که از اتاق خارج می‌شود آن را خاموش می‌نماید (میوتکس را باز می‌کند). در ادامه تعریف کلاس **Lightswitch** آمده است:

تعریف Lightswitch

```
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    def lock(self, semaphore):
        self.mutex.wait()
            self.counter += 1
            if self.counter == 1:
                semaphore.wait()
        self.mutex.signal()

    def unlock(self, semaphore):
        self.mutex.wait()
            self.counter -= 1
            if self.counter == 0:
                semaphore.signal()
        self.mutex.signal()
```

`lock` یک سمافور را به عنوان پارامتر دریافت می‌دارد و آن را چک نموده و حتی ممکن آن را بگیرد. اگر سمافور قفل باشد، نخ فراخوانده روی `semaphore` مسدود گردیده و تمامی نخ‌های بعدی روی `self.mutex` مسدود می‌گردند. زمانیکه سمافور باز هست، اولین نخ در حال انتظار مجدداً آن را قفل نموده و تمامی نخ‌های در حال انتظار ادامه می‌یابند.

[۷] کلید برق

اگر سمافور در ابتدا باز باشد، اولین نخ آن را قفل نموده و مابقی نخ‌ها ادامه می‌یابند.

`unlock` تا زمانیکه تمامی نخ‌هایی که `lock` را فراخوانده‌اند `unlock` را نیز فرا خوانند هیچ اثری نخواهد داشت. زمانیکه آخرین نخ `unlock` را فراخواند سمافور را باز می‌نماید.

با کمک این توابع، می‌توانیم کد خواننده را کمی ساده‌تر بازنویسی نماییم:

مقدار دهی اولیه خوانندگان-نویسندگان

```
readLightswitch = Lightswitch()
roomEmpty = Semaphore(1)
```

readLightswitch یک شیء Lightswitch اشتراکی است که مقدار شمارنده آن در ابتدا صفر است.

راه حل خوانندگان-نویسندگان (خواننده)

```
readLightswitch.lock(roomEmpty)
# critical section
readLightswitch.unlock(roomEmpty)
```

کد نویسنده بدون تغییر باقی می‌ماند.

همچنین این امکان وجود دارد که بجای ارسال roomEmpty به عنوان یک پارامتر به lockو unlock، ارجاعی به roomEmpty را به عنوان یک خصیصهٔ Lightswitch ذخیره نمود. این رهیافت جایگزین، کمتر مستعدّ خطا است، امّا تصور می‌کنم اگر هر یک از فراخوانی‌های lock و unlock سمافوری که روی آن عمل می‌کند را مشخص نماید خوانایی افزایش می‌یابد.

۳.۲.۴ قحطی

آیا خطر بن‌بست در راه‌حل قبل وجود دارد؟ برای اینکه بن‌بستی رخ دهد، باید این امکان برای یک نخ وجود داشته باشد که بر روی یک سمافور منتظر بماند در حالیکه سمافور دیگر را در اختیار دارد و به موجب آن مانع از این شود که خودش سیگنالی دریافت نماید.

در این مثال، بن‌بست ممکن نیست، امّا یک مشکل مرتبط وجود دارد که تقریبا به همان اندازه بد است: ممکن است نویسنده دچار قحطی شود.

اگر نویسنده‌ای آن زمان که چندین خواننده در ناحیه بحرانی قرار دارند سر برسد ممکن است برای همیشه در صف منتظر بماند در حالیکه خوانندگان می‌آیند و می‌روند. تا زمانیکه یک خواننده جدید پیش از اینکه آخرین خواننده از خواننده‌های جاری خارج شود برسد، همیشه حداقل یک خواننده در اتاق وجود خواهد داشت.

این وضعیت یک بن‌بست نیست زیرا که برخی نخ‌ها در حال پیشروی هستند، امّا این به طور کامل مطلوب نیست. برنامه‌ای نظیر این ممکن است تا زمانیکه بار روی سیستم کم است کار کند، زیرا که فرصت‌های زیادی برای نویسندگان وجود دارد. امّا همانطوری که بار سیسیتم افزایش یابد رفتار سیستم ممکن است به سرعت به زوال گراید (حداقل از دید نویسندگان).

معمّا: این راه‌حل را به گونه‌ای توسعه دهید تا زمانیکه یک نویسنده می‌رسد، خوانندگان موجود بتوانند خاتمه یابند ولی هیچ خوانندهٔ جدیدی نتواند وارد شود.

۴.۲.۴ راهنمایی خوانندگان-نویسندگان بدون قحطی

راهنمایی در ادامه آمده است. می‌توانید یک ترن‌استایل برای خوانندگان بیفزایید و به نویسندگان اجازه دهید آن را قفل نمایند. نویسندگان باید از طریق ترن‌استایل مشابهی گذر نمایند، امّا تا زمانیکه داخل آن هستند باید سمافور `roomEmpty` را بررسی نمایند. اگر یک نویسنده در ترن‌استایل گیر افتد اثر آن این است که خوانندگان را مجبور می‌نماید روی ترن‌استایل تشکیل صف دهند. سپس زمانیکه آخرین خواننده ناحیه بحرانی را ترک نمود، می‌توانیم مطمئن باشیم که حداقل یک نویسنده در ادامه وارد می‌شود (قبل از آنکه خوانندگان در صف بتوانند ادامه یابند).

مقدار دهی اولیه خوانندگان-نویسندگان بدون قحطی

```
1 readSwitch = Lightswitch()
2 roomEmpty = Semaphore(1)
3 turnstile = Semaphore(1)
```

`readSwitch` تعداد خوانندگان موجود در اتاق را نگاه می‌دارد؛ زمانیکه اولین خواننده وارد شد `roomEmpty` را قفل نموده و زمانیکه آخرین خواننده خارج شد آن را باز می‌نماید.

`turnstile` برای خواننده‌ها یک ترن‌استایل و برای نویسندگان یک میوتکس است.

۵.۲.۴ راه حل خوانندگان-نویسندگان بدون قحطی

کد نویسنده در ادامه آمده است:

راه حل نویسنده بدون قحطی

```
1 turnstile.wait()
2     roomEmpty.wait()
3     # critical section for writers
4 turnstile.signal()
5
6 roomEmpty.signal()
```

اگر یک نویسنده آن زمانیکه چندین خواننده در اتاق وجود دارد برسد، در خط ۲ مسدود می‌گردد، بدین معنی که ترن‌استایل قفل شده است. این کار، مانع ورود خوانندگان تا زمانیکه یک نویسنده در صف است می‌گردد. کد خواننده در ادامه آمده است:

راه حل خواننده بدون قحطی

```
1 turnstile.wait()
2 turnstile.signal()
3
4 readSwitch.lock(roomEmpty)
5     # critical section for readers
6 readSwitch.unlock(roomEmpty)
```

زمانیکه آخرین خواننده خارج می‌شود به `roomEmpty` سیگنال داده و نویسنده در حال انتظار را رفع انسداد می‌نماید. از آنجایی که هیچکدام از خوانندگان در حال انتظار نمی‌توانند از ترن‌استایل بگذرند، نویسنده بلافاصله وارد ناحیهٔ بحرانی خودش می‌شود.

زمانیکه نویسنده خارج می‌شود `turnstile` را سیگنال می‌دهد که نخ در حال انتظار اعم از خواننده یا نویسنده را رفع انسداد می‌نماید. بنابراین، این راه‌حل تضمین می‌نماید که حداقل یک نویسنده می‌تواند ادامه یابد، امّا هنوز این امکان برای یک خواننده وجود دارد آن زمان که نویسندگانی در صف وجود دارند وارد شود.

بسته به کاربرد، اعطای اولویت بیشتر به نویسندگان ممکن ایده خوبی باشد. برای مثال، اگر خوانندگان ضرب العجلی در بروزرسانی‌های خود نسبت به یک ساختمان داده داشته باشند بهتر است که تعداد خوانندگانی که داده قدیمی را پیش از اینکه نویسنده تغییر خود را اعمال نمایند می‌بینند کمینه باشند.

گرچه عموماً، بر عهده زمان‌بند و نه برنامه‌نویس است که تعیین نماید کدام نخ در حال انتظار رفع انسداد گردد. برخی زمان‌بندها از یک صف FIFO[۸] استفاده می‌کند و بدین معنی است که نخ‌ها بهمان ترتیبی که وارد صف شده‌اند رفع انسداد می‌شود. در دیگر زمان‌بندها، انتخاب ممکن است به صورت تصادفی، یا بر اساس الگوی اولویت بر مبنای اولویت نخ‌های در حال انتظار باشد.

[8] First-In-First-Out

اگر محیط برنامه‌نویسی شما این امکان را بدهد که برخی نخ‌ها را نسبت به مابقی اولویت دهید، آنگاه برای رسیدگی به مسأله فوق، راه آسان استفاده از همین امکان است. و اگر محیط برنامه‌نویسی چنین امکانی را به شما ندهد باید بدنبال راه‌حل دیگری باشید.

معمّا: راه حلی برای مسأله خوانندگان-نویسندگان ارائه دهید که اولویت را به نویسندگان دهد. به این معنی که اگر یک نویسنده رسید، هیچ خواننده‌ای مجاز به ورود نباشد تا آن زمان که تمامی نویسندگان سیستم را ترک گفته باشند.

۶.۲.۴ راهنمایی خوانندگان-نویسندگان با اولویت نویسنده

طبق معمول، راهنمایی در قالب متغیرهای مورد استفاده در راه‌حل آمده است.

مقداردهی اولیه خوانندگان-نویسندگان با اولویت نویسنده

```
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)
noWriters = Semaphore(1)
```

۷.۲.۴ راه حل نویسندگان-خوانندگان با اولویت نویسنده

کد خواننده در ادامه آمده است:

راه حل خواننده با اولویت نویسنده

```
noReaders.wait()
    readSwitch.lock(noWriters)
noReaders.signal()

    # critical section for readers

readSwitch.unlock(noWriters)
```

اگر یک خواننده درون ناحیه بحرانی باشد، noWriters را در اختیار می‌گیرد، امّا noReaders را خیر. بنابراین اگر یک نویسنده برسد می‌تواند noReaders را قفل نماید که سبب در صف قرار گرفتن خوانندگان بعدی می‌گردد.

زمانیکه آخرین خواننده خارج می‌شود، با سیگنال دهی به noWriters اجازه می‌دهد تا نویسندگانی که در صف قرار گرفته‌اند بکار خود ادامه دهند.

کد نویسنده:

راه حل نویسنده با اولویت نویسنده

```
writeSwitch.lock(noReaders)
    noWriters.wait()
        # critical section for writers
    noWriters.signal()
writeSwitch.unlock(noReaders)
```

زمانیکه یک نویسنده در ناحیه بحرانی است هر دوی noReaders و noWriters را در اختیار می‌گیرد. نسبتاً واضح است که اثر این، تضمین نمودن این است که هیچ خواننده و نویسندهٔ دیگری در ناحیه بحرانی قرار ندارد. به علاوه، اثر کمتر واضح این است که writeSwitch به چندین نویسنده اجازه می‌دهد تا روی noWriters در صف قرار گیرند، امّا تا زمانیکه نویسندگان حضور دارند، noReaders را قفل نگاه می‌دارد. فقط زمانیکه آخرین نویسنده خارج می‌شود، خوانندگان می‌توانند وارد شوند.

البته، یک اشکال این راه‌حل این است که اکنون ممکن است خوانندگان دچار قحطی شوند (یا حداقل با تاخیر طولانی مواجه شوند). برای برخی کاربردها شاید بهتر باشد که داده‌های قدیمی با زمان‌های بازگشتِ پیش‌بینی پذیر، اخذ گردد.

۳.۴ میوتکس بدون قحطی

در بخش قبل، با چیزی که من آن را **قحطی دسته‌ای** نامیدم مواجه شدیم، که در آن یک گروه از نخ‌ها (خوانندگان) سبب قحطی دسته دیگر (نویسندگان) می‌گردد. در سطحی ابتدایی‌تر، باید این موضوع **قحطی نخ** را بررسی نماییم (امکان اینکه یک نخ در حالیکه مابقی نخ‌ها به کار خود ادامه می‌دهند، به صورت نامتناهی منتظر بماند).

برای غالب برنامه‌های کاربردی همروند، قحطی پذیرفتی نیست، لذا باید ضرورت **انتظار محدود**[9] را اعمال کنیم، بدین معنی که زمان انتظار نخ بر روی یک سمافور (یا هر جای دیگری به همان منظور) باید ثبوتاً متناهی باشد.

تا حدّی، زمان‌بند مسبب قحطی است. هر زمان که چندین نخ آمادهٔ اجرا هستند، زمان‌بند تصمیم می‌گیرد که کدام یک، در یک پردازنده موازی، کدام مجموعه از نخ‌ها، اجرا شوند. اگر یک نخ هرگز زمانبندی نشود آنگاه دچار قحطی خواهد شد، بدون توجه به اینکه ما با سمافورها چه می‌کنیم.

لذا به منظور اینکه چیزی درباره قحطی بگوییم، باید با تعدادی پیش‌فرض درباره زمان‌بند شروع نماییم. اگر تمایل به یک فرض قوی درباره زمان‌بند داریم، می‌توانیم تصویر نماییم که زمان‌بند یکی از الگوریتم‌های بسیاری که می‌تواند ثابت شود که انتظار محدود را اعمال می‌کند، استفاده می‌نماید. اگر نمی‌دانیم که زمان‌بند از چه الگوریتمی استفاده می‌نماید، سپس می‌توانیم یک فرض ضعیف‌تر در نظر بگیریم:

خصوصیت ۱: اگر تنها یک نخ آماده اجرا هست، زمان‌بند باید اجازه اجرای آن را بدهد.

اگر بتوانیم خصوصیت ۱ را فرض کنیم، سپس می‌توانیم سیستمی بدون قحطی بسازیم. برای مثال، اگر یک تعداد متناهی از نخ‌ها وجود داشته باشد، سپس هر برنامه‌ای که شامل یک حصار باشد دچار قحطی نمی‌گردد، زیرا که نهایتاً تمامی نخ‌ها بجز یکی پشت حصار منتظر می‌ماند که در این نقطه آخرین نخ باید اجرا شود.

اگر چه، به طور کلی نوشتن برنامه‌هایی که دچار قحطی نمی‌شوند بدیهی نیست مگر اینکه فرض قوی‌تری نماییم:

خصوصیت ۲: اگر یک نخ آماده اجرا باشد، آنگاه زمانیکه برای اجرا منتظر می‌ماند باید متناهی باشد.

تا اینجای بحث، به طور ضمنی خصوصیت ۲ را فرض نموده‌ایم و به آن ادامه خواهیم داد. از طرف دیگر باید بدانید که بسیاری از سیستم‌های موجود از زمان‌بندهایی استفاده می‌نمایند که این خصوصیت را موکداً تضمین نمی‌نماید.

حتی با خصوصیت ۲، زمانیکه سمافورها را معرفی می‌کنیم، قحطی چهره نازیبایش را دومرتبه نشان می‌دهد. در تعریف یک سمافور، گفتیم زمانیکه یک نخ `signal` را اجرا می‌نماید، یکی از نخ‌های در حال انتظار را بیدار

[9] bounded waiting

می‌نماید. امّا هیچگاه نگفتم کدام را. قبل از اینکه چیزی درباره قحطی بگوییم باید پیش‌فرض‌هایی درباره رفتار سمافورها در نظر بگیریم.

ضعیف‌ترین فرضی که به منظور اجتناب از قحطی ممکن است به شرح زیر است:

> خصوصیت ۳: اگر زمانیکه یک نخ، `signal` را اجرا می‌نماید، نخ‌هایی در حال انتظار روی سمافور وجود داشته باشد، آنگاه یکی از نخ‌های در حال انتظار باید بیدار شود.

این التزام ممکن است واضح به نظر آید، امّا بدیهی نیست. این فرض جلوی یکی از گونه‌های مشکل‌ساز را خواهد گرفت که در آن یک نخ که به یک سمافور سیگنال می‌دهد در حالیکه نخ‌های دیگری در حال انتظار هستند، سپس به اجرای خود ادامه داده و روی همان سمافور منتظر شده و سیگنال خودش را می‌گیرد. اگر این نکته ممکن باشد، آنگاه هیچ کاری برای جلوگیری از قحطی نمی‌توانیم انجام دهیم.

با خصوصیت ۳، اجتناب از قحطی امکان‌پذیر می‌گردد، امّا حتی برای چیزی به سادگی میوتکس، این امر ساده نیست. برای مثال، سه نخ که در حال اجرای کد زیر هستند را تصویر نمایید:

حلقه میوتکس

```
1 while True:
2     mutex.wait()
3     # critical section
4     mutex.signal()
```

این دستور `while` یک حلقه بی‌پایان است؛ به عبارت دیگر، به محض اینکه یک نخ، ناحیه بحرانی را ترک می‌نماید به بالای حلقه بر می‌گردد و تلاش می‌کند که میوتکس را دوباره بگیرد.

تصور کنید نخ A میوتکس را گرفته و B و C منتظر هستند. زمانی که A خارج می‌شود B وارد می‌گردد امّا پیش از اینکه B خارج شود، A دور زده و در صف به C ملحق می‌شود. زمانیکه B خارج می‌شود، هیچ تضمینی وجود ندارد که پس از آن C وارد شود. در واقع، اگر پس از آن A وارد شود و B به صف ملحق شود، آنگاه ما به نقطه شروع بازگشته‌ایم و می‌توانیم این چرخه را تا ابد تکرار کنیم. C قحطی زده می‌شود.

وجود این الگو ثابت می‌نماید که میوتکس نسبت به قحطی آسیب‌پذیر است. یک راه‌حل برای این مسأله این است که پیاده‌سازی سمافور را به گونه‌ای تغییر دهیم که خصوصیت قوی‌تری را تضمین نماید.

> خصوصیت ۴: اگر یک نخ منتظر یک سمافور باشد، آنگاه تعداد نخ‌هایی که قبل از آن بیدار خواهند شد محدود است.

برای مثال، اگر صف تشکیل شده روی سمافور، از نوع FIFO باشد، آنگاه خصوصیت ۴ برقرار است زیرا که زمانیکه یک نخ به صف ملحق می‌شود تعداد نخ‌های جلوی آن، متناهی است و هیچ نخی که پس از آن می‌رسد نمی‌تواند جلوی آن قرار گیرد.

سمافوری که خصوصیت ۴ را دارد گاهی اوقات **سمافور قوی**[۱۰] گفته می‌شود؛ و سمافوری که تنها خصوصیت ۳ را دارد **سمافور ضعیف**[۱۱] گفته می‌شود. نشان داده‌ایم که با سمافورهای ضعیف، راه‌حل میوتکس ساده نسبت به قحطی آسیب‌پذیر هست. در واقع، حدس Dijkstra این بود که حل بدون قحطی مسأله میوتکس تنها با کمک سمافورهای ضعیف ممکن نیست.

در ۱۹۷۹ J.M. Morris با حل مسأله فوق و در نظر گرفتن اینکه تعداد نخ‌ها متناهی باشد، حدس Dijkstra را رد نمود[؟]. اگر شما به این مسأله علاقه‌مند هستید، بخش بعدی راه‌حل او را نشان می‌دهد. در غیر اینصورت، می‌توانید تصور کنید که سمافورها خصوصیت ۴ را دارند و به بخش ۴.۴ بروید.

معمّا: یک راه برای مسأله انحصار متقابل با کمک سمافورهای ضعیف بنویسید. راه‌حل شما باید شرط زیر را تضمین نماید: زمانیکه یک نخ می‌رسد و تلاش می‌کند به میوتکس وارد شود، روی تعداد نخ‌هایی که می‌توانند پیش از آن ادامه یابند باید یک حدی وجود داشته باشد. می‌توانید تصور نمایید که تعداد کلّ نخ‌ها متناهی است.

[10] strong semaphore [11] weak semaphore

۱.۳.۴ راهنمای میوتکس بدون قحطی

راه‌حل Morris مشابه با حصار قابل استفاده مجدد در بخش ۷.۳ است. این راه‌حل از دو ترن‌استایل برای ایجاد دو اتاق انتظار قبل از ناحیه بحرانی استفاده می‌نماید. این مکانیزم در دو فاز عمل می‌نماید. در طول فاز اول، ترن‌استایل اول باز است و دومی بسته، لذا نخ‌ها در اتاق دوم مجتمع می‌گردند. در طول فاز دوم، ترن‌استایل اول قفل است و لذا هیچ نخی جدیدی نمی‌تواند وارد شود و ترن‌استایل دوم باز است، بنابراین نخ‌های موجود می‌توانند وارد ناحیه بحرانی شوند.

گرچه ممکن است تعداد دلخواهی از نخ‌ها در اتاق انتظار باشند، امّا ورود هر کدام از آن‌ها به ناحیه بحرانی پیش از اینکه نخ‌های بعدی برسند تضمین شده است.

در ادامه متغیرهایی که در راه‌حل بکار بردم آمده است (در تلاش برای واضح‌تر ساختن ساختار، نام‌هایی را که Morris بکار برده بود، تغییر دادم).

راهنمایی میوتکس بدون قحطی

```
1 room1 = room2 = 0
2 mutex = Semaphore(1)
3 t1 = Semaphore(1)
4 t2 = Semaphore(0)
```

room1 و room2 تعداد نخ‌هایی که در اتاق‌های انتظار هستند را نگاهداری می‌نماید. mutex از شمارنده‌ها محافظت می‌نماید. t1 و t2 ترن‌استایل هستند.

۲.۳.۴ راه حل میوتکس بدون قحطی

در ادامه راه‌حل Morris آمده است.

الگوریتم Morris

```
1  mutex.wait()
2      room1 += 1
3  mutex.signal()
4
5  t1.wait()
6      room2 += 1
7      mutex.wait()
8      room1 -= 1
9
10     if room1 == 0:
11         mutex.signal()
12         t2.signal()
13     else:
14         mutex.signal()
15         t1.signal()
16
17 t2.wait()
18     room2 -= 1
19
20     # critical section
21
22     if room2 == 0:
23         t1.signal()
24     else:
25         t2.signal()
```

پیش از ورود به ناحیه بحرانی، یک نخ باید از دو ترن‌استایل بگذرد. این ترن‌استایل‌ها کد را به سه اتاق تقسیم می‌نمایند. اتاق ۱ خطوط ۲- ۸. اتاق ۲ خطوط ۶- ۱۸. اتاق ۳ مابقی کد است. شمارنده‌های room1 و room2 تعداد نخ‌های هر اتاق را در خود نگه می‌دارند.

شمارنده room1 به طور معمول با کمک mutex محافظت می‌شود امّا امر حفاظت از room2 بین t1 و t2 تقسیم شده است. به طور مشابه، مسئولیت دسترسی انحصاری به ناحیه بحرانی مشمول هر دوی t1 و t2 است. یک نخ به منظور ورود به ناحیه بحرانی، باید یکی از این دو و نه هر دو را بگیرد. آنگاه پیش از خروج، هر کدام را که گرفته باشد آزاد می‌نماید.

برای فهم اینکه این راه‌حل چگونه عمل می‌نماید، با دنبال نمودن یک نخ در تمام مسیر شروع می‌کنیم. زمانیکه به خط ۸، mutex و t1 را گرفته است. زمانیکه room1 را بررسی می‌نماید، که مقدار آن ۰ است، می‌تواند mutex را رها کرده و سپس ترن‌استایل دوم (t2) را باز نماید. در نتیجه، در خط ۱۷ منتظر نمانده و می‌تواند

بدون خطر مقدار room2 را یک واحد کاهش داده و وارد ناحیه بحرانی شود، زیرا نخ‌های بعدی باید روی t1 به صف شوند. با خروج از ناحیه بحرانی، مقدار room2 را صفر می‌بیند و t1 را رها می‌نماید که ما را به نقطه شروع بر می‌گرداند.

البته، راه‌حل اگر بیش از یک نخ وجود داشته باشد جذاب‌تر است. در این حالت، زمانیکه نخ مقدّم به خط ۸ می‌رسد، ممکن است دیگر نخ‌ها وارد اتاق انتظار شده و روی t1 صف تشکیل داده باشند. از آنجایی که room1 > 0 نخ مقدّم، t2 را قفل شده رها نموده و در عوض به t1 سیگنال می‌دهد تا اجازه ورود به اتاق ۲ را به دیگر نخ‌ها بدهد. از آنجایی که t2 هنوز قفل است، هیچ نخی نمی‌تواند وارد اتاق ۳ بشود.

نهایتاً (چون تعداد نخ‌ها متناهی است)، یک نخ پیش از اینکه دیگر نخ‌ها به اتاق ۱ وارد شوند به خط ۸ می‌رسد، که در این حالت t2 را باز نموده و به دیگر نخ‌ها اجازه می‌دهد که به اتاق ۳ وارد شوند. نخی که t2 را باز می‌کند همچنان t1 را نگاه می‌دارد، بنابراین اگر هر کدام از نخ‌های مقدم دوباره به ابتدای کد باز گردد، در خط ۵ مسدود می‌گردد.

از آنجایی که هر نخ خروجی از اتاق ۳ به t2 سیگنال می‌دهد، به دیگر نخ‌ها اجازه می‌دهد اتاق ۲ را ترک نمایند. زمانیکه آخرین نخ اتاق ۲ را ترک می‌نماید، t2 را قفل شده رها می‌کند و t1 را باز می‌نماید که این ما را به نقطه شروع باز می‌گرداند.

