

کتاب کوچک سما فورہا

آلن بی. دونی

مترجمین:

سید محمد جواد رضویان، سید علی آل طہ و محمد مہدی قاسمی نیا

نسخہ ۲/۲/۱

کتاب کوچک سمافورها

ویرایش دوم

نسخه ۲/۲/۱

حق نشر ۲۰۱۶ آلن بی. دونی

کپی، توزیع و/یا تغییر این سند تحت لایسنس زیر مجاز است:

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

(CC BY-NC-SA 4.0) <http://creativecommons.org/licenses/by-nc-sa/4.0>

فرم اصلی این کتاب یک 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 کمک لاتک، dvips و ps2pdf که همگی برنامه‌های کدباز هستند تایپ شده است. سورس لاتک این کتاب در آدرس <http://greenteapress.com/semaphores> موجود

است.^۱

^۱ در ترجمه این کتاب از زی‌لاتک و بسته زی‌پرشین استفاده شده است.

پیشگفتار

غالب کتاب‌های درسی سیستم‌های عامل در دوره کارشناسی بخشی در همگام سازی دارند که به طور معمول شامل معرفی اجزای اولیه‌ای (موتکس، سمافور، ناظر و متغیرهای شرطی) و مسائل کلاسیک مثل خواننده نویسنده و تولیدکننده مصرف کننده. وقتی که من در برکلی کلاسی سیستم‌عامل را داشتم، و در کالج کالبی این درس را تدریس کردم، به این نتیجه رسیدم که بیشتر دانشجویان قادر به درک راه حل ارائه شده برای اینگونه مسائل هستند، اما تنها برخی از این دانشجویان توانایی [ارئه چنین راه حل‌هایی] [ارئه همان راه حل‌ها] و حل مسائل مشابه را دارند.

یکی از دلایلی که دانشجویان نمی‌توانند به طور عمیق این قبیل مسایل را بفهمند، این است که وقت و تلاش بیشتری می‌برند از آنچیزی که کلاس‌ها در اختیارشان می‌گذارد. همگام‌سازی یکی از ماژول‌هایی است که نسبت به دیگر ماژول‌ها وقت بیشتری نیاز دارد. و من مطمئن نیستم که بتوانم برای این منظور دلایلی را شرح دهم، منتها من فکر می‌کنم که سمافورها یکی از چالشی‌ترین، جالب‌ترین و سرگرمی‌ترین بخش‌های سیستم‌عامل می‌باشد. 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 نوشتم. نوشتن کدهمگام‌سازی چالش‌های مختص به خود را دارد زیرا که با افزایش تعداد اجزا و تعداد تعاملات به طور غیر قابل کنترل افزایش می‌یابد.

با این وجود در بین راه حل‌هایی که دیدم، الگوهای یافتن و حداقل برخی رهیافت‌های روشمند درست برای ترکیب راه حل‌ها رسیدم. شانس این را داشتم که در زمانی که در کالج ویلسلی بودم، این کتاب را به همراه کتاب درسی استاندارد استفاده کردم و در زمان تدریس درس مبحث همگام‌سازی را به شکل موازی با درس تدریس می‌کردم. هر هفته به دانشجویان چند صفحه از کتاب را می‌دادم که با یک معما تمام می‌شد و گاهی اوقات به راهنمایی مختصر. و به آنها توصیه می‌کردم که به راهنمایی نگاه نکنند مگر اینکه گیر افتاده باشند. و همچنین ابزارهایی برای تست راه حل‌ها دادم، به تخته مغناطیسی کوچک که می‌تونستن کدهاشون رو بنویسند و یک بسته آهنربا برای نمایش تردهای در حال اجرا.

نتیجه بسیار چشمگیر بود، هر چه زمان بیشتری در اختیار دانشجویان می‌گذاشتم، عمق فهمشون

بیشتر می شد، مهمتر اینکه غالبشون قادر به حل بیشتر معماها بودند، و در برخی حالات همان راه حل های کلاسیک را می یافتند و یا راه حل جدیدی را ایجاد می کردند. وقتی که رفتم کالج گام بعدی را با ایجاد کلاس فوق برنامه همگام سازی برداشتم، که در آن کلاس این کتاب تدریس می شد و همچنین پیاده سازی دستورات اولیه همگام سازی در زبان اسمبلی x86 و پاسیکس و پیتون. دانشجویانی که این درس را گرفتند در یافتن خطاهای نسخه نخست کمک کردند و چندان از آنها راه حل هایی بهتر از راه حل های من ارائه دادند در پایان ترم از هر کدام آنها خواستم که یک مسأله جدید با ترجیحا با یک راه حل بنویسند. از این مشارکت ها در نسخه دوم استفاده کردم.

بخش باقی مانده از پیشگفتار: همچنین، پس از عرضه ی ویرایش اول، کنث ریک (Kenneth Reek) مقاله ی «الگوهای طراحی سمافورها» را در «گروه ویژه ی علاقمند به آموزش علوم کامپیوتر در ACM» ارائه داد. او در این مقاله مسأله ای را که من به آن «مسئله ی سوشی بار» می گویم معرفی و دو راه حل برای اثبات الگوهایی که وی آن ها را «دست به دست کردن باتوم» و «این کار را برای تو می کنم» نامید مطرح کرد. هنگامی که با این الگوها آشنا شدم، توانستم آن ها را در مسائل ویرایش اول کتاب به کار برم و راه حل هایی تولید کنم که به نظرم بهتر هستند. تغییر دیگر در نسخه دوم، نوع نگارش یا نحو آن است. بعد از آنی که نسخه اول را نوشتم، من زبان برنامه نویسی پی تون را که تنها یکی از عالیترین زبان های برنامه نویسی است بلکه یک زبان بسیار شبیه به شبه کد است را یاد گرفتم. در نتیجه من از یک شبه کد شبیه به C در ویرایش نخست به یک شبه کد شبیه به زبان پی تون تغییر دادم. در حقیقت، من یک شبیه ساز نوشتم که بسیاری از راه حل های ارائه شده در این کتاب را می تواند اجرا کند. خواننده هایی هم که با زبان پی تون آشنا نیستند نیز (ان شاء الله) می توانند آن را درک کنند. در مواردی که از ویژگی های زبان پی تون استفاده کردم، نحو زبان پی تون و نحوی کار کد را شرح داده ام. امیدوارم این تغییر زمینه خوانا تر کردن کتاب را بوجود آورده باشد. صفحه بندی این کتاب ممکن است کمی عجیب به نظر برسد! اما صفحات خالی نیز خود یک روش سودمند است، بعد از هر معما، یک فضای خالی را تا شبه راهنمایی که در صفحه بعد است، گذاشته ام و بعد از آن یک صفحه خالی دیگر برای حل مسأله تا صفحه نمایش راه حل نهایی. زمانی که من از این کتاب در کلاس استفاده می کنم، برخی از صفحات را از کتاب جدا می کنم و دانشجویانم آنها را بعدا صحافی می کنند! این سیستم صفحه بندی امکان جدا کردن معما را بدون صفحات مربوط به راهنمایی ها، محقق می کند. بعضی اوقات بخش مربوط به راهنمایی را تا می کنم و از دیده شدن آن جلوگیری می کنم تا دانشجویان خود به حل مسأله پرداخته و در زمان مناسب راه حل را ببینند. اگر کتاب را به شکل تک صفحه چاپ کنید (به شکل یک رو سفید! زمانی که پوالتان زیادی کرده باشد! مترجم) می توانید از چاپ صفحه های سفید خودداری کنید (ظاهرا نویسنده برای ایالت های اصفهان نشین آمریکا هستند! مترجم). این کتاب یک کتاب رایگان است، این بدین معنی است که هر شخصی می تواند آنرا بخواند، رونوشت برداری کند، اصلاح کند و حتی بازپخش کند و اینها به دلیل نوع لیسانس مورد

استفاده برای این کتاب است. امیدوارم افراد این کتاب را مناسب و کارا ببینند، اما بیشتر از آن امیدوارم که آنها برای ادامه فرایند توسعه ایرادات و پیشنهادات خود و همینطور مطالب بیشتر خود را برایم ارسال کنند. با تشکر آلن دونی

نیدهام، ماساچوست

سه شنبه، ۱۲ خرداد ۱۳۸۳

لیست همکاران

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

- بسیاری از مسائل این کتاب گونه دیگری از مسائل کلاسیکی است که ابتدا در مقالات تخصصی آمده‌اند و سپس در کتب مرجع. هر کجا که منبع یک مسأله یا راه حل را بدانم در متن به آن اشاره خواهم داشت.
- همچنین از دانشجویان Wellesley College که با ویرایش اول این کتاب کار کرده‌اند تشکر می‌نمایم و نیز دانشجویان Olin College که با ویرایش دوم کتاب سر و کار داشتند.
- Se Won تصحیح کوچکی —لکن مهم— را در ارائه راه حل Tanenbaum نسب به مسأله فیلسوف‌های در حال غذا خوردن ارسال نموده است.
- Daniel Zingaro در مسأله 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 نسخه قبلی راه حل مسأله H_2O نشان داده است و سال بعد از آن نیز تعدادی خطاهای تایپی را متذکر شده است.
- Thomas Hansen یک خطای تایپی را در مسأله Cigarette smokers یافته است.
- Pascal Rütten به چندین اشکال تایپی اشاره نموده است از جمله تلفظ نادرست Edsger Dijkstra.
- Marcelo Johann خطایی را در راه حل مسأله Dining Savages یافته و آن را اصلاح کرده است.
- Roger Shipman تمام اصلاحات به علاوه یک گونه جذاب از مسأله Barrier را ارسال نموده است.
- Jon Cass یک از قلم افتادگی را در مسأله فیلسوف‌های در حال غذا خوردن مشخص نموده است.

- Krzysztof Kościuszkiewicz چندین اصلاح از جمله از قلم افتادن خطی در تعریف کلاس Fifo را فرستاده است.
- Fritz Vaandrager از دانشگاه Radboud هلند و دانشجویانش Manuel, Marc Schoolderman و Lars Lockefeer و Stampe ابزاری بنام UPPAAL را به منظور بررسی چندین راه حل این کتاب بکار برده و خطاهایی را در راه حل‌های ارائه شده برای مساله‌های 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 در راه حل مساله رقص انحصاری را یافته است.
- Eus یک خطای تایپی در فصل سوم را پیدا کرده است.
- Tak-Shing Chan یک خطای خارج از محدوده^۲ را در `counter_mutex.c` یافته است.
- Roman V. Kiseliiov 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 راه حل Kenneth Reek نسبت به مساله Sushi Bar یافته است.
- Martin Storsjö تصحیحی در مساله generalized smokers را ارسال نموده است.
- Cris Hawkins به یک متغیر بدون استفاده اشاره نموده است.
- Adolfo Di Mare یک "and" از جا افتاده را یافته است.
- Simon Ellis یک خطای تایپی را یافته است.
- Benjamin Nash یک خطای تایپی و خطایی در یک راه حل و مشکل دیگری را یافته است.
- Alejandro Pulver مشکلی را در راه حل مساله Barbershop یافته است.

²out-of-bounds

فهرست مطالب

آ	پیشگفتار
۱	۱ معرفی
۱	۱.۱ به‌هنگام سازی
۲	۲.۱ مدل اجرایی
۵	۳.۱ تسلل به کمک پیام‌دهی
۷	۴.۱ عدم قطعیت
۷	۵.۱ متغیرهای اشتراکی
۸	۱.۵.۱ نوشتن‌های همروند
۸	۲.۵.۱ بروزرسانی‌های همروند
۱۰	۳.۵.۱ انحصار متقابل با تبادل پیام
۱۱	۲ سمافورها
۱۱	۱.۲ تعریف
۱۲	۲.۲ نحو
۱۴	۳.۲ چرا سمافورها؟
۱۵	۳ الگوهای همگام سازی پایه
۱۵	۱.۳ علامت‌دهی
۱۶	۲.۳ Sync.py
۱۶	۳.۳ قرار ملاقات
۱۹	۱.۳.۳ اشاره ای در خصوص قرار ملاقات

۲۱	راه حل قرار ملاقات	۲.۳.۳
۲۱	بن بست #۱	۳.۳.۳
۲۲	Mutex	۴.۳
۲۳	راهنمای انحصار متقابل	۱.۴.۳
۲۵	راه حل انحصار متقابل	۲.۴.۳
۲۶	Multiplex	۵.۳
۲۷	راه حل مالتی پلکس	۱.۵.۳
۲۸	حصار	۶.۳
۲۹	راهنمای حصار	۱.۶.۳
۳۱	نا راه حل حصار	۲.۶.۳
۳۳	بن بست #۲	۳.۶.۳
۳۵	راه حل حصار	۴.۶.۳
۳۷	بن بست #۳	۵.۶.۳
۳۷	حصار با قابلیت استفاده مجدد	۷.۳
۳۹	نا راه حل حصار با قابلیت استفاده مجدد #۱	۱.۷.۳
۴۱	مساله حصار با قابلیت استفاده مجدد #۱	۲.۷.۳
۴۳	نا راه حل حصار با قابلیت استفاده مجدد #۲	۳.۷.۳
۴۵	راهنمای حصار با قابلیت استفاده مجدد	۴.۷.۳
۴۷	راه حل حصار با قابلیت استفاده مجدد	۵.۷.۳
۴۹	ترن استایل از پیش باگذاری شده	۶.۷.۳
۵۱	اشياء حصار	۷.۷.۳
۵۲	صف	۸.۳
۵۳	راهنمایی برای معما	۱.۸.۳
۵۵	راه حل صف	۲.۸.۳
۵۷	راهنمای صف انحصاری	۳.۸.۳
۵۹	راه حل صف انحصاری	۴.۸.۳
۶۱	مسائل همگام سازی کلاسیک	۴
۶۱	مسئله تولیدکننده-مصرف کننده	۱.۴
۶۳	راهنمای تولیدکننده-مصرف کننده	۱.۱.۴

۶۵	راه حل تولیدکننده- مصرف کننده	۲.۱.۴
۶۷	بن بست #۴	۳.۱.۴
۶۷	تولیدکننده- مصرف کننده با یک بافر متناهی	۴.۱.۴
۶۹	راهنمای بافر محدود تولیدکننده- مصرف کننده	۵.۱.۴
۷۱	راه حل بافر محدود تولیدکننده- مصرف کننده	۶.۱.۴
۷۱	مساله خوانندگان- نویسندگان	۲.۴
۷۳	راهنمای خوانندگان- نویسندگان	۱.۲.۴
۷۵	راه حل خوانندگان- نویسندگان	۲.۲.۴
۷۸	قحطی	۳.۲.۴
۸۱	راهنمایی خوانندگان- نویسندگان بدون قحطی	۴.۲.۴
۸۳	راه حل خوانندگان- نویسندگان بدون قحطی	۵.۲.۴
۸۵	راهنمایی خوانندگان- نویسندگان با اولویت نویسنده	۶.۲.۴
۸۷	راه حل نویسندگان- خوانندگان با اولویت نویسنده	۷.۲.۴
۸۹	میوتکس بدون قحطی	۳.۴
۹۳	راهنمای میوتکس بدون قحطی	۱.۳.۴
۹۵	راه حل میوتکس بدون قحطی	۲.۳.۴
۹۹	غذا خوردن فیلسوف ها	۴.۴
۱۰۳	بن بست #۵	۱.۴.۴
۱۰۵	راهنمایی غذا خوردن فیلسوف ها #۱	۲.۴.۴
۱۰۷	راه حل غذا خوردن فیلسوف ها #۱	۳.۴.۴
۱۰۹	راه حل غذا خوردن فیلسوف ها #۲	۴.۴.۴
۱۱۱	راه حل تنبام	۵.۴.۴
۱۱۳	قطعی تنبام	۶.۴.۴
۱۱۵	مساله سیگاری ها	۵.۴
۱۱۹	بن بست #۶	۱.۵.۴
۱۲۱	راهنمایی مساله سیگاری ها	۲.۵.۴
۱۲۳	راه حل مساله سیگاری	۳.۵.۴
۱۲۴	تعمیم مساله سیگاری ها	۴.۵.۴
۱۲۵	راهنمای تعمیم مساله سیگاری ها	۵.۵.۴
۱۲۷	راه حل تعمیم یافته مساله سیگاری ها	۶.۵.۴

۱۲۹	۵ مسائل همگام‌سازی کمتر- کلاسیک
۱۲۹	۱.۵ مساله غذاخوردن وحشی‌ها
۱۳۱	۱.۱.۵ راهنمایی غذاخوردن وحشی‌ها
۱۳۳	۲.۱.۵ راه حل غذاخوردن وحشی‌ها
۱۳۵	۲.۵ مساله آرایشگاه
۱۳۷	۱.۲.۵ راهنمایی آرایشگاه
۱۳۹	۲.۲.۵ راه حل آرایشگاه
۱۴۱	۳.۵ آرایشگاه FIFO
۱۴۳	۱.۳.۵ راهنمایی آرایشگاه FIFO
۱۴۵	۲.۳.۵ راه حل آرایشگاه FIFO
۱۴۷	۴.۵ مسأله آرایشگاه هیلزر
۱۴۸	۱.۴.۵ راهنمایی آرایشگاه هیلزر
۱۴۹	۲.۴.۵ راه حل آرایشگاه هیلزر
۱۵۳	۵.۵ مساله بابا نوئل
۱۵۵	۱.۵.۵ راهنمایی مساله بابا نوئل
۱۵۷	۲.۵.۵ راه حل مساله بابا نوئل
۱۵۹	۶.۵ ساخت H_2O
۱۶۱	۱.۶.۵ راهنمایی H_2O
۱۶۳	۲.۶.۵ راه حل H_2O
۱۶۴	۷.۵ مساله عبور از رودخانه
۱۶۷	۱.۷.۵ راهنمایی عبور از رودخانه
۱۶۹	۲.۷.۵ راه حل عبور از رودخانه
۱۷۱	۸.۵ The coaster roller problem
۱۷۳	۱.۸.۵ hint Coaster Roller
۱۷۵	۲.۸.۵ solution Coaster Roller
۱۷۷	۳.۸.۵ problem Coaster Roller Multi-car
۱۷۹	۴.۸.۵ hint Coaster Roller Multi-car
۱۸۱	۵.۸.۵ solution Coaster Roller Multi-car