برای اینکه ببینیم این راه‌حل چگونه از قحطی جلوگیری می‌نماید، بهتر است که عملکردش را در دو فاز بررسی نماییم. در فار اول، نخ‌ها در اتاق ۱ بررسی شده، مقدار room1 را یک واحد افزایش داده، و سپس در یک زمان به اتاق ۲ سرازیر می‌شوند. تنها راه قفل نگاه داشتن t2، وجود یک جریان ادامه‌دار از نخ‌های به اتاق ۱ است. از آنجایی که تعداد نخ‌ها متناهی است این جریان نهایتاً پایان می‌پذیر و در آن نقطه t1 قفل می‌ماند و t2 باز می‌شود.

در فاز دوم، نخ‌ها به اتاق ۳ سرازیر می‌شود. از آنجایی که تعداد نخ‌های موجود در اتاق ۲ متناهی است و هیچ نخ جدیدی نمی‌تواند وارد شود ، در نهایت آخرین نخ خارج می‌گردد و در آن زمان t2 قفل شده و t1 باز می‌شود.

در پایان فاز اول، می‌دانیم که هیچ نخی روی t1 منتظر نیست زیرا که room1 = 0. و در پایان فاز دوم، می‌دانیم که هیچ نخی روی t2 منتظر نیست زیرا که room2 = 0.

با یک تعداد متناهی از نخ‌ها، قحطی تنها در صورتی ممکن است که یک نخ بتواند دور زده و از مابقی سبقت گیرد. امّا این مکانیزم ترن‌استایل مضاعف، چنین امری را ناممکن می‌سازد لذا قحطی غیر ممکن است.

نکته اینکه با وجود سمافورهای ضعیف جلوگیری از قحطی حتی برای ساده‌ترین مسائل همگام‌سازی بسیار سخت است. در ادامه کتاب، هر زمان که از قحطی صحبت می‌کنیم، سمافورهای قوی را مد نظر خواهیم داشت.

۴.۴ غذا خوردن فیلسوف‌ها

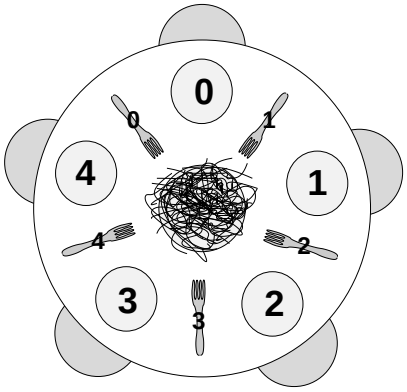
مسأله غذا خوردن فیلسوف‌ها در سال ۱۹۶۵ توسط دایکسترا مطرح گردید؛ زمانیکه دایناسورها بر زمین حکمرانی می‌کردند[؟]. این مسأله در انواع گوناگونی مطرح شده لکن ویژگی‌های استاندارد آن یک میز با پنج بشقاب، پنج چنگال (چوب) و یک کاسه بزرگ از اسپاگتی است. پنج فیلسوف که نشانگر نخ‌های در حال تعامل هستند، کنار میز آمده و حقله زیر را اجرا می‌نمایند:

حلقه پایه فیلسوف

```
while True:
   think()
   get_forks()
   eat()
   put_forks()
```

چنگال‌ها، منابعی را نشان می‌دهند که نخ‌ها باید برای پیشرفت به صورت انحصاری آن‌ها را در اختیار داشته باشند. آن چیزی که مسأله را جذاب، غیرواقعی و غیربهداشتی می‌نماید این است که فیلسوف‌ها برای خوردن نیاز به دو تا چنگال دارند، لذا یک فیلسوف گرسنه باید منتظر همسایه‌اش بماند تا چنگال را زمین بگذارد.

تصور کنید که فیلسوف‌ها یک متغیر محلی `i` دارند که هر کدام از آن‌ها را با مقداری بین ۰ تا ۴ مشخص می‌کند. به طور مشابه، چنگال‌ها از ۰ تا ۴ شماره‌گذاری شده‌اند، لذا فیلسوف i چنگال i را در سمت راست خود دارد و چنگال $i+1$ را در سمت چپ. دیاگرام این وضعیت در تصویر آمده است:



با فرض آنکه بدانیم فیلسوف‌ها چگونه `think` و `eat` می‌نمایند، وظیفه ما این است نسخه‌ای از `get_forks` و `put_forks` را بنویسیم که شرایط زیر را برآورده سازد:

- در یک زمان تنها یک فیلسوف بتواند یک چنگال را در اختیار داشته باشد.
- امکان بروز بن‌بست وجود نداشته باشد.

- یک فیلسوف نباید به واسطه انتظار برای به دست آوردن یک چنگال دچار قحطی شود.
- این امکان وجود داشته باشد که بیش از یک فیلسوف بتوانند به صورت همزمان غذا خورند.

آخرین نیازمندی، بیان دیگری از این مطلب است که راه‌حل باید کارا باشد؛ به این معنی که باید حداکثر مقدار همروندی را اجازه دهد.

درباره اینکه `eat` و `think` چقدر طول می‌کشد هیچ فرضی در نظر نمی‌گیریم، بجز آنکه `eat` باید در نهایت خاتمه یابد. در غیر اینصورت، اعمال محدودیت سوم ناممکن است—اگر یک فیلسوف یک از چنگال‌ها را تا ابد نگه دارد، هیچ چیز نمی‌تواند مانع قحطی همسایه‌ها شود.

برای اینکه ارجاع فیلسوف‌ها به چنگال‌هایشان را ساده نماییم می‌توانیم از توابع `left` و `right` استفاده کنیم.

کدام چنگال؟

```
1 def left(i): return i
2 def right(i): return (i + 1) % 5
```

اپراتور % زمانیکه مقدار به ۵ برسد آن به ۰ بر می‌گرداند؛ `(4 + 1) % 5 = 0`.

از آنجایی که لازم است دسترسی انحصاری به چنگال‌ها را فراهم آوریم، طبیعی است که از لیستی از سمافورها استفاده کنیم، هر کدام برای یک چنگال. در ابتدا تمامی چنگال‌ها موجودند.

متغیرهای غذا خوردن فیلسوف‌ها

```
1 forks = [Semaphore(1) for i in range(5)]
```

این نماد برای مقداردهی اولیه یک لیست ممکن است برای خوانندگانی که پایتون را به کار نبرده‌اند ناآشنا باشد. تابع `range`، لیستی با پنج عنصر بر می‌گرداند؛ برای هر عنصر این لیست، پایتون یک سمافور با مقدار اولیه ۱ ساخته و نتیجه را در یک لیست به نام `forks` گرد می‌آورد.

تلاشی ابتدایی برای `get_fork` و `put_fork` در ادامه آمده است:

ناراه‌حل غذا خوردن فیلسوف

```
1 def get_forks(i):
2     fork[right(i)].wait()
3     fork[left(i)].wait()
4
5 def put_forks(i):
6     fork[right(i)].signal()
7     fork[left(i)].signal()
```

واضح است که این راه‌حل اولین شرط را برآورده می‌نماید، امّا می‌توانیم مطمئن باشیم که دو شرط بعدی برآورده نمی‌شود، زیرا که اگر چنین بود، اصلاً مسأله جذابی نبوده و شما می‌توانستید به مطالعه فصل ۵ بپردازید.

معمّا: مشکل کجاست؟

۱.۴.۴ بن‌بست #۵

اشکال کار در این است که میز گرد است. در نتیجه، هر فیلسوف می‌تواند یک چنگال را بر گرفته و سپس برای همیشه منتظر چنگال دیگر بماند. بن‌بست!

معمّا: راه حلی برای این مسأله ارائه دهید که از بن‌بست جلوگیری نماید.

راهنمایی: یک راه اجتناب از بن‌بست این است که شرایطی که بن‌بست را ممکن می‌سازند را در نظر گرفته و سپس یکی از آن‌ها را تغییر دهیم. در این مورد، بن‌بست خیلی شکننده است-یک تغییر خیلی کوچک آن را در هم می‌شکند.

۲.۴.۴ راهنمایی غذا خوردن فیلسوف‌ها #۱

اگر تنها چهار فیلسوف در یک زمان مجاز به نشستن سر میز باشند، بن‌بست غیر ممکن می‌گردد.

ابتدا، خودتان را متقاعد نمایید که این ادعا درست است، سپس کدی بنویسید که تعداد فیلسوف‌ها را سر میز محدود نمایید.

۳.۴.۴ راه حل غذا خوردن فیلسوف‌ها #۱

اگر تنها چهار فیلسوف سر میز باشند، آنگاه در بدترین حالت هر کدام یک چنگال را بر می‌دارد. سپس، تنها یک چنگال روی میز باقی مانده و آن چنگال دو همسایه دارد که هر کدام یک چنگال دیگر در دست دارند. بنابراین، هر کدام از همسایه‌ها می‌توانند چنگال باقی‌مانده را برداشته و غذا خورد.

تعداد فیلسوف‌های سر میز را می‌توانیم با مالتی‌پلکسی با نام `footman` که مقدار اولیه ۴ دارد کنترل کنیم. راه‌حل مشابه زیر است:

راه حل غذا خوردن فیلسوف‌ها #۱

```
def get_forks(i):
    footman.wait()
    fork[right(i)].wait()
    fork[left(i)].wait()

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
    footman.signal()
```

علاوه بر اجتناب از بن‌بست، این راه‌حل تضمین می‌نماید که هیچ فیلسوفی دچار قحطی نشود. تصور کنید که شما سر میز نشسته‌اید و هر دو همسایه شما مشغول غذا خوردن هستند. شما در انتظار برای چنگال سمت راست مسدوده شده‌اید. در نهایتا همسایه سمت راست شما، چنگال را زمین خواهد گذاشت چرا که `eat` نمی‌تواند تا ابد ادامه داشته باشد. از آنجاییکه شما تنها نخی هستید که منتظر آن چنگال هستید، لزوماً پس از آن چنگال به دست خواهید آورد. با استدلالی مشابه، در انتظار برای چنگال سمت چپ‌تان نیز دچار قحطی نخواهید شد.

بنابراین، زمانیکه یک فیلسوف می‌تواند سر میز بگذراد محدود است. همچنین این نکته دلالت بر این دارد که زمان انتظار برای ورود به اتاق تا زمانیکه `footman` خصوصیت ۴ را دارد، محدود است (بخش ۳.۴ را ببینید).

این راه‌حل نشان می‌دهد که با کنترل کردن تعداد فیلسوف‌ها، می‌توانیم از بن‌بست اجتناب نماییم. راه دیگر برای اجتناب از بن‌بست تغییر دادن ترتیبی است که فیلسوف‌ها چنگال‌ها را بر می‌دارند. در ناراه‌حل اولیه، فیلسوف‌ها "راست‌دست"[۱۲] هستند؛ به این معنی که ابتدا چنگال سمت راست را بر می‌دارند. امّا اگر فیلسوف ۰ "چپ‌دست"[۱۳] باشد چه اتفاقی می‌افتد؟

معمّا: ثابت کنید که اگر حداقل یکی از فیلسوف‌ها چپ‌دست و حداقل یکی راست‌دست باشد، آنگاه بن‌بست غیر ممکن است.

راهنمایی: بن‌بست تنها زمانی رخ می‌دهد که ۵ فیلسوف یک چنگال را برای در دست دارند و تا ابد منتظر چنگال دیگر می‌مانند. در غیر اینصورت، یکی از آن‌ها می‌تواند هر دو چنگال را برداشته، غذا خورده و خارج شود.

12 13

اثبات با برهان خلف است. ابتدا، تصور کنید که بن‌بست ممکن باشد. سپس یکی از فیلسوف‌هایی که در بن‌بست گیر کرده است انتخاب نمایید. اگر آن فیلسوف چپ‌دست باشد، می‌توانید ثابت نمایید که تمامی فیلسوف‌ها چپ‌دست هستند، که این خود یک تناقض است. به طور مشابه، اگر راست‌دست باشد می‌تواند ثابت نمایید که همگی راست‌دست هستند. از هر دو طریق به تناقض می‌رسید؛ بنابراین بن‌بست ناممکن است.

۴.۴.۴ راه حل غذاخوردن فیلسوف‌ها #۲

در راه‌حل متقارن مسأله غذاخوردن فلیسوف‌ها، لازم است حداقل یک چپ دست و حداقل یک راست دست سر میز باشند. در این حالت، بن‌بست غیر ممکن است. راهنمایی قبل طرح کلی اثبات را بیان می‌کند. در ادامه جزئیات آمده است.

دوباره، اگر بن‌بست ممکن باشد، زمانی رخ می‌دهد که تمامی ۵ فیلسوف یک چنگال را نگاه داشته و منتظر چنگال دیگر بمانند. اگر فرض کنیم فیلسوف j چپ دست باشد آنگاه او باید چنگال سمت چپ را نگاه داشته و منتظر چنگال سمت راست باشد. بنابراین همسایه سمت راست او (فیلسوف k) باید چنگال سمت چپش را نگاه داشته و منتظر همسایه راست خود باشد؛ به عبارت دیگر فیلسوف k باید چپ دست باشد. با تکرار استدلال مشابه می‌توانیم ثابت کنیم که تمامی فیلسوف‌ها چپ دست هستند که با حکم اولیه که در آن حداقل یک راست دست وجود دارد در تناقض است. لذا بن‌بست ممکن نیست.

استدلالی مشابه آنچه برای راه‌حل قبل به کار بردیم ثابت می‌کند که قحطی نیز غیر ممکن است.

۵.۴.۴ راه حل تننبام

در راه‌حل قبل هیچ چیز نادرستی وجود ندارد امّا تنها برای تکمیل بحث، بگذارید نگاهی به برخی راه حل‌های جایگزین بیندازیم. یکی از شناخته شده ترینِ این راه حل‌ها، همانی است که در کتاب معروف سیستم‌عامل تننبام آمده است[؟]. برای هر فیلسوف یک متغیر وضعیت وجود دارد که نشان می‌دهد فیلسوف در کدامیک از حالات تفکر، خوردن و یاد انتظار برای خوردن (گرسنگی) است و یک سمافور که نشان می‌دهد آیا فیلسوف می‌تواند غذا خوردن را آغاز نماید نیز وجود دارد.

متغیرهای راه‌حل تننبام

```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

مقدار اولیه `state` یک لیست ۵ تایی از `'thinking'` است. `sem` یک لیست ۵ تایی از سمافورهایی است با مقدار اولیه ۰. و امّا کد:

راه‌حل تننبام

```
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
    state[left (i)]  != 'eating' and
    state[right (i)] != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

تابع `test` بررسی می‌کند که آیا فیلسوف iام می‌تواند شروع به غذا خوردن نماید، و این در صورتی است که او گرسنه بوده و هیچکدام از همسایه‌هایش در حال غذا خوردن نباشند. اگر چنین باشد، `test` به سمافور i سیگنال می‌دهد.

دو راه وجود دارد که یک فیلسوف می‌تواند غذا بخورد. در حالت اول، فیلسوف `get_forks` را اجرا کرده، چنگال‌های موجود را یافته و بلافاصله مشغول خوردن می‌شود. در حالت دوم، یکی از همسایه‌ها در حال غذا خوردن است و فیلسوف روی سمافور خودش مسدود گردیده است. نهایتاً یکی از همسایه‌ها دست از غذا می‌کشد و در این زمان `test` را روی هر دو همسایه خود اجرا می‌نماید. ممکن است که هر دو بررسی موفقیت آمیز باشد، و در این حالت همسایه‌ها می‌توانند به صورت همروند مشغول غذا خوردن شوند. ترتیب دو بررسی اهمیتی ندارد.

به منظور دسترسی به `state` یا فراخوانی `test`، یک نخ باید `mutex` را بگیرد. بنابراین، عمل بررسی و بروزرسانی آرایه، اتمی است.از آنجایی که یک فیلسوف تنها زمانی می‌تواند مشغول خوردن شود که بدانیم هر دو چنگال موجود است، دسترسی انحصاری به چنگال‌ها تضمین شده است.

هیچ بن‌بستی ممکن نیست، زیرا تنها سمافوری که بیش از یک فیلسوف به آن دسترسی دارد، `mutex` است و هیچ نخی تا زمانی که `mutex` را در اختیار دارد `wait` را اجرا نمی‌نماید.

امّا دومرتبه، قحطی ممکن ولی در اینجا خیلی مستلزم دقت و مهارت است.

معمّا: یا خودتان را متقاعد نمایید راه‌حل تننبام از قحطی جلوگیری می‌نماید و یا یک الگوی تکرارشونده بیابید اجازه می‌دهد یک نخ دچار قحطی شود در حالیکه مابقی نخ‌ها به کار خود ادامه می‌دهند.

۶.۴.۴ قحطی تنبام

متاسفانه، این راه‌حل مصون از قحطی نیست. Gingras نشان داد که الگوهای تکرارشونده‌ای وجود دارد در آن یک نخ برای همیشه منتظر مانده در حالیکه دیگر نخ‌ها می‌آیند و می‌روند[؟].

تصور کنید که می‌خواهیم فیلسوف ۰ دچار قحطی شود. در ابتدا، ۲ و ۴ سر میز هستند و ۱ و ۳ گرسنه هستند. تصور کنید که ۲ بر می‌خیزد و یک سر میز می‌نشیند؛ سپس ۴ برخواسته و ۳ می‌نشیند. اکنون در وضعیتی قرینه موقعیت شروع هستیم.

اگر ۳ برخیزد و ۴ بنشیند، و سپس برخواسته و ۲ بنشیند، به نقطه شروع بازگشته‌ایم. این حلقه را می‌توانیم تا ابد تکرار نماییم و در این صورت فیلسوف ۰ دچار قحطی می‌شود.

لذا راه‌حل تنبام تمامی ملزومات را بار آورده نمی‌نماید.

۵.۴ مسأله سیگاری‌ها

مسأله سیگاری‌ها در ابتدا توسط Suhas Patil مطرح شد [؟]، و ادعا نمود که این مسأله به سمافورها قابل حل نیست. این ادعا با تعدادی شرط همراه است، امّا در هر حالت مسأله جذاب و چالش برانگیز است.

چهار نخ در مسأله وجود دارد: یک عامل و سه سیگاری. سیگاری‌ها تا ابد، در حلقهٔ ابتدا انتظار برای مواد مورد نیاز و سپس ساختن و کشیدن سیگار هستند. مواد مورد نیاز شامل تنباکو، کاغذ و کبریت است.

فرض می‌کنیم که عامل یک منبع لایزال از این سه ماده لازم دارد و هر سیگاری یک منبع نامتناهی از یکی از این سه مواد لازم را دارد، بدین معنی که یکی از سیگاری‌ها کبریت، دیگری کاغذ و سومی نیز تنباکو دارد.

عامل مکراراً دو ماده متفاوت را به صورت تصادفی انتخاب نموده و آن‌ها را به سیگاری‌ها عرضه می‌کند. بسته به اینکه چه موادی انتخاب شده باشد، فرد سیگاری با ماده مکمل خود می‌تواند دو منبع را برداشته و ادامه دهد.

برای مثال، اگر عامل تنباکو و کاغذ بر دارد، فرد سیگاری که کبریت دارد می‌تواند هر دو ماده را برداشته و یک سیگار ساخته و سپس به عامل سیگنال دهد.

برای توضیح فرض قبل، عامل نشانگر یک سیستم‌عامل است که منابع را تخصیص می‌دهد، و سیگاری‌ها نشانگر برنامه‌هایی هستند که به منابع نیاز دارند. مسأله این است که اطمینان دهیم اگر منابعی موجود هستند که می‌توانند اجازه ادامه فعالیت برنامه‌های بیشتری را بدهند آن برنامه‌ها باید بیدار شوند. بر عکس، می‌خواهیم از بیدار نمودن برنامه‌هایی که نمی‌توانند به کار خود ادامه دهند اجتناب نماییم.

بر طبق این فرض، سه نسخه از این مسأله وجود دارد که اغلب در کتاب‌ها مشاهده می‌شود:

نسخه غیرممکن: نسخه Patil محدودیت‌هایی روی راه‌حل تحمیل می‌نماید. اول اینکه، شما اجازه تغییر کد عامل را ندارید. اگر عامل نشانگر یک سیستم‌عامل باشد، این فرض که شما نمی‌خواهید کد سیستم عامل را هر زمان که یک برنامه جدید می‌آید تغییر دهید بی معنی نیست. محدودیت دوم این است که شما نمی‌توانید از عبارات شرطی یا یک آرایه‌ای از سمافورها استفاده نمایید. با این محدودیت‌ها، مسأله قابل حل نیست، امّا همانطور که Patil اشاره نموده است محدودیت دوم کاملاً تصنعی است [؟]. با محدودیت‌هایی نظیر این دو، مسائل بسیار غیر قابل حل خواهد شد.

نسخه جالب: این نسخه محدودیت اول (عدم امکان تغییر کد عامل) را دارد ولی مابقی را در نظر نمی‌گیرد.

نسخه بدیهی: در برخی کتاب‌ها، در خود مسأله آمده است که عامل باید بر مبنای مواد موجود به آن فرد سیگاری که می‌تواند ادامه دهد سیگنال ارسال نماید. این نسخه از مسأله، جذابیتی ندارد زیرا که تمام فرض اولیه، مواد افزودنی و سیگارها را غیرضروری می‌نماید. همچنین در عمل، احتمالاً اینکه عامل، اطلاعی از سایر نخ‌ها و آنچه آن‌ها نیاز دارند داشته باشد ایده خوبی نباشد. در نهایت، این نسخه از مسأله نیز بسیار ساده است.

طبیعتاً بر روی نسخه جالب تمرکز می‌نمایم. برای تکمیل بیان مسأله، باید کد عامل را مشخص نماییم. عامل، سمافورهای زیر را بکار می‌برد:

سمافورهای عامل

```
agentSem = Semaphore(1)
tobacco = Semaphore(0)
paper = Semaphore(0)
match = Semaphore(0)
```

عامل در واقع از سه نخ همروند تشکیل شده است: عامل A، عامل B، عامل C. هر کدام از آن‌ها روی agentSem منتظر می‌مانند؛ هر گاه که agentSem سیگنالی دریافت کند، یکی از عامل‌ها بر می‌خیزد و از طریق سیگنال‌دهی به دو سمافور، مواد مورد نیاز را فراهم می‌آورد.

کد عامل A

```
agentSem.wait()
tobacco.signal()
paper.signal()
```

کد عامل B

```
agentSem.wait()
paper.signal()
match.signal()
```

کد عامل C

```
agentSem.wait()
tobacco.signal()
match.signal()
```

این مسأله آنقدرها هم ساده نیست و راه‌حل طبیعی آن کار نمی‌کند. نوشتن کدی مانند زیر، وسوسه انگیز است:

سیگاری با کبریت

```
tobacco.wait()
paper.wait()
agentSem.signal()
```

سیگاری با تنباکو

```
paper.wait()
match.wait()
agentSem.signal()
```

سیگاری با کاغذ

```
1 tobacco.wait()
2 match.wait()
3 agentSem.signal()
```

مشکل این راه‌حل کجاست؟

۱.۵.۴ بن‌بست #۶

مشکل راه‌حل قبل امکان وقوع بن‌بست است. تصور کنید که عامل تنباکو و کاغذ عرضه می‌نماید. از آنجایی که فرد سیگاری که در دست خود کبریت دارد منتظر `tobacco` ممکن است رفع انسداد گردد. امّا آن فرد سیگاری که در دست خود تنباکو دارد منتظر `paper` است، لذا ممکن است او نیز رفع انسداد گردد (احتمال آن نیز زیاد است). سپس اولین نخ روی `paper` مسدود گردیده و دومی نیز روی `match` مسدود می‌گردد و بن‌بست!.

۲.۵.۴ راهنمایی مسأله سیگاری‌ها

راه‌حل Parnas، "سه نخ کمکی فروشندهٔ غیر مجاز"[۱۴] را به کار می‌برد که آن‌ها به سیگنال‌هایی که از عامل می‌رسد پاسخ می‌دهند، میزان موجود مواد مورد نیاز را نگه می‌دارند و به سیگاری مناسب سیگنال می‌دهند.

متغیرها و سمافورهای اضافی به شرح زیر است:

راهنمایی مسأله سیگاری‌ها

```
isTobacco = isPaper = isMatch = False
tobaccoSem = Semaphore(0)
paperSem = Semaphore(0)
matchSem = Semaphore(0)
```

متغیرهای بولی نشان می‌دهند که آیا یک ماده مورد نظر وجود دارد یا خیر. فروشنده‌ها `tobaccoSem` را بکار برده تا به آن فرد سیگاری که تنباکو در دست دارد سیگنال بدهد و به طرقی مشابه به سمافورهای دیگر.

[14] pusher

۳.۵.۴ راه حل مسأله سیگاری

کد یکی از فروشندگان در ادامه آمده است:

فروشنده A

```
tobacco.wait()
mutex.wait()
    if isPaper:
        isPaper = False
        matchSem.signal()
    elif isMatch:
        isMatch = False
        paperSem.signal()
    else:
        isTobacco = True
mutex.signal()
```

این فروشنده هر زمان که تنباکو موجود باشد فعال می‌شود. اگر مقدار isPaper برابر true باشد، می‌داند که فروشندهٔ B نیز در حال حاضر فعال است، لذا می‌تواند به آن فرد سیگاری که در دست خود کبریت دارد سیگنال دهد. به طور مشابه، اگر کبریت موجود باشد می‌تواند به آن فرد سیگاری که در دست خود کاغذ دارد سیگنال دهد.

اگر نسخت فروشنده A اجرا شود، سپس هر دوی isPaper و isMatch را false می‌بیند و نمی‌تواند به هیچ‌کدام از افراد سیگاری سیگنال دهد لذا مقدار isTobacco را true می‌نماید.

فروشنده‌های دیگر نیز چنین هستند. از آنجایی که تمام کار اصلی را فروشنده‌ها انجام می‌دهند، کد سیگاری بدیهی می‌شود.