۱۸۳	problems Not-so-classical ۶
۱۸۳	problem search-insert-delete The ۱.۶
۱۸۵	hint Search-Insert-Delete ۱.۱.۶
۱۸۷	solution Search-Insert-Delete ۲.۱.۶
۱۸۸	problem bathroom unisex The ۲.۶
۱۸۹	hint bathroom Unisex ۱.۲.۶
۱۹۱	solution bathroom Unisex ۲.۲.۶
۱۹۳	problem bathroom unisex No-starve ۳.۲.۶
۱۹۵	solution bathroom unisex No-starve ۴.۲.۶
۱۹۵	problem crossing Baboon ۳.۶
۱۹۶	Problem Hall Modus The ۴.۶
۱۹۹	hint problem Hall Modus ۱.۴.۶
۲۰۱	solution problem Hall Modus ۲.۴.۶
۲۰۵	problems classical remotely Not ۷
۲۰۵	problem bar sushi The ۱.۷
۲۰۷	hint bar Sushi ۱.۱.۷
۲۰۹	non-solution bar Sushi ۲.۱.۷
۲۱۱	non-solution bar Sushi ۳.۱.۷
۲۱۳	۱# solution bar Sushi ۴.۱.۷
۲۱۵	۲# solution bar Sushi ۵.۱.۷
۲۱۶	problem care child The ۲.۷
۲۱۷	hint care Child ۱.۲.۷
۲۱۹	non-solution care Child ۲.۲.۷
۲۲۱	solution care Child ۳.۲.۷
۲۲۱	problem care child Extended ۴.۲.۷
۲۲۳	hint care child Extended ۵.۲.۷
۲۲۵	solution care child Extended ۶.۲.۷
۲۲۷	problem party room The ۳.۷
۲۲۹	hint party Room ۱.۳.۷

۲۳۱	solution party Room	۲.۳.۷
۲۳۵	problem Bus Senate The	۴.۷
۲۳۷	hint problem Bus	۱.۴.۷
۲۳۹	۱# solution problem Bus	۲.۴.۷
۲۴۱	۲# solution problem Bus	۳.۴.۷
۲۴۳	problem Hall Faneuil The	۵.۷
۲۴۵	Hint Problem Hall Faneuil	۱.۵.۷
۲۴۷	solution problem Hall Faneuil	۲.۵.۷
۲۵۱	Hint Problem Hall Faneuil Extended	۳.۵.۷
۲۵۳	solution problem Hall Faneuil Extended	۴.۵.۷
۲۵۷	problem Hall Dining	۶.۷
۲۵۹	hint problem Hall Dining	۱.۶.۷
۲۶۱	solution problem Hall Dining	۲.۶.۷
۲۶۲	problem Hall Dining Extended	۳.۶.۷
۲۶۳	hint problem Hall Dining Extended	۴.۶.۷
۲۶۵	solution problem Hall Dining Extended	۵.۶.۷

۲۶۷		Python in Synchronization	۸
۲۶۸	problem checker Mutex	۱.۸
۲۷۱	hint checker Mutex	۱.۱.۸
۲۷۳	solution checker Mutex	۲.۱.۸
۲۷۵	problem machine coke The	۲.۸
۲۷۷	hint machine Coke	۱.۲.۸
۲۷۹	solution machine Coke	۲.۲.۸

۲۸۱		C in Synchronization	۹
۲۸۱	exclusion Mutual	۱.۹
۲۸۲	code Parent	۱.۱.۹
۲۸۲	code Child	۲.۱.۹
۲۸۳	errors Synchronization	۳.۱.۹
۲۸۵	hint exclusion Mutual	۴.۱.۹

۲۸۷	solution exclusion Mutual	۵.۱.۹
۲۸۹	semaphores own your Make	۲.۹
۲۹۱	hint implementation Semaphore	۱.۲.۹
۲۹۳	implementation Semaphore	۲.۲.۹
۲۹۵	detail implementation Semaphore	۳.۲.۹
۲۹۹		threads Python up Cleaning	آ
۲۹۹	methods Semaphore	۱.آ
۳۰۰	threads Creating	۲.آ
۳۰۱	interrupts keyboard Handling	۳.آ
۳۰۵		threads POSIX up Cleaning	ب
۳۰۵	code Pthread Compiling	۱.ب
۳۰۶	threads Creating	۲.ب
۳۰۸	threads Joining	۳.ب
۳۰۹	Semaphores	۴.ب

فصل ۱

معرفی

۱.۱ به‌هنگام‌سازی

اصطلاحاً همگام‌سازی به معنی وقوع هم‌زمان دو چیز است. در سیستم‌های کامپیوتری همگام‌سازی کلی‌تر است. این به معنی رابطه مابین رویدادهاست، در هر تعداد از رویدادها و هر نوع رابطه (قبل، حین، بعد). غالباً برنامه‌نویسان با محدودیت‌های همگام‌سازی مواجه‌اند، که این محدودیت‌ها الزاماتی در ارتباط با ترتیب این رخدادها می‌باشد.

تسلسل: رخداد الف پیش از رخداد ب اتفاق می‌افتد.

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

در زندگی واقعی غالباً محدودیت‌های همگامی‌سازی را با کمک یک ساعت بررسی و اعمال می‌کنیم. چگونه می‌فهمیم که رخداد الف قبل از رخداد ب رخ داده است؟ با دانستن زمان رخداد هر دو واقعه را بدانیم، می‌توانیم زمان‌ها را با هم مقایسه کنیم. در سیستم‌های کامپیوتری غالباً نمی‌توانیم از ساعت در محدودیت‌های همگام‌سازی‌های کامپیوتری را برآورده کنیم، زیرا که هیچ ساعت جهانی به دلیل اینکه زمان دقیق وقوع رویدادها را نمی‌دانیم. این کتاب درباره تکنیک‌های نرم‌افزار برای اعمال‌های محدودیت‌های همگام‌سازی در کامپیوتر است.

۲.۱ مدل اجرایی

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، اول اجرا می‌گردد.

در دو صورت همگام‌سازی پیچیده 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 پردازنده دیگر اجرا شده است.

و یا ممکن است یک پردازنده چندین نخ اجرایی داشته باشد. نخ دنباله‌ای از دستورات است که به 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 نخ دیگری منتقل شود و به همین ترتیب ادامه دهد.

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

در همگام‌سازی، تفاوتی بین مدل موازی و مدل چند نخ وجود ندارد. مساله یکی است—در یک پردازنده (یا یک نخ) ترتیب اجرا مشخص است اما بین پردازنده‌ها (یا نخ‌ها) بیان این ترتیب غیر ممکن است.

یک مثال واقعی این مساله را روشن‌تر می‌نمایند. تصور کنید که شما و دوستان 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. اما 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 اما همیشه در میزان دقت ساعت‌ها محدودیت وجود دارد. به علاوه، در بیشتر وقت‌ها 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.

معملاً: 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؟

۳.۱ تسلل به کمک پیام‌دهی

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 ایده پایه آن، تبادل پیام^۱، راه حل واقعی برای بسیاری از مسائل همگام‌سازی می‌باشد. جدول زمانی زیر را در نظر بگیرید.

نخ A (شما)	نخ B (Bob)
1 Eat breakfast	1 Eat breakfast
2 Work	2 Wait for a call
3 Eat lunch	3 Eat lunch
4 Call Bob	

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

$$a_1 < a_2 < a_3 < a_4$$

$$b_1 < b_2 < b_3$$

که رابطه $a_1 < a_2$ به معنای وقوع a_1 پیش از a_2 است. ولی در کل هیچ راهی برای مقایسه رخدادهای نخ‌های مختلف نداریم؛ برای مثال ایده‌ای از اینکه چه کسی ابتدا صبحانه می‌خورد نداریم (آیا $b_1 < a_1$ است؟). اما با کمک تبادل پیام (تماس تلفنی) می‌توانیم بگوییم چه کسی زودتر ناهار خورده است ($a_3 < b_3$). با فرض اینکه باب هیچ دوست دیگری نداشته باشد هیچ تماسی جز از شما دریافت نخواهد کرد بنابراین ($b_2 > a_4$). با ترکیب تمامی روابط، داریم

$$b_3 > b_2 > a_4 > a_3$$

که ثابت می‌کند شما قبل از باب ناهار خورده‌اید. در این حالت، می‌گوییم شما و باب به صورت متوالی^۲ ناهار خورده‌اید زیرا ترتیب وقایع را می‌دانیم. از طرف دیگر صبحانه را به صورت همروند^۳ خورده‌اید زیرا که ترتیب مشخص نیست. مواقعی که درباره رخدادهای همروند صحبت می‌کنیم، اینکه بگوییم آن‌ها در یک زمان یا به صورت

¹message passing

²sequential

³concurrent

همزمان رخ می‌دهد بی‌راه نیست هر چند که دقیق هم نیست. 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.

گاهی اوقات پس از اجرای برنامه می‌توانیم بگوییم که کدامیک ابتدا رخ داده است اما 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**^۴ هستند به این معنی که با نگاه به برنامه امکان اینکه بگوییم با اجرای آن چه چیزی رخ خواهد داد، وجود ندارد. در ادامه یک برنامه ساده غیر قطعی آمده است:

نخ A	نخ B
۱ print "yes"	۱ 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.

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

۵.۱ متغیرهای اشتراکی

بیشتر مواقع، غالب متغیرها در اکثر نخ‌ها **محلی**^۵ 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 خواهد داشت زیرا که نخ‌ها دخالتی در آن متغیرها ندارند.

اما گاهی اوقات برخی متغیرها بین دو یا چند نخ به صورت **اشتراکی**^۶ هستند؛ این یکی از شیوه‌های تعامل نخ‌ها با یکدیگر است. برای مثال، یک راه 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 مساله 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

^۴non-determinism

^۵local

^۶shared

است. دو بخش بعدی با این تعاملات سر و کار خواهد داشت. خواندن همروند متغیرهای اشتراکی که گونه دیگری از این تعامل است عموماً مشکل همگام‌سازی تولید نمی‌نماید.

۱.۵.۱ نوشتن‌های همروند

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

نخ A	نخ B
<pre> 1 x = 5 2 print x </pre>	<pre> 1 x = 7 </pre>

کدام مقدار x چاپ خواهد شد؟ در پایان اجرای تمام این دستورات، مقدار x چیست؟ این بستگی به ترتیب اجرای هر یک از دستورات، که به آن مسیر اجرا^۷ گفته می‌شود، دارد. $a_1 < a_2 < b_1$ یکی از مسیرهای ممکن است که در آن خروجی برنامه ۵ است، درحالی‌که مقدار نهایی ۷ خواهد بود.

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

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

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

پاسخ به چنین سؤالاتی یکی از بخش‌های مهم برنامه‌نویسی همروند است: مسیرهای ممکن کدام‌ها هستند و هر یک از این مسیرها چه تأثیراتی 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^۸ است که مقدار جدید، 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	نخ B
<pre> 1 count = count + 1 </pre>	<pre> 1 count = count + 1 </pre>

^۷execution path

^۸increment

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 روزرسانی شامل دو گام است: یک خواندن و یک نوشتن. اگر کد را، با یک متغیر موقتی temp بازنویسی نماییم، این مشکل واضح‌تر خواهد شد.

نخ A	نخ B
<pre> 1 temp = count 2 count = temp + 1 </pre>	<pre> 1 temp = count 2 count = temp + 1 </pre>

اکنون مسیر اجرای زیر را در نظر بگیرید

$$a_1 < b_1 < b_2 < a_2$$

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 برخوردار هستند زیرا که همیشه این امکان وجود ندارد که با نگاه کردن به یک برنامه سطح بالا بگوییم کدام عملیات در یک گام انجام شده و کدام‌ها وقفه‌پذیر هستند. در واقع، برخی کامپیوترها دستور افزایشی را فراهم می‌آورند که به صورت سخت‌افزاری پیاده‌سازی شده است و وقفه‌پذیر نیست. عملی که وقفه‌پذیر نباشید اتمی^۹ گفته می‌شود.

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

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

معمول‌ترین محدودیت انحصار متقابل^{۱۰} 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.

معملاً: فرض کنید ۱۰۰ تا نخ برنامه زیر را به صورت همزمان اجرا می‌نمایند (اگر با زبان پایتون آشنا نیستند حلقه for یکصد مرتبه روزرسانی انجام می‌دهد):

^۹atomic

^{۱۰}mutual exclusion

```

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.

۳.۵.۱ انحصار متقابل با تبادل پیام

همانند تسلیل، 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 کنترل می‌نمایید. غالب زمان‌ها، هر دو شما چراغ‌های اخطار را مشاهده می‌نمایید اما هر دو اجازه دارید برای ناهار دست از کار بکشید. اینکه چه کسی اول ناهار می‌خورد اهمیتی ندارد اما مهم است که ناهار هر دوی شما همزمان نباشد تا راکتور بدون نظارات باقی نماند!

معملاً: 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 است. در نرم‌افزار، سمافور ساختمان داده‌ای است که برای حل انواع 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 ایده اصلی یکسان است.

۱.۲ تعریف

سمافور شبیه یک عدد صحیح منتهی با سه تفاوت است:

۱. زمانیکه سمافوری را ایجاد می‌نمایید مقدار اولیه آن را می‌توانید هر عدد صحیحی قرار دهید، اما پس از آن تنها اعمال مجاز، افزایش و کاهش آن هم به اندازه یک واحد است و مقدار جاری سمافور را نمی‌تواند بخوانید.
۲. زمانیکه نخ‌ی سمافوری را کاهش می‌دهد اگر نتیجه مقداری منفی باشد، نخ خودش را مسدود^۱ 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.

¹block

وقتی می‌گوییم یک نخ خود را مسدود کرده است منظور این است که به زمان‌بند اعلام می‌نماید که دیگر نمی‌تواند ادامه دهد. تا زمانیکه واقع‌ای 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، اما این تعریف پی‌آمدهای در پی دارد که ممکن است بخواهید درباره آن‌ها ببینید.

- به طور کلی، نمی‌توان از قبل گفت که کاهش سمافور توسط یک نخ منجر به مسدود شدن آن می‌شود یا نه (در حالت‌های خاص ممکن است بتوانید ثابت کنید که مسدود می‌شود یا خیر).
- پس از اینکه نخ 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 کدام از این دو نخ، بلافاصله ادامه می‌یابد.
- زمانیکه به یک سمافور سیگنال می‌دهید ضرورتاً نمی‌دانید که آیا نخ 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 بلاک شدن کاهش یابند. 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 نخ در حالت انتظار نیست اما اگر نخ یک سمافور را کاهش دهد، مسدود خواهد شد.

۲.۲ نحو

در بیشتر محیط‌های برنامه‌نویسی، پیاده‌سازی سمافور به صورت بخشی از زبان برنامه‌نویسی یا سیستم عامل 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 شیوه عملکرد سمافورها استفاده می‌کنیم. نحو ایجاد یک سمافور و مقداردهی اولیه آن در ادامه آمده است.

نحو مقداردهی اولیه سمافور

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

^۲waking

^۳pseudo-language

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

اعمال سمافور

```
1 fred.increment()
2 fred.decrement()
```

و

اعمال سمافور

```
1 fred.signal()
2 fred.wait()
```

و

اعمال سمافور

```
1 fred.V()
2 fred.P()
```

وجود این همه نام ممکن تعجب بر انگیز باشد اما بی‌دلیل نیست. `increment` و `decrement` بیان می‌نمایند که این اعمال چه انجام می‌دهند. `signal` و `wait` شرح می‌دهند که به 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

بقیه اسمی 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