فرد سیگاری با تنباکو

```
tobaccoSem.wait()
makeCigarette()
agentSem.signal()
smoke()
```

Parnas راه‌حل مشابهی ارائه می‌دهد که متغیرهای بولی را به صورت بیتی در یک متغیر صحیح ذخیره نموده است و سپس عدد صحیح را به عنوان اندیس آرایه‌ای از سمافورها بکار می‌بندد. با این شیوه، راه‌حل او از شرط (یکی از محدودیت‌های تصنعی) اجتناب می‌نماید. کد حاصل کمی خلاصه‌تر است، امّا عملکردش آنقدر واضح نیست.

۴.۵.۴ تعمیم مسأله سیگاری‌ها

Parnas پیشنهاد نمود که اگر عامل را به این صورت دستکاری کنیم که نیازی نباشد که عامل پس از گذاشتن مواد لازم صبر نماید، آنگاه مسأله سیگاری‌ها دشوارتر خواهد می‌گردد. در این حالت، باید چند نمونه از یک ماده روی میز موجود باشد.

معمّا: راه‌حل قبل را به گونه‌ای دستکاری کنید که با این تغییر مطابقت داشته باشد.

۵.۵.۴ راهنمای تعمیم مسأله سیگاری‌ها

اگر عامل، منتظر سیگاری‌ها نماند، ممکن است موارد لازم روی میز انباشته شود. بجای استفاده از مقادیر بولی به منظور ردگیری موارد لازم، به اعداد صحیح برای شمارش آن‌ها نیاز داریم.

راهنمای تعمیم مسأله سیگاری‌ها

```
numTobacco = numPaper = numMatch = 0
```

۶.۵.۴ راه حل تعمیم‌یافته مسأله سیگاری‌ها

کد تغییر یافته فروشنده A در ادامه آمده است:

A فروشنده

```
tobacco.wait()
mutex.wait()
    if numPaper:
        numPaper -= 1
        matchSem.signal()
    elif numMatch:
        numMatch -= 1
        paperSem.signal()
    else:
        numTobacco += 1
mutex.signal()
```

یک راه تصویرسازی این مسأله این است که تصویر نمایید آنزمان که عاملی اجرا می‌شود، دو فروشنده ساخته و به هر کدام از آن‌ها یکی از مواد لازم را می‌دهد و آن‌ها را همراه با سایر فروشندگان در یک اتاق قرار می‌دهد. به سبب میوتکس، فروشنده‌ها در اتاقی که سه سیگاری خوابیده و یک میز وجود دارد به ترتیب وارد می‌شوند. هر فروشنده یکی پس از دیگری وارد اتاق شده و مواد روی میز را بررسی می‌نماید. اگر او بتواند یک مجموعه کامل از مواد لازم سیگار را گرد آورد، آن‌ها از روی میز برداشته و سیگاری متناظر را بیدار می‌نماید. و اگر نتواند، مواد همراه خود را روی میز رها کرده و اتاق را بدون اینکه کسی را بیدار کند ترک می‌نماید.

این مثالی از الگویی است که آن را «جدول امتیاز»[۱۵] خوانده و بعداً چندین مرتبه آن را خواهیم دید. متغیرهای numPaper، numTobacco و numMatch وضعیت سیستم را نگاه می‌دارند. از آنجایی که هر نخ از طریق میوتکس به ترتیب وارد می‌شود، مثل اینکه به جدول امتیاز نگاه کرده باشد وضعیت را بررسی نموده و مطابق آن عکس العمل نشان می‌دهد.

[15] scoreboard

فصل ۵

مسائل همگام‌سازی کمتر-کلاسیک

۱.۵ مسأله غذاخوردن وحشی‌ها

این مسأله از **برنامه‌نویسی همروند** Andrews اقتباس شده است[؟].

> قبیله‌ای از وحشی‌ها، از یک دیگ بزرگ که می‌تواند M پرس ازمُبلّغ پخته شده را در خود نگاه دارد به صورت مشترک شام می‌خورند. زمانیکه یک وحشی می‌خواهد غذا بخورد، از درون دیگ از خود پذیرایی می‌کند مگر اینکه دیگ خالی باشد. اگر دیگ خالی بود، وحشی آشپز را بیدار نموده و سپس منتظر او می‌ماند تا دومرتبه دیگر را پر نماید.

هر تعداد نخ وحشی می‌تواند کد زیر را اجرا نماید:

کد ناهمگام یک وحشی

```
1 while True:
2     getServingFromPot()
3     eat()
```

و نخ یک آشپز کد زیر را اجرا می‌نماید:

کد ناهمگام آشپز

```
1 while True:
2     putServingsInPot(M)
```

محدودیت‌های همگام‌سازی عبارتند از:

- اگر دیگ خالی باشد وحشی‌ها نمی‌توانند `getServingFromPot` را فراخوانند.
- آشپز تنها در صورتی می‌تواند `putServingsInPot` را فراخواند که دیگ خالی باشد.

معمّا: کدی برای وحشی‌ها و آشپز اضافه نمایید که محدودیت‌های همگام‌سازی را برآورده نماید.

۱.۱.۵ راهنمایی غذاخوردن وحشی‌ها

همانند مسأله تولیدکننده–مصرف‌کننده در اینجا نیز وسوسه می‌شویم که برای نگهداری تعداد پرس‌ها از سمافور استفاده نماییم. امّا به منظور سیگنال‌دهی به آشپز، آن زمانی که دیگ خالی است، یک پیش از اینکه مقدار سمافور را کاهش دهد باید بداند که آیا باید منتظر بماند؟ و ما نمی‌توانیم چنین کاری کنیم.

یک جایگزین این است که جدول امتیاز را به منظور نگهداری تعداد پرس‌ها بکار بریم. اگر یک وحشی شمارنده را صفر بیابد، آشپز را بیدار نموده و منتظر دریافت سیگنال پرشدن دیگ می‌شود. متغیرهایی را که به کار برده‌ایم در ادامه مشاهده می‌نمایید:

Dining Savages hint

```
1 \begin{lstlisting}[title=\rl    }         {]{}
2 servings = 0
3 mutex = Semaphore(1)
4 emptyPot = Semaphore(0)
5 fullPot = Semaphore(0)
```

جای تعجب نیست که `emptyPot` نشانگر خالی بودن دیگ است و `fullPot` بیانگر پربودن دیگ است.

۲.۱.۵ راه حل غذاخوردن وحشی‌ها

راه‌حل در اینجا ترکیبی از الگوی جدول امتیاز با یک قرار ملاقات است. کد آشپز در ادامه آمده است.

راه حل غذاخوردن وحشی‌ها (آشپز)

```
while True:
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()
```

کد وحشی‌ها تنها کمی پیچیده‌تر است. از آنجایی که هر وحشی از میوتکس می‌گذرد، دیگ را بررسی می‌نماید. اگر دیگ خالی باشد، به آشپز سیگنال داده و منتظر می‌ماند و الا servings را کاهش داده و پرسی را از دیگ بر می‌دارد.

راه حل غذاخوردن وحشی‌ها (وحشی)

```
while True:
    mutex.wait()
        if servings == 0:
            emptyPot.signal()
            fullPot.wait()
            servings = M
        servings -= 1
        getServingFromPot()
    mutex.signal()

    eat()
```

اینکه وحشی بجای آشپز دستور servings = M را اجرا می‌کند شاید کمی عجیب بنظر آید. واقعاً ضرورتی به اینکار نبود، زمانی که آشپز putServingsInPot را اجرا می‌نماید، می‌دانیم آن وحشی که میوتکس را در اختیار دارد روی fullPot منتظر می‌ماند. لذا آشپز به servings دسترسی اَمنی دارد. امّا در این حالت، تصمیم گرفتم که وحشی این کار را انجام دهد چنانکه با نگاه به کد نیز واضح است که تمامی دسترسی‌های به servings درون میوتکس انجام می‌پذیرد.

این راه حل، بدون بن‌بست است. تنها امکان بن‌بست زمانی رخ می‌دهد که آن وحشی‌ای که mutex را نگاه داشته منتظر fullPot می‌ماند. زمانیکه او منتظر است، سایر وحشی‌ها روی mutex در صف انتظار قرار می‌گیرند. اما نهایتاً آشپز اجرا شده و به fullPot سیگنال می‌دهد، و این منجر به این می‌شود که وحشی منتظر ادامه یافته و میوتکس را آزاد نماید.

آیا این راه‌حل فرض نموده است است دیگ نخ-ایمن[۱] است و یا تضمین می‌نماید که putServingsInPot و getServingFromPot به صورت انحصاری انجام شوند.

[۱] در برنامه‌نویسی چند نخی، کدی را نخ-ایمن گوییم که تضمین نماید ساختمان داده‌های اشتراکی بدون دخالت‌های ناخواسته دیگر نخ‌ها، بروزرسانی شود.

۲.۵ مسأله آرایشگاه

مسأله اصلی آرایشگر بوسیله Dijkstra پیشنهاد شد. یک گونه دیگر آن در کتاب اصول سیستم‌های عامل Silberschatz و Galvin آمده است.

> یک آرایشگاه شامل یک صف انتظار با n صندلی و اتاق آرایشگر با صندلی آرایشگر است. اگر هیچ مشتری وجود نداشته باشد آرایشگر می‌خوابد. اگر یک مشتری وارد آرایشگاه شود و تمامی صندلی‌ها اشغال شده باشد، آنگاه مشتری مغازه را ترک می‌نماید. اگر آرایشگر مشغول باشد، اما صندلی موجود باشد، مشتری روی یکی از صندلی‌های خالی می‌نشیند. اگر آرایشگر خواب باشد، مشتری او را بیدار می‌نماید. برنامه‌ای بنویسید که آرایشگر و مشتری‌ها را هماهنگ نماید.

برای اینکه مسأله را کمی واقعی‌تر نماییم، اطلاعات زیر را به آن می‌افزاییم:

- نخ‌های مشتری باید تابعی به نام `getHairCut` را فراخوانند.
- اگر زمانیکه آرایشگاه پر است یک نخ مشتری برسد، می‌تواند `balk` را که هیچ مقداری بر نمی‌گرداند، فراخواند.
- نخ آرایشگر باید `cutHair` را فراخواند.
- زمانیکه آرایشگر `cutHair` را فرا می‌خواند، باید دقیقاً تنها یک نخ باشد که به طور همزمان `getHairCut` را فرا می‌خواند.

راه حلی ارائه دهید که شرایط فوق را تضمین نماید.

۱.۲.۵ راهنمایی آرایشگاه

راهنمایی آرایشگاه

```
n = 4
customers = 0
mutex = Semaphore(1)
customer = Semaphore(0)
barber = Semaphore(0)
customerDone = Semaphore(0)
barberDone = Semaphore(0)
```

n تعداد کل مشتری‌هایی است که می‌توانند در آرایشگاه باشند: سه نفر در اتاق انتظار و یک نفر روی صندلی آرایش.

customers تعداد مشتری‌های درون آرایشگاه را می‌شمرد و بوسیله mutex حفاظت می شود.

آرایشگر روی customer منتظر می‌ماند تا یک مشتری وارد شده و سپس مشتری روی barber می‌ماند تا زمانیکه آرایشگر به او سیگنال نشتن روی صندلی آرایش را بدهد.

پس از آرایش مو، مشتری به customerDone سیگنال می‌دهد و روی barberDone منتظر می‌ماند.

۲.۲.۵ راه حل آرایشگاه

این راه‌حل یک جدول امتیاز و دو قرار ملاقات را ترکیب می‌نماید. کد مشتری‌ها در ادامه آمده است.

راه حل آرایشگاه (مشتری)

```
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
mutex.signal()

customer.signal()
barber.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

اگر n مشتری در آرایشگاه وجود داشته باشد، هر مشتری که می‌رسد بلافاصله `balk` را فرا می‌خواند و الّا هر مشتری به `customer` سیگال داده و روی `barber` منتظر می‌ماند.

کد آرایشگر در ادامه آمده است.

راه حل آرایشگاه (آرایشگر)

```
customer.wait()
barber.signal()

# cutHair()

customerDone.wait()
barberDone.signal()
```

هر زمان که یک مشتری سیگنال می‌دهد، آرایشگر بیدار شده، به `barber` سیگنال می‌دهد، و مشغول آرایش یک نفر می‌شود. اگر آن زمانیکه آرایشگر مشغول است مشتری دیگری برسد، آنگاه در تکرار بعدی آرایشگر بدون اینکه بخوابد از سمافور `customer` می‌گذرد.

اسامی `customer` و `barber` بر مبنای قرارداد نامگذاری یک قرار ملاقات هستند، لذا `customer.wait()` به معنای "انتظار برای یک مشتری" است و نه اینکه "مشتری‌های در اینجا منتظرند".

قرار ملاقات دوم با استفاده از customerDone و barberDone، تضمین می‌نماید کار آرایش فعلی تمام شده باشد پیش از اینکه آرایشگر به ابتدای حلقه برگردد و به مشتری بعدی اجازه ورود به ناحیه بحرانی دهد.

این راه‌حل در sync_code/barber.py آمده است (ر.ک. ۲.۳).

۳.۵ آرایشگاه FIFO

در راه‌حل قبل تضمینی وجود ندارد که مشتریان به همان ترتیبی که می‌رسند سرویس دریافت کنند. تا سقف n مشتری می‌توانند از ترن‌استایل گذر کنند، به customer سیگنال داده، و روی barber منتظر بمانند. زمانیکه آرایشگر به barber سیگنال دهد، هر یک از مشتریان ممکن است ادامه دهد.

این راه‌حل را به گونه‌ای تغییر دهید که مشتریان به همان ترتیبی که از ترن‌استایل عبور می‌کنند سرویس دریافت نمایند.

راهنمایی: می‌توانید به نخ جاری به صورت self ارجاع دهید، لذا وقتی می‌نویسد self.sem = Semaphore(0)، هر نخ سمافور خودش را می‌گیرد.

۱.۳.۵ راهنمایی آرایشگاه FIFO

من از لیستی از سمافورها به نام queue در راه‌حل خود استفاده می‌کنم.

راهنمایی آرایشگاه FIFO

```
n = 4
customers = 0
mutex = Semaphore(1)
customer = Semaphore(0)
customerDone = Semaphore(0)
barberDone = Semaphore(0)
queue = []
```

زمانی‌که هر یک از نخ‌ها از ترن‌استایل عبور می‌کند، یک نخ ساخته و آن را در صف قرار می‌دهد.

به جای انتظار روی barber، هر نخ روی سمافور خودش منتظر می‌ماند. وقتیکه آرایشگر بیدار می‌شود، یک نخ را از صف خارج کرده و به آن سیگنال می‌دهد.

۲.۳.۵ راه حل آرایشگاه FIFO

در ادامه کد تغییریافتهٔ مشتریان آمده است:

راه حل آرایشگاه FIFO (مشتری)

```
self.sem = Semaphore(0)
mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue.append(self.sem)
mutex.signal()

customer.signal()
self.sem.wait()

# getHairCut()

customerDone.signal()
barberDone.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

و کد آرایشگر به این صورت است:

راه حل آرایشگاه FIFO (آرایشگر)

```
customer.wait()
mutex.wait()
    sem = queue.pop(0)
mutex.signal()

sem.signal()

# cutHair()

customerDone.wait()
barberDone.signal()
```

توجه نمایید که آرایشگر باید `mutex` را بگیرد تا به صف دسترسی داشته باشد.

این راه‌حل در `sync_code/barber2.py` آمده است (ر.ک. ۲.۳).

۴.۵ مسألهٔ آرایشگاه هیلزر

ویلیام استالینگز [؟] یک نسخهٔ پیچیده‌تر از مسألهٔ آرایشگاه را ارائه می‌دهد، که آن را مدیون رالف هیلزر در داشنگاه ایالتی کالیفرنیا در چیکو می‌داند.

> آرایشگاه ما سه تا صندلی، سه تا آرایشگر و اتاق انتظاری دارد که چهار مشتری می‌توانند روی یک کاناپه قرار گیرند و مابقی بایستند. طبق قوانین آتش‌نشانی تعداد کل مشتری‌های داخل آرایشگاه نباید از ۲۰ تا تجاوز نماید.
>
> اگر ظرفیت مشتری‌های داخل آرایشگاه تکمیل باشد، مشتری جدید وارد مغازه نخواهد شد. زمانیکه داخل است اگر روی کاناپه جا باشد می‌نشیند در غیر اینصورت می‌ایستد. زمانیکه یک آرایشگر آزاد باشد، آن مشتری که بیشترین زمان را روی کاناپه بوده سرویس دریافت می‌کند و اگر مشتریان ایستاده وجود داشته باشند آن فردی که مدت زمان بیشتری را در آرایشگاه بوده است جای او را روی کاناپه خواهد گرفت. زمانیکه آرایش یک مشتری تمام شد، هر آرایشگر می‌تواند اجرت را دریافت دارد، امّا از آنجایی که تنها یک صندوق وجود دارد در هر زمان تنها یک مشتری می‌تواند پرداخت خود را انجام دهد. آرایشگران زمان خود را بین آرایش، دریافت وجه و خوابیدن روی صندلی در انتظار مشتری تقسیم می‌نمایند.

به عبارت دیگر، محدودیت‌های همگام‌سازی زیر اعمال می‌شود:

- مشتریان توابع زیر را به ترتیب فرا می‌خوانند: `enterShop`، `sitOnSofa`، `getHairCut`، `pay`.
- آرایشگران `cutHair` و `acceptPayment` را فرا می‌خوانند.
- مشتریان اگر ظرفیت آرایشگاه پر باشد نمی‌توانند `enterShop` را فراخوانند.
- اگر کاناپه پر باشد، مشتری تازه وارد نمی‌تواند `sitOnSofa` را فراخواند.
- زمانیکه یک مشتری `getHairCut` را فرا می‌خواند متناظراً یک آرایشگر باید `cutHair` را به صورت همزمان اجرا نماید و بالعکس.
- فراخوانی `getHairCut` به صورت همزمان حداکثر توسط سه مشتری و اجرای `cutHair` به طور همزمان توسط حداکثر سه آرایشگر ممکن باشد.
- مشتری باید قبل از اینکه آرایشگر بتواند `acceptPayment` را فراخواند، `pay` را اجرا نماید.
- آرایشگر قبل از خروج مشتری باید `acceptPayment` را اجرا نماید.

معمّا: کدی بنویسید که محدودیت‌های همگام‌سازی آرایشگاه هیلزر را اعمال نماید.

۱.۴.۵ راهنمایی آرایشگاه هیلزر

متغیرهایی که در راه‌حل بکار رفته در ادامه آمده است:

راهنمایی آرایشگاه هیلزر

```
n = 20
customers = 0
mutex = Semaphore(1)
sofa = Semaphore(4)
customer1 = Semaphore(0)
customer2 = Semaphore(0)
barber = Semaphore(0)
payment = Semaphore(0)
receipt = Semaphore(0)
queue1 = []
queue2 = []
```

mutex از customers که تعداد مشتری‌های درون آرایشگاه را در خود دارد و از queue1 که لیست سمافورهای نخ‌های منتظر برای نشستن روی کاناپه است حفاظت می‌نماید.

mutex2 از queue2 که لیست سمافورهای نخ‌های منتظر صندلی است حفاظت می‌کند.

sofa یک مالتی‌پلکس است که حداکثر تعداد مشتری‌های روی کاناپه را اعمال می‌کند.

customer1 پیغام می‌دهد که یک مشتری در queue1 وجود دارد و customer2 پیغام می‌دهد که یک مشتری در queue2 وجود دارد.

payment پیغام می‌دهد که یک مشتری پرداخت داشته است و receipt پیغام می‌دهد که آرایشگر اجرت را دریافت کرده است.

۲.۴.۵ راه حل آرایشگاه هیلزر

این راه‌حل به طور قابل توجهی از آنچه انتظار داشتم پیچیده‌تر است. شاید در ذهن هیلزر راه‌حل ساده‌تری وجود داشته است لکن این بهترین چیزی است که می‌توانستم ارائه دهم.

راه حل آرایشگاه هیلزر (مشتری)

```
self.sem1 = Semaphore(0)
self.sem2 = Semaphore(0)

mutex.wait()
    if customers == n:
        mutex.signal()
        balk()
    customers += 1
    queue1.append(self.sem1)
mutex.signal()

# enterShop()
customer1.signal()
self.sem1.wait()

sofa.wait()
    # sitOnSofa()
    self.sem1.signal()
    mutex.wait()
        queue2.append(self.sem2)
    mutex.signal()
    customer2.signal()
    self.sem2.wait()
sofa.signal()

# sitInBarberChair()

# pay()
payment.signal()
receipt.wait()

mutex.wait()
    customers -= 1
mutex.signal()
```

اولین پاراگراف مشابه راه‌حل قبلی است. زمانیکه یک مشتری می‌رسد، شمارنده را بررسی نموده آنگاه یا از ورود امتناع ورزیده و یا خودش را به صف می‌افزاید. سپس به آرایشگر سیگنال می‌دهد.

زمانیکه مشتری از صف خارج می‌شود، وارد مالتی‌پلکس می‌گردد، روی مبل می‌نشیند و خود را به صف دوم

افزاید.

زمانیکه از آن صف خارج می‌شود، آرایش شده، پرداخت انجام داده و خارج می‌شود.

راه حل آرایشگاه هیلزر (آریشگر)

```
customer1.wait()
mutex.wait()
    sem = queue1.pop(0)
    sem.signal()
    sem.wait()
mutex.signal()
sem.signal()

customer2.wait()
mutex.wait()
    sem = queue2.pop(0)
mutex.signal()
sem.signal()

barber.signal()
# cutHair()

payment.wait()
# acceptPayment()
receipt.signal()
```

هر آرایشگر منتظر یک مشتری می‌ماند تا وارد شده، به سمافور مشتری سیگنال می‌دهد تا او را از صف خارج نماید، سپس منتظر او می‌ماند تا درخواست نشستن روی کاناپه بدهد. این روند، نیاز FIFO را برآورده می‌کند.

آرایشگر منتظر مشتری می‌ماند تا به صف دوم ملحق شود و سپس با سیگنال دادن به او اجازه می‌دهد که یک صندلی را مطالبه کند.

هر آرایشگر تنها به یک مشتری اجازه می‌دهد که روی صندلی بنشیند، لذا حداکثر تا سه آرایش همزمان می‌تواند صورت گیرد. از آنجایی که تنها یک صندوق وجود دارد، مشتری باید `mutex` را بگیرد. مشتری و آرایشگر نزد صندوق قرارا ملاقات گذاشته و سپس هر دو خارج می‌شوند.

این راه حل، محدودیت‌های همگام‌سازی را برآورده می‌نماید، امّا نهایت بهره‌برداری را از کاناپه نمی‌کند. از آنجایی که تنها سه آرایشگر وجود دارد، هیچگاه بیش از سه مشتری نمی‌تواند روی کاناپه وجود داشته باشد، لذا ضرورتی به مالتی‌پلکس وجود ندارد.

این راه‌حل در `sync_code/barber3.py` آمده است (ر.ک. ۲.۳).

تنها راهی که برای حل مسأله به ذهنم می‌رسد این است که یک نوع سومی از نخ ایجاد نمایم که من آن را راهنما می‌نامم. راهنمایان `queue1` مدیریت نموده و آرایشگران `queue2` را مدیریت می‌نمایند. اگر چهار راهنما و سه آرایشگر وجود داشته باشد کاناپه می‌تواند به طور کامل مورد استفاده قرار گیرد.

این راه‌حل در sync_code/barber4.py آمده است (ر.ک. ۲.۳).

۵.۵ مسأله بابا نوئل

این مسأله از کتاب سیستم‌های عامل ویلیام استالینگز گرفته شده است [؟]، امّا او در این مسأله نیز خود را مدیون John Trono از کالج Michael در ورمُنت می‌داند.

> بابا نوئل در مغازه خود در قطب شمال می‌خوابد و فقط در صورتی بیدار می‌شود که یا (۱) تمام نه گوزن از تعطیلات خود اقیانوس آرام جنوبی باز گردند یا (۲) برخی از پری‌ها در ساخت اسباب‌بازی‌ها مشکل داشته باشند؛ به منظور اینکه اجازه دهیم بابا نوئل کمی بخوابد، پری‌ها تنها در صورتی می‌توانند بابا نوئل را بیدار نمایند که سه تا از آن‌ها با مشکل مواجه شوند. زمانیکه سه پری مشکلشان حل شود، هر پری دیگری که آرزوی دیدن بابا نوئل را دارد باید صبر کند تا آن پری‌ها بازگردند. اگر بابا نوئل بیدار شود و سه پری را پشت در مغازه‌اش منتظر بیابد و همچنین دریابد که آخرین گوزن شمالی دوباره از مناطق گرمسیری آمده است، بابا نوئل تصمیم می‌گیرد که پری‌ها می‌توانند تا پس از کریسمس منتظر بمانند زیرا که از آن مهمتر، این است که سورتمه خود را آماده کند. (فرض شده است که گوزن شمالی نمی‌تواند مناطق گرمسیری را ترک کند و لذا آن‌ها تا آخرین لحظه ممکن در آنجا می‌مانند.) آخرین گوزن شمالی که می‌رسد باید بابا نوئل را ببرد در حالیکه گوزن‌های دیگر در یک کلبه گرم پیش از اینکه به سورتمه بسته شوند منتظر هستند.

در اینجا تعدادی مشخصه‌های اضافی آمده است:

- پس از اینکه نهمین گوزن رسید، بابا نوئل باید `prepareSleigh` را فراخواند و سپس تمامی گوزن‌ها `getHitched` را فراخوانند.
- پس از اینکه سومین پری می‌رسد، بابا نوئل باید `helpElves` را فراخواند. به صورت همزمان، سه پری نیز باید `getHelp` فراخوانند.
- هر سه پری پیش از اینکه پری دیگری وارد شود (شمارنده پری‌ها را افزایش دهد) باید `getHelp` را فراخوانند.