```
1 fred.increment_and_wake_a_waiting_process_if_any()
2 fred.decrement_and_block_if_the_result_is_negative()
```

گمان نمی‌کنم کسی به این زودی این اسمی را بپذیرد. با این وجود، به منظور استفاده، `signal` و `wait` را انتخاب می‌کنم.

^۴ constructor

^۵ اگر زبان شما هلندی باشد `V` و `P` آنقدرها هم بی‌معنی نیستند.

۳.۲ چرا سمافورها؟

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

- سمافورها محدودیت‌هایی تعمّدی تحمیل می‌نمایند که به برنامه‌نویس‌ها کمک می‌کند تا از خطاها بدور باشند.
- راه‌حلی که از سمافورها استفاده می‌نمایند اغلب تمیز و سازمان‌یافته است به گونه‌ای که اثبات درستی آن‌ها را ساده می‌سازد.
- سمافورها می‌توانند به صورتی کارا در بسیاری از سیستم‌ها پیاده‌سازی شوند، لذا راه‌حلی 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 است و نشان داده می شود چگونه با استفاده از سمافورها آن ها را حل کنیم. این مسائل شامل موضوعات مختلفی می شود از جمله تسلسل و انحصار متقابل — که قبلاً با آن ها آشنا شده ایم —.

۱.۳ علامت دهی

ساده ترین شکل استفاده از یک سمافور احتمالاً مکانیزم **علامت دهی**^۱ است، و به این معناست که یک نخپیمای به نخ دیگر می فرستد تا وقوع رخدادی را اعلام کند. علامت دهی این امکان را فراهم می آورد تا مطمئن شویم که یک قطعه کد از یک نخ، حتماً قبل از قطعه کدی دیگر در نخ دیگری اجرا خواهد شد؛ به عبارت دیگر، مسئله تسلسل را حل می کند. فرض کنید یک سمافور با نام `sem` و مقدار اولیه ۰ داریم، و نخ های `A` و `B` هر دو به آن دسترسی دارند.

نخ A		نخ B	
1	<code>statement a1</code>	1	<code>sem.wait()</code>
2	<code>sem.signal()</code>	2	<code>statement b1</code>

کلمه `statement` نشان دهنده یک عبارت دلخواه در برنامه است. برای اینکه مثال مشخص تر شود، فرض کنید `a1` یک خط از یک فایل را می خواند، و `b1` آن خط را در صفحه نمایش نشان می دهد. سمافور در این برنامه تضمین می کند که نخ `A` عملیات `a1` را، قبل از آنکه نخ `B` عملیات `b1` را شروع کند، به طور

¹signaling

کامل انجام داده است.

روش کار به این صورت است: اگر اول نخ B به عبارت wait برسد، با مقدار اولیه، یعنی صفر، مواجه می شود و بلاک [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 می یابد. بهرحال ترتیب a1 و b1 تضمین می شود. این شیوه استفاده از سمافورها، پایه و اساس نام های signal و wait است، و در این مورد، این اسامی به راحتی به خاطر سپرده می شوند. اما متأسفانه، موارد دیگری را خواهیم دید که این اسم ها کمتر به ما کمک می کنند.

حالا که از اسامی بامعنی صحبت می کنیم، باید بدانیم که sem دارای این شرایط نیست. اگر امکان داشته باشد، ایده ی 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: درباره ی استفاده از sync بنویسید، از signal.py شروع کنید.

چرا نخ B به initComplete علامت می دهد؟

۳.۳ قرار ملاقات

معمّاً: 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		نخ B	
1	statement a1	1	statement b1
2	statement a2	2	statement b2

می خواهیم مطمئن شویم که a1 پیش از b2 رخ می دهد و نیز b1 قبل از a2 اتفاق می افتد. هنگام نوشتن راه حل خود، نام و مقدار اولیه سمافورها را حتماً مشخص نمایید (اشاره کوچکی وجود دارد). راه حل شما نباید قید و بندهای زیادی داشته باشد. مثلاً، ترتیب a1 و b1 برای ما اهمیتی ندارد. در راه حل شما، باید امکان هر ترتیبی وجود داشته باشد.

نام این مسئله همگام سازی، قرار ملاقات است. ایده 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 اشاره ای در خصوص قرار ملاقات

اگر خوش شانس باشید می‌توانید به یک راه حل برسید، ولی اگر هم نرسیدید، این اشاره برای شماست. 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 می‌دهد 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	نخ B
<pre> 1 statement a1 2 aArrived.signal() 3 bArrived.wait() 4 statement a2 </pre>	<pre> 1 statement b1 2 bArrived.signal() 3 aArrived.wait() 4 statement b2 </pre>

هنگام کار بر روی مساله قبلی، ممکن است کدی مانند زیر را امتحان کرده باشید:

نخ A	نخ B
<pre> 1 statement a1 2 bArrived.wait() 3 aArrived.signal() 4 statement a2 </pre>	<pre> 1 statement b1 2 bArrived.signal() 3 aArrived.wait() 4 statement b2 </pre>

اگرچه این راه حل کار می‌کند اما احتمالاً بهیچگی 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.

دوباره 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 باشد.

۳.۳.۳ بن بست ۱#

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

نخ A	نخ B
<pre> 1 statement a1 2 bArrived.wait() 3 aArrived.signal() 4 statement a2 </pre>	<pre> 1 statement b1 2 aArrived.wait() 3 bArrived.signal() 4 statement b2 </pre>

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

این وضعیت بن بست^۲ نامیده می شود و بوضوح یک راه حل ناموفق برای مساله همگام سازی است. در این حالت، خطا واضح است اما درک امکان رخداد بن بست اغلب همیشه به این روشنی نیست. مثال های بیشتری را بعدا خواهیم دید.

۴.۳ Mutex

دومین کاربرد رایج سمافورها اعمال انحصار متقابل است. پیش از این یکی از کاربردهای انحصار متقابل در کنترل دسترسی همروند به متغیرهای اشتراکی را دیده ایم. mutex تضمین می نماید که در هر زمان تنها یک نخ به متغیر اشتراکی دسترسی دارد.

mutex شبیه یک توکن^۳ است از نخ به نخ دیگر منتقل می شود و در هر زمان به یک نخ اجازه ادامه فعالیت می دهد. برای نمونه، در رمان سالار مگس ها^۴ یک گروه از بچه ها از یک صدف به عنوان mutex بهره می برند. 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 می تواند صحبت کند.^۵

به طور مشابه، به منظور دسترسی یک نخ به متغیر اشتراکی، باید که mutex را بگیرد و زمانی که کارش تمام شد آن را رها کند. در هر زمان تنها یک نخ می تواند mutex را در اختیار داشته باشد. معمما: سمافورهایی به مثال زیر بیفزایید تا انحصار متقابل متغیر اشتراکی count را اعمال نمایند.

نخ A	نخ B
$\text{count} = \text{count} + 1$	$\text{count} = \text{count} + 1$

^۲ deadlock

^۳ token

^۴ The Lord of the Flies

^۵ اگرچه این تعبیر در اینجا مفید است ولی می تواند گمراه کننده نیز باشد، همانطوری که در بخش ۶.۵ خواهید دید.

۱.۴.۳ راهنمای انحصار متقابل

سمافوری با نام mutex و مقدار اولیه 1 ایجاد نمایید. این مقدار 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 mutex را آزاد نماید.

۲.۴.۳ راه حل انحصار متقابل

راه حلی را در ادامه می بینیم:

نخ A	نخ B
<pre> 1 mutex.wait() 2 # critical section 3 count = count + 1 4 mutex.signal() </pre>	<pre> 1 mutex.wait() 2 # critical section 3 count = count + 1 4 mutex.signal() </pre>

از آنجایی که مقدار اولیه mutex \ است، اولین نخ که wait در کد خود می رسد می تواند بلافاصله ادامه یابد. البته عمل انتظار روی سمافور موجب کاهش مقدار آن می گردد لذا دومین نخ به wait می رسد باید منتظر پیام دهی نخ اول بماند.

عملیات بروزرسانی متغیر حاشیه دار شده است تا نشان دهد که درون یک mutex قرار دارد. در این مثال، هر دو نخ کد یکسانی را اجرا می نمایند. گاهی اوقات این نوع راه حل را ^۶مقارن می نامیم. اگر نخ ها کدهای مختلفی را اجرا می نمایند این راه حل نامقارن^۷ گفته می شود. راه حل های مقارن اغلب راحت تر تعمیم داده می شوند. در این حالت، راه حل mutex می تواند 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 می خواند هیچ دو نخ به صورت همزمان به متغیر count دسترسی نخواهند داشت. اغلب کدی که به حفاظت نیاز دارد ناحیه بحرانی^۸ نامیده می شود، زیرا که جلوگیری از دسترسی همزمان، اهمیتی حیاتی دارد.

در استعارات رایج علم کامپیوتر، گاهی اوقات به طرق دیگری درباره mutex ها صحبت می شود. در تعبیری که تاکنون استفاده نمودیم، mutex توکنی است که از یک نخ به نخ دیگر انتقال داده می شود. در تعبیر دیگر، از ناحیه بحرانی به عنوان اتافی یاد می شود و در هر زمان تنها یک نخ اجازه دارد که داخل آن باشد. در این تعبیر، mutex ها قفل^۹ نامیده می شوند و گفته می شود یک نخ پیش از ورود، mutex را قفل نموده و هنگام خروج آن را باز می نماید. گرچه گهگاهی، کاربران تعبیر را خلط نموده و صحبت از گرفتن^{۱۰} و رهانمودن^{۱۱} یک قفل می نمایند که این تعبیر، به آن اندازه با معنی نیست. هر دو تعبیر، بالقوه مفید و بالقوه گمراه کننده هستند. هنگام کار بر روی مساله بعدی، هر دو شیوه

^۶symmetric

^۷asymmetric

^۸critical section

^۹lock

^{۱۰}getting

^{۱۱}releasing

تفکر را امتحان نموده و ببینید کدام یک شماره به راه حل می‌رساند.

۵.۳ Multiplex

معما: راه حل 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^{۱۲} نامیده می‌شود. در دنیای واقعی، مساله مالتی‌پلکس در یک کلوپ شبانه شلوغ زمانی رخ می‌دهد که یک حداکثر، برای تعداد افرادی که مجاز به حضور در ساختمان در یک زمان هستند 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 فردی دیگری اجازه ورود می‌یابد. تضمین این محدودیت با سمافورها ممکن است مشکل به نظر برسد اما تقریباً بدیهی است.

¹²multiplex

۱.۵.۳ راه حل مالتی پلکس

به منظور اینکه اجرای چند نخ را در ناحیه بحرانی ممکن سازیم تنها مقدار اولیه سمافور را n —حداکثر تعداد نخ‌های مجاز—قرار می‌دهیم.

در هر زمان، مقدار سمافور نشانگر تعداد نخ‌هایی است که می‌توانند داخل شوند. اگر این مقدار صفر باشد آنگاه نخ بعدی تا زمانی که یک 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 راه حل متقارن است، به طور قراردادی تنها یک کپی از کد نمایش داده می‌شود اما شما باید چندین کپی از کد را که به صورت هم‌روند در چندین نخ اجرا می‌شود در نظر بگیرید.

راه حل مالتی پلکس

```
1 multiplex.wait()
2   critical section
3 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 راه حل 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 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 می‌گیرد؛ 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 واقع، گاهی اوقات باجه‌های بلیط فروشی سیستمی مشابه این را بکار می‌برند. به مشتری‌هایی که در صف هستند هستند توکن‌هایی داده می‌شود. هر توکن به دارنده آن اجازه خرید یک بلیط را می‌دهد.

۶.۳ حصار^{۱۳}

دوباره مساله قرار ملاقات از بخش ۳.۳ در نظر بگیرید. یک محدودیت راه حلی که ارائه دادیم این است که برای بیشتر از دو نخ کار نمی‌کند.

معماً: راه حل قرار ملاقات را تعمیم دهید. هر نخ باید کد زیر را اجرا نماید:

کد حصار

1 rendezvous
2 critical point

لازمه همگام‌سازی این است که هیچ نخ 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 برای n ذخیره شده است و تمامی نخ‌ها به آن دسترسی دارند.

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

¹³Barrier

۱.۶.۳ راهنمای حصار

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

راهنمای حصار

```
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 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 نا راه حل حصار

ابتدا راه حلی را ارائه می‌دهیم که به طور کامل درست نمی‌باشد چرا که بررسی چنین راه حل‌هایی برای فهم اینکه چه چیزی نادرست است مفید است.

نا راه حل حصار

```
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10
11 critical point
```

از آنجایی که count به وسیله یک mutex محافظت می‌شود، تعداد نخ‌هایی که رد می‌شود را می‌شمارد. $n - 1$ نخ اول زمانی که به حصار می‌رسد منتظر می‌مانند چرا در ابتدا قفل است. زمانی که n مین نخ می‌رسد حصار را باز می‌نماید. معماً: مشکل این راه حل چیست؟

۳.۶.۳ بن بست ۲#

مشکل راه حل قبلی، بن بست است به عنوان مثال، تصور کنید $n = 5$ و چهار نخ منتظر حصار هستند. مقدار سمافور، منفی 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 می‌یابد و سمافور به ۳- افزایش می‌یابد.

اما پس از آن دیگر هیچ نخ‌ی به سمافور پیام نداده و هیچ کدام از دیگر نخ‌ها نمی‌تواند از مانع عبور نماید. این دومین مثال از بن بست است.

معمّا: آیا این کد همیشه یک بن بست تولید می‌نماید؟ آیا می‌توانید یک مسیر اجرایی از طریق این کد بیایید که منجر به بن بست نشود؟
معمّا: این مشکل را حل کنید.

۴.۶.۳ راه حل حصار

بالاخره، کد راه حل صحیح مساله حصار در ادامه آمده است.

راه حل حصار

```

1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10 barrier.signal()
11
12 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: turnstile^{۱۴}، زیرا که در یک زمان به تنها یک نخ اجازه عبور می‌دهد و می‌تواند قفل شود تا تمام نخ‌ها را نگه دارد.

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

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

¹⁴turnstile

۵.۶.۳ بن بست #۳

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

راه حل بد حصار

```

1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5     if count == n: barrier.signal()
6
7     barrier.wait()
8     barrier.signal()
9 mutex.signal()
10
11 critical point

```

این ایده بدی است زیرا که می‌تواند سبب بن بست گردد.

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

در اینجا، بن بست کاملاً واضح است، اما یک منبع رایج بن بست‌ها را نشان می‌دهد: مسدود شدن روی یک سمافور در حالیکه یک mutex در اختیار دارد.

۷.۳ حصار با قابلیت استفاده مجدد

اغلب، مجموعه نخ‌های همکار، یک سری از گام‌های یک حلقه را اجرا نموده و پس از هر گام نزد 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.

معملاً: راه حل حصار را چنان بازنویسی کنید که پس از عبور تمام نخ‌ها، ترن استایل دوباره قفل شود.

۱.۷.۳ نا راه حل حصار با قابلیت استفاده مجدد #۱

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

نا راه حل حصار با قابلیت استفاده مجدد

```
1 rendezvous
2
3
4 mutex.wait()
5     count += 1
6 mutex.signal()
7
8 if count == n: turnstile.signal()
9
10 turnstile.wait()
11 turnstile.signal()
12
13 critical point
14
15 mutex.wait()
16     count -= 1
17 mutex.signal()
18
19 if count == 0: turnstile.wait()
```

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

معمّا: مشکل چیست؟

۲.۷.۳ مساله حصار با قابلیت استفاده مجدد #۱

یک مشکلی در خط ۸ کد قبل وجود دارد.

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

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

۳.۷.۳ نا راه حل حصار با قابلیت استفاده مجدد #۲

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

نا راه حل حصار با قابلیت استفاده مجدد

```
1 rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n: turnstile.signal()
6 mutex.signal()
7
8 turnstile.wait()
9 turnstile.signal()
10
11 critical point
12
13 mutex.wait()
14     count -= 1
15     if count == 0: turnstile.wait()
16 mutex.signal()
```

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

متأسفانه 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. معماً مشکل را بیابید و آن را رفع نمایید.

۴.۷.۳ راهنمای حصار با قابلیت استفاده مجدد

همانطوری که تا کنون نیز نوشته شده است، این کد به نخ‌ی که زودتر از ناحیه بحرانی خود خارج می‌شود اجازه عبور از mutex دوم را داده و سپس در حلقه کد خود از mutex 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 مساله 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 اولی قبل و دومی باز است. زمانی که تمامی نخ‌ها به ترن‌استایل اول برسند، آن دومی را قفل و اولی را باز می‌نماییم. آن هنگام که تمامی نخ‌ها به ترن‌استایل دوم برسند اولی را دوباره قفل کرده — که این کار از بازگشت نخ‌ها به ابتدای حلقه جلوگیری می‌نماید — و سپس دومی را باز می‌نماییم.

۵.۷.۳ راه حل حصار با قابلیت استفاده مجدد

راهنمای حصار با قابلیت استفاده مجدد

```

1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile2.wait()      # lock the second
7         turnstile.signal()     # unlock the first
8 mutex.signal()
9
10 turnstile.wait()              # first turnstile
11 turnstile.signal()
12
13 # critical point
14
15 mutex.wait()
16     count -= 1
17     if count == 0:
18         turnstile.wait()      # lock the first
19         turnstile2.signal()   # unlock the second
20 mutex.signal()
21
22 turnstile2.wait()             # second turnstile
23 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 تمام نخ‌های ناحیه بحرانی را اجرا نمایند. متأسفانه، این راه حل از نابدیهی‌ترین انواع کد همگام‌سازی است: اطمینان یافتن از صحت این راه حل سخت است. اغلب اینکه یک مسیر خاص از طریق برنامه سبب خطایی گردد، آنچنان آشکار نیست. بدتر اینکه، بررسی پیاده‌سازی یک راه حل آنچنان کمکی نمی‌کند. خطا ممکن است خیلی به ندرت اتفاق افتد، زیرا که مسیر بخصوصی که سبب خطا می‌گردد ممکن است نیازمند ترکیبی فوق‌العاده بی‌بدیل از شرایط باشد. تولید و اشکال زدایی چنین خطاهایی با روش‌های مرسوم تقریباً ناممکن است.

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

نوع اثبات مد نظر بسیار غیر رسمی است. می‌توانیم از مزیت ساختار کد و اصطلاحاتی که توسعه داده‌ایم برای اثبات و سپس نشان دادن یک تعدادی از ادعاهای سطح متوسط درباره برنامه استفاده کنیم.

برای نمونه:

۱. فقط n امین نخ می تواند ترن استایل ها را قفل یا باز نماید.
 ۲. قبل اینکه یک نخ بتواند اولین ترن استایل را باز نماید، باید دومی را بسته باشد و بالعکس؛ بنابراین برای یک نخ، عبور از دیگر نخ ها به اندازه بیش از یک ترن استایل ناممکن است.
- با یافتن انواع صحیح عبارات به منظور اعلام و اثبات، گاهی اوقات می توانید یک راه موجز برای متقاعد کردن خودتان (یا یک همکار شکاک) بیابید که کدتان ضد گلوگه است.

۶.۷.۳ ترن استایل از پیش باگذاری شده

یک چیز جالب در رابطه با ترن استایل این است که یک جزء همه منظوره است که می‌توانید آن را در راه حل‌های مختلفی بکار ببرید. اما یک اشکال آن این است که نخ‌ها را مجبور می‌نماید که به صورت ترتیبی عبور نمایند و این ممکن است سبب تعویض بسترهای^{۱۵} زیادی بیش از حد نیاز گردد.

در مساله حصار با قابلیت استفاده مجدد، می‌توانیم راه حل را ساده‌تر کنیم اگر نخ‌ی که ترن استایل را از حالت قفل خارج می‌نماید آن را با تعداد سیگنال کافی به منظور عبور نخ‌های لازم، از پیش بارگذاری نماید^{۱۶}.

نحو مورد استفاده در اینجا فرض می‌کند که signal می‌تواند پارامتری دریافت دارد که تعداد سیگنال‌ها را مشخص می‌نماید. این، یک ویژگی استاندارد نیست، اما 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 مشکلی نیست.

راه حل حصار با قابلیت استفاده مجدد

```

1 # rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n:
6         turnstile.signal(n)           # unlock the first
7     mutex.signal()
8
9     turnstile.wait()                   # first turnstile
10
11 # critical point
12
13 mutex.wait()
14     count -= 1
15     if count == 0:
16         turnstile2.signal(n)          # unlock the second
17     mutex.signal()
18
19     turnstile2.wait()                  # second turnstile

```

زمانیکه n امین نخ می‌رسد، ترن استایل اول را با یک سیگنال بازای هر نخ از پیش بارگذاری می‌نماید. زمانیکه n امین نخ از ترن استایل عبور می‌نماید آخرین توکن را در اختیار گرفته و ترن استایل را دوباره قفل می‌نماید.

¹⁵ context switch

¹⁶ با تشکر از Matt Tesch بابت این راه حل!

اتفاق مشابهی در ترن استایل دوم رخ می دهد، آن زمان که آخرین نخ از mutex عبور نموده آن را باز می نماید.

۷.۷.۳ اشیاء حصار

به طور معمول، یک حصار را درون یک شیء محصور^{۱۷} می‌نمایند. در اینجا برای تعریف کلاس، نحو پایتون را بکار می‌بریم.

کلاس Barrier

```

1 class Barrier:
2     def __init__(self, n):
3         self.n = n
4         self.count = 0
5         self.mutex = Semaphore(1)
6         self.turnstile = Semaphore(0)
7         self.turnstile2 = Semaphore(0)
8
9     def phase1(self):
10        self.mutex.wait()
11        self.count += 1
12        if self.count == self.n:
13            self.turnstile.signal(self.n)
14        self.mutex.signal()
15        self.turnstile.wait()
16
17    def phase2(self):
18        self.mutex.wait()
19        self.count -= 1
20        if self.count == 0:
21            self.turnstile2.signal(self.n)
22        self.mutex.signal()
23        self.turnstile2.wait()
24
25    def wait(self):
26        self.phase1()
27        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 متد روی آن عمل می‌نماید. از آنجایی که هر شیء حصار، `mutex` و ترن‌استایل‌های خود را دارد، `self.mutex` به `mutex` شیء جاری اشاره می‌نماید. در اینجا مثالی مشاهده می‌نمایید که یک شیء Barrier را ایجاد نموده و روی آن منتظر می‌ماند:

¹⁷encapsulate

واسط Barrier

```
1 barrier = Barrier(n)           # initialize a new barrier
2 barrier.wait()                 # wait at a barrier
```

کدی که حصار را بکار می برد می تواند phase1 و phase2 را به صورت مجزا فراخواند اگر چیزی وجود داشته باشد که لازم باشد در آن بین اجرا شود.

۸.۳ صف

سمافورها می توانند به عنوان یک صف نیز استفاده شوند. در این حالت، مقدار اولیه 0 است و معمولاً کد به گونه ای نوشته شده است که امکان علامت دهی وجود ندارد مگر اینکه یک نخ، در حال انتظار وجود داشته باشد، لذا مقدار سمافور هیچگاه مثبت نیست.

به عنوان مثال، تصور کنید که نخ ها، نمایشگر رقصنده هایی باشند و دو نوع رقص —جلودار و دنباله رو— در دو صف، پیش از ورود به اتاق رقص منتظر هستند. هنگامی که یک جلودار می رسد، بررسی می کند که آیا هیچ دنباله روئی منتظر هست یا خیر. اگر چنین باشد، هر دو می توانند داخل شوند در غیر این صورت می بایست منتظر بماند.

به طور مشابه، زمانی که یک دنباله رو می رسد، او ابتدا چک می کند که آیا جلوداری وجود دارد یا خیر و منطبق با داخل شده یا صبر می نماید.

معما: کدی بنویسید برای جلودارها و دنباله روها که این شرایط در آن صدق کند.

۱.۸.۳ راهنمایی برای معما

در اینجا متغیرهایی آمده است که در راه حل خود بکار برده‌ام:

راهنمایی صف

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

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

۲.۸.۳ راه حل صف

کد مربوط به جلو دارها را در اینجا آمده است:

راه حل صف (جلودارها)

```
1 followerQueue.signal()
2 leaderQueue.wait()
3 dance()
```

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

راه حل صف (دنباله‌روها)

```
1 leaderQueue.signal()
2 followerQueue.wait()
3 dance()
```

این راه حل به همان اندازه که نشان می‌دهد ساده است، تنها یک قرار ملاقات است. هر جلودار دقیقاً به یک دنباله‌رو علامت می‌دهد و هر دنباله‌رو به یک جلودار علامت می‌دهد، لذا این ضمانت می‌کند که جلودارها و دنباله‌روها تنها به صورت جفت جفت اجازه اقدام داشته باشند. اما اینکه آیا آن‌ها واقعاً به صورت جفت جفت وارد عمل می‌شوند واضح نیست. برای هر تعداد از نخ‌ها امکان تجمیع قبل از اجرای dance وجود دارد و لذا برای هر تعداد از جلودارها dance پیش از دنباله‌روها ممکن است. 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 که شما را آورده است برقصید^{۱۸}.

معمداً: یک راه حل برای این مساله «صف انحصاری» ارائه کنید.

^{۱۸} متن آهنگ اجرا شده توسط Shania Twain

۳.۸.۳ راهنمای صف انحصاری

متغیرهایی را که در راه حل خود بکار برده‌ام در ادامه مشاهده می‌نمایید:

راهنمای صف

```
1 leaders = followers = 0
2 mutex = Semaphore(1)
3 leaderQueue = Semaphore(0)
4 followerQueue = Semaphore(0)
5 rendezvous = Semaphore(0)
```

leaders و followers شمارنده‌هایی هستند که تعداد رقاص‌های هر نوع که در حال انتظار هستند در خود نگه می‌دارند. موتکس دسترسی انحصاری به شمارنده‌ها را ضمانت می‌کند. leaderQueue و followerQueue صف‌هایی هستند که رقاص‌ها در آن منتظر می‌مانند. rendezvous برای بررسی اینکه هر دو نخ رقص را خود به پایان رسانده‌اند بکار می‌رود.

۴.۸.۳ راه حل صف انحصاری

قطعه کد مربوط به جلودارها:

راه حل صف (جلودارها)

```

1 mutex.wait()
2 if followers > 0:
3     followers--
4     followerQueue.signal()
5 else:
6     leaders++
7     mutex.signal()
8     leaderQueue.wait()
9
10 dance()
11 rendezvous.wait()
12 mutex.signal()

```

زمانی که یک جلودار می‌رسد، میوتکسی که `followers` و `leaders` را محافظت می‌کند می‌گیرد. 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 تضمین می‌کند که تنها یک نخ دنباله‌رو وجود دارد که `dance` را به طور همزمان 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)

```

1 mutex.wait()
2 if leaders > 0:
3     leaders--
4     leaderQueue.signal()
5 else:
6     followers++
7     mutex.signal()
8     followerQueue.wait()
9
10 dance()
11 rendezvous.signal()

```

زمانی که یک دنباله‌رو می‌رسد، چک می‌کند که آیا جلوداری در حالت انتظار هست. 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 راه حل من همیشه یک جلودار این کار را انجام می‌دهد.

زمانی که یک سمارفور به عنوان یک صف استفاده می‌شود،^{۱۹} 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 سمارفورهای صف به ندرت مثبت می‌باشد. اگرچه این امکان نیز وجود دارد. ببینید آیا می‌توانید بفهمید چگونه ممکن است.

^{۱۹} سمارفوری که به عنوان یک صف بکار می‌رود بسیار شبیه به یک متغیر شرطی است. تفاوت اصلی در این است که نخ‌ها باید میوتکس را به طور صریح پیش از انتظار رها نموده و پس از آن به صورت صریح آن را دوباره در اختیار گیرند (البته اگر آن را نیاز داشته باشد).

فصل ۴

مسائل همگام‌سازی کلاسیک

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

گرچه برای اکثر موارد، این مسائل در دنیای واقع 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 مساله کلاسیک، یک فرمول‌بندی کلاسیک ارائه می‌دهیم و همچنین شباهات آن را به مساله 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 می‌افزاید؛ مصرف‌کننده‌ها عناصر را حذف نموده و پردازش می‌کنند.

برنامه‌های رویداد-محور مثال‌های خوبی هستند. یک "رویداد"^۱، 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 عمل درحال انتظار تکمیل می‌شود.

¹event

هر زمان که یک رویداد رخ می‌دهد، یک نخ تولیدکننده یک شیء 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، مصرف‌کننده‌ها "مدیریت‌کننده رویداد"^۲ نامیده می‌شوند.

چندین محدودیت همگام‌سازی وجود دارد که برای درست کار کردن سیستم نیاز به تحمیل آن‌ها داریم.

- 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 تولیدکننده عنصر جدیدی را بیفزاید مسدود می‌گردد.

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

کد پایه تولیدکننده

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

همچنین فرض کنید مصرف‌کننده‌ها عملیات زیر را اجرا می‌کنند:

کد پایه 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.

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

²event handler

۱.۱.۴ راهنمای تولیدکننده-مصرف کننده

در اینجا متغیرهایی آمده است که ممکن است بخواهید استفاده کنید:

مقداردهی اولیه تولیدکننده-مصرف کننده

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

بدون تعجب، 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، نشان‌دهنده تعداد نخ‌های مصرف‌کننده در صف می‌باشد.

event یک متغیر محلی^۳ است، که در اینجا بدین معنی است که هر نخ نسخه خود را دارد. تاکنون فرض کرده‌ایم که تمام نخ‌ها به تمام متغیرها دسترسی دارند، اما 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 پشت‌زمان اجرای خودش را داشته باشد، آنگاه هر متغیر تخصیص داده شده در پشت، مخصوص به نخ^۴ است.
- اگر نخ‌ها به عنوان اشیا نشان داده شوند، می‌توانیم به هر شیء نخ یک خصیصه اضافه نماییم.
- اگر نخ‌ها IDهای یکتا داشته باشند، می‌توانیم ID را به عنوان اندیس یک آرایه یا جدول درهم‌سازی به کار ببریم، 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 اشتراکی اند، لذا پیش فرض متغیرها اشتراکی هستند مگر اینکه به صراحت به صورت local اعلان شوند.

^۳local variable

^۴thread-specific

۲.۱.۴ راه حل تولیدکننده-مصرفکننده

کد تولیدکننده از راه حل من در ادامه آمده است.

راه حل تولیدکننده

```

1 event = waitForEvent()
2 mutex.wait()
3     buffer.add(event)
4     items.signal()
5 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 `items` دادن سیگنال یک واحد آن را افزایش می‌دهد. کد مصرفکننده نیز مشابه است.

راه حل مصرفکننده

```

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

```

دوباره، عملیات بافر توسط یک میوتکس حفاظت می‌شود اما 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 راه حل درست است، این امکان وجود دارد که کارایی آن را قدری بهبود بخشید. تصور کنید زمانی که یک تولید 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 پرهزینه‌ای هستند: انجام غیرضروری آن‌ها می‌تواند به کارایی برنامه آسیب زند. لذا احتمالاً بازنویسی کد تولیدکننده به صورت زیر بهتر باشد.

راه حل بهبود یافته تولیدکننده

```

1 event = waitForEvent()
2 mutex.wait()
3     buffer.add(event)
4 mutex.signal()
5 items.signal()

```

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

یک چیز دیگری در رابطه با این راه حل 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. اما 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` را می‌تواند نادقیق باشد.

ممکن آن را بخواهیم بوسیله چک کردن بافر را درون میوتکس مدیریت کنیم:

راه حل معیوب مصرف‌کننده

```

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

این ایده بدی است.

معملاً: چرا؟

۳.۱.۴ بن بست #۴

اگر مصرفکننده کد زیر را اجرا نماید

راه حل معیوب مصرفکننده

```

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

می تواند سبب بروز بن بست گردد. تصور نمایید که بافر خالی است. یک مصرفکننده سر رسیده و میوتکس را گرفته و سپس روی `items` مسدود می گردد. زمانیکه تولیدکننده سر می رسد، روی `mutex` مسدود شده و سیستم وارد یک توقف تمام اعیار گردد. این مشکل یک خطای رایج در کد همگام سازی است: هر زمان که برای یک سمافور منتظر می مانید در حالیکه یک میوتکس را در دست دارید، خطر بن بست وجود دارد. زمانیکه یک راه حل را نسبت به مساله همگام سازی بررسی می نمایید، باید این نوع از بن بست را مد نظر داشته باشید.

۴.۱.۴ تولیدکننده-مصرفکننده با یک بافر متناهی

در این مثالی که بالا توضیح داده شد (نخ های کنترل کننده رویداد)، بافر اشتراکی معمولاً نامتناهی است (به طور دقیق تر، با منابع سیستم نظیر حافظه فیزیکی و فضای مبادله^۵ محدود شده است). گرچه در هسته سیستم عامل، محدودیت هایی روی فضای موجود نیز وجود دارد. بافرهایی نظیر درخواست دیسک یا بسته های شبکه معمولاً سائز ثابتی دارند. در این گونه موارد، یک محدودیت همگام سازی دیگر نیز داریم:

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

فرض کنید که اندازه بافر را می دانیم. آن `bufferSize` بنامید. از آنجاییکه سمافوری داریم که تعداد عناصر را نگاه می دارد، وسوسه می شویم کدی به صورت زیر بنویسیم.

راه حل معیوب بافر محدود

```

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

^۵swap space

اما نمی‌توانیم چنین کنیم. به یاد آورید مقدار جاری یک سемаفور را نمی‌توان بررسی نمود؛ wait و signal تنها عملیات ما هستند.

معملاً: کد تولیدکننده - مصرف‌کننده‌ای بنویسید که محدودیت بافر - متناهی را مدیریت کند.

۵.۱.۴ راهنمای بافر محدود تولیدکننده-مصرفکننده

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

مقداردهی اولیه 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 مصرفکننده بعدی سیگنال دهد مسدود گردد.

۶.۱.۴ راه حل بافر محدود تولیدکننده-مصرفکننده

و اما راه حل.

راه حل بافر محدود مصرفکننده

```

1 items.wait()
2 mutex.wait()
3     event = buffer.get()
4 mutex.signal()
5 spaces.signal()
6
7 event.process()

```

کد تولیدکننده تا حدودی متقارن است:

راه حل بافر محدود تولیدکننده

```

1 event = waitForEvent()
2
3 spaces.wait()
4 mutex.wait()
5     buffer.add(event)
6 mutex.signal()
7 items.signal()

```

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

۲.۴ مساله خوانندگان-نویسندگان

مساله کلاسیک بعدی، که مساله خواننده-نویسنده گفته می‌شود، مربوط می‌شود به هر راه حلی که یک ساختمان داده، پایگاه داده یا سیستم فایل خوانده شده و توسط نخ‌های همروند ویرایش می‌شود. زمانیکه ساختمان داده نوشته شده یا ویرایش می‌گردد ضروری است که دیگر نخ‌ها از خواندن منع گردند تا از دخالت در ویرایشی که در حال انجام است جلوگیری نموده و داده‌های نامعتبر یا ناپایدار خوانده نشود. همانند مساله تولیدکننده-مصرفکننده، راه حل مساله، 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 زمانی که نخ دیگری (اعم از خواننده یا نویسنده) در ناحیه بحرانی وجود دارد، نمی‌تواند وارد شود و زمانی که 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 نمی‌نماید، اما وجود یک دسته خاص در ناحیه بحرانی دسته‌های دیگر را مانع می‌گردد.

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

^۶categorical mutual exclusion

۱.۲.۴ راهنمای خوانندگان-نویسندگان

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

مقداردهی اولیه خوانندگان-نویسندگان

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

شمارنده readers تعداد خوانندگان در اتاق را نگاه می‌دارد. mutex شمارنده اشتراکی را حفاظت می‌نماید.

اگر هیچ نخی (اعم از خواننده یا نویسنده) در ناحیه بحرانی نباشد roomEmpty مقدار 1 دارد و الا 0. نام roomEmpty بر اساس قاعده‌ای که خود قرار گذاشتیم انتخاب گردیده تا اینکه 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 راه حل خوانندگان-نویسندگان

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

راه حل نویسندگان

```
1 roomEmpty.wait()
2     critical section for writers
3 roomEmpty.signal()
```

آیا نویسنده آن زمانیکه از اتاق خارج می‌شود، می‌تواند از خالی بودن آن مطمئن باشد؟ بله، زیرا که می‌داند هیچ نخه 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 ورود نویسندگان را ممنوع نماید. 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 آنجاییکه میوتکس را در اختیار گرفته، هر خواننده بعدی روی mutex تشکیل صف می‌دهند.

راه حل خوانندگان

```
1 mutex.wait()
2     readers += 1
3     if readers == 1:
4         roomEmpty.wait()      # first in locks
5     mutex.signal()
6
7 # critical section for readers
8
9 mutex.wait()
10    readers -= 1
11    if readers == 0:
12        roomEmpty.signal()    # last out unlocks
13    mutex.signal()
```

کد پس از ناحیه بحرانی یکسان است. آخرین خواننده‌ای که اتاق را ترک می‌نماید لامپ خاموش کرده —بدین معنی که به roomEmpty سیگنال می‌دهد— که ورود نویسنده منتظر را ممکن می‌سازد.

دوباره، برای اینکه نشان دهیم این کد درست است، مفید است تعدادی از ادعاهایی که درباره نحوه 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 روی roomEmpty در صف قرار گیرد اما چندین نویسنده ممکن است در صف قرار گیرند.