بابا نوئل باید در یک حلقه اجرا شود لذا او می‌تواند به مجموعه بسیاری از پری‌ها کمک کند. می‌توانیم تصور نماییم که دقیقا ۹ گوزن شمالی وجود دارد، امّا هر تعداد از پری ممکن است.

۱.۵.۵ راهنمایی مسأله بابا نوئل

راهنمایی مسأله بابا نوئل

```
elves = 0
reindeer = 0
santaSem = Semaphore(0)
reindeerSem = Semaphore(0)
elfTex = Semaphore(1)
mutex = Semaphore(1)
```

elves و reindeer شمارنده‌هایی هستند که بوسیله mutex محافظت می‌شوند. پری‌ها و گوزن‌ها از mutex برای تغییر شمارنده‌ها استفاده می‌کنند؛ بابا نوئل نیز آن را می‌گیرد تا متغیرها را بررسی نماید.

بابا نوئل روی santaSem منتظر می‌ماند تا یا یک پری یا یک گوزن به او سیگنال دهد.

گوزن‌ها روی reindeerSem منتظر می‌مانند تا بابا نوئل به آن‌ها سیگنال دهد که به چراگاه وارد شده و به سورتمه بسته شوند.

پری‌ها elfTex را بکار می‌برند تا از ورود پری اضافی، آن زمانی که سه پری در حال گرفتن کمک هستند جلوگیری نمایند.

۲.۵.۵ راه حل مسأله بابا نوئل

کد بابا نوئل بسیار آسان است. به یاد داشته باشید که کد او همیشه در حلقه اجرا می‌شود.

راه حل مسأله بابا نوئل (بابا نوئل)

```
santaSem.wait()
mutex.wait()
    if reindeer >= 9:
        prepareSleigh()
        reindeerSem.signal(9)
        reindeer -= 9
    else if elves == 3:
        helpElves()
mutex.signal()
```

زمانیکه بابا نوئل بیدار می‌شود، بررسی می‌نماید کدامیک از دو شرط بر قرار است و متناسب با آن با گوزن‌ها و یا با پری‌های منتظر تعامل می‌نماید. اگر ۹ گوزن در حال انتظار باشند، بابا نوئل `prepareSleigh` را فراخوانده و سپس نه بار به `reindeerSem` سیگنال می‌دهد تا به گوزن‌ها اجازه دهد که `getHitched` را فراخوانند. اگر پری‌های در حال انتظاری وجود داشته باشد، بابا نوئل فقط `helpElves` را فرا می‌خواند. هیچ نیازی به این نیست که پری‌ها منتظر بابا نوئل شوند؛ زمانیکه آن‌ها به `santaSem` سیگنال می‌دهند می‌توانند بلافاصله `getHelp` را فراخوانند.

بابا نوئل نباید شمارنده `elves` را کاهش دهد زیرا که پری‌ها در راه خروج‌شان اینکار را انجام می‌دهند.

کد گوزن‌ها در ادامه آمده است:

راه حل مسأله بابا نوئل (گوزن‌ها)

```
mutex.wait()
    reindeer += 1
    if reindeer == 9:
        santaSem.signal()
mutex.signal()

reindeerSem.wait()
getHitched()
```

گوزن نهم به بابا نوئل سیگنال می‌دهد و به گوزن‌های دیگر که روی `reindeerSem` منتظر هستند ملحق می‌گردد. زمانیکه بابا نوئل سیگنال می‌دهد تمامی گوزن‌ها `getHitched` را اجرا می‌نمایند .

کد پری‌ها نیز مشابه است، بجز اینکه زمانیکه سومین پری می‌رسد باید ورود پری بعدی را مانع شود تا آن هنگامی که سه تای اول `getHelp` را اجرا نمایند.

راه حل مسأله بابا نوئل (پری‌ها)

```
elfTex.wait()
mutex.wait()
    elves += 1
    if elves == 3:
        santaSem.signal()
    else
        elfTex.signal()
mutex.signal()

getHelp()

mutex.wait()
    elves -= 1
    if elves == 0:
       elfTex.signal()
mutex.signal()
```

اولین دو پری در همان زمانی که mutex را آزاد می‌نمایند elfTex را نیز آزاد می‌نمایند، امّا آخرین پری elfTex را نگاه می‌دارد که مانع از ورود پری‌های دیگر می‌گردد تا زمانیکه تمامی سه پری getHelp را فراخوانند. آخرین پری که خارج می‌شود elfTex را آزاد می‌نماید و این به دسته بعدی پری‌ها اجازه ورود می‌دهد.

۶.۵ ساخت H_2O

این مسأله برای حداقل یک دهه جزء اصلی کلاس سیستم عامل در U.C. Berkeley بود. بنظر می‌رسد که بر مبنای تمرینی در کتاب «برنامه‌نویسی همروند» اندرو می‌باشد [؟].

دو نوع نخ وجود دارد: اکسیژن و هیدروژن. به منظور ترکیب این نخ‌ها به مولکول‌های آب، باید حصاری بسازیم که هر نخ تا زمانیکه یک مولکول کامل آماده ادامه باشد منتظر بماند.

هر نخ که از حصار عبور می‌نماید، باید bond فراخواند. شما باید تضمین نمایید که تمامی نخ‌های یک مولکول پیش از نخ‌های مولکول بعدی bond را فراخوانند.

به عبارت دیگر:

- اگر زمانیکه یک نخ اکسیژن به حصار می‌رسد هیچ نخ هیدروژنی حاضر نباشد، باید برای دو نخ هیدروژن منتظر بماند.
- اگر زمانیکه یک نخ هیدروژن به حصار می‌رسد هیچ نخ دیگری حاضر نباشد، باید منتظر یک نخ اکسیژن و یک نخ دیگر هیدروژن بماند.

نباید نگران تطابق صریح نخ‌ها باشیم، بدین معنی که نخ‌ها از اینکه با چه نخ‌های دیگری جفت می‌شوند ضرورتاً اطلاعی ندارند. نکته کلیدی تنها این است نخ‌ها به صورت مجموعه‌های کامل از حصار عبور می‌نمایند؛ بنابراین اگر ما دنباله نخ‌هایی که bond را فرا می‌خوانند بررسی کنیم و آن‌ها را به گروه‌های سه‌تایی تقسیم کنیم هر گروه باید شامل یک نخ اکسیژن و دو نخ هیدروژن باشد.

معمّا: یک کد همگام‌سازی برای مولکول‌های اکسیژن و هیدروژن بنویسید که این شرایط را برآورده نمایند.

۱.۶.۵ راهنمایی H_2O

متغیرهایی که در راه‌حل بکار برده‌ام را در زیر مشاهده می‌نمایید:

راهنمایی ساخت آب

```
mutex = Semaphore(1)
oxygen = 0
hydrogen = 0
barrier = Barrier(3)
oxyQueue = Semaphore(0)
hydroQueue = Semaphore(0)
```

oxygen و hydrogen شمارنده‌هایی هستند که بوسیله mutex حفاظت می‌شوند. barrier جایی است که هر مجموعه‌ای از سه نخ پس از فراخوانی bond و پیش از اجازه فعالیت به نخ‌های بعدی، یکدیگر را ملاقات می‌کنند.

oxyQueue سمافوری است که نخ‌های اکسیژن روی آن منتظر می‌مانند؛ hydroQueue سمافوری است که نخ‌های هیدروژن روی آن منتظر می‌مانند؛ از آنجایی که از قرارداد نام‌گذاری برای صف‌ها استفاده می‌نمایم، لذا oxyQueue.wait() بمعنای «به صف اکشیژن محلق شو» است و oxyQueue.signal() بمعنای «یک نخ اکسیژن از صف آزاد نما» می‌باشد.

۲.۶.۵ راه حل H_2O

در ابتدا hydroQueue و oxyQueue قفل هستند. زمانیکه یک نخ اکسیژن می‌رسد دو بار به hydroQueue سیگنال می‌دهد تا به دو هیدروژن اجازه ادامه کار دهد. سپس نخ اکسیژن منتظر نخ‌های هیدروژن می‌ماند تا برسند.

کد اکسیژن

```
1  mutex.wait()
2  oxygen += 1
3  if hydrogen >= 2:
4      hydroQueue.signal(2)
5      hydrogen -= 2
6      oxyQueue.signal()
7      oxygen -= 1
8  else:
9      mutex.signal()
10
11 oxyQueue.wait()
12 bond()
13
14 barrier.wait()
15 mutex.signal()
```

هر نخ اکسیژن که وارد می‌شود، میوتکس را می‌گیرد و جدول امتیاز را بررسی می‌نماید. اگر حداقل دو نخ هیدروژن منتظر وجود داشته باشد، به دو تای آن‌ها و خودش سیگنال می‌دهد و سپس با هم پیوند شیمیایی برقرار می‌نمایند. اگر دو نخ هیدروژن وجود نداشته باشد، میوتکس را آزاد نموده و منتظر می‌ماند.

پس از پیوند (خط ۱۲)، نخ‌ها نزد حصار منتظر می‌مانند تا تمامی هر سه نخ با هم تشکیل پیوند دهد و سپس نخ اکسیژن میوتکس را آزاد می‌نماید. از آنجایی که تنها یک نخ اکسیژن در هر مجموعه وجود دارد، تضمین می‌شود که به mutex تنها یک بار سیگنال داده می‌شود.

کد هیدروژن نیز مشابه است:

کد هیدروژن

```
1  mutex.wait()
2  hydrogen += 1
3  if hydrogen >= 2 and oxygen >= 1:
4      hydroQueue.signal(2)
5      hydrogen -= 2
6      oxyQueue.signal()
7      oxygen -= 1
8  else:
9      mutex.signal()
10
```

```
11 hydroQueue.wait()
12 bond()
13
14 barrier.wait()
```

یک ویژگی غیرمعمول این راه‌حل این است که نقطه خروج از میوتکس مبهم است. در برخی حالات، نخ‌ها وارد میوتکس شده، شمارنده را بروزرسانی نموده و از میوتکس خارج می‌شوند. امّا زمانیکه آن نخ تشکیل دهنده یک مجموعه کامل می‌رسد، باید به منظور ممانعت از نخ‌های دیگر میوتکس را نگه دارد تا زمانیکه مجموعه فعلی bond را فراخواند.

پس از فراخوانی bond، سه نخ نزد حصار منتظر می‌مانند. زمانیکه حصار باز می‌شود، می‌دانیم که تمامی سه نخ bond را فراخوانده‌اند و یکی از آن‌ها میوتکس را نگه داشته است. نمی‌دانیم که کدام نخ میوتکس را نگه داشته است امّا از آنجایی تنها یکی از آن‌ها میوتکس را آزاد می‌نماید دانستنش اهمیتی ندارد. با توجه به اینکه می‌دانیم تنها یک نخ اکسیژن وجود دارد، آن را قادر به انجام اینکار می‌سازیم.

ممکن است این راه‌حل نادرست به نظر آید، زیرا که تا کنون عموماً اینگونه درست بود که یک نخ باید قفلی را نگه دارد تا بتواند آن را آزاد نماید. امّا هیچ قاعده‌ای نمی‌گوید که این نکته باید درست باشد. اینجا یکی از آن حالت‌هایی است که نگاه به میوتکس به عنوان توکنی که نخ‌ها باید آن گرفته و آزاد نمایند کمی گمراه کننده است.

۷.۵ مسأله عبور از رودخانه

این مسأله از مجموعه مسائل نوشته توسط انتونی ژوزف[۲] در دانشگاه برکلی[۳] است، امّا اینکه نویسندهٔ اصلی خود او است یا نه را نمی‌دانم. این مسأله از این جنبه که یک گونه خاص از حصار در آن وجود دارد که تنها در ترکیب‌های معینی، به نخ‌ها اجازه عبور می‌دهد مشابه مسأله H_2O است.

یک جایی نزدیک ردموند[۴] ایالت واشنگتن، یک قایق پارویی وجود دارد که بوسیله هر دوی هکرهای لینوکس و کارمندان مایکروسافت برای عبور از یک رودخانه بکار می‌رود. قایق دقیقاً چهار نفر در خود جا می‌دهد و ساحل رودخانه را با تعداد بیشتر یا کمتری ترک نخواهد کرد. به منظور تضمین امنیت مسافران، همنشینی یک هکر با سه کارمند مایکروسافت مجاز نمی‌باشد و بالعکس. هر ترکیب دیگری امن است.

هر نخ که سوار قایق می‌شود باید تابع board را صدا زند. تضمین نمایید که تمامی چهار نخ سوار بر قایق، board را پیش از هر نخ دیگری که متعلق به سری بعدی است صدا زند.

پس از اینکه هر چهار نخ board را صدا زدند، دقیقاً یکی از آن‌ها باید تابع rowBoat را فراخواند تا نشان دهد که آن نخ پاروها را خواهد گرفت. اینکه کدام نخ این تابع را صدا می‌زند تا آن زمانیکه یکی اینکار انجام می‌دهد اهمیتی ندارد.

[2] Anthony Joseph [3] U.C. Berkeley [4] Redmond

نگران جهت حرکت سفر نباشید. فرض کنید که تنها رفت آمد از یک جهت مورد توجه ما است.

۱.۷.۵ راهنمایی عبور از رودخانه

متغیرهایی که در حل مسأله بکار برده‌ام در ادامه آمده است:

راهنمایی عبور از رودخانه

```
1 barrier = Barrier(4)
2 mutex = Semaphore(1)
3 hackers = 0
4 serfs = 0
5 hackerQueue = Semaphore(0)
6 serfQueue = Semaphore(0)
7 local isCaptain = False
```

hackers و serfs تعداد هکرها و کارمندان منتظر سوار شدن را می‌شمرد. از آنجایی که هر دو اینها بوسیله mutex محافظت می‌شوند، می‌توانیم شرایط هر دو متغیر را بدون نگرانی درباره بروزرسانی نابهنگام بررسی نماییم. این یک مثال دیگری از یک جدول امتیاز است.

hackerQueue و serfQueue به ما اجازه می‌دهند که تعداد هکرها و کارمندانی که گذر می‌نمایند را کنترل نماییم. barrier تضمین می‌کند که تمامی چهار نخ، پیش از اینکه کاپیتان rowBoat را فراخواند board را فراخوانند.

isCaptain یک متغیر محلی است که نشان می‌دهد کدام نخ باید rowBoat را فراخواند.

۲.۷.۵ راه حل عبور از رودخانه

ایده اصلی این راه‌حل این است که هر نخ ورودی یکی از شمارنده‌ها را بروزرسانی نموده و سپس بررسی می‌نماید که آیا مجموعه را تکمیل می‌نماید، خواه اینکه چهارمین نخ از همنوعان خودش باشد و یا اینکه یک جفت ترکیبی از جفت‌های ممکن را تکمیل نماید.

کد هکرها را ارائه خواهم کرد؛ کد کارمندان مایکروسافت هم قرینه آن است (البته بجز اینکه ۱۰۰۰ بار بزرگتر، پر از باگ، و شامل یک مرورگر توکار است.)

راه حل عبور از رودخانه

```
mutex.wait()
    hackers += 1
    if hackers == 4:
        hackerQueue.signal(4)
        hackers = 0
        isCaptain = True
    elif hackers == 2 and serfs >= 2:
        hackerQueue.signal(2)
        serfQueue.signal(2)
        serfs -= 2
        hackers = 0
        isCaptain = True
    else:
        mutex.signal()      # captain keeps the mutex

hackerQueue.wait()

board()
barrier.wait()

if isCaptain:
    rowBoat()
    mutex.signal()          # captain releases the mutex
```

از آنجایی که هر نخ از طریق بخش انحصار متقابل به ترتیب وارد می‌شود، بررسی می‌نماید که یک خدمه کامل آماده سوار شدن قایق هست یا خیر؟ اگر چنین بود، به نخ‌های متناسب سیگنال داده، خودش را به عنوان کاپیتان معرفی نموده، و میوتکس می‌گیرد تا مانع نخ‌های اضافی شود تا آن زمان که قایق رانده شود.

تعداد نخ‌هایی که سوار شده‌اند را حصار نگاه می‌دارد. زمانیکه آخرین نخ می‌رسد، تمامی نخ‌ها با هم ادامه می‌یابند. کاپیتان `rowBoat` را فراخوانده و در نهایت میوتکس را آزاد می‌نماید.

۸.۵ مسأله ترن هوایی

این مسأله از کتاب «برنامه‌نویسی همروند» اندرو[۵] اقتباس شده است، امّا او این مسأله را متعلق به تز کارشناسی‌ارشد جی.اس. هرمن[۶] می‌داند.

> فرض کنید n نخ رهگذر و یک ماشین وجود دارد. رهگذران پی‌درپی منتظر سوار شدن ماشین هستند؛ ماشینی که گنجایش C رهگذر را دارد و $C < n$. این ماشین زمانی می‌تواند در مسیر حرکت کند که پر باشد.

جزئیات بیشتر در ادامه آمده است:

- رهگذران باید `board` و `unboard` فرا خوانند.
- ماشین باید `load`، `run` و `unload` را فراخواند.
- رهگذران نمی‌توانند سوار شوند مگر اینکه ماشین `load` را فراخوانده باشد.
- ماشین نمی‌تواند حرکت کند مگر اینکه C رهگذر سوار شده باشند.
- مسافران نمی‌تواند پیاده شوند جز اینکه ماشین `unload` را فرخوانده باشد.

معمّا: کدی برای رهگذران و ماشین بنویسید که این محدودیت‌ها را اعمال نماید:

[5] Andrews [6] J. S. Herman

۱.۸.۵ راهنمایی ترن هوایی

راهنمایی ترن هوایی

```
mutex = Semaphore(1)
mutex2 = Semaphore(1)
boarders = 0
unboarders = 0
boardQueue = Semaphore(0)
unboardQueue = Semaphore(0)
allAboard = Semaphore(0)
allAshore = Semaphore(0)
```

mutex از passengers محافظت می‌نماید- تعداد مسافرانی را که boardCar را فراخونده‌اند می‌شمارد

.

رهگذران پیش از سوار شدن روی boardQueue و پیش پیاده‌شدن روی unboardQueue منتظر می‌مانند.

allAboard نشان می‌دهد که ماشین پر است. allAshore نشان می‌دهد که ماشین خالی شده است.

۲.۸.۵ راه حل ترن هوایی

کد نخ ماشین در ادامه آمده است:

راه حل ترن هوایی (ماشین)

```
load()
boardQueue.signal(C)
allAboard.wait()

run()

unload()
unboardQueue.signal(C)
allAshore.wait()
```

زمانیکه ماشین می‌رسد، به C رهگذر سیگنال می‌دهد، سپس منتظر آخرین رهگذر می‌شود که به `allAboard` سیگنال می‌دهد. پس از حرکت، به C اجازه پیاده‌شدن را می‌دهد و سپس منتظر `allAshore` می‌گردد.

راه حل ترن هوایی (رهگذر)

```
boardQueue.wait()
board()

mutex.wait()
   boarders += 1
   if boarders == C:
       allAboard.signal()
       boarders = 0
mutex.signal()

unboardQueue.wait()
unboard()

mutex2.wait()
   unboarders += 1
   if unboarders == C:
       allAshore.signal()
       unboarders = 0
mutex2.signal()
```

طبیعتاً مسافران پیش از سوارشدن منتظر ماشین می‌شوند و پیش از ترک آن نیز منتظر توقف ماشین می‌مانند. آخرین مسافری که پیاده می‌شود به ماشین سیگنال می‌دهد و شمارنده مسافران را ریست می‌نماید.

۹.۵ مسأله ترن هوایی چند ماشینی

راه‌حل ارائه شده، قابل تعمیم به حالتی که بیش از یک ماشین وجود داشته باشد نیست. به منظور انجام اینکار، باید تعدادی محدودیت اضافی را برآورده نماییم:

- تنها یک ماشین در هر لحظه می‌تواند مسافرگیری نماید.
- چندین ماشین به صورت همزمان می‌توانند روی مسیر باشند.
- از آنجایی که ماشین‌ها نمی‌توانند از یکدیگر سبقت بگیرند، باید به همان ترتیبی که مسافرگیری نموده‌اند، مسافران پیاده نمایند.
- تمام نخ‌های سوار یک ماشین باید پیش از سوارشدن نخ‌های بعدی از ماشین پیاده شوند.

معمّا: راه‌حل قبلی را چنان تغییر دهید که محدودیت‌های اضافی را نیز اعمال کند. می‌توانید فرض کنید که m ماشین وجود دارد و هر کدام یک متغیر محلی به نام `i` دارند که محتوی شناسه‌ای بین ۰ و $m - ۱$ است.

۱.۹.۵ راهنمایی ترن هوایی چند ماشینی

دو لیست از سمافورها را به منظور نگهداشتن ترتیب ماشین‌ها بکار برده‌ام. یکی از این لیست‌ها نشانگر ناحیه مسافرگیری و لیست دیگر نشانگر ناحیه پیاده‌کردن مسافران است. هر لیست شامل یک سمافور بازای هر ماشین است. در هر لحظه، فقط یک سمافور از هر لیست باز است، لذا به این وسیله می‌توانیم، ترتیب را در سوارکردن یا پیاده‌کردن نخ‌ها اعمال کنیم. در ابتدا، تنها سمافورهای ماشین ۰ باز هستند. هر ماشین که وارد محل سوارکردن (پیاده کردن) می‌شود، روی سمافور خودش منتظر می‌ماند؛ و هر زمان که خارج می‌شود به ماشین بعدی در خط سیگنال می‌دهد.

راهنمایی ترن هوایی چند ماشینی

```
1 loadingArea = [Semaphore(0) for i in range(m)]
2 loadingArea[0].signal()
3 unloadingArea = [Semaphore(0) for i in range(m)]
4 unloadingArea[0].signal()
```

تابع `next` شناسه ماشین بعدی در دنباله را محاسبه می‌نماید (بعد از $m-1$ به ۰ باز می‌گردد):

پیاده‌سازی `next`

```
1 def next(i):
2     return (i + 1) % m
```

۲.۹.۵ راه حل ترن هوایی چند ماشینی

کد تغییر یافته ماشین در ادامه آمده است:

راه حل ترن هوایی چند ماشینی (ماشین)

```
loadingArea[i].wait()
load()
boardQueue.signal(C)
allAboard.wait()
loadingArea[next(i)].signal()

run()

unloadingArea[i].wait()
unload()
unboardQueue.signal(C)
allAshore.wait()
unloadingArea[next(i)].signal()
```

کد رهگذران بدون تغییر باقی می‌ماند.

فصل ۶

مسائل نه-چندان-کلاسیک

۱.۶ مسأله جستجو-درج-حذف

این مسأله از کتاب «برنامه‌نویسی همروند» اندرو است [؟].

> سه نوع نخ به یک لیست پیوندی یکطرفه به صورت مشترک دسترسی دارند: نخ‌های جستجوگر، درج کننده و حذف کننده. جستجوگرها تنها لیست را بررسی می‌نمایند؛ بنابراین این نوع نخ‌ها می‌توانند به صورت همزمان با یکدیگر عمل کنند. درج کننده‌ها عناصر جدید را به انتهای لیست می‌افزایند؛ درج‌ها باید به صورت انحصاری انجام شود تا از درج همزمان عناصر جدید توسط دو درج کننده ممانعت بعمل آید. با اینوجود یک درج می‌تواند به موازات هر تعداد جستجو انجام شود. در نهایت حذف‌کننده‌ها، عناصر از هر جایی در لیست حذف می‌کنند. حداکثر یک فرآینده حذف کننده در هر زمان می‌تواند به لیست دسترسی داشته باشد، و همچنین حذف باید به صورت انحصاری و بدون حضور جستجوگرها و درج کننده‌ها انجام شود.

معمّا: کدی برای جستجوگرها، درج کننده‌ها و حذف کننده‌ها بنویسید که این نوع از انحصار متقابل دسته‌ای سه‌گانه را اعمال نماید.

۱.۱.۶ راهنمایی جستجو-درج-حذف

راهنمایی جستجو-درج-حذف

```
insertMutex = Semaphore(1)
noSearcher = Semaphore(1)
noInserter = Semaphore(1)
searchSwitch = Lightswitch()
insertSwitch = Lightswitch()
```

`insertMutex` تضمین می‌نماید که در یک زمان تنها یک درج کننده در ناحیه بحرانی خودش است. `noSearcher` و `noInserter` نشان می‌دهند که هیچ جستجوگر و درج کننده‌ای در نواحی بحرانی خودشان قرار ندارند؛ یک حذف کننده باید هر دوی اینها را نگاه دارد تا بتواند وارد شود.

`searchSwitch` و `insertSwitch` توسط جستجوگرها و درج کننده‌ها برای بیرون نگاه داشتن حذف کننده استفاده می‌شود.

۲.۱.۶ راه حل جستجو-درج-حذف

راه‌حل در ادامه آمده است:

[illegible] (searchSwitch.wait(noSearcher)

```
searchSwitch.wait(noSearcher)
# critical section
searchSwitch.signal(noSearcher)
```

تنها چیزی که یک جستجوگر باید نگران آن باشد حذف کننده است. اولین جستجوگری که وارد می‌شود noSearcher را گرفته و آخرینی که خارج می‌شود آن را آزاد می‌نماید.

[illegible] (insertSwitch.wait(noInserter)

```
insertSwitch.wait(noInserter)
insertMutex.wait()
# critical section
insertMutex.signal()
insertSwitch.signal(noInserter)
```

مشابهاً، اولین درج کننده noInserter گرفته و آخرینی که خارج می‌شود آن را آزاد می‌نماید. از آنجایی که جستجوگرها و درج کننده‌ها برای سمافورهای متفاوتی رقابت می‌کنند، می‌توانند به صورت همزمان در ناحیه بحرانی خودشان باشند. امّا insertMutex تضمین می‌نماید که تنها یک درج کننده در یک لحظه در ناحیه بحرانی باشد.