- زمانی که 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

```

1 class Lightswitch:
2     def __init__(self):
3         self.counter = 0
4         self.mutex = Semaphore(1)
5
6     def lock(self, semaphore):
7         self.mutex.wait()
8         self.counter += 1
9         if self.counter == 1:
10            semaphore.wait()
11        self.mutex.signal()
12
13    def unlock(self, semaphore):
14        self.mutex.wait()
15        self.counter -= 1
16        if self.counter == 0:
17            semaphore.signal()
18        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 فراخواننده روی semaphore مسدود گردیده و تمامی نخ‌های بعدی روی

^۷کلید برق

`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.

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 فراخوانند 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

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

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

راه حل خوانندگان – نویسندگان (خواننده)

```
1 readLightswitch.lock(roomEmpty)
2 # critical section
3 readLightswitch.unlock(roomEmpty)
```

کد نویسنده بدون تغییر باقی می‌ماند.

همچنین این امکان وجود دارد که بجای ارسال roomEmpty به عنوان یک پارامتر به lock و unlock، ارجاعی به roomEmpty را به عنوان یک خصیصه Lightswitch ذخیره نمود. این رهیافت جایگزین، کمتر مستعد خطا است، اما 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 راه حل 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، اما یک مشکل مرتبط وجود دارد که تقریباً 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، اما 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. اما همانطوری که بار سیستم افزایش یابد رفتار سیستم ممکن است به سرعت به زوال گراید (حداقل از دید نویسندگان).

معمّا: این راه حل را به گونه‌ای توسعه دهید تا زمانیکه یک نویسنده می‌رسد، خوانندگان موجود بتوانند خاتمه یابند ولی هیچ خواننده 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، اما تا زمانی که 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 زمانی که 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؛ زمانی که اولین خواننده وارد شد `roomEmpty` را قفل نموده و زمانی که 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 ناحیه 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 راه حل تضمین می‌نماید که حداقل یک نویسنده می‌تواند ادامه یابد، اما 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 می‌بیند 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

^۸First-In-First-Out

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.

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

معمّا: راه حلی برای مساله 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 راه حل آمده است.

مقداردهی اولیه خوانندگان-نویسندگان با اولویت نویسنده

```
1 readSwitch = Lightswitch()  
2 writeSwitch = Lightswitch()  
3 noReaders = Semaphore(1)  
4 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

```

1 noReaders.wait()
2   readSwitch.lock(noWriters)
3 noReaders.signal()
4
5   # critical section for readers
6
7 readSwitch.unlock(noWriters)

```

اگر یک خواننده درون ناحیه بحرانی باشد، noWriters را در اختیار می‌گیرد، اما 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

```

1 writeSwitch.lock(noReaders)
2   noWriters.wait()
3   # critical section for writers
4   noWriters.signal()
5 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 نویسنده 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، اما تا زمانیکه نویسندگان حضور دارند، noReaders را قفل نگاه می‌دارد. فقط زمانیکه آخرین نویسنده خارج می‌شود، خوانندگان می‌توانند وارد شوند.

البته، یک اشکال این راه حل 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 بازگشت 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**^۹ 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 حدی، زمان‌بند مسبب قحطی است. هر زمان که چندین نخ آماده 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 اینجا 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 نمی‌نمایند.

^۹bounded waiting

حتی با خصوصیت ۲، زمانیکه سمافورها را معرفی می‌کنیم، قحطی چهره نازیبايش را دومرتبه نشان می‌دهد. در تعریف یک سمافور، گفتیم زمانیکه یک نخ signal را اجرا می‌نماید، یکی از نخ‌های در حال انتظار را بیدار می‌نماید. اما 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 نخ‌هایی وجود دارد که روی یک سمافور در حال انتظار هستند در حالیکه یک نخ signal را اجرا می‌نماید، آنگاه یکی از نخ‌های در حال انتظار باید بیدار شوند.

این التزام ممکن است واضح به نظر آید، اما 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، اما 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 اما 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 راه حل برای این مساله 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، راه حل میوتکس ساده نسبت به قحطی آسیب‌پذیر هست. در واقع، حدس Dijkstra این بود که حل بدون قحطی مساله میوتکس تنها با کمک سمافورهای ضعیف ممکن نیست.

در ۱۹۷۹ J.M. Morris با حل مساله فوق و در نظر گرفتن اینکه تعداد نخ‌ها متناهی باشد، حدس Dijkstra را رد نمود [۶]. اگر شما به این مساله علاقه‌مند هستید، بخش بعدی راه حل او را نشان می‌دهد.

در غیر اینصورت، می‌توانید تصور کنید که سمافورها خصوصیت ۴ را دارند و به بخش ۴.۴ بروید.

معملاً: یک راه برای مساله انحصار متقابل با کمک سمافورهای ضعیف بنویسید. راه حل 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 کل نخ‌ها متناهی است.

¹⁰strong semaphore

¹¹weak semaphore

۱.۳.۴ راهنمای میوتکس بدون قحطی

راه حل Morris مشابه با حصار قابل استفاده مجدد در بخش ۷.۳ است. این راه حل از دو ترن استایل برای ایجاد دو اتاق انتظار قبل از ناحیه بحرانی استفاده می‌نماید. این مکانیزم در دو فاز عمل می‌نماید. در طول فاز اول، ترن استایل اول باز است و دومی بسته، لذا نخ‌ها در اتاق دوم مجتمع می‌گردند. در طول فاز دوم، ترن استایل اول قفل است و لذا هیچ نخ جدیدی نمی‌تواند وارد شود و ترن استایل دوم باز است، بنابراین نخ‌های موجود می‌توانند وارد ناحیه بحرانی شوند. گرچه ممکن است تعداد دلخواهی از نخ‌ها در اتاق انتظار باشند، اما ورود هر کدام از آنها به ناحیه بحرانی پیش از اینکه نخ‌های بعدی برسند تضمین شده است. در ادامه متغیرهایی که در راه حل 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 نگهداری می‌نماید. mutex از شمارنده‌ها محافظت می‌نماید. t1 و t2 ترن استایل هستند.

۲.۳.۴ راه حل میوتکس بدون قحطی

در ادامه راه حل 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 ترن استایل بگذرد. این ترن استایل ها کد را به سه اتاق تقسیم می نمایند. اتاق ۱ خطوط ۲-۸. اتاق ۲ خطوط ۶-۱۸. اتاق ۳ مابقی کد است. شمارنده های room1 و room2 تعداد نخ های هر اتاق را در خود نگه می دارند.

شمارنده room1 به طور معمول با کمک mutex محافظت می شود اما 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 می نماید.

برای فهم اینکه این راه حل چگونه عمل می نماید، با دنبال نمودن یک نخ در تمام مسیر شروع می کنیم. زمانیکه به خط ۸، mutex و t1 را گرفته است. زمانیکه room1 بررسی می نماید، که مقدار آن ۰ است، می تواند mutex را رها کرده و سپس ترن استایل 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 t_1 به صف شوند. با خروج از ناحیه بحرانی، مقدار room2 را صفر می‌بیند و t_1 را رها می‌نماید که ما را به نقطه شروع بر می‌گرداند.

البته، راه حل اگر بیش از یک نخ وجود داشته باشد جذاب‌تر است. در این حالت، زمانیکه نخ مقدم به خط λ می‌رسد، ممکن است دیگر نخ‌ها وارد اتاق انتظار شده و روی t_1 صف تشکیل داده باشند. از آنجایی که $room1 > 0$ نخ مقدم، t_2 را قفل شده رها نموده و در عوض به t_1 سیگنال می‌دهد تا اجازه ورود به اتاق ۲ را به دیگر نخ‌ها بدهد. از آنجایی که t_2 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 λ می‌رسد، که در این حالت t_2 را باز نموده و به دیگر نخ‌ها اجازه می‌دهد که به اتاق ۳ وارد شوند. نخ‌ی که t_2 را باز می‌کند همچنان t_1 را نگاه می‌دارد، بنابراین اگر هر کدام از نخ‌های مقدم دوباره به ابتدای کد باز گردد، در خط λ مسدود می‌گردد.

از آنجایی که هر نخ خروجی از اتاق ۳ به t_2 سیگنال می‌دهد، به دیگر نخ‌ها اجازه می‌دهد اتاق ۲ را ترک نمایند. زمانیکه آخرین نخ اتاق ۲ را ترک می‌نماید، t_2 را قفل شده رها می‌کند و t_1 را باز می‌نماید که این ما را به نقطه شروع باز می‌گرداند.

برای اینکه ببینیم این راه حل چگونه از قحطی جلوگیری می‌نماید، بهتر است که عملکردش را در دو فاز بررسی نماییم. در فاز 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 t_2 ، وجود یک جریان ادامه‌دار از نخ‌های به اتاق ۱ است. از آنجایی که تعداد نخ‌ها متناهی است این جریان نهایتاً پایان می‌پذیرد و در آن نقطه t_1 قفل می‌ماند و t_2 باز می‌شود.

در فاز دوم، نخ‌ها به اتاق ۳ سرازیر می‌شوند. 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 t_2 قفل شده و t_1 باز می‌شود.

در پایان فاز اول، می‌دانیم که هیچ نخ‌ی روی t_1 منتظر نیست زیرا که $room1 = 0$. و در پایان فاز دوم، می‌دانیم که هیچ نخ‌ی روی t_2 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. اما 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

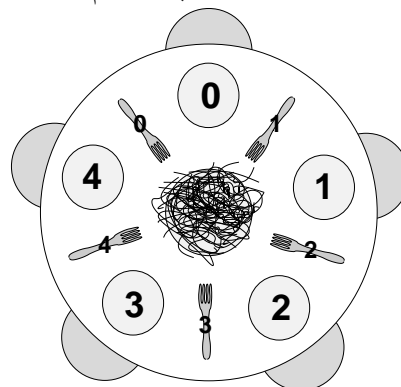
مساله غذا خوردن فیلسوف‌ها در سال ۱۹۶۵ توسط دایکسترا مطرح گردید؛ زمانیکه دایناسورها بر زمین حکمرانی می‌کردند [۳]. این مساله در انواع گوناگونی مطرح شده لکن ویژگی‌های استاندارد آن یک میز با پنج بشقاب، پنج چنگال (چوب) و یک کاسه بزرگ از اسپاگتی است. پنج فیلسوف که نشانگر نخ‌های در حال تعامل هستند، کنار میز آمده و حلقه زیر را اجرا می‌نمایند:

حلقه پایه فیلسوف

```
1 while True:
2     think()
3     get_forks()
4     eat()
5     put_forks()
```

چنگال‌ها، منابعی را نشان می‌دهند که نخ‌ها باید برای پیشرفت به صورت انحصاری آن‌ها را در اختیار داشته باشند. آن چیزی که مساله 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 `put_forks` و `get_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 راه حل 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
```

ایراتور % زمانیکه مقدار به ۵ برسد آن به ۰ بر می‌گردد؛ $0 = 5 \% 1 + 4$. 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:

نا راه حل 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 راه حل اولین شرط را برآورده می‌نماید، اما می‌توانیم مطمئن باشیم که دو شرط بعدی برآورده نمی‌شود، زیرا که اگر چنین بود، اصلاً مساله جذابی نبوده و شما می‌توانستید به مطالعه فصل ۵ بپردازید.

معمّاً: 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!

معملاً: راه حلی برای این مساله 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، بن بست غیر ممکن می‌گردد. ابتدا، خودتان را متقاعد نمایید که این ادعا درست است، سپس کدی بنویسید که تعداد فیلسوف‌ها را سر میز محدود نماید.

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 نامی پلکسی با نام `footman` که مقدار اولیه ۴ دارد کنترل کنیم. راه حل مشابه زیر است:

راه حل غذا خوردن فیلسوف‌ها ۱#

```

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

```

علاوه بر اجتناب از بن‌بست، این راه حل 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 نخ می‌خورید 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، زمانی که یک فیلسوف می‌تواند سر میز بگذرد محدود است. همچنین این نکته دلالت بر این دارد که زمان انتظار برای ورود به اتاق تا زمانی که `footman` خصوصیت ۴ را دارد، محدود است (بخش ۳.۴ را ببینید).

این راه حل 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 نا راه حل اولیه، فیلسوف‌ها “راست‌دست”^{۱۲} هستند؛ به این معنی که ابتدا چنگال سمت راست را بر می‌دارند. اما اگر فیلسوف “چپ‌دست”^{۱۳} باشد چه اتفاقی می‌افتد؟

معملاً ثابت کنید که اگر حداقل یکی از فیلسوف‌ها چپ‌دست و حداقل یکی راست‌دست باشد، آنگاه

¹²

¹³

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.

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 غذا خوردن فیلسوف‌ها #۲

در راه حل متقارن مساله غذا خوردن فیلسوف‌ها، 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 z 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 راه حل قبل به کار بردیم ثابت می‌کند که قطعی نیز غیر ممکن است.

۵.۴.۴ راه حل تنبام

در راه حل قبل هیچ چیز نادرستی وجود ندارد اما تنها برای تکمیل بحث، بگذارید نگاهی به برخی راه حل‌های جایگزین بیندازیم. یکی از شناخته شده ترین این راه حل‌ها، همانی است که در کتاب معروف سیستم عامل تنبام آمده است [۱۱]. 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 راه حل تنبام

```
1 state = ['thinking'] * 5
2 sem = [Semaphore(0) for i in range(5)]
3 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 اما کد:

راه حل تنبام

```
1 def get_fork(i):
2     mutex.wait()
3     state[i] = 'hungry'
4     test(i)
5     mutex.signal()
6     sem[i].wait()
7
8 def put_fork(i):
9     mutex.wait()
10    state[i] = 'thinking'
11    test(right(i))
12    test(left(i))
13    mutex.signal()
14
15 def test(i):
16     if state[i] == 'hungry' and
17        state[left(i)] != 'eating' and
18        state[right(i)] != 'eating':
19         state[i] = 'eating'
20         sem[i].signal()
```

تابع test بررسی می‌کند که آیا فیلسوف i 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 موفقیت‌آمیز 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. اما دومرتبه، قطعی ممکن ولی در اینجا خیلی مستلزم دقت و مهارت است. معماً: یا خودتان را متقاعد نمایید راه حل تنبام 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 قطعی تنبام

متأسفانه، این راه حل 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 [۴].

تصور کنید که می‌خواهیم فیلسوف ° 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 ° دچار قحطی می‌شود.

لذا راه حل تنبام تمامی ملزومات را بار آورده نمی‌نماید.

۵.۴ مساله سیگاری‌ها

مساله سیگاری‌ها در ابتدا توسط Suhas Patil مطرح شد [۸]، و ادعا نمود که این مساله به سمافورها قابل حل نیست. این ادعا با تعدادی شرط همراه است، اما در هر حالت مساله جذاب و چالش برانگیز است. چهار نخ در مساله 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 مکرراً 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 سیستم عامل است که منابع را تخصیص می‌دهد، و سیگاری‌ها نشانگر برنامه‌هایی هستند که به منابع نیاز دارند. مساله 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 مساله وجود دارد که اغلب در کتاب‌ها مشاهده می‌شود:

نسخه غیرممکن: نسخه Patil محدودیت‌هایی روی راه حل تحمیل می‌نماید. اول اینکه، شما اجازه تغییر کد عامل را ندارید. اگر عامل نشانگر یک سیستم عامل 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، مساله قابل حل نیست، اما همانطور که Patil اشاره نموده است محدودیت دوم کاملاً تصنعی است [۷]. 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 مساله آمده است که عامل باید بر مبنای مواد موجود به آن فرد سیگاری که می‌تواند ادامه دهد سیگنال ارسال نماید. این نسخه از مساله، جذابیتی ندارد زیرا که تمام فرض اولیه، مواد افزودنی و سیگارها را غیرضروری می‌نماید. همچنین در عمل، احتمالاً اینکه

عامل، اطلاعاتی از سایر نخ‌ها و آنچه آن‌ها نیاز دارند داشته باشد ایده خوبی نباشد. در نهایت، این نسخه از مساله نیز بسیار ساده است.

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

سمافورهای عامل

```
1 agentSem = Semaphore(1)
2 tobacco = Semaphore(0)
3 paper = Semaphore(0)
4 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

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

کد عامل B

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

کد عامل C

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

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

سیگاری با کبریت

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

سیگاری با تنباکو

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

سیگاری با کاغذ

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

مشکل این راه حل کجاست؟

۱.۵.۴ بن بست #۶

مشکل راه حل قبل امکان وقوع بن بست 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. اما 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 بن بست!

۲.۵.۴ راهنمایی مساله سیگاری‌ها

راه حل Parnas، “سه نخ کمکی فروشنده 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 مساله سیگاری‌ها

```
1 isTobacco = isPaper = isMatch = False
2 tobaccoSem = Semaphore(0)
3 paperSem = Semaphore(0)
4 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.

¹⁴pusher

۳.۵.۴ راه حل مساله سیگاری

کد یکی از فروشندگان در ادامه آمده است:

فروشنده A

```

1 tobacco.wait()
2 mutex.wait()
3     if isPaper:
4         isPaper = False
5         matchSem.signal()
6     elif isMatch:
7         isMatch = False
8         paperSem.signal()
9     else:
10        isTobacco = True
11 mutex.signal()

```

این فروشنده هر زمان که تنباکو موجود باشد فعال می‌شود. اگر مقدار `isPaper` برابر `true` باشد، می‌داند که فروشنده 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 نسخه 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

```

1 tobaccoSem.wait()
2 makeCigarette()
3 agentSem.signal()
4 smoke()

```

Parnas راه حل 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، راه حل او از شرط (یکی از محدودیت‌های تصنعی) اجتناب می‌نماید. کد حاصل کمی خلاصه‌تر است، اما عملکردش آنقدر واضح نیست.

۴.۵.۴ تعمیم مساله 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 مساله سیگاری‌ها دشوارتر خواهد می‌گردد. در این حالت، باید چند نمونه از یک ماده روی میز موجود باشد.

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

۵.۵.۴ راهنمای تعمیم مساله 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 مساله سیگاری‌ها

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


۶.۵.۴ راه حل تعمیم‌یافته مساله سیگاری‌ها

کد تغییر یافته فروشنده A در ادامه آمده است:

A فروشنده

```

1 tobacco.wait()
2 mutex.wait()
3     if numPaper:
4         numPaper -= 1
5         matchSem.signal()
6     elif numMatch:
7         numMatch -= 1
8         paperSem.signal()
9     else:
10        numTobacco += 1
11 mutex.signal()

```

یک راه تصویرسازی این مساله 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.

فصل ۵

مسائل همگام‌سازی کمتر- کلاسیک

۱.۵ مساله غذاخوردن وحشی‌ها

این مساله از برنامه‌نویسی همروند Andrews اقتباس شده است [۱].

قبیله‌ای از وحشی‌ها، از یک دیگ بزرگ که می‌تواند M پرس از مبلّغ 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 مساله تولیدکننده-مصرف‌کننده در اینجا نیز وسوسه می‌شویم که برای نگهداری تعداد پرس‌ها از سمافور استفاده نماییم. اما 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 ببریم. 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 پر بودن دیگ است.

۲.۱.۵ راه حل غذاخوردن وحشی‌ها

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

راه حل غذاخوردن وحشی‌ها (آشپز)

```
1 while True:
2     emptyPot.wait()
3     putServingsInPot(M)
4     fullPot.signal()
```

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

راه حل غذاخوردن وحشی‌ها (وحشی)

```
1 while True:
2     mutex.wait()
3     if servings == 0:
4         emptyPot.signal()
5         fullPot.wait()
6         servings = M
7         servings -= 1
8         getServingsFromPot()
9     mutex.signal()
10
11 eat()
```

اینکه وحشی بجای آشپز دستور $M = \text{servings}$ 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 آسانی دارد. اما 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 راه حل فرض نموده است است دیگر نخ-ایمن^۱ است و یا تضمین می‌نماید که `putServingsInPot` و `getServingsFromPot` به صورت انحصاری انجام شوند.

^۱ در برنامه‌نویسی چند نخ، کدی را نخ-ایمن گوئیم که تضمین نماید ساختمان داده‌های اشتراکی بدون دخالت‌های ناخواسته

۲.۵ مساله آرایشگاه

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

```
1 n = 4
2 customers = 0
3 mutex = Semaphore(1)
4 customer = Semaphore(0)
5 barber = Semaphore(0)
6 customerDone = Semaphore(0)
7 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 می‌شود. 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 راه حل یک جدول امتیاز و دو قرار ملاقات را ترکیب می‌نماید. کد مشتری‌ها در ادامه آمده است.