[illegible] (noSearcher.wait()

```
noSearcher.wait()
noInserter.wait()
# critical section
noInserter.signal()
noSearcher.signal()
```

از آنجایی که حذف کننده هر دوی noSearcher و noInserter را می‌گیرد، دسترسی انحصاری تضمین شده است. البته، هر زمانی که ببینیم نخی بیش از یک سمافور را گرفته است، باید بن‌بست بررسی نماییم. با بررسی نمودن چند سناریو، شما باید قادر باشید که خودتان را متقاعد نمایید که این راه‌حل عاری از بن بست است.

به عبارت دیگر، مانند بسیاری از مسائل انحصار متقابل دسته‌ای، این مسأله نیز مستعد قحطی است. همانطوری که در مسأله خوانندگان-نویسندگان دیدیم، گاهی اوقات می‌توانیم این مشکل را با اختصاص اولویت به یک دسته از نخ‌ها بر طبق معیارهای خاص برنامه حل کنیم. امّا به طور کلی نوشتن یک راه‌حل کارا (که حداکثر درجه همزمانی را ممکن سازد) و از قحطی جلوگیری نماید سخت است.

۲.۶ مسأله سرویس بهداشتی عمومی

این مسأله را زمانی که یکی از دوستانم موقعیت خود را در تدریس فیزیک در کالج کالبی[۱] رها نمود و در زیراکس[۲] مشغول به کار شد نوشتم[۳].

او در یک اتاقک در زیرزمین یک ساختمان کار می‌کرد و نزدیک‌ترین توالت زنانه دو طبقه بالاتر بود. او به رئیس خود پیشنهاد داد که توالت مردانه طبقه خودشان مختص به جنس خاصی نباشد.

رئیس موافقت نمود در صورتیکه محدودیت همگام‌سازی زیر بتواند فراهم آید:

- مردها و زن‌ها به طور همزمان نمی‌توانند با هم در دستشویی باشند.
- هرگز نباید بیش از سه کارمند ساعت کاری خود را در دستشویی تلف کنند.

البته راه‌حل باید از بن‌بست جلوگیری نماید. گرچه فعلاً نگران قحطی نباشید. می‌توانید فرض کنید که با سرویس بهداشتی تمام سمافورهای مورد نیاز شما فراهم آمده است.

---

[۳] بعداً دریافتم که مسأله‌ای تقریبا یکسان با این مسأله در کتاب «برنامه‌نویسی همروند» اندرو وجود دارد [؟].

[1] Colby College [2] Xerox

۱.۲.۶ راهنمایی سرویس بهداشتی عمومی

متغیرهایی که در راه‌حل بکار برده‌ام در ادامه آمده است:

راهنمایی سرویس بهداشتی عمومی

```
1 empty = Semaphore(1)
2 maleSwitch = Lightswitch()
3 femaleSwitch = Lightswitch()
4 maleMultiplex = Semaphore(3)
5 femaleMultiplex = Semaphore(3)
```

اگر اتاق خالی باشد مقدار empty برابر ۰ است در غیر اینصورت ۱. maleSwitch این امکان را به مردان می‌دهند که مانع ورود زن‌ها به اتاق شوند. زمانیکه اولین مرد وارد می‌شود، لایت‌سوئیچ، empty را قفل نموده تا مانع ورووود بانوان گردد؛ زمانیکه آخرین مرد خارج می‌شود، empty را باز نموده تا ورود بانوان را ممکن سازد. بانوان نیز عملی مشابه با استفاده از femaleSwitch انجام می‌دهند.

maleMultiplex و femaleMultiplex تضمین می‌نمایند که بیش از سه زن یا سه مزد در یک لحظه در سیستم نباشند.

۲.۲.۶ راه حل سرویس بهداشتی عمومی

کد بانوان در ادامه آمده است:

راه حل سرویس بهداشتی عمومی (بانوان)

```
femaleSwitch.lock(empty)
    femaleMultiplex.wait()
        # bathroom code here
    femaleMultiplex.signal()
female Switch.unlock(empty)
```

کد آقایان نیز مشابه است.

آیا با این راه‌حل مشکلی دارد؟

۳.۲.۶ مسأله سرویس بهداشتی عمومی بدون قحطی

مشکل راه‌حل قبلی این است که قحطی را ممکن می‌سازد. در حالیکه یک مرد منتظر ورود است ممکن است یک صف طولانی از بانوان و وارد شوند و بالعکس.

معمّا: مشکل را حل کنید:

۴.۲.۶ راه حل سرویس بهداشتی عمومی بدون قحطی

همانطوری که پیش از این دیدیم، می‌توانیم از ترن‌استایل استفاده کنیم تا یک نوعِ نخ مانع جریان دیگر انواع نخ گردد. این مرتبه به کد آقایان نگاه خواهیم انداخت:

راه حل سرویس بهداشتی عمومی بدون قحطی(آقایان)

```
turnstile.wait()
    maleSwitch.lock(empty)
turnstile.signal()

    maleMultiplex.wait()
        # bathroom code here
    maleMultiplex.signal()

maleSwitch.unlock (empty)
```

تا زمانیکه مردانی در اتاق وجود دارند، مردان تازه از راه رسیده از ترن‌استایل عبور خواهند نمود و وارد می‌شوند. آن زمانیکه یک مرد می‌رسد اگر یک تعدادی خانم داخل اتاق باشند، مرد مورد نظر درون ترن‌استایل مسدود شده و از ورود تمامی کسانی که پس از او می‌رسند (اعم از زن یا مرد) جلوگیری خواهد شد تا زمانیکه افراد داخل اتاق آنجا را ترک نمایند. در این نقطه، مردی که درون ترن‌استایل هست وارد شده و ورود سایر مردان را نیز ممکن می‌سازد.

کد بانوان نیز مشابه است، لذا اگر مردانی درون اتاق باشند و یک خانم برسد، آن خانم درون ترن‌استایل گیر خواهد افتاد و مانع ورود مردان دیگر می‌شود.

این راه‌حل ممکن است کارا نباشد. اگر سیستم مشغول باشد، اغلب چندین نخ مرد و زن وجود خواهد داشت که روی ترن‌استایل صف بسته‌اند. هر زمان که `empty` سیگنال داده می‌شود، یک نخ ترن‌استایل را ترک نموده و نخ دیگر وارد خواهد شد. اگر نخ جدید از جنس مخالف باشد آن نخ بی‌درنگ مسدود خواهد شد و مانع نخ‌های دیگر می‌گردد. بنابراین معمولاً در هر زمان تنها ۱ تا ۲ نخ در سرویس بهداشتی هستند و سیستم از تمام ظرفیت همزمانی موجود بهره نخواهد برد.

۳.۶ مسأله عبورکردن میمون

این مسأله از کتاب «سیستم‌های عامل: طراحی و پیاده‌سازی» تننبام اخذ شده است [؟]. یک جایی در پارک ملی کروگر در افریقای جنوبی[۴] یک دره عمیق با یک تک طناب که بین این دره کشیده شده است وجود دارد. میمون‌ها می‌توانند با کمک دست‌هایشان با آویزان شدن از آن طناب از روی دره عبور نمایند، امّا اگر دو میمون در دو جهت

[4] Kruger National Park, South Africa

مخالف در میانه راه به یکدیگر برسند، تا سر حد مرگ خواهند جنگید. علاوه بر این، طناب تنها تحمل وزن ۵ میمون را دارد. اگر میمون‌های بیشتری به طور همزمان روی طناب باشند پاره خواهد شد.

تصور کنید که می‌توانیم استفاده از سمافورها را به میمون‌ها آموزش دهیم، دوست داریم که یک برنامه همگام‌سازی با ویژگی‌های زیر طراحی نماییم:

- هنگامیکه یک میمون شروع به عبور نموده است، تضمین شده باشد بدون اینکه با میمونی از روبرو مواجه شود به طرف دیگر برسد.
- هیچگاه بیشتر از ۵ میمون روی طناب نباشد.
- یک جریان پیوسته از میمون‌ها که در یک جهت عبور می‌کند نباید برای همیشه مانع میمون‌های جهت مخالف شود (بدون قحطی).

به دلایلی که واضح است راه‌حل این مسأله را نخواهم گفت.

۴.۶ مسأله تالار Modus

این مسأله توسط Nathan Karst یکی از دانشجویان Olin که طی زمستان ۲۰۰۵ در تالار Modus[۵] زندگی می‌کرده نوشته شده است.

> زمستان، خصوصاً پس از یک بارش سنگین برف، ساکنان تالار Modus یک مسیر خندق مانند بین محل اقامت مقوایی‌شان و بقیه محوطه کالج ایجاد نمودند. هر روز برخی از ساکنین برای رفت و آمد بین کلاس‌ها، غذا خوردن و معاشرت از طریق این مسیر اقدام می‌نمایند؛ از دانشجویان تنبلی که روزانه برای رفتن به Tier 3 از ماشین استفاده می‌کنند صرف نظر می‌کنیم. همچنین از جهت عابرین در حال حرکت نیز چشم‌پوشی می‌کنیم. به دلایلی نامعلومی، دانشجویانی که در تالار غربی زندگی می‌کنند گاهی اوقات، زندگی در Mods را ضروری می‌دانند.
>
> متاسفانه، مسیر آنقدر عریض نیست که دو نفر بتوانند در کنار هم راه بروند. اگر دو نفر از Mods در یک نقطه از مسیر همدیگر را ببینند، یک نفر از آن‌ها با خوشحالی از مسیر خارج شده داخل برف می‌رود تا جا به دیگری دهد. وضعیت مشابهی برای دو ساکن ResHall در عبور از مسیرها رخ خواهد داد. گرچه اگر یک نفر از Mods (heathe n) با یک نفر از ResHall (prude) به هم برسند، یک کشمکش خشن به وقوع خواهد پیوست و گروهی که در اکثریت است پیروز است؛ بدین معنی که آن دسته‌ای که جمعیت بیشتری دارد گروه دیگر را وادار به انتظار می‌نماید.

[۵] تالار Modus یکی از چندین نام مستعار برای ساختمان‌های ماژولار است که همچنین به نام Mods نیز شناخته می‌شود و جایی است که برخی دانشجویان در زمانی که تالار اقامتی دوم در حال ساخته شدن است در آنجا زندگی می‌کنند.

این مسأله شبیه مسأله عبور میمون‌ها است (منتهی با بیش از یک راه)، به همراه یک پیچیدگی مضاعف که کنترل ناحیه بحرانی بوسیله قانون اکثریت تعیین می‌شود. این قانون، پتانسیل یک راه‌حل کارا و بدون قحطی را نسبت به مسأله انحصار دسته‌ای دارد.

قحطی ممکن نیست زیرا تا زمانیکه یک دسته، ناحیه بحرانی را کنترل می‌نماید، اعضای دسته دیگر به صف می‌پیوندند تا زمانیکه تبدیل به اکثریت شوند. سپس آن‌ها می‌توانند تا زمانیکه منتظر خالی شدن ناحیه بحرانی هستند مانع ورود رقیبان جدید گردند. انتظار می‌رود این راه حل، کارا باشد زیرا که نخ‌ها را در قالب دسته‌ها عبور داده لذا حداکثر همروندی در ناحیه بحرانی را ممکن می‌سازد.

معمّا: کدی بنویسید که انحصار دسته‌ای با قانون اکثریت را پیاده‌سازی نماید.

۱.۴.۶ راهنمایی مسأله تالار Modus

متغیرهایی که در راه‌حل بکار برده‌ام در ادامه آمده است.

راهنمایی مسأله تالار Modus

```
heathens = 0
prudes = 0
status = 'neutral'
mutex = Semaphore(1)
heathenTurn = Semaphore(1)
prudeTurn = Semaphore(1)
heathenQueue = Semaphore(0)
prudeQueue = Semaphore(0)
```

heathens و prudes شمارنده هستند و status وضعیت میدان را نگاه می‌دارد که می‌تواند 'neutral'، 'heathens rule'، 'prudes rule'، 'transition to heathens' و یا 'transition to prudes'. هر سه تای این متغیرها با mutex در یک الگوی جدول امتیاز معمولی محافظت می‌شوند.

heathenTurn و prudeTurn دسترسی به میدان را به نحوی کنترل می‌نمایند که می‌توانیم در مدت انتقال مانع یکی از دو طرف شویم.

heathenQueue و prudeQueue جایی هستند که نخ‌ها پس از ورود و پیش از گرفتن میدان، منتظر می‌مانند.

۲.۴.۶ راه حل مسأله تالار Modus

کد heathenها در ادامه آمده است:

راه حل مسأله Modus

```
heathenTurn.wait()
heathenTurn.signal()

mutex.wait()
heathens++

if status == 'neutral':
    status = 'heathens rule'
    mutex.signal()
elif status == 'prudes rule':
    if heathens > prudes:
        status = 'transition to heathens'
        prudeTurn.wait()
    mutex.signal()
    heathenQueue.wait()
elif status == 'transition to heathens':
    mutex.signal()
    heathenQueue.wait()
else
    mutex.signal()

# cross the field

mutex.wait()
heathens--

if heathens == 0:
    if status == 'transition to prudes':
        prudeTurn.signal()
    if prudes:
        prudeQueue.signal(prudes)
        status = 'prudes rule'
    else:
        status = 'neutral'

if status == 'heathens rule':
    if prudes > heathens:
        status = 'transition to prudes'
        heathenTurn.wait()

mutex.signal()
```

هر دانشجویی که می‌رسد باید حالات زیر را در نظر گیرد:

- اگر میدان خالی بود، دانشجو مدعی عبور heathenها می‌شود.
- اگر الان heathenها مسئول هستند ولی ورودی جدید توازن را بهم زند، او ترن‌استایل prude را قفل نموده و سیستم به حالت transition می‌رود.
- اگر prudeها مسئول بودند امّا ورودی جدید توازن به هم نزند، این فرد به صف ملحق می‌گردد.
- اگر سیستم در حال انتقال کنترل به heathen است، ورودی جدید به صف ملحق می‌شود.
- در غیر اینصورت، نتیجه می‌گیرم که یا heathenها مسئول هستند و یا سیستم در حال انتقال کنترل به prude است. در هر حالت، این نخ می‌تواند ادامه یابد.

به طور مشابه، هر دانشجویی که خارج می‌شود باید چندین حالت را در نظر گیرد.

- اگر او آخرین فرد heathen باشد که می‌رود باید شرایط زیر را در نظر گیرد:
  - اگر سیستم در حالت transition است، بدین معنی که ترن‌استایل prude قفل شده است، آن فرد باید آن را باز نماید.
  - اگر prudeهایی منتظر هستند، او باید به آن‌ها سیگنال داده و `status` را چنان بروزرسانی نماید که prudeها مسئول هستند. و اگر چنین نباشد وضعیت جدید، ``neutral'' است.
- اگر آن فرد، آخرین نفر heathen که می‌رود نباشد، هنوز باید این امکان را بررسی نماید که آیا رفتن او، توازن را بر هم می‌زند یا نه؟ در این حالت، او ترن‌استایل heathen را می‌بندد و انتقال را آغاز می‌نماید.

یک مشکل بالقوه این راه‌حل این است که هر تعداد نخ ممکن است در خط ۳ متوقف شوند، جاییکه آن‌ها باید از ترن‌استایل گذشته باشند ولی هنوز وارد نشده‌اند. تا زمانیکه آن نخ‌ها وارد شوند، شمرده نمی‌شوند، لذا توازن قدرت ممکن است بازگوی تعداد نخ‌هایی که از ترن‌استایل عبور نموده‌اند نباشد. همچنین زمانیکه تمام نخ‌ها که وارد شده‌اند خارج نیز شده باشند یک انتقال پایان می‌پذیرد. در آن نقطه، ممکن است نخ‌هایی (از هر دو نوع) وجود داشته باشند که از ترن‌استایل گذر کرده باشند.

این رفتارها کارایی را تحت تاثیر قرار می‌دهد —این راه‌حل همروندی حداکثری را تضمین نمی‌کند— امّا این نکات، اگر بپذیرید که «قانون اکثریت» تنها به نخ‌هایی که برای رای‌گیری ثبت‌نام نموده‌اند اعمال می‌شود، درستی راه‌حل را تحت تاثیر خود قرار نمی‌دهد.

فصل ۷

مسائل تقریباً غیر کلاسیک

۱.۷ مسأله بار سوشی

این مسأله از یک مسأله پیشنهاد شده توسط Kenneth Reek الهام گرفته شده است.[؟]. که یک بار سوشی با پنج صندلی را تصور کنید. اگر زمانیکه شما می‌رسید یک صندلی خالی وجود داشته باشد، شما می‌توانید بلافاصله بنشینید. امّا اگر زمانیکه می‌رسید تمامی پنج صندلی پر باشد به این معنی است که تمامی آن‌ها با هم مشغول غذا خوردن هستند و شما باید پیش از نشستن منتظر بمانید تا تمام آن‌ها بار را ترک نمایند.

معمّا: کدی برای مشتریان در حال ورود/خروج به/از بار سوشی بنویسید که این نیازها را اعمال نماید.

۱.۱.۷ راهنمایی بار سوشی

متغیرهای بکار رفته در ادامه آمده است:

راهنمایی بار سوشی

```
1 eating = waiting = 0
2 mutex = Semaphore(1)
3 block = Semaphore(0)
4 must_wait = False
```

eating و waiting تعداد نخ‌های منتظر و نشسته در بار را در خود نگاه می‌دارند. mutex از هر دو شمارنده محافظت می‌نماید. must_wait نشان می‌دهد که بار پر است و مشتری‌های جدید باید روی block مسدود گردند.

۲.۱.۷ ناراه‌حل بار سوشی

راه‌حل نادرستی که Reek برای نشان‌دادن یکی از مشکلات این مسأله به کار می‌برد در ادامه آمده است:

ناراه‌حل بار سوشی

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()

    mutex.wait()      # reacquire mutex
    waiting -= 1

eating += 1
must_wait = (eating == 5)
mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0:
    n = min(5, waiting)
    block.signal(n)
    must_wait = False
mutex.signal()
```

معمّا: این راه‌حل چه مشکلی دارد؟

۳.۱.۷ ناراه‌حل بارسوشی

مشکل در خط ۷ است. اگر یک مشتری زمانیکه بار پر است برسد تا زمانیکه منتظر است تا سایر مشتریان بار را ترک کنند باید از میوتکس صرف نظر کند. زمانیکه آخرین مشتری بار را ترک می‌کند، او به `block` سیگنال می‌دهد که منجر به بیدار شدن حداقل تعدادی از مشتریان منتظر می‌شود و `must_wait` را `False` می‌کند.

امّا زمانیکه مشتریان بیدار می‌شوند، آن‌ها باید میوتکس را بگیرند و این بدین معنی است که آن‌ها باید با نخ‌های که جدیداً وارد می‌شوند رقابت نمایند. اگر نخ‌های جدیدی که می‌رسند میوتکس را اول بگیرند، آنگاه آن‌ها می‌توانند پیش از نخ‌های منتظر تمامی صندلی‌ها را بگیرند. این فقط یک مسأله بی‌عدالتی نیست؛ زیرا که ممکن است بیش از ۵ نخ به طور همزمان در ناحیه بحرانی قرار گیرند که محدودیت‌های همگام‌سازی مسأله را خدشه‌دار می‌نمایند.

Reek دو راه‌حل برای این مشکل ارائه می‌دهد که در دو بخش بعدی آمده است.

معمّا: ببینید آیا می‌توانید دو راه‌حل صحیح متفاوت برای این مسأله ارائه دهید.

راهنمایی: هیچکدام از این راه حل‌ها هیچ متغیر اضافی بکار نمی‌برد.

۴.۱.۷ راه حل بار سوشی #۱

تنها دلیلی که یک مشتری در حال انتظار باید میوتکس را مجدداً بگیرد، بروز رسانی وضعیت eating و waiting است لذا یک راه برای حل این مشکل این است که مشتری در حال خروج —کسی که در حال حاضر میوتکس را دارد— این بروزرسانی را انجام دهد.

راه حل بار سوشی #۱

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()
else:
    eating += 1
    must_wait = (eating == 5)
    mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0:
    n = min(5, waiting)
    waiting -= n
    eating += n
    must_wait = (eating == 5)
    block.signal(n)
mutex.signal()
```

زمانیکه آخرین مشتری در حال خروج میوتکس را آزاد می‌کند، eating نیز بروزرسانی شده لذا مشتری‌هایی که جدیداً وارد می‌شوند وضعیت درست را می‌بینند و در صورت لزوم مسدود می‌شوند. Reek این الگو را "به خاطر تو این کار را انجام خواهم داد" نامیده است زیرا که نخ در حال خروج کار را انجام می‌دهد که منطقاً به نظر می‌رسد که متعلق به نخ‌های در حال انتظار است.

اشکال این راهکار این است که تایید اینکه وضعیت به درستی به روز رسانی می‌شود کمی سخت است.

۵.۱.۷ راه حل بار سوشی #۱

راه‌حل دیگر Reek بر پایه ایده غیر معمول است که ما می‌توانیم یک میوتکس را از نخی به نخ دیگر منتقل نماییم. به عبارت دیگر، یک نخ می‌تواند یک قفل را بگیرد و سپس نخ دیگری می‌تواند آن را آزاد نماید. تا زمانیکه هر دو نخ می‌دانند که قفل منتقل شده است هیچ مشکلی با این موضوع وجود ندارد.

راه حل بار سوشی #۲

```
mutex.wait()
if must_wait:
    waiting += 1
    mutex.signal()
    block.wait()     # when we resume, we have the mutex
    waiting -= 1

eating += 1
must_wait = (eating == 5)
if waiting and not must_wait:
    block.signal()            # and pass the mutex
else:
    mutex.signal()

# eat sushi

mutex.wait()
eating -= 1
if eating == 0: must_wait = False

if waiting and not must_wait:
    block.signal()            # and pass the mutex
else:
    mutex.signal()
```

اگر کمتر از ۵ مشتری در بار وجود داشته باشد و هیچ مشتری منتظر نباشد، یک مشتری جدید فقط مقدار `eating` را افزایش داده و میوتکس را رها می‌کند. پنجمین مشتری مقدار `must_wait` را برابر `True` می‌نماید.

اگر `must_wait` مقدار صحیح داشته باشد تا زمانیکه آخرین مشتری در بار مقدار آن را ناصحیح نموده و به `block` سیگنال دهد مشتری‌های جدید مسدود می‌شوند. این قابل درک است که نخ سیگنال دهنده از میوتکس صرف نظر نموده و نخ منتظر آن را می‌گیرد. در هر حال به یاد داشته باشید که این نکته بوسیله برنامه‌نویس غیر قابل درک است و چیزی است که در توضیحات، مستند شده است ولکن به وسیله معانی سمافورها اعمال نشده است. این وظیفه ما است که معنی درست را دریابیم.

زمانیکه نخ منتظر ادامه می‌یابد، در می‌یابیم که آن نخ میوتکس را دارد. اگر نخ‌های منتظر دیگری وجوپد داشته باشد، آن نخ به `block` سیگنال می‌دهد که دومرتبه میوتکس را به نخ منتظر می‌دهد. این فرآیند به این

صورت ادامه می‌یابد که اگر هیچ صندلی و یا نخ منتظر وجود نداشته باشد هر نخ میوتکس را به دیگری می‌دهد. در هر حالت، آخرین نخ میوتکس را آزاد نموده و می‌رود که بنشیند.

از آنجاییکه میوتکس از یک نخ به نخ دیگر منتقل می‌شود مشابه آنچه در مسابقه دو امدادی رخ می‌دهد Reek این الگو را "باتوم را منتقل کنید" می‌نامد. یک چیز خوب در رابطه با این راه‌حل این است که تایید پایدار بودن بروزرسانی `eating` و `waiting` آسان است. و یک نقطه ضعف آن این است که تایید صحیح بکار رفتن میوتکس سخت‌تر است.

۲.۷ مسأله مراقبت از کودکان

این مسأله را Max Hailperin در کتابش، «سیستم‌های عامل و میان‌افزارها»[1]، نوشته است [؟]. در یک مرکز مراقبت از کودکان، قوانین دولتی حضور دائم یک بزرگسال را برای هر سه بچه الزامی می‌نماید.

معمّا: کدی برای سه نخ کودک و یک نخ بزرگسال بنویسید که این محدودیت را در یک ناحیه بحرانی اعمال نماید.

[1] *Operating Systems and Middleware*

۱.۲.۷ راهنمایی مراقبت از کودکان

پیشنهاد Hailperin این است که این مسأله با یک سمافور تقریباً قابل حل است.

راهنمایی مراقبت از کودکان

```
1 multiplex = Semaphore(0)
```

`multiplex` تعداد توکن‌های موجود را می‌شمرد که هر توکن اجازه ورود یک نخ کودک را می‌دهد. با ورود برزگسالان، آن‌ها به `multiplex` سه بار سیگنال می‌دهند؛ و با خروجشان، سه بار `wait` را فرا می‌خوانند. امّا این راه‌حل یک اشکال دارد.

معمّا: این مشکل چیست؟

۲.۲.۷ ناراه‌حل مراقبت از کودکان

آنچه در ادامه آمده مشابه کد بزرگسال در ناراه‌حل Hailperin است:

ناراه‌حل مراقبت از کودکان (بزرگسال)

```
multiplex.signal(3)

# critical section

multiplex.wait()
multiplex.wait()
multiplex.wait()
```

مشکل احتمال وقوع بن‌بست است. تصور نمایید که سه کودک و دو بزرگسال در مرکز مراقبت از کودکان وجود دارد. مقدار `multiplex` برابر ۳ است، لذا هر کدام از بزرگسالان باید بتوانند که بروند. امّا اگر هر دو بزرگسال در یک لحظه بخواهند بروند، ممکن است توکن‌های موجود را بین خود تقسیم نمایند و هر دو مسدود شوند.

معمّا: این مشکل را با حداقل تغییرات، رفع نمایید.

۳.۲.۷ راه‌حل مرکز مراقبت از کودکان

افزودن یک میوتکس مشکل را حل می‌نماید.

راه‌حل مرکز مراقبت از کودکان (بزرگسال)

```
1
2  multiplex.signal(3)
3
4  # critical section
5
6  mutex.wait()
7      multiplex.wait()
8      multiplex.wait()
9      multiplex.wait()
10 mutex.signal()
```

اکنون سه عمل `wait` اتمی هستند. اگر سه توکن موجود باشد، نخی که میوتکس را می‌گیرد تمامی هر سه توکن را گرفته و خارج می‌شود. اگر توکن کمتری وجود داشته باشد، اولین نخ روی میوتکس مسدود خواهد شد و نخ‌های بعدی روی میوتکس تشکیل صف می‌دهند.

۴.۲.۷ مسأله توسعه‌یافته مرکز مراقبت

یک ویژگی این راه‌حل این استکه نخ بزرگسال در انتظار خروج می‌تواند جلوی ورود نخ‌های کودک را بگیرد.

تصور نمایید که ۴ کودک و ۲ بزرگسال وجود دارد، لذا مقدار multiplex برابر ۲ است. اگر یکی از بزرگسالان بخواهد که برود، باید دو توکن را نگاه دارد و سپس منتظر سومی بماند. اگر یک نخ کودک برسد، گرچه قانوناً می‌تواند وارد شود ولی منتظر خواهند ماند. از منظر بزرگسالی که تلاش می‌کند تا خارج شود، این امر خوب است، امّا اگر شما در تلاش برای حداکثر کردن بهره‌وری مرکز مراقبت از کودکان هستید این مسأله خوب نیست.

معمّا: راه‌حلی برای این مشکل ارائه دهید که از انتظارهای غیرضروری اجتناب نماید.

راهنمایی: به رقصندگان در بخش ۸.۳ بیندیشید.

۵.۲.۷ راهنمایی مسأله توسعه‌یافته مرکز مراقبت

متغیرهای بکار رفته در راه‌حل در اینجا آمده است:

راهنمایی مسأله توسعه‌یافته مرکز مراقبت

```
children = adults = waiting = leaving = 0
mutex = Semaphore(1)
childQueue = Semaphore(0)
adultQueue = Semaphore(0)
```

children، adults، waiting و leaving تعداد کودکان، بزرگسالان، کودکان در انتظار ورود و بزرگسالان در انتظار خروج را نگاه می‌دارند؛ این متغیرها بوسیله mutex محافظت می‌شوند.

در صورت لزوم، کودکان برای ورود روی childQueue منتظر می‌مانند. بزرگسالان به منظور خروج روی adultQueue منتظر می‌مانند.

۶.۲.۷ راه‌حل مسأله توسعه‌یافته مرکز مراقبت

این راه حل، از راه‌حل زیبای Hailperin کمی پیچیده‌تر است، امّا تقریبا ترکیبی از الگوهایی است که پیش از این دیده‌ایم: یک تابلوی امتیاز، دو صف و "بخاطر تو این کار را انجام خواهم داد".

کد کودک در ادامه آمده است.

راه‌حل توسعه‌یافته مرکز مراقبت از کودکان (کودک)

```
mutex.wait()
    if children < 3 * adults:
        children++
        mutex.signal()
    else:
        waiting++
        mutex.signal()
        childQueue.wait()

# critical section

mutex.wait()
    children--
    if leaving and children <= 3 * (adults-1):
        leaving--
        adults--
        adultQueue.signal()
mutex.signal()
```

وقتی کودکان وارد می‌شوند، بررسی می‌نمایند که آیا به اندازه کافی بزرگسال وجود دارد و یا (۱) `children` را یک واحد افزایش داده و وارد می‌شود و یا (۲) `waiting` را یک واحد افزایش داده و مسدود می‌گردد. زمانیکه کودکان خارج می‌شوند، بررسی می‌نمایند که آیا نخ بزرگسالی در انتظار برای خروج وجود دارد و اگر مقدور باشد به آن سیگنال می‌دهند.

کد بزرگسال در ادامه آمده است:

راه‌حل توسعه‌یافته مرکز مراقبت از کودکان (بزرگسال)

```
mutex.wait()
    adults++
    if waiting:
        n = min(3, waiting)
        childQueue.signal(n)
        waiting -= n
        children += n
mutex.signal()

# critical section

mutex.wait()
    if children <= 3 * (adults-1):
        adults--
        mutex.signal()
    else:
        leaving++
        mutex.signal()
        adultQueue.wait()
```

بزرگسالان همانطوری که وارد می‌شوند اگر کودکانی در حال انتظار باشند به آن‌ها سیگنال می‌دهند. قبل از خروج، بررسی می‌نمایند که آیا به اندازهٔ کافی بزرگسال باقی می‌ماند یا خیر؟. اگر به تعدادی کافی بودند، `adults` را یک واحد کم کرده و خارج می‌شوند. در غیر اینصورت `leaving` را یک واحد افزایش داده و مسدود می‌گردند. زمانیکه یک نخ بزرگسال در انتظار خروج است، به عنوان یکی از بزرگسالان موجود در ناحیه بحرانی شمرده می‌شود، لذا کودکان بیشتری می‌توانند وارد شوند.

۳.۷ مسأله اتاق پارتی

این مسأله را زمانیکه در کالج کالبی بودم نوشتم. در یک ترم به سبب اینکه یک دانشجو مدعی شده بود که شخصی از طرف رئیس دفتر دانشجویان در غیاب او اتاقش را وارسی کرده است منازعه‌ای رخ داد. اگرچه ادعا به صورت عمومی مطرح شد، رئیس دفتر دانشجویان قادر به اظهار نظر در مورد آن نبود، لذا هرگز نفهمیدیم واقعاً چه اتفاقی افتاده است. این مسأله را برای این نوشتن تا یکی از دوستانم را که معاون مسکن دانشجویی بود دست بیندازم.

محدودیت‌های همگام‌سازی زیر نسبت دانشجویان و مدیریت دانشجویی اعمال می‌گردد:

۱. هر تعداد دانشجو می‌تواند به طور همزمان در یک اتاق باشد.

۲. مدیریت دانشجویی تنها در صورتی می‌تواند وارد اتاقی شود که یا هیچ دانشجویی در اتاق نباشد (به منظور هدایت یک تفحص) و یا بیش از ۵۰ دانشجو در اتاق باشد (به منظور بر هم زدن پارتی).

۳. زمانیکه مدیریت دانشجویی در اتاق هست هیچ دانشجویی دیگری ممکن نیست وارد شود ولی دانشجویان ممکن است اتاق را ترک کنند.

۴. مدیریت دانشجویی ممکن نیست اتاق را ترک گوید مگر اینکه تمامی دانشجویان آن را ترک کرده باشند.

۵. تنها یک مدیریت دانشجویی وجود دارد لذا نباید بین چند مدیر انحصار اعمال کنید.

معمّا: برای دانشجویان و مدیریت دانشجویی یک کد همگام‌سازی بنویسید که تمامی این محدودیت‌ها را اعمال نماید.

۱.۳.۷ راهنمای اتاق پارتی

راهنمای اتاق پارتی

```
students = 0
dean = 'not here'
mutex = Semaphore(1)
turn = Semaphore(1)
clear = Semaphore(0)
lieIn = Semaphore(0)
```

students تعداد دانشجویان داخل اتاق را می‌شمرد و dean وضعیت مدیر است که علاوه بر 'not here' همچنین می‌تواند "waiting" و یا "in the room" . mutex از students و dean محافظت می‌نماید لذا این نیز مثالی دیگر یک جدول امتیاز است.

turn ترن‌استایلی است که مانع ورود دانشجویان به اتاق زمانیکه مدیر در اتاق است می‌گردد.

clear و lieIn به صورت قرار ملاقات‌هایی بین یک دانشجو و مدیر (که یک نمونه کامل دیگر از رسوایی است) بکار می‌رود.

۲.۳.۷ راه‌حل اتاق پارتی

این مسأله‌ای سخت است. من پیش اینکه با این یکی روبرو شوم روی نسخه‌های زیادی کار کردم. این نسخه که در ویرایش اول این کتاب عرضه گردید کاملاً درست بود، امّا گهگاهی مدیر ممکن بود وارد اتاق شود و سپس دریابد که نه می‌تواند به جستجو بپردازد و نه می‌تواند پارتی را متوقف کند، لذا او می‌بایست در سکوتی خجالت‌بار آنجا را ترک کند.

Matt Tesch یک راه‌حل نوشت که جلوی این حقارت را می‌گرفت، امّا نتیجه اینقدر پیچیده بود که به سختی توانستیم خود را متقاعد کنیم که راه‌حل درست است. امّا آن راه‌حل مرا به این راه‌حل که کمی خواناتر است رهنمون گردید.

راه‌حل اتاق پارتی (مدیر)

```
mutex.wait()
    if students > 0 and students < 50:
        dean = 'waiting'
        mutex.signal()
        lieIn.wait()     # and get mutex from the student.

    # students must be 0 or >= 50

    if students >= 50:
        dean = 'in the room'
        breakup()
        turn.wait()      # lock the turnstile
        mutex.signal()
        clear.wait()     # and get mutex from the student.
        turn.signal()    # unlock the turnstile

    else:                # students must be 0
        search()

dean = 'not here'
mutex.signal()
```

زمانیکه مدیر می‌رسد، سه حالت وجود دارد: اگر دانشجویانی داخل اتاق بوده و ۵۰ نفر یا بیشتر نباشد مدیر باید صبر کند. اگر ۵۰ یا تعدادی بیشتر دانشجو در اتاق باشد مدیر پارتی را متوقف نموده و منتظر خروج دانشجویان می‌ماند. اگر هیچ دانشجویی وجود نداشته باشد، مدیر به جستجو پرداخته و سپس اتاق را ترک می‌کند.

در دو حالت اول، مدیر باید برای یک قرار ملاقات با یک دانشجو صبر کند لذا از mutex صرف نظر می‌کند تا بن‌بستی رخ ندهد. زمانیکه مدیر بر می‌خیرد، باید جدول امتیاز را تغییر دهد بنابراین نیاز دارد تا میوتکس را دوباره بگیرد. این، مشابه وضعیتی است که در مسأله بار سوشی دیدیم. راه‌حلی که انتخاب کردم الگوی "باتوم را منتقل کنید" است.

راه‌حل اتاق پارتی (دانشجو)

```
mutex.wait()
    if dean == 'in the room':
        mutex.signal()
        turn.wait()
        turn.signal()
        mutex.wait()

    students += 1

    if students == 50 and dean == 'waiting':
        lieIn.signal()
# and pass mutex to the dean
    else:
        mutex.signal()

party()

mutex.wait()
    students -= 1

    if students == 0 and dean == 'waiting':
        lieIn.signal()          # and pass mutex to the dean
    elif students == 0 and dean == 'in the room':
        clear.signal()          # and pass mutex to the dean
    else:
        mutex.signal()
```

سه حالت وجود دارد که ممکن است یک دانشجو مجبور شود به مدیر سیگنال دهد. اگر مدیر در حال انتظار است، پنجاهمین دانشجویی که وارد می‌شود یا آخرین دانشجویی که خارج می‌شود باید به `lieIn` سیگنال دهد. اگر مدیر داخل اتاق است (و در حالیکه منتظر است تا تمامی دانشجویان خارج شوند)، آخرین دانشجویی که خارج می‌شود باید به `clear` سیگنال دهد. در تمامی سه حالت، قابل درک است که میوتکس از یک دانشجو به مدیر منتقل می‌گردد.

بخشی از این راه‌حل که ممکن است واضح نباشد این است چطور در می‌یابیم در خط ۷ از کد مدیر، مقدار `students` باید صفر و یا نباید کمتر از ۵۰ باشد. نکته در فهم این است تنها دو راه برای رسیدن به این دونقطه وجود دارد: یا شرط اولی نادرست بود که در اینصورت `students` یا صفر است و یا کمتر پنجاه نیست؛ یا مدیر زمانیکه یک دانشجو سیگنال داد روی `lieIn` منتظر بوده است و دوباره به این معنی است که `students` یا صفر است و یا کمتر از ۵۰ نیست.

۴.۷ مسأله اتوبوس سنا[۲]

این مسأله در اصل بر مبنای اتوبوس سنا در کالج Wellesley بود. مسافران به ایستگاه اتوبوس آمده و منتظر اتوبوس می‌شوند. زمانیکه اتوبوس می‌رسد، تمامی مسافران منتظر `boardBus` را فرا می‌خوانند امّا هر کسی که در حین سوارشدن مسافران می‌رسد باید منتظر اتوبوس بعدی بماند. ظرفیت اتوبوس ۵۰ نفر است، اگر بیش از ۵۰ نفر منتظر باشند، برخی باید منتظر اتوبوس بعدی بنمانند.

زمانیکه تمامی مسافران منتظر سوار شدند، اتوبوس می‌تواند `depart` را فراخواند. اگر اتوبوس زمانی برسد که هیچ مسافری وجود نداشته باشد می‌تواند بلافاصله حرکت کند.

معمّا: کدی بمنظور همگام‌سازی بنویسید که تمامی این محدودیت‌ها را اعمال کند.

[2] https://www.wellesley.edu/housingtransp/transportation/shuttlebuses

۱.۴.۷ راهنمایی مسأله اتوبوس

متغیرهایی که در راه‌حل بکار برده‌ام در ادامه آمده است:

راهنمای مسأله اتوبوس

```
riders = 0
mutex = Semaphore(1)
multiplex = Semaphore(50)
bus = Semaphore(0)
allAboard = Semaphore(0)
```

`mutex` از `riders` که تعداد مسافران منتظر را نگه می‌دارد محافظ می‌کند؛ `multiplex` تضمین می‌کند که بیش از ۵۰ مسافر در سکوی سوارشدن وجود نداشته باشد.

مسافران روی سمافور `bus` منتظر می‌مانند و این سمافور زمانیکه اتوبوس می‌رسد سیگنال دریافت می‌دارد. اتوبوس روی سمافور `allAboard` منتظر می‌ماند و این سمافور زمانیکه آخرین دانشجو سوار شد سیگنال دریافت می‌کند.

۲.۴.۷ راه حل مسأله اتوبوس #۱

کد اتوبوس در ادامه آمده است. دوباره از الگوی "باتوم را منتقل کنید" استفاده می‌کنیم.

راه حل مسأله اتوبوس (اتوبوس)

```
mutex.wait()
if riders > 0:
    bus.signal()        # and pass the mutex
    allAboard.wait()    # and get the mutex back
mutex.signal()

depart()
```

زمانیکه اتوبوس می‌رسد، mutex را می‌گیرد تا از ورود کسانیکه بعداً می‌رسند به سکوی سوارشدن جلوگیری نماید. اگر هیچ مسافری وجود نداشته اتوبوس بلافاصله محل را ترک می‌نماید. در غیر اینصورت، به bus سیگنال داده و منتظر می‌ماند تا مسافران سوار شوند.

کد مسافران نیز در اینجا آمده است:

راه حل مسأله اتوبوس (مسافران)

```
multiplex.wait()
    mutex.wait()
        riders += 1
    mutex.signal()

    bus.wait()              # and get the mutex
multiplex.signal()

boardBus()

riders -= 1
if riders == 0:
    allAboard.signal()
else:
    bus.signal()            # and pass the mutex
```

multiplex تعداد مسافران موجود در محل انتظار را کنترل می‌کند، گرچه مسافران تا زمانیکه riders را یک واحد افزایش ندهند وارد محل انتظار می‌گردند.

مسافران تا زمانیکه اتوبوس برسد روی سمافور bus منتظر می‌مانند. زمانیکه یک مسافر بیدار می‌شود (سیگنال دریافت می‌کند)، قابل فهم است که میوتکس را دارد. پس از سوار شدن هر مسافر یک واحد از مقدار riders کم می‌کند. اگر تعداد مسافران بیشتری در حال انتظار باشند، مسافری که در حال سوار شدن است به bus سیگنال داده و میوتکس را به مسافر بعدی منتقل می‌کند. آخرین مسافر به allAboard سیگنال داده و

میوتکس را به اتوبوس پس می‌دهد. نهایتاً اتوبوس میوتکس را رها کرده و حرکت می‌کند.

معمّا: آیا می‌توانید راه حلی برای این مسأله با کمک الگوی "بخاطر تو انجام این کار را خواهم داد" بیابید؟

۳.۴.۷ راه حل مسأله اتوبوس #۲

این راه‌حل را Grant Hutchins ارائه داد که متغیرهای کمتری نسبت به راه‌حل قبلی بکار می‌برد و مستلزم انتقال میوتکس نیست.

راه حل #۲ مسأله اتوبوس (مقداردهی اولیه)

```
waiting = 0
mutex = new Semaphore(1)
bus = new Semaphore(0)
boarded = new Semaphore(0)
```

waiting تعداد مسافران در ناحیه سوارشدن است که بوسیله mutex حفاظت می‌شود. bus زمانیکه اتوبوس رسیده است سیگنال می‌دهد؛ boarded سیگنال می‌دهد که یک مسافر سوار شده است.

کد اتوبوس در ادامه آمده است.

راه حل اتوبوس (اتوبوس)

```
mutex.wait()
n = min(waiting, 50)
for i in range(n):
    bus.signal()
    boarded.wait()

waiting = max(waiting-50, 0)
mutex.signal()

depart()
```

اتوبوس mutex را می‌گیرد و در طی فرآیند سوارشدن مسافران آن را نگاه می‌دارد. حلقه به نوبت به هر مسافر سیگنال داده و منتظر او می‌ماند تا سوار شود. با کنترل تعداد سیگنال‌ها، اتوبوس از اینکه بیش از ۵۰ مسافر سوار شوند جلوگیری می‌نماید.

زمانیکه تمامی مسافران سوار شدند، اتوبوس مقدار waiting را بروزرسانی می‌نماید که این مثالی از الگوی "بخاطر تو این کار را انجام خواهم داد" است.

کد مسافران از دو الگوی ساده استفاده می‌نماید: یک میوتکس و یک قرار ملاقات.

Bus problem solution (riders)

```
\begin{lstlisting}[title=\rl  }          ))}]{}
mutex.wait()
     waiting += 1
mutex.signal()

bus.wait()
```

```
board()
boarded.signal()
```

چالش: اگر مسافران زمانیکه اتوبوس در حال مسافرگیری است برسند و آن‌ها را مجبور کنید که منتظر اتوبوس بعدی بمانند ممکن است که رنجیده خاطر شوند. آیا می‌توانید راه حلی بیابید که اجازه دهد تا مسافرینی که بعداً می‌رسند بدون نقض دیگر محدودیت‌ها سوار اتوبوس شوند.

۵.۷ مسأله تالار Faneuil

این مسأله بوسیله Grant Hutchins تحت تاثیر یک دوست که برای سوگند شهروندی به تالار Faneuil بوستون رفته بود نوشته شده است.

"سه نوع نخ وجود دارد: مهاجران، تماشاگران، و یک قاضی. مهاجران باید در یک صف منتظر مانده، مدارک را به منظور احراز هویت ارائه داده و سپس بنشینند. قاضی در یک زمانی وارد ساختمان می‌شود. زمانی که قاضی در ساختمان است، هیچکس نمی‌تواند وارد شود و مهاجران نمی‌توانند محل را ترک کنند ولی تماشاگران می‌توانند سالن را ترک گویند. زمانیکه تمامی مهاجران احراز هویت شدند قاضی می‌تواند تابعیت آن‌ها را تایید کند. پس از تایید، مهاجران گواهی شهروندی خود را دریافت می‌کنند. قاضی در زمانی پس از تایید محل را ترک می‌گوید. تماشاگران اکنون نیز مانند قبل می‌توانند وارد شوند. پس از اینکه مهاجران گواهی خود را گرفتند ممکن است سالن را ترک کنند.

برای روشن‌تر نمودن این محدودیت‌ها، بگذارید نخ‌ها را مقید به اجرای اعمالی کنیم و محدودیت‌ها را روی این اعمال بگذاریم.

- مهاجران باید `enter`، `checkIn`، `sitDown`، `swear`، `getCertificate` و `leave` فرا خوانند.
- قاضی `enter`، `confirm` و `leave` را فرا می‌خواند.
- تماشاگران `enter`، `spectate` و `leave` را فرا می‌خوانند.
- زمانیکه قاضی در ساختمان است، هیچ کس نمی‌تواند `enter` را فراخواند و مهاجران نمی‌توانند `leave` را فرا خوانند.
- تا زمانیکه تمامی مهاجرانی که `enter` را فراخوانده‌اند `checkIn` را اجرا نکرده‌اند قاضی نمی‌تواند `confirm` را فراخواند.
- تا زمانیکه قاضی `confirm` اجرا نکرده باشد مهاجران نمی‌توانند `getCertificate` را فراخوانند.

۱.۵.۷ راهنمایی مسأله تالار Faneuil

راهنمایی مسأله تالار Faneuil

```
noJudge = Semaphore(1)
entered = 0
checked = 0
judge = 0
mutex = Semaphore(1)
confirmed = Semaphore(0)
allSignedIn = Semaphore(0)
```

noJudge مانند یک ترن‌استایل برای مهاجران و تماشاگران در حال ورود عمل می نماید؛ همچنین از entered که تعداد مهاجران درون اتاق را می‌شمرد محافظت می‌نماید. checked تعداد مهاجرانی که احراز هویت شده‌اند را می‌شمرد و بوسیله mutex حفاظت می‌شود. confirmed علامت می‌دهد که قاضی confirm را اجرا نموده است.

۲.۵.۷ راه حل مسأله تالار Faneuil

کد مهاجران در ادامه آمده است:

راهنمایی مسأله تالار Faneuil (مهاجر)

```
noJudge.wait()
enter()
entered++
noJudge.signal()

mutex.wait()
checkIn()
checked++

if judge = 1 and entered == checked:
    allSignedIn.signal()              # and pass the mutex
else:
    mutex.signal()

sitDown()
confirmed.wait()

swear()
getCertificate()

noJudge.wait()
leave()
noJudge.signal()
```

مهاجران برای ورود از یک ترن‌استایل گذر کرده و زمانیکه قاضی در اتاق است این ترن‌استایل قفل می‌گردد. مهاجران پس از ورود به منظور احراز هویت و بروزرسانی `checked` باید `mutex` را بگیرند. اگر یک قاضی در حال انتظار باشد آخرین مهاجری که احراز هویت می‌شود به `allSignedIn` سیگنال داده و میوتکس را به قاضی منتقل می‌کند.

کد قاضی در ادامه آمده است:

راهنمایی مسأله تالار Faneuil (قاضی)

```
noJudge.wait()
mutex.wait()

enter()
judge = 1

if entered > checked:
    mutex.signal()
```

```
    allSignedIn.wait()
# and get the mutex back.

confirm()
confirmed.signal(checked)
entered = checked = 0

leave()
judge = 0

mutex.signal()
noJudge.signal()
```

قاضی noJudge را نگه می‌دارد تا مانع ورود مهاجران و تماشاگران گردد و نیز mutex را نگه می‌دارد تا به entered و checked دسترسی داشته باشد.

اگر قاضی در لحظه‌ای برسد که تمامی مهاجرانی که داخل شده‌اند احراز هویت نیز شده باشند، می‌تواند بلافاصله کار را خود ادامه دهد. در غیر اینصورت، باید از میوتکس صرف نموده و منتظر بماند. زمانیکه آخرین مهاجر احراز هویت شد به allSignedIn سیگنال می‌دهد، و قابل درک است که قاضی دومرتبه میوتکس را خواهد گرفت.

قاضی پس از فراخوانی confirm، به confirmed بازای هر مهاجری که احراز هویت شده است یک مرتبه سیگنال می‌دهد و سپس مقدار شمارنده‌ها را صفر می‌کند (مثالی از "بخاطر تو این کار را انجام خواهم داد"). سپس قاضی محل را ترک نموده mutex و noJudge را رها می‌نماید.

پس از اینکه قاضی به confirmed سیگنال داد، مهاجران swear و getCertificate را به صورت همزمان فرا می‌خوانند و سپس قبل از خروج، منتظر ترن‌استایل noJudge می‌شوند که باز شود.

کد تماشاگران ساده است؛ تنها محدودیتی که آن‌ها باید رعایت کنند ترن‌استایل noJudge است.

راهنمایی مسأله تالار Faneuil (تماشاگر)

```
noJudge.wait()
enter()
noJudge.signal()

spectate()

leave()
```

توجه: در این راه‌حل ممکن است مهاجران پس از اینکه گواهی خود را گرفتند بوسیله قاضی دیگری که برای سوگند دادن دسته بعدی مهاجران در حال ورود است گیر بیفتند. اگر چنین اتفاقی بیفتد، آن‌ها باید در تمام مدت زمان مراسم سوگند گروه دیگر صبر کنند.

معمّا: این راه‌حل را چنان دستکاری کنید که محدودیت اضافی زیر را اعمال نماید. پس از اینکه قاضی خارج می‌شود تمامی مهاجرانی سوگند خورده‌اند باید قبل از اینکه قاضی بعدی وارد شود خارج شوند.

۳.۵.۷ راهنمایی مسأله توسعه‌یافته تالار Faneuil

راه‌حل من از متغیرهای اضافی زیر بهر می‌برد:

راهنمایی مسأله تالار Faneuil

```
1 exit = Semaphore(0)
2 allGone = Semaphore(0)
```

از آنجایی که مسأله توسعه‌یافته مستلزم یک قرار ملاقات اضافی است، آن را با دو سمافور حل می‌نماییم.