راه حل آرایشگاه (مشتری)

```

1 mutex.wait()
2     if customers == n:
3         mutex.signal()
4         balk()
5         customers += 1
6 mutex.signal()
7
8 customer.signal()
9 barber.wait()
10
11 # getHairCut()
12
13 customerDone.signal()
14 barberDone.wait()
15
16 mutex.wait()
17     customers -= 1
18 mutex.signal()

```

اگر n مشتری در آرایشگاه وجود داشته باشد، هر مشتری که می‌رسد بلافاصله balk را فرا می‌خواند و آلا هر مشتری به customer سیگنال داده و روی barber منتظر می‌ماند. کد آرایشگر در ادامه آمده است.

راه حل آرایشگاه (آرایشگر)

```

1 customer.wait()
2 barber.signal()
3
4 # cutHair()
5
6 customerDone.wait()
7 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 راه حل 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. زمانی که 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 راه حل خود استفاده می کنم.

راهنمایی آرایشگاه FIFO

```
1 n = 4
2 customers = 0
3 mutex = Semaphore(1)
4 customer = Semaphore(0)
5 customerDone = Semaphore(0)
6 barberDone = Semaphore(0)
7 queue = []
```

زمانی که هر یک از نخ ها از ترن استایل عبور می کند، یک نخ ساخته و آن را در صف قرار می دهد. به جای انتظار روی barber، هر نخ روی سمافور خودش منتظر می ماند. وقتی که آرایشگر بیدار می شود، یک نخ را از صف خارج کرده و به آن سیگنال می دهد.

۲.۳.۵ راه حل آرایشگاه FIFO

در ادامه کد تغییر یافته مشتریان آمده است:

راه حل آرایشگاه FIFO (مشتری)

```

1 self.sem = Semaphore(0)
2 mutex.wait()
3     if customers == n:
4         mutex.signal()
5         balk()
6         customers += 1
7         queue.append(self.sem)
8 mutex.signal()
9
10 customer.signal()
11 self.sem.wait()
12
13 # getHairCut()
14
15 customerDone.signal()
16 barberDone.wait()
17
18 mutex.wait()
19     customers -= 1
20 mutex.signal()

```

و کد آرایشگر به این صورت است:

راه حل آرایشگاه FIFO (آرایشگر)

```

1 customer.wait()
2 mutex.wait()
3     sem = queue.pop(0)
4 mutex.signal()
5
6 sem.signal()
7
8 # cutHair()
9
10 customerDone.wait()
11 barberDone.signal()

```

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

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

۴.۵ مسئله آرایشگاه هیلزر

ویلیام استالینگز [۱۰] یک نسخه پیچیده‌تر از مسئله آرایشگاه را ارائه می‌دهد، که آن را مدیون رالف هیلزر در دانشگاه 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، اما 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 راه حل بکار رفته در ادامه آمده است:

راهنمایی آرایشگاه هیلزر

```

1 n = 20
2 customers = 0
3 mutex = Semaphore(1)
4 sofa = Semaphore(4)
5 customer1 = Semaphore(0)
6 customer2 = Semaphore(0)
7 barber = Semaphore(0)
8 payment = Semaphore(0)
9 receipt = Semaphore(0)
10 queue1 = []
11 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.

`queue2` از `mutex2` 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 راه حل به طور قابل توجهی از آنچه انتظار داشتم پیچیده تر است. شاید در ذهن هیلزر راه حل ساده تری وجود داشته است لکن این بهترین چیزی است که می توانستم ارائه دهم.

راه حل آرایشگاه هیلزر (مشتری)

```

1 self.sem1 = Semaphore(0)
2 self.sem2 = Semaphore(0)
3
4 mutex.wait()
5     if customers == n:
6         mutex.signal()
7         balk()
8         customers += 1
9         queue1.append(self.sem1)
10    mutex.signal()
11
12 # enterShop()
13 customer1.signal()
14 self.sem1.wait()
15
16 sofa.wait()
17     # sitOnSofa()
18     self.sem1.signal()
19     mutex.wait()
20         queue2.append(self.sem2)
21     mutex.signal()
22     customer2.signal()
23     self.sem2.wait()
24 sofa.signal()
25
26 # sitInBarberChair()
27
28 # pay()
29 payment.signal()
30 receipt.wait()
31
32 mutex.wait()
33     customers -= 1
34    mutex.signal()

```

اولین پاراگراف مشابه راه حل قبلی است. زمانیکه یک مشتری می رسد، شمارنده را بررسی نموده آنگاه یا از ورود امتناع ورزیده و یا خودش را به صف می افزاید. سپس به آرایشگر سیگنال می دهد. زمانیکه مشتری از صف خارج می شود، وارد مالتی پلکس می گردد، روی مبل می نشیند و خود را به صف

دوم افزایش.

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

راه حل آرایشگاه هیلزر (آرایشگر)

```

1 customer1.wait()
2 mutex.wait()
3     sem = queue1.pop(0)
4     sem.signal()
5     sem.wait()
6 mutex.signal()
7 sem.signal()
8
9 customer2.wait()
10 mutex.wait()
11     sem = queue2.pop(0)
12 mutex.signal()
13 sem.signal()
14
15 barber.signal()
16 # cutHair()
17
18 payment.wait()
19 # acceptPayment()
20 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، اما 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 راه حل در `sync_code/barber3.py` آمده است (ر.ک. ۲.۳).

تنها راهی که برای حل مساله 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 راه حل در `sync_code/barber4.py` آمده است (ر.ک. ۲.۳).

۵.۵ مساله بابا نوئل

این مساله از کتاب سیستم‌های عامل ویلیام استالینگز گرفته شده است [۱۰]، اما او در این مساله نیز خود را مدیون John Trono از کالج Michael در ورمنت 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 مشکلمان 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 فراخواند.
- پس از اینکه سومین پری می‌رسد، بابا نوئل باید `helpElves` را فراخواند. به صورت همزمان، سه پری نیز باید `getHelp` فراخواند.
- هر سه پری پیش از اینکه پری دیگری وارد شود (شمارنده پری‌ها را افزایش دهد) باید `getHelp` را فراخواند.

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

۱.۵.۵ راهنمایی مساله بابا نوئل

راهنمایی مساله بابا نوئل

```
1 elves = 0
2 reindeer = 0
3 santaSem = Semaphore(0)
4 reindeerSem = Semaphore(0)
5 elfTex = Semaphore(1)
6 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 مساله بابا نوئل

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

راه حل مساله بابا نوئل (بابا نوئل)

```

1 santaSem.wait()
2 mutex.wait()
3     if reindeer >= 9:
4         prepareSleigh()
5         reindeerSem.signal(9)
6         reindeer -= 9
7     else if elves == 3:
8         helpElves()
9 mutex.signal()

```

زمانیکه بابا نوئل بیدار می‌شود، بررسی می‌نماید کدامیک از دو شرط برقرار 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 مساله بابا نوئل (گوزن‌ها)

```

1 mutex.wait()
2     reindeer += 1
3     if reindeer == 9:
4         santaSem.signal()
5 mutex.signal()
6
7 reindeerSem.wait()
8 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 مساله بابا نوئل (پری‌ها)

```
1 elfTex.wait()
2 mutex.wait()
3     elves += 1
4     if elves == 3:
5         santaSem.signal()
6     else
7         elfTex.signal()
8 mutex.signal()
9
10 getHelp()
11
12 mutex.wait()
13     elves -= 1
14     if elves == 0:
15         elfTex.signal()
16 mutex.signal()
```

اولین دو پری در همان زمانی که mutex را آزاد می‌نمایند elfTex را نیز آزاد می‌نمایند، اما آخرین پری elfTex را نگاه می‌دارد که مانع از ورود پری‌های دیگر می‌گردد تا زمانی که 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 مساله 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 [۱].

دو نوع نخ وجود دارد: اکسیژن و هیدروژن. به منظور ترکیب این نخ‌ها به مولکول‌های آب، باید حصار ی 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.

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

۱.۶.۵ راهنمایی H₂O

متغیرهایی که در راه حل بکار برده‌ام را در زیر مشاهده می‌نمایید:

راهنمایی ساخت آب

```
1 mutex = Semaphore(1)
2 oxygen = 0
3 hydrogen = 0
4 barrier = Barrier(3)
5 oxyQueue = Semaphore(0)
6 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 اکسیژن محلق شو» است و oxyQueue.signal() بمعنای «یک نخ اکسیژن از صف آزاد نما» می‌باشد.

۲.۶.۵ راه حل H₂O

در ابتدا hydroQueue و oxyQueue قفل هستند. زمانی که 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 hydroQueue.wait()
11 bond()
12
13
14 barrier.wait()

```

یک ویژگی غیرمعمول این راه حل 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. اما 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 اما 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 راه حل 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. اما 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 مساله عبور از رودخانه

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

یک جایی نزدیک ردmond^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.

^۲Anthony Joseph

^۳U.C. Berkeley

^۴Redmond

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.

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 مساله 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 فراخواند.

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 راه حل 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 هموعان خودش باشد و یا اینکه یک جفت ترکیبی از جفت‌های ممکن را تکمیل نماید.

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

راه حل عبور از رودخانه

```

1 mutex.wait()
2     hackers += 1
3     if hackers == 4:
4         hackerQueue.signal(4)
5         hackers = 0
6         isCaptain = True
7     elif hackers == 2 and serfs >= 2:
8         hackerQueue.signal(2)
9         serfQueue.signal(2)
10        serfs -= 2
11        hackers = 0
12        isCaptain = True
13    else:
14        mutex.signal()          # captain keeps the mutex
15
16 hackerQueue.wait()
17
18 board()
19 barrier.wait()
20
21 if isCaptain:
22     rowBoat()
23     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.

problem coaster roller The 1.5

it attributes he but, [1] *Programming Concurrent* Andrews's from is problem This
thesis. Master's Herman's S. J. to

pas- The thread. car a and threads passenger n are there Suppose
pas- C hold can which car. the in rides take to wait repeatedly sengers
it when only tracks the around go can car The $C < n$ where sengers.
full. is

details: additional some are Here

- .unboard and board invoke should Passengers
- .unload and run, load invoke should car The
- load invoked has car the until board cannot Passengers
- boarded. have passengers C until depart cannot car The
- .unload invoked has car the until unboard cannot Passengers

constraints. these enforces that car and passengers the for code Write Puzzle:

hint Coaster Roller ٨.٥

Roller Coaster hint

```

1 mutex = Semaphore(1)
2 mutex2 = Semaphore(1)
3 boarders = 0
4 unboarders = 0
5 boardQueue = Semaphore(0)
6 unboardQueue = Semaphore(0)
7 allAboard = Semaphore(0)
8 allAshore = Semaphore(0)

```

have that passengers of number the counts which ,passengers protects mutex

.boardCar invoked

un- before unboardQueue and boarding before boardQueue on wait Passengers

full. is car the that indicates allAboard boarding.

solution Coaster Roller ٧.٨.٥

thread: car the for code my is Here

Roller Coaster solution (car)

```

1 load()
2 boardQueue.signal(C)
3 allAboard.wait()
4
5 run()
6
7 unload()
8 unboardQueue.signal(C)
9 allAshore.wait()

```

signal to one last the for waits then passengers. C signals it arrives. car the When
for waits then disembark. to passengers C allows it departs. it After .allAboard
.allAshore

Roller Coaster solution (passenger)

```

1 boardQueue.wait()
2 board()
3
4 mutex.wait()
5     boarders += 1
6     if boarders == C:
7         allAboard.signal()
8         boarders = 0
9 mutex.signal()
10
11 unboardQueue.wait()
12 unboard()
13
14 mutex2.wait()
15     unboarders += 1
16     if unboarders == C:
17         allAshore.signal()
18         unboarders = 0
19 mutex2.signal()

```

stop to car the for wait and naturally. boarding. before car the for wait Passengers
passenger the resets and car the signals board to passenger last The leaving. before
counter.

problem Coaster Roller Multi-car 3.8.5

In car, one than more is there where case the to generalize not does solution This
constraints: additional some satisfy to have we that, do to order

time, a at boarding be can car one Only •

concurrently, track the on be can cars Multiple •

they order same the in unload to have they other, each pass can't cars Since •
boarded.

threads the of any before disembark must carload one from threads the All •
carloads, subsequent from

You constraints, additional the handle to solution previous the modify Puzzle:

that i named variable local a has car each that and cars, m are there that assume can
. $m - \backslash$ and \circ between identifier an contains

hint Coaster Roller Multi-car 9.8.5

loading the represents One order. in cars the keep to semaphores of lists two used I
 for semaphore one contains list Each area. unloading the represents one and area
 enforces that so unlocked. is list each in semaphore one only time. any At car. each
 • Car for semaphores the only Initially. unload. and load can threads order the
 own its on waits it unloading) (or loading the enters car each As unlocked. are
 line. in car next the signals it leaves it as semaphore:

Multi-car Roller Coaster hint

```

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

```

sequence the in car next the of identifier the computes next function The
 :(\circ to $m - 1$ from around (wrapping

Implementation of next

```

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

```


solution Coaster Roller Multi-car 8.8.5

cars: the for code modified the is Here

Multi-car Roller Coaster solution (car)

```
1 loadingArea[i].wait()
2 load()
3 boardQueue.signal(C)
4 allAboard.wait()
5 loadingArea[next(i)].signal()
6
7 run()
8
9 unloadingArea[i].wait()
10 unload()
11 unboardQueue.signal(C)
12 allAshore.wait()
13 unloadingArea[next(i)].signal()
```

unchanged. is passengers the for code The

فصل ٦

problems Not-so-classical

١.٦ The search-insert-delete problem

This is one of Andrews's *Programming Concurrent* [١].

Three kinds of threads share access to a singly-linked list: deleters, inserters, and searchers. Deleters remove items from the list; inserters add new items to the list; searchers examine the list. Each thread must execute its operations mutually exclusively with respect to the list. However, deleters and inserters may execute in parallel, as may searchers. Finally, a searcher may execute in parallel with a deleter or an inserter, but a deleter and an inserter may not execute in parallel. At any time, the list must be in a state such that deleters and inserters can process the list without causing a deletion or insertion to fail.

Puzzle: Write code for searchers, deleters, and inserters that enforces this kind of mutual exclusion, three-way

hint Search-Insert-Delete ١.١.٩

Search-Insert-Delete hint

```

1 insertMutex = Semaphore(1)
2 noSearcher = Semaphore(1)
3 noInserter = Semaphore(1)
4 searchSwitch = Lightswitch()
5 insertSwitch = Lightswitch()

```

time. a at section critical its in is inserter one only that ensures insertMutex
no and searchers no are there that (surprise) indicate noInserter and noSearcher
enter. to these of both hold to needs deleter a sections: critical their in inserters
exclude to inserters and searchers by used are insertSwitch and searchSwitch
deleters.

solution Search-Insert-Delete 2.1.6

solution: my is Here

Search-Insert-Delete solution (searcher)

```
1 searchSwitch.wait(noSearcher)
2 # critical section
3 searchSwitch.signal(noSearcher)
```

searcher first The deleter. a is about worry to needs searcher a thing only The
it. releases out one last the 'noSearcher takes in

Search-Insert-Delete solution (inserter)

```
1 insertSwitch.wait(noInserter)
2 insertMutex.wait()
3 # critical section
4 insertMutex.signal()
5 insertSwitch.signal(noInserter)
```

it. releases out one last the and noInserter takes inserter first the Similarly.
their in be can they semaphores. different for compete inserters and searchers Since
in is inserter one only that ensures insertMutex But concurrently. section critical
time. a at room the

Search-Insert-Delete solution (deleter)

```
1 noSearcher.wait()
2 noInserter.wait()
3 # critical section
4 noInserter.signal()
5 noSearcher.signal()
```

guaranteed is it noInserter and noSearcher both holds deleter the Since
one than more holding thread a see we time any course. Of access. exclusive
you scenarios. few a out trying By deadlocks. for check to need we semaphore.
free. deadlock is solution this that yourself convince to able be should
prone is one this problems. exclusion categorical many like hand. other the On
sometimes can we problem. Readers-Writers the in saw we As starvation. to
to according threads of category one to priority giving by problem this mitigate

solu- efficient an write to difficult is it general in But criteria. application-specific starvation. avoids that concurrency) of degree maximum the allows that (one tion

problem bathroom unisex The ۲.۶

at physics teaching position her left mine of friend a when \ problem this wrote I
 Xerox. at job a took and College Colby
 the and monolith. concrete a of basement the in cubicle a in working was She
 that Uberboss the to proposed She up. floors two was bathroom women's nearest
 on like of sort bathroom. unisex a to floor her on bathroom men's the convert they
 McBeal. Ally
 constraints synchronization following the that provided agreed. Uberboss The
 maintained: be can

time. same the at bathroom the in women and men be cannot There •
 company squandering employees three than more be never should There •
 bathroom. the in time

worry don't though. now. For deadlock. avoid should solution the course Of
 the all with equipped is bathroom the that assume may You starvation. about
 need. you semaphores

[۱] *Programming Concurrent* Andrews's in appears problem identical nearly a that learned I Later

hint bathroom Unisex 1.2.6

solution: my in used I variables the are Here

Unisex bathroom hint

```

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

```

otherwise. ◦ and empty is room the if \ is empty
 en- male first the When room. the from women bar to men allows maleSwitch
 un- it exits. male last the When women: barring ,empty locks lightswitch the ters.
 .femaleSwitch using likewise do Women enter. to women allowing ,empty locks
 than more no are there that ensure femaleMultiplex and maleMultiplex
 time. a at system the in women three and men three

solution bathroom Unisex ۲.۲.۶

code: female the is Here

Unisex bathroom solution (female)

```
1 femaleSwitch.lock(empty)
2   femaleMultiplex.wait()
3     # bathroom code here
4   femaleMultiplex.signal()
5 female Switch.unlock(empty)
```

similar. is code male The

solution? this with problems any there Are

problem bathroom unisex No-starve ۳.۲.۶

of line long A starvation. allows it that is solution previous the with problem The
 versa. vice and waiting. man a is there while enter and arrive can women
 problem. the fix Puzzle:

solution bathroom unisex No-starve ۴.۲.۶

stop to thread of kind one allow to turnstile a use can we before, seen have we As
code: male the at look we'll time This thread. of kind other the of flow the

No-starve unisex bathroom solution (male)

```

1 turnstile.wait()
2   maleSwitch.lock(empty)
3 turnstile.signal()
4
5   maleMultiplex.wait()
6   # bathroom code here
7   maleMultiplex.signal()
8
9 maleSwitch.unlock (empty)
```

turnstile the through pass will arrivals new room. the in men are there as long As
block will male the arrives, male a when room the in women are there If enter. and
entering from female) and (male arrivals later all bar will which turnstile. the inside
enters, turnstile the in male the point that At leave. occupants current the until
enter. to males additional allowing possibly
female arriving an room the in men are there if so similar. is code female The

men. additional barring turnstile. the in stuck get will
often will there then busy, is system the If efficient. be not may solution This
empty time Each turnstile. the on queued female. and male threads. several be
new the If enter. will another and turnstile the leave will thread one signaled. is
threads. additional barring block. promptly will it gender. opposite the is thread
the and time. a at bathroom the in threads ۲-۱ only be usually will there Thus.
concurrency. available the of advantage full take not will system

problem crossing Baboon ۳.۶

Im- and Design Systems: Operating Tanenbaum's from adapted is problem This
Park, National Kruger in somewhere canyon deep a is There .[۱۱] plementation
canyon the cross can Baboons canyon. the spans that rope single a and Africa, South

opposite in going baboons two if but rope, the on hand-over-hand swinging by
Further- deaths, their to drop and fight will they middle, the in meet directions
baboons more are there If baboons, Δ hold to enough strong only is rope the more,

break, will it time, same the at rope the on
to like would we semaphores, use to baboons the teach can we that Assuming
properties: following the with scheme synchronization a design

side other the to get to guaranteed is it cross, to begun has baboon a Once •
way, other the going baboon a into running without

rope, the on baboons Δ than more never are There •

ba- bar not should direction one in crossing baboons of stream continuing A •
starvation). (no indefinitely way other the going boons

clear, be should that reasons for problem this to solution a include not will I

Problem Hall Modus The 4.6

in living students Olin the of one Karst, Nathan by written was problem This
. $\Upsilon \circ \Delta$ of winter the during Υ Hall Modus

Modus of denizens the winter, this snowfall heavy particularly a After
shantytown cardboard their between path trench-like a created Hall
and to walk residents the of some day Every campus, of rest the and
indo- the ignore will we path, the via civilization and food class, from
the ignore also will We . Υ Tier to drive to daily chose who students lent
rea- unknown some For traveling, are pedestrians which in direction
necessary it find occasionally would Hall West in living students son,

Mods, the to venture to

students some that Mods, aka buildings, modular the for nicknames several of one is Hall Modus Υ
built, being was hall residence second the while in lived

to people two allow to enough wide not is path the Unfortunately,
 the on point some at meet persons Mods two If side-by-side, walk
 accommodate to drift high neck the into aside step gladly will one path,
 cross inhabitants ResHall two if occur will situation similar A other, the
 violent a however, meet, prude ResHall a and heathen Mods a If paths,
 of strength by solely determined victors the with ensue will skirmish
 the force will population larger the with faction the is, that numbers:
 wait, to other

with one), than ways more (in problem Crossing Baboon the to similar is This
 rule, majority by determined is section critical the of control that twist added the
 categor- the to solution starvation-free and efficient an be to potential the has This
 problem, exclusion ical
 section, critical the controls faction one while because avoided is Starvation
 majority, a achieve they until queue in accumulate faction other the of members
 critical the for wait they while entering from opponents new bar can they Then
 move to tend will it because efficient be to solution this expect I clear, to section
 section, critical the in concurrency maximum allowing batches, in through threads
 rule, majority with exclusion categorical implements that code write Puzzle:

hint problem Hall Modus ۱.۴.۶

solution. my in used I variables the are Here

Modus problem hint

```

1 heathens = 0
2 prudes = 0
3 status = 'neutral'
4 mutex = Semaphore(1)
5 heathenTurn = Semaphore(1)
6 prudeTurn = Semaphore(1)
7 heathenQueue = Semaphore(0)
8 prudeQueue = Semaphore(0)

```

field. the of status the records status and counters. are prudes and heathens
or heathens' to 'transition rule', 'prudes rule', 'heathens 'neutral', be can which
pat- scoreboard usual the in mutex by protected are three All prudes'. to 'transition
tern.

one bar can we that so field the to access control prudeTurn and heathenTurn

transition. a during other the or side

and in checking after wait threads where are prudeQueue and heathenQueue

field. the taking before

solution problem Hall Modus ۲.۴.۶

heathens: for code the is Here

Modus problem solution

```
1 heathenTurn.wait()
2 heathenTurn.signal()
3
4 mutex.wait()
5 heathens++
6
7 if status == 'neutral':
8     status = 'heathens rule'
9     mutex.signal()
10 elif status == 'prudes rule':
11     if heathens > prudes:
12         status = 'transition to heathens'
13         prudeTurn.wait()
14     mutex.signal()
15     heathenQueue.wait()
16 elif status == 'transition to heathens':
17     mutex.signal()
18     heathenQueue.wait()
19 else
20     mutex.signal()
21
22 # cross the field
23
24 mutex.wait()
25 heathens--
26
27 if heathens == 0:
28     if status == 'transition to prudes':
29         prudeTurn.signal()
30     if prudes:
31         prudeQueue.signal(prudes)
32         status = 'prudes rule'
33     else:
34         status = 'neutral'
35
36 if status == 'heathens rule':
37     if prudes > heathens:
38         status = 'transition to prudes'
39         heathenTurn.wait()
40
41 mutex.signal()
```

cases: following the consider to has he in. checks student each As

heathens. the for claim lays student the empty. is field the If •

balance. the tipped has arrival new the but charge. in currently heathens the If •

mode. transition to switches system the and turnstile prude the locks he

joins he balance. the tip doesn't arrival new the but charge. in prudes the If •

queue. the

the joins arrival new the control. heathen to transitioning is system the If •

queue.

system the or charge. in are heathens the either that conclude we Otherwise •

proceed. can thread this case. either In control. prude to transitioning is

cases. several consider to has she out. checks student each as Similarly.

following: the consider to has she out. check to heathen last the is she If •

locked. is turnstile prude the that means that transition. in is system the If —

it. open to has she so

the so status updates and them signals she waiting. prudes are there If —

'neutral'. is status new the not. If charge. in are prudes

possibility the check to has still she out. check to heathen last the not is she If •

heathen the closes she case. that In balance. the tip will departure her that

transition. the starts and turnstile

could threads of number any that is solution this of difficulty potential One

yet not but turnstile the passed have would they where ,٣ Line at interrupted be

power of balance the so counted. not are they in. check they Until in. checked

transi- a Also. turnstile. the passed have that threads of number the reflect not may

that At out. checked also have in checked have that threads the all when ends tion

turnstile. the passed have that types) both (of threads be may there point.

max- guarantee not does solution efficiency—this affect may behaviors These
“majority that accept you if correctness. affect don’t they concurrency—but inum
vote. to registered have that threads to applies only rule”

فصل ٧

classical remotely Not problems

problem bar sushi The ١.٧

Imagine .[٩] Reek Kenneth by proposed problem a by inspired was problem This
seat a take can you seat, empty an is there while arrive you If seats. Δ with bar sushi a
them of all that means that full, are seats Δ all when arrive you if But immediately.
before leave to party entire the for wait to have will you and together, dining are
down, sit you
enforces that bar sushi the leaving and entering customers for code write Puzzle:
requirements, these

hint bar Sushi 1.1.7

used: I variables the are Here

Sushi bar hint

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

and bar the at sitting threads of number the of track keep waiting and eating
has (or is bar the that indicates must_wait counters. both protects mutex waiting.
.block on block to have customers incoming som full, been)

non-solution bar Sushi ٢.١.٧

this of difficulties the of one illustrate to uses Reek solution incorrect an is Here
problem.

Sushi bar non-solution

```

1 mutex.wait()
2 if must_wait:
3     waiting += 1
4     mutex.signal()
5     block.wait()
6
7     mutex.wait()      # reacquire mutex
8     waiting -= 1
9
10 eating += 1
11 must_wait = (eating == 5)
12 mutex.signal()
13
14 # eat sushi
15
16 mutex.wait()
17 eating -= 1
18 if eating == 0:
19     n = min(5, waiting)
20     block.signal(n)
21     must_wait = False
22 mutex.signal()

```

solution? this with wrong what's Puzzle:

non-solution bar Sushi ३.१.१

up give to has he full. is bar the while arrives customer a If . १ Line at is problem The
customer last the When leave. can customers other that so waits he while mutex the
customers. waiting the of some least at up wakes which ,block signals she leaves.

.must_wait clears and

that and back, mutex the get to have they up, wake customers the when But
and arrive threads new If threads. new incoming with compete to have they means
is This threads. waiting the before seats the all take could they first, mutex the get
the in be to threads १ than more for possible is it injustice: of question a just not

constraints. synchronization the violates which concurrently, section critical
two next the in appear which problem. this to solutions two provides Reek

sections.

solutions! correct different two with up come can you if see Puzzle:

variables. additional any uses solution neither Hint:

۱# solution bar Sushi ۴.۱.۷

state the update to is mutex the reacquire to has customer waiting a reason only The
departing the make to is problem the solve to way one so .waiting and eating of
updating. the do mutex. the has already who customer.

Sushi bar solution #1

```

1 mutex.wait()
2 if must_wait:
3     waiting += 1
4     mutex.signal()
5     block.wait()
6 else:
7     eating += 1
8     must_wait = (eating == 5)
9     mutex.signal()
10
11 # eat sushi
12
13 mutex.wait()
14 eating -= 1
15 if eating == 0:
16     n = min(5, waiting)
17     waiting -= n
18     eating += n
19     must_wait = (eating == 5)
20     block.signal(n)
21 mutex.signal()

```

been already has eating mutex. the releases customer departing last the When
necessary. if block and state right the see customers arriving newly so updated.
doing is thread departing the because you.” for it do “I’ll pattern this calls Reek
threads. waiting the to belong to logically. seems. that work
the that confirm to difficult more little a it is that is approach this of drawback A
correctly. updated being is state

solution bar Sushi 1.7

can we that notion counterintuitive the on based is solution alternative Reek's acquire can thread one words. other In another! to thread one from mutex a transfer understand threads both as long As it. release can thread another then and lock a this. with wrong nothing is there transferred. been has lock the that

Sushi bar solution #2

```

1 mutex.wait()
2 if must_wait:
3     waiting += 1
4     mutex.signal()
5     block.wait()      # when we resume, we have the mutex
6     waiting -= 1
7
8 eating += 1
9 must_wait = (eating == 5)
10 if waiting and not must_wait:
11     block.signal()    # and pass the mutex
12 else:
13     mutex.signal()
14
15 # eat sushi
16
17 mutex.wait()
18 eating -= 1
19 if eating == 0: must_wait = False
20
21 if waiting and not must_wait:
22     block.signal()    # and pass the mutex
23 else:
24     mutex.signal()


```

entering an waiting. one no and bar the at customers 5 than fewer are there If sets customer fifth The mutex. the releases and eating increments just customer .must_wait

bar the at customer last the until block customers entering set. is must_wait If gives thread signaling the that understood is It .block signals and must_wait clears this that though. mind. in Keep it. receives thread waiting the and mutex the up comments. the in documented and programmer. the by understood invariant an is

right. it get to us to up is It semaphores. of semantics the by enforced not but there If mutex. the has it that understand we resumes. thread waiting the When wait- a to mutex the passes again. which. block signals it waiting. threads other are next the to mutex the passing thread each with continues. process This thread. ing last the case. either In threads. waiting more no or chairs more no are there until down. sit to goes and mutex the releases thread one from passed being is mutex the since baton.” the “Pass pattern this calls Reek solution this about thing nice One race. relay a in baton a like next the to thread A consistent. are waiting and eating to updates that confirm to easy is it that is correctly. used being is mutex the that confirm to harder is it that is drawback

problem care child The ٢.٧

Mid- and Systems Operating textbook his for problem this wrote Hailperin Max one always is there that require regulations state center. care child a At .[] dleware children. three every for present adult con- this enforces that threads adult and threads child for code Write Puzzle: section. critical a in straint

hint care Child 2.7

semaphore. one with problem this solve *almost* can you that suggests Hailperin

Child care hint

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

a allows token each where available. tokens of number the counts multiplex
 they as times: three multiplex signal they enter. adults As enter. to thread child
 solution. this with problem a is there But times. three wait they leave.
 problem? the is what Puzzle:

non-solution care Child 2.2.7

non-solution: Hailperin's in like looks code adult the what is Here

Child care non-solution (adult)

```

1 multiplex.signal(3)
2
3 # critical section
4
5 multiplex.wait()
6 multiplex.wait()
7 multiplex.wait()

```

and children three are there that Imagine deadlock. potential a is problem The
 adult either so, 3 is multiplex of value The center. care child the in adults two
 they time. same the at leave to start adults both if But leave. to able be should
 block. both and them. between tokens available the divide might
 change. minimal a with problem this solve Puzzle:

solution care Child ۳.۲.۷

problem: the solves mutex a Adding

Child care solution (adult)

```

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

```

available. tokens three are there If atomic. are operations wait three the Now
fewer are there If exit. and tokens three all get will mutex the gets that thread the
will threads subsequent and mutex the in block will thread first the available. tokens
mutex. the on queue

problem care child Extended ۴.۲.۷

child prevent can leave to waiting thread adult an that is solution this of feature One
entering. from threads
is multiplex the of value the so adults. two and children ۴ are there that Imagine
waiting block then and tokens two take will she leave. to tries adults the of one If .۴
to legal be would it though even wait will it arrives. thread child a If third. the for
fine. just be might that leave. to trying adult the of view of point the From enter.
not. it's center. care child the of utilization the maximize to trying are you if but
waiting. unnecessary avoids that problem this to solution a write Puzzle:

.۸.۳ Section in dancers the about think Hint:

hint care child Extended ۵.۲.۷

solution: my in used I variables the are Here

Extended child care hint

```

1 children = adults = waiting = leaving = 0
2 mutex = Semaphore(1)
3 childQueue = Semaphore(0)
4 adultQueue = Semaphore(0)

```

chil- of number the of track keep leaving and waiting .adults .children
 pro- are they leave: to waiting adults and enter: to waiting children adults. dren.
 .mutex by tected
 on wait Adults necessary. if enter: to childQueue on wait Children
 leave. to adultQueue

solution care child Extended ٩.٢.٧

mostly is it but solution. elegant Hailperin's than complicated more is solution This
 “I'll and queues. two scoreboard. a before: seen have we patterns of combination a
 you”. for it do
 code: child the is Here

Extended child care solution (child)

```

1 mutex.wait()
2     if children < 3 * adults:
3         children++
4         mutex.signal()
5     else:
6         waiting++
7         mutex.signal()
8         childQueue.wait()
9
10 # critical section
11
12 mutex.wait()
13     children--
14     if leaving and children <= 3 * (adults-1):
15         leaving--
16         adults--
17         adultQueue.signal()
18 mutex.signal()

```

(١) either and adults enough are there whether check they enter. children As
 they When block. and waiting increment (٢) or enter and children increment
 possible. if it signal and leave to waiting thread adult an for check they exit.

adults: for code the is Here

Extended child care solution (adult)

```

1 mutex.wait()
2     adults++
3     if waiting:
4         n = min(3, waiting)
5         childQueue.signal(n)
6         waiting -= n
7         children += n
8 mutex.signal()
9
10 # critical section
11
12 mutex.wait()
13     if children <= 3 * (adults-1):
14         adults--
15         mutex.signal()
16     else:
17         leaving++
18         mutex.signal()
19         adultQueue.wait()

```

they leave, they Before any. if children, waiting signal they enter, adults As
 exit, and adults decrement they so. If left, adults enough are there whether check
 to waiting is thread adult an While block, and leaving increment they Otherwise
 can children additional so section, critical the in adults the of one as counts it leave,
 enter.

problem party room The ۳.۷

con- a was there semester One College. Colby at was I while problem this wrote I
 Students of Dean the from someone that student a by allegation an over troversy
 public, was allegation the Although absence. his in room his searched had Office
 out found never we so case, the on comment to able wasn't Students of Dean the
 the was who mine. of friend a tease to problem this wrote I happened. really what
 Housing. Student of Dean
 of Dean the and students to apply constraints synchronization following The
 Students:

time. same the at room a in be can students of number Any .۱

the in students no are there if room a enter only can Students of Dean The .۲
 room the in students Δ^0 than more are there if or search) a conduct (to room
 party). the up break (to

enter, may students additional no room, the in is Students of Dean the While .۳
 leave. may students but

left. have students all until room the leave not may Students of Dean The .۴
 exclusion enforce to have not do you so Students. of Dean one only is There .۵
 deans. multiple among

Students of Dean the for and students for code synchronization write Puzzle:
 constraints. these of all enforces that

hint party Room ١.٣.٧

Room party hint

```

1 students = 0
2 dean = 'not here'
3 mutex = Semaphore(1)
4 turn = Semaphore(1)
5 clear = Semaphore(0)
6 lieIn = Semaphore(0)

```

of state the is dean and room, the in students of number the counts students
 students protects mutex room". the "in or "waiting" be also can which Dean, the
 scoreboard. a of example another yet is this so ,dean and
 the in is Dean the while entering from students keeps that turnstile a is turn
 room.

Dean the and student a between rendezvouses as used are lieIn and clear
 scandal!). of kind other whole a is (which

solution party Room ۲.۳.۷

one. this to got I before versions of lot a through worked I hard. is problem This occasionally but correct. mostly was edition first the in appeared that version The break nor search neither could he that find then and room the enter would Dean the silence. embarrassed in off skulk to have would he so party. the up was result the but humiliation. this spared that solution a wrote Tesch Matt cor- was it that ourselves convincing time hard a had we that enough complicated readable. more bit a is which one. this to me led solution that But rect.