راهنمایی دیگر: استفاده مجدد از الگوی "باتوم را منتقل کنید" را مفید یافتم.

۴.۵.۷ راه حل مسأله توسعه‌یافته تالار Faneuil

نیمه بالایی این راه‌حل مشابه قبل است. تفاوت از خط ۲۱ شروع می شود. در اینجا مهاجران منتظر قاضی می‌مانند تا خارج شود.

راهنمایی مسأله تالار Faneuil (مهاجر)

```
1  noJudge.wait()
2  enter()
3  entered++
4  noJudge.signal()
5
6  mutex.wait()
7  checkIn()
8  checked++
9
10 if judge = 1 and entered == checked:
11     allSignedIn.signal()              # and pass the mutex
12 else:
13     mutex.signal()
14
15 sitDown()
16 confirmed.wait()
17
18 swear()
19 getCertificate()
20
21 exit.wait()                          # and get the mutex
22 leave()
23 checked--
24 if checked == 0:
25     allGone.signal()                  # and pass the mutex
26 else:
27     exit.signal()                     # and pass the mutex
```

برای قاضی، تفاوت از خط ۱۸ شروع می‌شود. زمانیکه قاضی آماده ترک محل است، نمی‌تواند noJudge را آزاد کند زیرا که این کار ممکن است سبب ورود مهاجران بیشتر و یا یک قاضی دیگر شود. در عوض، به exit سیگنال می‌دهد و این کار خروج یک مهاجر را مجاز می‌سازد و mutex را منتقل می‌کند.

مهاجری که سیگنال را دریافت می‌کند مقدار checked را یک واحد کاهش می‌دهد و سپس باتوم را به مهاجر بعدی می‌دهد. آخرین مهاجری که خارج می شود به allGone سیگنال داده و میوتکس را دومرتبه به قاضی می‌دهد. این بازگرداندن خیلی ضروری نیست ولی این ویژگی خوب را دارد که قاضی هر دوی mutex و noJudge را رها می‌نماید تا خیلی تر و تمیز به این مرحله پایان دهد.

راهنمایی مسأله تالار Faneuil (قاضی)

```
noJudge.wait()
mutex.wait()

enter()
judge = 1

if entered > checked:
    mutex.signal()
    allSignedIn.wait()
# and get the mutex back.

confirm()
confirmed.signal(checked)
entered = 0

leave()
judge = 0

exit.signal()                          # and pass the mutex
allGone.wait()                         # and get it back
mutex.signal()
noJudge.signal()
```

کد تماشاگر در مسأله توسعه‌یافته بدون تغییر است.

۶.۷ مسأله سالن غذاخوری

این مسأله بوسیله Jon Pollack در کلاس Synchronization من در کالج Olin نوشته شده است.

دانشجویان در سالن غذاخوری `dine` را فراخوانده و سپس `leave` را صدا می‌زنند. پس از فراخوانی `dine` و پیش از فراخواندن `leave`، وضعیت یک دانشجو "آماده برای خروج" در نظر گرفته می‌شود.

محدودیت همگام‌سازی که در مورد دانشجویان اعمال می‌شود این است به منظور حفظ وجهه اجتماعی، یک دانشجو هرگز ممکن نیست که سر یک میز تنها بنشیند. یک دانشجو در صورتی ممکن است تنها نشسته باشد که اگر پیش از اینکه او `dine` را تمام کرده باشد هر دانشجوی دیگری که `dine` را فراخوانده `leave` را فراخواند.

معمّا: کدی بنویسید که این محدودیت را اعمال نماید.

۱.۶.۷ راهنمایی مسأله سالن غذاخوری

راهنمایی مسأله سالن غذاخوری

```
1 eating = 0
2 readyToLeave = 0
3 mutex = Semaphore(1)
4 okToLeave = Semaphore(0)
```

eating و readyToLeave شمارنده‌هایی هستند که بوسیله mutex حفاظت می‌شوند لذا این یک الگوی جدول امتیاز معمول است.

اگر یک دانشجو آماده ترک میز باشد، اما دانشجوی دیگری ممکن است سر میز تنها بماند، او باید روی okToLeave منتظر مانده تا دانشجوی دیگری این وضعیت را تغییر داده و سیگنال دهد.

۲.۶.۷ راه حل مسأله سالن غذاخوری

اگر این محدودیت‌ها را تحلیل نمایید، متوجه خواهید شد که تنها یک وضعیت وجود دارد که یک دانشجو باید در آن صبر کند و آن این است که یک دانشجو در حال خوردن غذا است و یک دانشجو تمایل به ترک میز دارد. امّا تنها دو راه برای خلاصی از این وضعیت وجود دارد: ممکن است دانشجوی دیگری برای غذا خوردن برسد و یا دانشجویی که غذا می‌خورد غذایش تمام شود.

در هر حالت، دانشجویی که به دانشجوی منتظر سیگنال می‌دهد شمارنده‌ها را بروزرسانی می‌نماید لذا دانشجوی منتظر نباید مجدداً میوتکس را بگیرد. این، مثال دیگری از الگوی "به خاطر تو این کار را انجام خواهم داد" است.

راه حل مسأله سالن غذاخوری

```
getFood()

mutex.wait()
eating++
if eating == 2 and readyToLeave == 1:
    okToLeave.signal()
    readyToLeave--
mutex.signal()

dine()

mutex.wait()
eating--
readyToLeave++

if eating == 1 and readyToLeave == 1:
    mutex.signal()
    okToLeave.wait()
elif eating == 0 and readyToLeave == 2:
    okToLeave.signal()
    readyToLeave -= 2
    mutex.signal()
else:
    readyToLeave--
    mutex.signal()

leave()
```

زمانیکه دانشجو وارد می‌شود، اگر ببیند که یک دانشجو در حال غذاخوردن است و یک دانشجو منتظر ترک کردن است اجازه می‌دهد که دانشجوی منتظر برود و مقدار `readyToLeave` را برای او یک واحد کاهش می‌دهد.

پس از صرف غذا، دانشجو سه حالت را بررسی می‌نماید:

- اگر تنها یک دانشجو در حال غذا خوردن سر میز باشد، دانشجویی که قصد ترک دارد باید از میوتکس صرف نظر نموده و منتظر بماند.
- اگر دانشجویی که در حال ترک میز است دریابد که دانشجوی دیگری منتظر اوست، او به آن دانشجوی منتظر سیگنال داده و مقدار شمارنده را برای هر دوی‌شان بروزرسانی می‌نماید.
- در غیر اینصورت، فقط مقدار `readyToLeave` را یک واحد کاهش داده و میز را ترک می‌نماید.

۳.۶.۷ مسأله توسعه‌یافته سالن غذاخوری

مسأله سالن غذاخوری اگر یک مرحله دیگر به آن بیفزاییم کمی چالشی‌تر می‌شود. همانطوری که دانشجویان برای خوردن غذا می‌آیند `getFood`، `dine` و سپس `leave` را فرا می‌خوانند. پس از فراخوانی `getFood` و پیش از فراخوانی `dine` یک دانشجو در وضعیت "آماده برای خوردن" در نظر گرفته می‌شود. به طور مشابه، پس از فراخونی `dine` یک دانشجو در وضعیت "آماده برای خروج" در نظر گرفته می‌شود.

محدودیت همگام‌سازی مشابهی اعمال می‌گردد: یک دانشجو هرگز به تنهایی سر یک میز نمی‌نشیند. یک دانشجو در صورتی تنها در نظر گرفته می‌شود اگر

- زمانیکه هیچ کس دیگری سر میز نباشد و هیچ کس دیگری آماده غذا خوردن نباشد او `dine` را فراخواند و یا
- هر کس دیگری که `dine` را فراخونده پیش از اینکه او `dine` را تمام کند `leave` را فراخواند.

معمّا: کدی بنویسید که این محدودیت‌ها را اعمال نماید.

۴.۶.۷ راهنمای مسأله توسعه‌یافته سالن غذاخوری

متغیرهایی که در راه‌حل خود بکار برده‌ام در ادامه آمده است:

راهنمای مسأله توسعه‌یافته سالن غذاخوری

```
readyToEat = 0
eating = 0
readyToLeave = 0
mutex = Semaphore(1)
okToSit = Semaphore(0)
okToLeave = Semaphore(0)
```

readyToEat، eating و readyToLeave شمارنده‌هایی هستند که بوسیله mutex محافظت می‌شوند.

اگر یک دانشجو در وضعیتی است که نمی‌تواند ادامه دهد، باید روی okToSit یا okToLeave منتظر بماند تا دانشجوی دیگری وضعیت را تغییر داده و سیگنال دهد.

همچنین بازای هر نخ از یک متغیر به نام hasMutex استفاده کردم تا با کمک آن بتوانم بررسی نمایم که آیا آن نخ میوتکس را دارد یا نه.

۵.۶.۷ راه حل مسأله توسعه‌یافته سالن غذاخوری

دومرتبه، اگر محدودیت‌ها را تحلیل نماییم، در می‌یابیم که تنها یک وضعیت وجود دارد که یک دانشجوی آماده غذاخوردن باید صبر کند و آن این است که هیچ کسی در حال غذاخوردن نبوده هیچ کس دیگری نیز آماده غذاخوردن نیست. و تنها راه خروج از این وضعیت این است که فرد دیگری که آماده غذاخوردن است برسد.

راه حل مسأله توسعه‌یافته سالن غذاخوری

```
getFood()

mutex.wait()
readyToEat++
if eating == 0 and readyToEat == 1:
    mutex.signal()
    okToSit.wait()
elif eating == 0 and readyToEat == 2:
    okToSit.signal()
    readyToEat -= 2
    eating += 2
    mutex.signal()
else:
    readyToEat--
    eating++
    if eating == 2 and readyToLeave == 1:
        okToLeave.signal()
        readyToLeave--
    mutex.signal()

dine()

mutex.wait()
eating--
readyToLeave++
if eating == 1 and readyToLeave == 1:
    mutex.signal()
    okToLeave.wait()
elif eating == 0 and readyToLeave == 2:
    okToLeave.signal()
    readyToLeave -= 2
    mutex.signal()
else:
    readyToLeave--
    mutex.signal()

leave()
```

مشابه راه‌حل قبلی، در این راه‌حل نیز الگوی "بخاطر تو این کار را انجام خواهم داد" را بکار بردم چنانکه دانشجوی منتظر نباید دومرتبه میوتکس را بگیرد.

تفاوت اصلی بین این راه‌حل و راه‌حل قبلی در این است که اولین دانشجویی که به یک میز خالی می‌رسد باید منتظر بماند و دانشجوی دوم به هر دو اجازه می‌دهد که ادامه یابند. در هر حالت، نمی‌بایست وجود دانشجویان در حال انتظار برای خروج را بررسی کنیم چرا که سر یک میز خالی کسی نیست که بتواند آن را ترک نماید!

فصل ۸

همگام‌سازی در پایتون

با استفاده از شبه‌کد، از برخی جزئیات نازیبای همگام‌سازی در دنیای واقعی اجتناب کرده‌ایم. در این فصل، نگاهی به کدهای واقعی همگام‌سازی در پایتون می‌اندازیم و در فصل بعد به کدهای C می‌نگریم.

پایتون یک محیط چندنخی منطقاً خوشایند فراهم می‌آورد که با اشیاء سمافور کامل می‌گردد. این محیط چندنخی، نقطه ضعف‌هایی نیز دارد امّا چندین کد تمیزکاری در پیوست آ وجود که باعث می‌شود اوضاع کمی بهتر شود.

مثالی ساده در ادامه آمده است:

```
from threading_cleanup import *

class Shared:
    def __init__(self):
        self.counter = 0

def child_code(shared):
    while True:
        shared.counter += 1
        print shared.counter
        time.sleep(0.5)

shared = Shared()
children = [Thread(child_code, shared) for i in range(2)]
for child in children: child.join()
```

خط اول کد تمیزکاری از پیوست آ را اجرا می‌نماید؛ این خط را در مثال‌های بعدی نشان نخواهیم داد.

Shared یک نوعْ شیء است که حاوی متغیرهای اشتراکی خواهد بود. متغیرهای عمومی نیز بین نخ‌ها

مشترک هستند، امّا ما هیچ متغیری عمومی را در این مثال‌ها بکار نخواهیم برد. متغیرهایی که به واسطه تعریف شدن داخل یک تابع محلی محسوب می‌شوند باز به این معنی که مختص به یک نخ باشند نیز محلی هستند.

کد فرزند یک حلقه بی‌پایان است که مقدار counter را یک واحد افزایش داده، مقدار جدید را چاپ کرده و سپس به مدت ۵.۰ به خواب می‌رود.

نخ والد shared و دو فرزند را ایجاد نموده و سپس منتظر فرزندان می‌ماند تا خاتمه یابند (که در این حالت چنین نخواهد بود).

۱.۸ مسأله بررسی‌کننده میوتکس

دانشجویان کوشا در همگام‌سازی درخواهند یافت که فرزندان بروزرسانی‌های ناهمگامی نسبت به counter اعمال می‌کنند که این کار، امن نیست. اگر این کد را اجرا کنید ممکن است خطاهایی ببینید، امّا احتمالاً نخواهید دید. مسأله‌ای ناخوشایند درباره خطاهای همگام‌سازی این است که غیر قابل پیش‌بینی هستند، بدین معنی که حتی با تست‌های گسترده نیز ممکن آن خطاها آشکار نکند.

برای تشخیص خطاها معمولا ضروری است که عمل جستجوی خطا را خودکار نماییم. در این حالت، می‌توانیم خطاها را با دنبال کردن مقادیر counter تشخیص دهیم.

```
class Shared:
    def __init__(self, end=10):
        self.counter = 0
        self.end = end
        self.array = [0]* self.end

def child_code(shared):
    while True:
        if shared.counter >= shared.end: break
        shared.array[shared.counter] += 1
        shared.counter += 1

shared = Shared(10)
children = [Thread(child_code, shared) for i in range(2)]
for child in children: child.join()
print shared.array
```

در این مثال، shared حاوی یک لیست است (که به غلط array نامیده شده است) و تعداد دفعاتی که هر مقدار شمارنده استفاده شده است را نگه می‌دارد. با هر اجرای حلقه، فرزندان مقدار counter را بررسی نموده و اگر مقدار آن از end بیشتر شود خاتمه می‌یابند. اگر مقدار از حد مجاز نگذرد، از مقدار counter به عنوان اندیسی برای array استفاده نموده و مدخل متناظر را یک واحد افزایش می‌دهند. سپس مقدار counter را

یک واحد اضافه می‌نمایند.

اگر هر چیز به درستی عمل کند، هر مدخل آرایه باید دقیقاً یک بار به میزان یک واحد افزایش یابد. زمانیکه فرزندان خاتمه می‌یابند، والد از `join` باز می‌گردد و مقدار `array` را چاپ می‌کند. زمانیکه برنامه را اجرا کردم مقدار زیر را گرفتم

```
1] ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,[1
```

که مایوسانه صحیح است. اگر اندازه آرایه را افزایش دهیم، ممکن است انتظار خطاهای بیشتری را داشته باشیم، امّا بررسی نمودن نتیجه نیز مشکل‌تر می‌شود.

می‌توانیم چک کردن را با ایجاد یک هیستوگرام از نتایج آرایه به صورت خودکار درآوریم:

```
class Histogram(dict):
    def __init__(self, seq=[]):
        for item in seq:
            self[item] = self.get(item, 0) + 1

print Histogram(shared.array)
```

اکنون زمانیکه برنامه را اجرا کنم مقدار زیر را دریافت می‌دارم

```
{1: 10}
```

بدین معنی که مقدار ۱ همانطوری که انتظار می‌رفت ۱۰ مرتبه ظاهر شده است. هیچ خطایی تاکنون رخ نداده، امّا اگر مقدار end را بزرگتر کنیم، اوضاع جالب‌تر می‌شود:

```
end = 100, {1: 100}
end = 1000, {1: 1000}
end = 10000, {1: 10000}
end = 100000, {1: 27561, 2: 72439}
```

وای! زمانیکه end به اندازه کافی بزرگ باشد که تعویض بسترهای زیادی بین فرزندان وجود داشته باشد، خطاهای همگام‌سازی نمایان می‌شود. در این حالت، مقدار خیلی زیادی خطا دریافت می‌کنیم که نشان می‌دهد برنامه در یک الگوی تکراری افتاده است که نخ‌ها دائما در ناحیه بحرانی دچار وقفه می‌گردند.

این مثال یکی از خطرات خطاهای همگام‌سازی را نشان می‌دهد که این خطاها ممکن است بندرت رخ دهند امّا تصادفی نیستند. اگر یک خطا در هر میلیون، یک مرتبه رخ دهد، بدین معنی نیست که یک میلیون بار پشت سر هم اتفاق نیفتد.

معمّا: کد همگام‌سازی به این برنامه اضافه نمایید که دسترسی انحصاری به متغیرهای اشتراکی را اعمال نماید. می‌توانید کد این بخش را از greenteapress.com/semaphores/counter.py دانلود کنید.

۱.۱.۸ راهنمایی بررسی‌کننده میوتکس

نسخه‌ای از Shared که بکار برده‌ام در ادامه آمده است:

```
class Shared:
    def __init__(self, end=10):
        self.counter = 0
        self.end = end
        self.array = [0]* self.end
        self.mutex = Semaphore(1)
```

تنها تغییر، سمافوری است که mutex نامگذاری شده است که تعجبی هم ندارد.

۲.۱.۸ راه حل بررسی‌کننده میوتکس

راه‌حل در ادامه آمده است:

```
def child_code(shared):
    while True:
        shared.mutex.wait()
        if shared.counter < shared.end:
            shared.array[shared.counter] += 1
            shared.counter += 1
            shared.mutex.signal()
        else:
            shared.mutex.signal()
            break
```

گرچه این مسأله، مشکل‌ترین مسأله همگام‌سازی این کتاب نیست، امّا ممکن است شما درک درست جزئیات را کمی مستلزم زیرکی بدانید. بخصوص که فراموشی سیگنال دهی به میوتکس قبل از شکستن حلقه سبب بن‌بست می‌گردد.

من این راه‌حل را با `end = 1000000` اجرا کردم و نتیجه زیر را گرفتم:

```
{1: 1000000}
```

البته، این بدین معنی نیست که راه‌حل من درست است امّا برای شروع خوب است.

۲.۸ مسأله دستگاه خودکار کوکاکولا

برنامه بعدی، عملیات افزودن/برداشتن کوکاکولا به/از دستگاه خودکار کوکاکولا توسط تولیدکنندگان/مصرف‌کنندگان را شبیه‌سازی می‌نماید.

```
import random

class Shared:
    def __init__(self, start=5):
        self.cokes = start

def consume(shared):
    shared.cokes -= 1
    print shared.cokes

def produce(shared):
    shared.cokes += 1
    print shared.cokes

def loop(shared, f, mu=1):
    while True:
        t = random.expovariate(1.0/mu)
        time.sleep(t)
        f(shared)

shared = Shared()
fs = [consume]*2 + [produce]*2
threads = [Thread(loop, shared, f) for f in fs]
for thread in threads: thread.join()
```

ظرفیت ماشین ۱۰ کوکاکولا است و ماشین در ابتدا نیمه‌پر است. لذا متغیر اشتراکی cokes برابر ۵ است.

برنامه ۴ نخ ایجاد می‌نماید: دو تولیدکننده و دو مصرف‌کننده. هر دوی آن‌ها تابع loop را اجرا می‌نمایند امّا تولیدکننده produce و مصرف‌کننده consume را فرا می‌خواند. این توابع دسترسی ناهمگامی به یک متغیر اشتراکی دارند که این نوع دسترسی در اینجا پذیرفتنی نیست.

در هر اجرای حلقه، تولیدکنندگان و مصرف‌کنندگان برای یک مدتی به خواب می‌روند که مقدار آن به صورت تصادفی از یک توزیع نمایی با میانگین mu تعیین می‌شود. از آنجایی که دو تولیدکننده و دو مصرف‌کننده وجود دارد، در هر ثانیه به طور میانگین دو کوکاکولا به ماشین افزوده شده و دو کوکاکولا مصرف می‌گردد.

بنابراین به طور متوسط میزان کوکاکولاها ثابت است، امّا در کوتاه مدت، این مقدار می‌تواند به صورت فزاینده‌ای متفاوت باشد. اگر برنامه را برای مدتی اجرا کنید، احتمالا مقدار cokes را زیر صفر و یا بیشتر از ۱۰ خواهید دید. البته که هیچکدام از این دو نباید اتفاق بیفتند.

معمّا: کدی به این برنامه بیفزایید که محدودیت‌های همگام‌سازی زیر را اعمال نماید:

- دسترسی به cokes باید به صورت انحصاری باشد.
- اگر مقدار کوکاکولا صفر است، مصرف‌کننده باید تا زمانیکه کوکاکولا افزوده شود مسدود گردد.
- اگر مقدار کوکاکولا ۱۰ است، تولیدکننده باید تا زمانیکه کوکاکولایی مصرف شود مسدود گردد.

برنامه را می‌توانید از آدرس زیر دانلود کنید: greenteapress.com/semaphores/coke.py

۱.۲.۸ مسأله دستگاه خودکار کوکاکولا

متغیرهای اشتراکی که در راه‌حل خود بکار برده‌ام به شرح زیر است:

Listing:

```
class Shared:
    def __init__(self, start=5, capacity=10):
        self.cokes = Semaphore(start)
        self.slots = Semaphore(capacity-start)
        self.mutex = Semaphore(1)
```

الان cokes یک سمافور است (بجای عدد صحیح ساده)، که سبب می‌شود چاپ مقدار آن مستلزم مقداری زیرکی گردد. البته که شما هرگز نباید به مقدار یک سمافور دسترسی داشته باشید و پایتون با روش معمول خود، هیچکدام از شیوه‌های تقلب که در برخی از پیاده‌سازی‌ها را مشاهده می‌نمایید فراهم نمی‌کند.

امّا جالب است بدانید که مقدار یک سمافور در یک ویژگی خصوصی به نام _Semaphore__value ذخیره شده است. همچنین، موردی که نمی‌دانید این است که در واقع پایتون هیچ محدودیتی در دسترسی به ویژگی‌های خصوصی اعمال نمی‌نماید. البته که شما هرگز نباید از این امکان استفاده کنید امّا فکر کردم دانستن این نکته شاید برایتان جالب باشد.

۲.۲.۸ راه حل دستگاه خودکار کوکاکولا

اگر شما بقیه کتاب را خوانده باشید، با چیزی که حداقل به خوبی کد زیر است نباید هیچ مشکلی داشته باشید:

```
def consume(shared):
    shared.cokes.wait()
    shared.mutex.wait()
    print shared.cokes.value()
    shared.mutex.signal()
    shared.slots.signal()

def produce(shared):
    shared.slots.wait()
    shared.mutex.wait()
    print shared.cokes._Semaphore__value
    shared.mutex.signal()
    shared.cokes.signal()
```

اگر این برنامه را برای مدتی اجرا کنید، باید بتوانید تایید کنید که تعداد کوکاکولاها در ماشین هرگز منفی و یا بیشتر از ۱۰ نمی‌شود. بنابراین این راه‌حل به نظر درست می‌آید.

لااقل تا الان.

فصل ۹

همگام‌سازی در C

در این بخش یک برنامه چندنخی و همگام به زبان C خواهیم نوشت. پیوست ب کدهای کاربردی فراهم می‌آورد که من از آن‌ها به منظور دلچسب‌تر نمودن کد C استفاده می‌کنم. مثال‌های این بخش مبتنی بر آن کدهاست.

۱.۹ انحصار متقابل

با تعریف ساختاری که حاوی متغیرهای اشتراکی است شروع خواهم کرد:

```
typedef struct {
  int counter;
  int end;
  int *array;
} Shared;

Shared *make_shared (int end)
{
  int i;
  Shared *shared = check_malloc (sizeof (Shared));

  shared->counter = 0;
  shared->end = end;

  shared->array = check_malloc (shared->end * sizeof(int));
  for (i=0; i<shared->end; i++) {
    shared->array[i] = 0;
  }
  return shared;
```

```
}
```

counter یک متغیر اشتراکی است که بوسیله نخ‌های همروند مقدارش یک واحد یک واحد افزایش می‌یابد تا به مقدار end برسد. از array به منظور بررسی خطاهای همگام‌سازی، با ردگیری مقدار counter پس از هر افزایش، استفاده خواهیم کرد.

۱.۱.۹ کد والد

کدی که نخ والد به منظور ایجاد نخ‌ها و سپس انتظار برای اتمام آن‌ها بکار می‌برد در ادامه آمده است:

```
int main ()
{
  int i;
  pthread_t child[NUM_CHILDREN];

  Shared *shared = make_shared (100000);

  for (i=0; i<NUM_CHILDREN; i++) {
    child[i] = make_thread (entry, shared);
  }

  for (i=0; i<NUM_CHILDREN; i++) {
    join_thread (child[i]);
  }

  check_array (shared);
  return 0;
}
```

حلقهٔ اول نخ‌های فرزند را ایجاد می‌نماید؛ حلقه دوم منتظر می‌شود تا آن‌ها کارشان را تمام کنند. زمانیکه کار آخرین فرزند پایان یابد، نخ والد check_array را به منظور بررسی خطاها فراخوانی می‌نماید. make_thread و join_thread در پیوست ب تعریف شده‌اند.

۲.۱.۹ کد فرزند

کدی که توسط هر کدام از فرزندان اجرا می‌شود در ادامه آمده است:

```
void child_code (Shared *shared)
{
  while (1) {
    if (shared->counter >= shared->end) {
```

```
      return;
    }
    shared->array[shared->counter]++;
    shared->counter++;
  }
}
```

در هر اجرای حلقه، نخ‌های فرزند از counter به عنوان اندیس array استفاده نموده و عنصر متناظر را یک واحد افزایش می‌دهد. سپس مقدار counter را یک واحد افزایش داده و اتمام کار را بررسی می‌کند.

۳.۱.۹ خطاهای همگام‌سازی

اگر هر چیز به درستی عمل همایند، هر عنصر array باید یک واحد افزایش یابد. لذا به منظور بررسی خطاها، تنها تعداد عناصری را که نیستند 1 می‌شماریم.

```
void check_array (Shared *shared)
{
  int i, errors=0;

  for (i=0; i<shared->end; i++) {
    if (shared->array[i] != 1) errors++;
  }
  printf ("%d errors.\n", errors);
}
```

این برنامه را (بهمراه کدهای تمیزکاری)می‌توانید از آدرس زیر دانلود نمایید:

greenteapress.com/semaphores/counter.c

اگر برنامه را کامپایل و اجرا نمایید، خروجی برنامه باید مشابه زیر باشد:

```
Starting child at counter 0
10000
20000
30000
40000
50000
60000
70000
```

```
80000
90000
Child done.
Starting child at counter 100000
Child done.
Checking...
0 errors.
```

البته که تعامل میان فرزندان بسته به جزئیات سیستم عامل و همچنین دیگر برنامه‌های در حال اجرا روی کامپیوتر دارد. در مثالی که دیدید، پیش از اینکه نخ‌های دیگری آغاز به کار نمایند یک نخ، از 0 تا end حلقه را به طور کامل و به تنهایی طی می‌کند لذا تعجبی ندارد که هیچ خطای همگام‌سازی وجود نداشته باشد.

امّا همانطور که end بزرگتر می‌شود، تعویض بسترهای بیشتری بین فرزندان وجود دارد. در سیستم من، خطاها زمانی شروع شد که end به ۱۰۰۰۰۰۰۰ رسید.

معمّا: سمافورها را به منظور اعمال دسترسی انحصاری به متغیرهای اشتراکی بکار برید و برنامه را دومرتبه اجرا نمایید تا نشان دهید که خطایی وجود ندارد.

۴.۱.۹ راهنمایی انحصار متقابل

نسخه‌ای از Shared که در راه‌حل بکار برده‌ام، در زیر آمده است:

Listing:

```
typedef struct {
  int counter;
  int end;
  int *array;
  Semaphore *mutex;
} Shared;

Shared *make_shared (int end)
{
  int i;
  Shared *shared = check_malloc (sizeof (Shared));

  shared->counter = 0;
  shared->end = end;

  shared->array = check_malloc (shared->end * sizeof(int));
  for (i=0; i<shared->end; i++) {
    shared->array[i] = 0;
  }
  shared->mutex = make_semaphore(1);
  return shared;
}
```

خط ۵ متغیر mutex را به عنوان یک سمافور اعلان می‌نماید و خط ۲۰ مقدر اولیه 1 را به آن اختصاص می‌دهد.

۵.۱.۹ راه حل انحصار متقابل

نسخه همگام‌سازی شده کد فرزند در ادامه آمده است:

```
void child_code (Shared *shared)
{
  while (1) {
    sem_wait(shared->mutex);
    if (shared->counter >= shared->end) {
      sem_signal(shared->mutex);
      return;
    }

    shared->array[shared->counter]++;
    shared->counter++;
    sem_signal(shared->mutex);
  }
}
```

هیچ نکته شگفت‌انگیزی در اینجا وجود ندارد؛ تنها پیچیدگی مسأله این است که به یاد داشته باشید که پیش از دستور `return` میوتکس را آزاد نمایید.

کد این راه‌حل را می‌توانید از آدرس زیر دانلود نمایید. greenteapress.com/semaphores/counter_mutex.c

۲.۹ سمافورهای خودتان را بسازید

رایج‌ترین ابزارهای همگام‌سازی برای برنامه‌هایی از Pthreads استفاده می‌نمایند میوتکس‌ها و متغیرهای شرطی هستند و نه سمافورها. برای شرحی از این ابزارها، کتاب «برنامه‌نویسی با نخ‌هایPOSIX»[1] اثر Butenhof را پیشنهاد می‌کنم.

معمّا: میوتکس‌ها و متغیرهای شرطی را مطالعه نموده و سپس با استفاده از آن‌ها یک پیاده‌سازی از سمافورها بنویسید.

ممکن است بخواهید از کد کاربردی زیر را در راه‌حل خود استفاده کنید. این بسته‌بندی من از میوتکس‌های Pthreads است.

```
typedef pthread_mutex_t Mutex;

Mutex *make_mutex ()
{
  Mutex *mutex = check_malloc (sizeof(Mutex));
  int n = pthread_mutex_init (mutex, NULL);
  if (n != 0) perror_exit ("make_lock failed");
  return mutex;
}

void mutex_lock (Mutex *mutex)
{
  int n = pthread_mutex_lock (mutex);
  if (n != 0) perror_exit ("lock failed");
}

void mutex_unlock (Mutex *mutex)
{
  int n = pthread_mutex_unlock (mutex);
  if (n != 0) perror_exit ("unlock failed");
}
```

[1] *Programming with POSIX Threads*

و بسته‌بندی من از متغیرهای شرطی Pthreads:

```
typedef pthread_cond_t Cond;

Cond *make_cond ()
{
  Cond *cond = check_malloc (sizeof(Cond));
  int n = pthread_cond_init (cond, NULL);
  if (n != 0) perror_exit ("make_cond failed");
  return cond;
}

void cond_wait (Cond *cond, Mutex *mutex)
{
  int n = pthread_cond_wait (cond, mutex);
  if (n != 0) perror_exit ("cond_wait failed");
}

void cond_signal (Cond *cond)
{
  int n = pthread_cond_signal (cond);
  if (n != 0) perror_exit ("cond_signal failed");
}
```

۱.۲.۹ راهنمایی پیاده‌سازی سمافور

تعریف ساختار سمافورهایم در ادامه آمده است:

```
typedef struct {
  int value, wakeups;
  Mutex *mutex;
  Cond *cond;
} Semaphore;
```

مقدار سمافور در value قرار می‌گیرد. wakeups تعداد سیگنال‌های در حال انتظار را می‌شمارد که در واقع تعداد نخ‌هایی است که بیدار شده‌اند امّا هنوز اجرایشان ادامه نیافته است. دلیل وجود wakeups برای برآورده شدن خصوصیت ۳ سمافورها است که در بخش ۳.۴ شرح داده شد.

mutex دسترسی انحصاری به value و wakeups را فراهم می‌آورد؛ اگر نخ‌ها روی سمافور منتظر باشند در واقع روی متغیر شرطی cond منتظر می‌مانند.

کد مقداردهی اولیه این ساختار در ادامه آمده است:

```
Semaphore *make_semaphore (int value)
{
  Semaphore *semaphore = check_malloc (sizeof(Semaphore));
  semaphore->value = value;
  semaphore->wakeups = 0;
  semaphore->mutex = make_mutex ();
  semaphore->cond = make_cond ();
  return semaphore;
}
```

۲.۲.۹ پیاده‌سازی سمافور

پیاده‌سازی من از سمافورها با کمک میوتکس‌ها و متغیرهای شرطی Pthread در ادامه آمده است:

```
1  void sem_wait (Semaphore *semaphore)
2  {
3    mutex_lock (semaphore->mutex);
4    semaphore->value--;
5
6    if (semaphore->value < 0) {
7      do {
8        cond_wait (semaphore->cond, semaphore->mutex);
9      } while (semaphore->wakeups < 1);
10     semaphore->wakeups--;
11   }
12   mutex_unlock (semaphore->mutex);
13 }
14
15 void sem_signal (Semaphore *semaphore)
16 {
17   mutex_lock (semaphore->mutex);
18   semaphore->value++;
19
20   if (semaphore->value <= 0) {
21     semaphore->wakeups++;
22     cond_signal (semaphore->cond);
23   }
24   mutex_unlock (semaphore->mutex);
25 }
```

غالب این کد سر راست است؛ تنهای چیزی که ممکن است کمی پیچیدگی داشته باشد حلقه `do...while` در خط ۷ است. این کاربردی غیر معمول از یک متغیر شرطی است، امّا در اینجا ضروری است.

معمّا: چرا نمی‌توانیم حلقه `do...while` را با یک حلقه `while` جایگزین نماییم؟

۳.۲.۹ جزئیات پیاده‌سازی سمافور

با یک حلقه `while`، این پیاده‌سازی خصوصیت ۳ را نخواهد داشت. ممکن است یک نخ سیگنال داده و در ادامه اجرا سیگنال خودش را بگیرد.

با حلقه `do...while`، زمانیکه یک نخ سیگنال می‌دهد تضمین می‌شود که یکی از نخ‌های در حال انتظار، سیگنال را دریافت نماید حتی اگر نخ دیگری پیش از ادامه یافتن یکی از نخ‌های منتظر، میوتکس را در خط ۳ بگیرد[۲].

[۲] «««< HEAD البته این تضمین تقریبی است. چرا که یک بیداری نادرست در زمانی مناسب می‌تواند این تضمین را خدشه‌دار کند. برای توضیحات بیشتر لینک `http://en.wikipedia.org/wiki/Spurious_wakeup` را مطالعه نمایید.

کتاب‌نامه

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[2] Edsger Dijkstra. Cooperating sequential processes. 1965. Reprinted in *Programming Languages*, F. Genuys, ed., Academic Press, New York 1968.

[3] Armando R. Gingras. Dining philosophers revisited. *ACM SIGCSE Bulletin*, 22(3):21–24, 28, September 1990.

[4] Max Hailperin. *Operating Systems and Middleware: Supporting Controlled Interaction.* Thompson Course Technology, 2006.

[5] Joseph M. Morris. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 8:76–80, February 1979.

[6] David L. Parnas. On a solution to the cigarette smokers' problem without conditional statements. *Communications of the ACM*, 18:181–183, March 1975.

[7] Suhas Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. Technical report, MIT, 1971.

[8] Kenneth A. Reek. Design patterns for semaphores. In *ACM SIGCSE*, 2004.

[9] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, fourth edition, 2000.

[10] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, second edition, 2001.

پیوست آ

تمیزکاری نخ‌های پایتون

در مقایسه با بسیاری از محیط‌های چندنخی، نخ‌های پایتون خیلی خوب هستند، امّا چند تا ویژگی وجود دارد که مرا آزار می‌دهد. خوشبختانه، با یک کد کوچک تمیزکاری می‌توان این مسأله را برطرف نمود.

آ.۱ متدهای سمافور

در ابتدا متدهای سمافورهای پایتون `acquire` و `release` نامیده می شد که انتخابی کاملاً معقول است، امّا پس از چندین سال کار روی این کتاب، به `signal` و `wait` عادت کرده‌ام. خوشبختانه می‌توانم با تعریف زیرکلاس نسخه سمافور موجود در ماژول `threading` شیوه خودم را داشته باشم.

تغییر نام سمافور

```
import threading

class Semaphore(threading._Semaphore):
    wait = threading._Semaphore.acquire
    signal = threading._Semaphore.release
```

زمانیکه این کلاس تعریف شده باشد، می‌توانید با استفاده از نحو این کتاب به ایجاد و دستکاری سمافورها بپردازید.

مثال سمافور

```
mutex = Semaphore()
mutex.wait()
mutex.signal()
```

آ.۲ ایجاد نخ

ویژگی دیگری از ماژول threading که مرا می‌رنجاند، واسط[1] ایجاد و شروع نخ‌ها است. شیوه معمول، متغیرهای کلمه‌کلیدی و دو گام نیاز دارد.

مثال نخ (شیوه معمول)

```
import threading

def function(x, y, z):
    print x, y, z

thread = threading.Thread(target=function, args=[1, 2, 3])
thread.start()
```

در این مثال، ایجاد نخ هیچ اثر فوری ندارد. امّا زمانیکه start را فراخوانی می‌کنید، نخ جدید، تابع هدف را با آرگومان‌های داده شده اجرا می‌نماید. این شیوه در زمانیکه شما باید کاری را روی نخ پیش از شروع آن انجام دهید عالی است، ولی من تقریباً هرگز چنین نمی‌کنم. همچنین گمان می‌کنم که آرگومان‌های target و args نازیبا است.

خوشبختانه هر دوی این مسائل را می‌توانیم با چهار خط کد حل نماییم.

کلاس نخ تمیزکاری شده

```
class Thread(threading.Thread):
    def __init__(self, t, *args):
        threading.Thread.__init__(self, target=t, args=args)
        self.start()
```

اکنون می‌توانیم نخ‌ها را با یک واسط زیباتر ایجاد نماییم و آن‌ها به طور خودکار شروع می‌شوند.

مثال نخ (به روش من)

```
thread = Thread(function, 1, 2, 3)
```

این روش وامدار شیوه‌ای است که می‌پسندم و آن ایجاد چندین نخ با یک ادراک لیست است.

مثال چند نخی

```
threads = [Thread(function, i, i, i) for i in range(10)]
```

[1] interface

آ.۳ کنترل وقفه‌های صفحه کلید

یکی دیگر از مشکلات کلاس `threading` این است که `Thread.join` با کلیدهای `Ctrl-C` -که سیگنال `SIGINT` را تولید می‌نماید و پایتون آن را به یک `KeyboardInterrupt` ترجمه می‌نماید- دچار وقفه نمی‌شود.

لذا اگر کد زیر را بنویسید:

برنامه توقف‌ناپذیر

```
import threading, time

class Thread(threading.Thread):
    def __init__(self, t, *args):
        threading.Thread.__init__(self, target=t, args=args)
        self.start()

def parent_code():
    child = Thread(child_code, 10)
    child.join()

def child_code(n=10):
    for i in range(n):
        print i
        time.sleep(1)

parent_code()
```

درخواهید یافت که کد فوق با Ctrl-C یا SIGINT دچار وقفه نمی‌شود[۲]. راه‌حل من برای این مسأله از os.fork و os.wait استفاده می‌کند لذا این راه‌حل تنها روی سیستم‌های UNIX و Macintosh کار می‌کند. شیوه کار بدین صورت است: پیش از ایجاد نخ‌های جدید، برنامه تابع watcher را که یک فرآیند جدید را ایجاد می‌کند فراخوانی می‌نماید. فرآیند جدید برمی‌گردد و مابقی برنامه را اجرا می‌کند. فرآیند اصلی منتظر تکمیل فرآیند فرزند می‌ماند، از این رو نام آن watcher است.

The watcher

```
import threading, time, os, signal, sys

class Thread(threading.Thread):
    def __init__(self, t, *args):
        threading.Thread.__init__(self, target=t, args=args)
        self.start()

def parent_code():
    child = Thread(child_code, 10)
    child.join()

def child_code(n=10):
    for i in range(n):
        print i
```

[۲] در زمان این نگارش، این باگ به شماره ۱۱۶۷۹۳۰ گزارش شده است ولی هنوز باز بوده و وضعیت آن نامشخص است.

```
        time.sleep(1)

def watcher():
    child = os.fork()
    if child == 0: return
    try:
        os.wait()
    except KeyboardInterrupt:
        print 'KeyboardInterrupt'
        os.kill(child, signal.SIGKILL)
    sys.exit()

watcher()
parent_code()
```

اگر این نسخه از برنامه را اجرا کنید، باید بتوانید با استفاده از Ctrl-C آن را دچار وقفه نمایید. مطمئن نیستم ولی گمان می‌کنم که ارسال SIGINT به فرآیند watcher تضمین شده است لذا این یکی از آن مسائلی است که نخ‌های والد و فرآیند باید با آن برخورد نمایند.

تمامی این کد را در فایلی تحت نام threading_cleanup.py قرار داده‌ام که شما می‌توانید آن را از greenteapress.com/semaphores/threading_cleanup.py دانلود نمایید.

مثال‌های فصل ۸ با درک این نکته که این کد پیش از کد مثال، اجرا می‌گردد ارائه شده است.

پیوست ب

تمیزکاری نخ‌های POSIX

در این بخش، چند کد کاربردی که از آن به منظور خوشایندتر نمودن چند نخی در C استفاده می‌کنم را ارائه می‌دهم. مثال‌های بخش ۹ بر پایه این کد هستند.

احتمالاً محبوب‌ترین استاندارد چندنخی که در C استفاده می‌شود POSIX Threads یا به اختصار Pthreads است. استاندارد POSIX یک مدل نخ و یک واسط برای ایجاد و کنترل نخ‌ها تعریف می‌نماید. غالب نسخه‌های UNIX یک پیاده‌سازی از Pthreads را فراهم می‌نمایند.

ب.۱ کامپایل کد Pthreads

استفاده از Pthreads شبیه استفاده از غالب کتاب‌های C است:

- در ابتدای برنامه‌تان باید فایل‌های سرآمد[1] را وارد نمایید.
- کدی می‌نویسید که توابع تعریف‌شده توسط Pthreads را فراخوانی می‌نماید.
- با کامپایل برنامه، آن را به کتابخانه Pthreads متصل می‌نمایید.

من در مثال‌هایم از سرآمدهای زیر استفاده می‌کنم:

سرآمدها

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

[1] headers files

```
#include <pthread.h>
#include <semaphore.h>
```

دوتای اول سرآمدهای استاندارد هستند؛ سرآمد سوم برای Pthreads و چهارمی هم برای سمافورها. به منظور کامپایل با کتابخانه Pthreads در gcc، می‌توانید از گزینه l- در خط فرمان استفاده نمایید:

```
gcc -g -O2 -o array array.c -lpthread
```

دستور فوق array.c را با اطلاعات اشکال‌زدایی و بهینه‌سازی کامپایل نموده، به کتابخانه Pthreads متصل کرده و فایل اجرایی با نام array ایجاد می‌نماید.

اگر شما به زبانی مانند پایتون که مکانیزم مدیریت استثناء را فراهم می‌آورد عادت دارید، احتمالاً زبان‌هایی نظیر C که برای بررسی شرایط خطا نیاز به تصریح دارند برای شما آزار دهنده خواهند بود. اغلب برای کمتر کردن این مشکل، درون توابع خودم، فراخوانی توابع کتابخانه‌ای را همراه با کد بررسی خطای آن‌ها، مجتمع می‌نمایم. برای نمونه، نسخه malloc که مقدار خروجی را بررسی می‌نماید در ادامه آمده است.

```
void *check_malloc(int size)
{
  void *p = malloc (size);
  if (p == NULL) {
    perror ("malloc failed");
    exit (-1);
  }
  return p;
}
```

ب.۲ ایجاد نخ‌ها

با آن توابع Pthread که می‌خواهم از آن‌ها استفاده کنم کاری مشابه انجام داده‌ام: در ادامه بسته‌بندی خود را از pthread_create مشاهده می‌نمایید.

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
  int n;
  pthread_t thread;

  n = pthread_create (&thread, NULL, entry, (void *)shared);
  if (n != 0) {
    perror ("pthread_create failed");
    exit (-1);
  }
```

```
  return thread;
}
```

مقدار بازگشتی pthread_create از نوع pthread_t است که می‌توانید آن را به صورت دستگیره‌ای[۲] برای نخ جدید در نظر بگیرید. نیازی نیست نگران پیاده‌سازی pthread_t باشید امّا باید بدانید که از لحاظ معنایی یک نوع ابتدایی است[۳]. بدین معنی است که می‌توانید یک دستگیره نخ را به عنوان یک مقدار تغییرناپذیر در نظر بگیرید، لذا می‌توانید بدون مشکلی مقدار آن کپی یا ارسال نمایید. در اینجا بدین سبب به این نکته اشاره نمودم که این مطلب برای سمافورها صحیح نمی‌باشد و تا دقیقه‌ای دیگر به آن می‌پردازم.

اگر pthread_create موفقیت‌آمیز باشد، مقدار 0 را بر می‌گرداند و تابع من دستگیره نخ جدید را بر می‌گرداند. اگر خطایی رخ دهد، pthread_create یک کد خطا بر می‌گرداند و تابع من پیام خطا را چاپ نموده و خارج می‌شود.

پارامترهای pthread_create مقداری توضیح می‌طلبد. با پارامتر دوم، Shared، شروع می‌کنیم که یک ساختار تعریف شده توسط کاربر است که محتوی متغیرهای اشتراکی است. دستور typedef زیر یک نوع جدید ایجاد می‌نماید.

```
typedef struct {
  int counter;
} Shared;
```

در این جا، تنها متغیر اشتراکی counter است. تابع make_shared فضای لازم برای ساختار Shared را تخصیص داده و محتوای آن را مقداردهی اولیه می‌نماید.

```
Shared *make_shared ()
{
  int i;
  Shared *shared = check_malloc (sizeof (Shared));
  shared->counter = 0;
  return shared;
}
```

اکنون یک داده‌ساختار اشتراکی داریم، اجازه دهید به pthread_create برگردیم. اولین پارامتر اشاره‌گری به یک تابع است که اشاره‌گر void را دریافت نموده و یک اشاره‌گر void بر می‌گرداند. اگر نحو تعریف این نوع برای دیدگانتان ناآشنا است، بدانید که تنها نیستید. در هر حال، هدف این پارامتر، تعیین تابعی است که با اجرای نخ جدید، شروع خواهد شد. به طور قراردادی این تابع entry نامگذاری می‌شود.

```
void *entry (void *arg)
```

[۳] مثل یک عدد صحیح، برای نمونه این عدد صحیح همان چیزی است که pthread_t در تمام پیاده‌سازی‌هایی که از آن می‌دانم است.

[2] handle

```
{
  Shared *shared = (Shared *) arg;
  child_code (shared);
  pthread_exit (NULL);
}
```

پارامتر entry باید به صورت یک اشاره‌گر void تعریف شود، امّا در این برنامه می‌دانیم که واقعاً اشاره‌گری به یک ساختار Shared است لذا می‌توانیم مطابق آن تبدیل نوع را انجام داده و سپس آن را به child_code ارسال کنیم و این کد کار اصلی را انجام می‌دهد.

زمانیکه child_code باز می‌گردد، pthread_exit را فراخوانی می‌نماییم و این تابع می‌تواند برای ارسال یک مقدار به هر نخی (معمولاً نخ والد) که به این نخ ملحق می‌شود، بکار رود. در این جا، نخ فرزند کار خاصی انجام نمی‌دهد، لذا مقدار NULL ارسال می‌کنیم.

ب.۳ الحاق نخ‌ها

زمانیکه یک نخ بخواهد منتظر اتمام نخ دیگری بشود، تابع pthread_joins را صدا می‌زند. در ادامه بسته‌بندی pthread_join آمده است:

```
void join_thread (pthread_t thread)
{
  int ret = pthread_join (thread, NULL);
  if (ret == -1) {
    perror ("pthread_join failed");
    exit (-1);
  }
}
```

پارامتر، دستگیره نخی است که می‌خواهید منتظر آن بمانید. تمامی کاری این تابع انجام می‌دهد فراخوانی pthread_join و بررسی نتیجه است.

ب.۴ سمافورها

استاندارد POSIX یک واسط برای سمافورها تعریف می‌نماید. این واسط بخشی از Pthreads نیست، امّا غالب سیستم‌های مبتنی برای یونیکس که Pthreads را پیاده‌سازی می‌نمایند سمافورها را نیز فراهم می‌آورند. اگر با Pthreads مواجه شدید که سمافور ندارد خودتان می‌توانید سمافور خودتان را ایجاد نمایید؛ ر.ک. بخش ۲.۹.

سمافورهای POSIX از نوع sem_t هستند. نیازی به اطلاع از پیاده‌سازی این نوع ندارید، امّا باید بدانید که از لحاظ معنایی تنها یک ساختار است به این معنی که اگر آن را به یک متغیر نسبت دهید یک کپی از محتوای یک ساختار ایجاد نموده‌اید. کپی‌کردن یک سمافور تقریباً ایدهٔ بدی است. در POSIX، رفتار کپی تعریف نشده است.

در برنامه‌هایم، از حروف بزرگ برای اشاره به انواعی استفاده می‌کنم که معنای ساختاری دارند و همیشه آن را به اشاره‌گرها دستکاری می‌کنم. خوشبختانه براحتی می‌توان یک بسته‌بندی برای sem_t ایجاد نمود که رفتاری مشابه یک شیء محض داشته باشد. در ادامه typedef و بسته‌بندی ایجاد و مقداردهی اولیه سمافورها آمده است:

```
typedef sem_t Semaphore;

Semaphore *make_semaphore (int n)
{
  Semaphore *sem = check_malloc (sizeof(Semaphore));
  int ret = sem_init(sem, 0, n);
  if (ret == -1) {
    perror ("sem_init failed");
    exit (-1);
  }
  return sem;
}
```

تابع make_semaphore مقدار اولیه سمافور را به عنوان پارامتر می‌گیرد. فضای لازم برای یک Semaphore را تخصیص داده و آن را مقداردهی اولیه می‌نماید و اشاره‌گری به Semaphore بر می‌گرداند.

تابع sem_init فرم قدیمی گزارش خطای یونیکس را استفاده می‌کند که در صورت بروز خطا مقدار -1 را بر می‌گرداند. یک چیز جالب در رابطه با این توابع بسته‌بندی این است که لازم نیست به خاطر بسپاریم که هر تابع چه سبک گزارش خطا را بکار می‌برد.

با این تعاریف، می‌توانیم کدی به زبان C بنویسیم که تقریباً شبیه یک زبان برنامه‌نویسی واقعی است:

```
Semaphore *mutex = make_semaphore(1);
sem_wait(mutex);
sem_post(mutex);
```

سمافورهای POSIX بجای signal از post استفاده می‌کنند و این آزاردهنده است امّا می‌توانیم آن را برطرف کنیم:

```
int sem_signal(Semaphore *sem)
{
  return sem_post(sem);
}
```

تمیزکاری تا همین حد کافی است.