Room party solution (dean)

```

1 mutex.wait()
2     if students > 0 and students < 50:
3         dean = 'waiting'
4         mutex.signal()
5         lieIn.wait()      # and get mutex from the student.
6
7         # students must be 0 or >= 50
8
9         if students >= 50:
10            dean = 'in the room'
11            breakup()
12            turn.wait()    # lock the turnstile
13            mutex.signal()
14            clear.wait()   # and get mutex from the student.
15            turn.signal()  # unlock the turnstile
16
17        else:              # students must be 0
18            search()
19
20 dean = 'not here'
21 mutex.signal()

```

room. the in students are there if cases: three are there arrives. Dean the When breaks Dean the more. or ۵۰ are there If wait. to has Dean the more. or ۵۰ not but Dean the students. no are there If leave. to students the for waits and party the up leaves. and searches so student. a with rendezvous a for wait to has Dean the cases. two first the In to has he up. wakes Dean the When deadlock. a avoid to mutex up give to has he

the to similar is This back. mutex the get to needs he so scoreboard. the modify
the “Pass the is chose I solution The problem. Bar Sushi the in saw we situation
pattern. baton”

Room party solution (student)

```

1 mutex.wait()
2     if dean == 'in the room':
3         mutex.signal()
4         turn.wait()
5         turn.signal()
6         mutex.wait()
7
8     students += 1
9
10    if students == 50 and dean == 'waiting':
11        lieIn.signal()           # and pass mutex to the dean
12    else:
13        mutex.signal()
14
15 party()
16
17 mutex.wait()
18     students -= 1
19
20    if students == 0 and dean == 'waiting':
21        lieIn.signal()           # and pass mutex to the dean
22    elif students == 0 and dean == 'in the room':
23        clear.signal()          # and pass mutex to the dean
24    else:
25        mutex.signal()

```

the If Dean, the signal to have might student a where cases three are There
 .lieIn signal to has out one last the or in student thΔ◦ the then waiting, is Dean
 student last the leave), to students the all for (waiting room the in is Dean the If
 the from passes mutex the that understood is it cases, three all In .clear signals out

Dean, the to student

of V Line at know we how is obvious be not may that solution this of part One
 realize to is key The .Δ◦ than less not or ◦ be must students that code Dean's the
 was conditional first the either point: this to get to ways two only are there that
 was Dean the or :Δ◦ than less not or ◦ either is students that means which false,
 is students that again, means, which signaled, student a when lieIn on waiting
 .Δ◦ than less not or ◦ either

problem Bus Senate The ٩.٧

Riders College. Wellesley at bus Senate the on based originally was problem This
riders waiting the all arrives, bus the When bus. a for wait and stop bus a to come
for wait to has boarding is bus the while arrives who anyone but ,boardBus invoke
people Δ than more are there if people: Δ is bus the of capacity The bus. next the
bus. next the for wait to have will some waiting.
the If .depart invoke can bus the boarded, have riders waiting the all When
immediately. depart should it riders, no are there when arrives bus
constraints. these of all enforces that code synchronization Write Puzzle:

hint problem Bus ۱.۴.۷

solution: my in used I variables the are Here

Bus problem hint

```

1 riders = 0
2 mutex = Semaphore(1)
3 multiplex = Semaphore(50)
4 bus = Semaphore(0)
5 allAboard = Semaphore(0)

```

waiting: are riders many how of track keeps which ,riders protects mutex

area. boarding the in riders ۵۰ than more no are there sure makes multiplex

waits bus The arrives. bus the when signaled gets which ,bus on wait Riders

board. to student last the by signaled gets which ,allAboard on

\# solution problem Bus ۲.۴.۷

pattern. baton” the “Pass the using are we Again. bus. the for code the is Here

Bus problem solution (bus)

```

1 mutex.wait()
2 if riders > 0:
3     bus.signal()          # and pass the mutex
4     allAboard.wait()      # and get the mutex back
5 mutex.signal()
6
7 depart()
```

entering from arrivals late prevents which ,mutex gets it arrives. bus the When
it Otherwise. immediately. departs it riders. no are there If area. boarding the
board. to riders the for waits and bus signals
riders: the for code the is Here

Bus problem solution (riders)

```

1 multiplex.wait()
2     mutex.wait()
3     riders += 1
4     mutex.signal()
5
6     bus.wait()              # and get the mutex
7 multiplex.signal()
8
9 boardBus()
10
11 riders -= 1
12 if riders == 0:
13     allAboard.signal()
14 else:
15     bus.signal()           # and pass the mutex
```

although area. waiting the in riders of number the controls multiplex The
.riders increments she until area waiting the enter doesn't rider a speaking. strictly
understood is it up. wakes rider a When arrives. bus the until bus on wait Riders
are there If .riders decrements rider each boarding. After mutex. the has she that
next the to mutex the pass and bus signals rider boarding the waiting. riders more
bus. the to back mutex the passes and allAboard signals rider last The rider.

departs. and mutex the releases bus the Finally.
you” for it do “I’ll the using problem this to solution a find you can Puzzle:
pattern?

۲# solution problem Bus ۳.۴.۷

the than variables fewer uses which solution. this with up came Hutchins Grant
vari- the are Here mutexes. any around passing involve doesn't and one. previous
ables:

Bus problem solution #2 (initialization)

```
1 waiting = 0
2 mutex = new Semaphore(1)
3 bus = new Semaphore(0)
4 boarded = new Semaphore(0)
```

by protected is which area. boarding the in riders of number the is waiting
has rider a that signals boarded arrived: has bus the when signals bus .mutex
boarded.

bus. the for code the is Here

Bus problem solution (bus)

```
1 mutex.wait()
2 n = min(waiting, 50)
3 for i in range(n):
4     bus.signal()
5     boarded.wait()
6
7 waiting = max(waiting-50, 0)
8 mutex.signal()
9
10 depart()
```

loop The process. boarding the throughout it holds and mutex the gets bus The
of number the controlling By board. to her for waits and turn in rider each signals
boarding. from riders Δ° than more prevents bus the signals.
example an is which .waiting updates bus the boarded. have riders the all When
pattern. you” for it do “I’ll the of
rendezvous. a and mutex a patterns: simple two uses riders the for code The

Bus problem solution (riders)

```
1 mutex.wait()
2     waiting += 1
3 mutex.signal()
```

```
4  
5 bus.wait()  
6 board()  
7 boarded.signal()
```

annoyed be might they boarding, is bus the while arrive riders if Challenge:
late allows that solution a find you Can one. next the for wait them make you if
constraints? other the violating without board to arrivals

problem Hall Faneuil The ۵.۷

who friend a by inspired was who Hutchins. Grant by written was problem This
 Boston. in Hall Faneuil at Citizenship of Oath her took
 Im- judge. one a and spectators. immigrants. threads: of kinds three are “There
 judge the point. some At down. sit then and in. check line. in wait must migrants
 the and enter. may one no building. the in is judge the When building. the enters
 in. check immigrants all Once leave. may Spectators leave. not may immigrants
 immigrants the confirmation. the After naturalization. the confirm can judge the
 after point some at leaves judge The Citizenship. U.S. of certificates their up pick
 their get immigrants After before. as enter now may Spectators confirmation. the
 leave.” may they certificates.

functions some threads the give let’s specific. more requirements these make To
 functions. those on constraints put and execute. to

- swear sitDown checkIn enter invoke must Immigrants
 .leave and getCertificate
- .leave and confirm enter invokes judge The
- .leave and spectate enter invoke Spectators
- may immigrants and enter may one no building. the in is judge the While
 .leave not
- enter invoked have who immigrants all until confirm not can judge The
 .checkIn invoked also have
- .confirm executed has judge the until getCertificate not can Immigrants

Hint Problem Hall Faneuil ۱.۵.۷

Faneuil Hall problem hint

```

1 noJudge = Semaphore(1)
2 entered = 0
3 checked = 0
4 mutex = Semaphore(1)
5 confirmed = Semaphore(0)

```

pro- also it spectators: and immigrants incoming for turnstile a as acts noJudge
checked room. the in immigrants of number the counts which entered tects
.mutex by protected is it in: checked have who immigrants of number the counts
.confirm executed has judge the that signals confirmed

solution problem Hall Faneuil ۲.۵.۷

immigrants: for code the is Here

Faneuil Hall problem solution (immigrant)

```

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()
12     # and pass the mutex
13 else:
14     mutex.signal()
15
16 sitDown()
17 confirmed.wait()
18
19 swear()
20 getCertificate()
21
22 noJudge.wait()
23 leave()
24 noJudge.signal()

```

the in is judge the while enter: they when turnstile a through pass Immigrants

locked. is turnstile the room.

.checked update and in check to mutex get to have immigrants entering. After

and allSignedIn signals in check to immigrant last the waiting. judge a is there If

judge. the to mutex the passes

judge: the for code the is Here

Faneuil Hall problem solution (judge)

```

1 noJudge.wait()
2 mutex.wait()
3
4 enter()
5 judge = 1

```

```

6
7 if entered > checked:
8     mutex.signal()
9     allSignedIn.wait()
# and get the mutex back.
10
11 confirm()
12 confirmed.signal(checkered)
13 entered = checked = 0
14
15 leave()
16 judge = 0
17
18 mutex.signal()
19 noJudge.signal()

```

and entering, from spectators and immigrants bar to noJudge holds judge The
 .checked and entered access can he so mutex
 also has entered has who everyone when instant an at arrives judge the If
 mu- the up give to has she Otherwise, immediately, proceed can she in, checked
 is it .allSignedIn signals and in checks immigrant last the When wait, and tex
 back, mutex the get will judge the that understood
 immigrant every for once confirmed signals judge the ,confirm invoking After
 you”), for it do “I’ll of example (an counters the resets then and in, checked has who
 .noJudge and mutex releases and leaves judge the Then
 and swear invoke immigrants ,confirmed signals judge the After
 to turnstile noJudge the for wait then and concurrently, getCertificate
 leaving, before open
 the is obey to have they constraint only the easy: is spectators for code The
 turnstile, noJudge

Faneuil Hall problem solution (spectator)

```

1 noJudge.wait()
2 enter()
3 noJudge.signal()
4
5 spectate()
6
7 leave()

```

get they after stuck. get to immigrants for possible is it solution this in Note:
 immigrants. of batch next the in swear to coming judge another by certificate. their
 in-ceremony. swearing another through wait to have might they happens. that If
 the after that constraint additional the handle to solution this modify Puzzle:
 judge the before leave must in sworn been have who immigrants all leaves. judge
 again. enter can

Hint Problem Hall Faneuil Extended ۳.۵.۷

variables: additional following the uses solution My

Faneuil Hall problem hint

```

1 exit = Semaphore(0)
2 allGone = Semaphore(0)

```

it solve can we rendezvous. additional an involves problem extended the Since

semaphores. two with

again. pattern baton” the “pass the use to useful it found I hint: other One

solution problem Hall Faneuil Extended ۶.۵.۷

.۶ Line at starts difference The before. as same the is solution this of half top The
leave. to judge the for here wait Immigrants

Faneuil Hall problem solution (immigrant)

```

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()
12 # and pass the mutex
13 else:
14     mutex.signal()
15
16 sitDown()
17 confirmed.wait()
18
19 swear()
20 getCertificate()
21
22 exit.wait() # and get the mutex
23 leave()
24 checked--
25 if checked == 0:
26     allGone.signal() # and pass the mutex
27 else:
28     exit.signal() # and pass the mutex

```

leave. to ready is judge the When .۶ Line at starts difference the judge. the For
possibly and immigrants. more allow would that because .noJudge release can't she
to immigrant one allows which .exit signals she Instead. enter. to judge. another
.mutex passes and leave.

ba- the passes then and checked decrements signal the gets that immigrant The
passes and allGone signals leave to immigrant last The immigrant. next the to ton
has it but necessary. strictly not is pass-back This judge. the to back mutex the

phase the end to noJudge and mutex both releases judge the that feature nice the
cleanly.

Faneuil Hall problem solution (judge)

```
1 noJudge.wait()
2 mutex.wait()
3
4 enter()
5 judge = 1
6
7 if entered > checked:
8     mutex.signal()
9     allSignedIn.wait()
10 # and get the mutex back.
11
12 confirm()
13 confirmed.signal(checked)
14 entered = 0
15
16 leave()
17 judge = 0
18
19 exit.signal() # and pass the mutex
20 allGone.wait() # and get it back
21 mutex.signal()
22 noJudge.signal()
```

unchanged. is problem extended the for code spectator The

problem Hall Dining ۶.۷

Olin at class Synchronization my during Pollack Jon by written was problem This
College.

dine invoking After .leave then and dine invoke hall dining the in Students
leave”. to “ready considered is student a leave invoking before and
main- to order in that: is students to applies that constraint synchronization The
is student A alone. table a at sit never may student a suave, social of illusion the tain
leave invokes dine invoked has who else everyone if alone sitting be to considered
.dine finished has she before
constraint. this enforces that code write Puzzle:

hint problem Hall Dining ۶.۷

Dining Hall problem hint

```

1 eating = 0
2 readyToLeave = 0
3 mutex = Semaphore(1)
4 okToLeave = Semaphore(0)

```

usual the is this so «mutex by protected counters are readyToLeave and eating

pattern. scoreboard

table. the at alone left be would student another but leave. to ready is student a If

signals. and situation the changes student another until okToLeave on waits she

solution problem Hall Dining ۶.۶.۷

where situation one only is there that realize will you constraints. the analyze you If
to wants who student one and eating student one is there if wait. to has student a
might student another situation: this of out get to ways two are there But leave.

finish. might student dining the or eat. to arrive
counters. the updates student waiting the signals who student the case. either In
example another is This back. mutex the get to have doesn't student waiting the so
pattern. you" for it do "I'll the the of

Dining Hall problem solution

```

1  getFood()
2
3  mutex.wait()
4  eating++
5  if eating == 2 and readyToLeave == 1:
6      okToLeave.signal()
7      readyToLeave--
8  mutex.signal()
9
10 dine()
11
12 mutex.wait()
13 eating--
14 readyToLeave++
15
16 if eating == 1 and readyToLeave == 1:
17     mutex.signal()
18     okToLeave.wait()
19 elif eating == 0 and readyToLeave == 2:
20     okToLeave.signal()
21     readyToLeave -= 2
22     mutex.signal()
23 else:
24     readyToLeave--
25     mutex.signal()
26
27 leave()

```

waiting one and eating student one sees she if in. checking is student is When
him. for readyToLeave decrements and hook the off waiter the lets she leave. to

cases: three checks student the dining. After
 up give to has student departing the eating, left student one only is there If •
 wait. and mutex the
 him signals she her. for waiting is someone that finds student departing the If •
 them. of both for counter the updates and
 leaves. and readyToLeave decrements just she Otherwise. •

problem Hall Dining Extended ٣.٩.٧

As step. another add we if challenging more little a gets problem Hall Dining The
 invoking After .leave then and dine .getFood invoke they lunch to come students
 Similarly, eat”. to “ready considered is student a ,dine invoking before and getFood
 leave”. to “ready considered is student a dine invoking after
 table a at sit never may student a applies: constraint synchronization same The
 either if alone sitting be to considered is student A alone.
 to ready one no and table the at else one no is there while dine invokes She •
 or eat.
 finished has she before leave invokes dine invoked has who else everyone •
 .dine

constraints. these enforces that code write Puzzle:

hint problem Hall Dining Extended ۶.۶.۷

solution: my in used I variables the are Here

Extended Dining Hall problem hint

```

1 readyToEat = 0
2 eating = 0
3 readyToLeave = 0
4 mutex = Semaphore(1)
5 okToSit = Semaphore(0)
6 okToLeave = Semaphore(0)

```

.mutex by protected all counters. are readyToLeave and eating ,readyToEat
 or okToSit on waits she proceed. cannot she where situation a in is student a If
 signals. and situation the changes student another until okToLeave
 whether of track keep help to hasMutex named variable per-thread a used also I
 mutex. the holds thread a not or

solution problem Hall Dining Extended ۵.۶.۷

where situation one only is there that realize we constraints. the analyze we if Again.
 else one no and eating one no is there if wait. to has eat to ready is who student a
 eat. to ready is who arrives else someone if is out way only the And eat. to ready

Extended Dining Hall problem solution

```

1  getFood()
2
3  mutex.wait()
4  readyToEat++
5  if eating == 0 and readyToEat == 1:
6      mutex.signal()
7      okToSit.wait()
8  elif eating == 0 and readyToEat == 2:
9      okToSit.signal()
10     readyToEat -= 2
11     eating += 2
12     mutex.signal()
13 else:
14     readyToEat--
15     eating++
16     if eating == 2 and readyToLeave == 1:
17         okToLeave.signal()
18         readyToLeave--
19         mutex.signal()
20
21 dine()
22
23 mutex.wait()
24 eating--
25 readyToLeave++
26 if eating == 1 and readyToLeave == 1:
27     mutex.signal()
28     okToLeave.wait()
29 elif eating == 0 and readyToLeave == 2:
30     okToLeave.signal()
31     readyToLeave -= 2
32     mutex.signal()
33 else:
34     readyToLeave--
35     mutex.signal()
36
37 leave()

```

waiting a that so pattern you” for it do “I’ll the used I solution, previous the in As
back. mutex the get to have doesn’t student
the that is one previous the and solution this between difference primary The
allows student second the and wait. to has table empty an at arrives who student first
waiting students for check to have don’t we case. either It proceed. to students both
table! empty an leave can one no since leave. to

Python in Synchronization

synchronization of details ugly the of some avoided have we pseudocode. using By
 Python: in code synchronization real at look we'll chapter this In world. real the in
 C. at look we'll chapter next the in
 complete environment. multithreading pleasant reasonably a provides Python
 in code cleanup some is there but foibles. few a has It objects. Semaphore with
 better. little a things makes that Appendix
 example: simple a is Here

Listing:

```

1 from threading_cleanup import *
2
3 class Shared:
4     def __init__(self):
5         self.counter = 0
6
7     def child_code(shared):
8         while True:
9             shared.counter += 1
10            print shared.counter
11            time.sleep(0.5)
12
13 shared = Shared()
14 children = [Thread(child_code, shared) for i in range(2)]
15 for child in children: child.join()
```

out line this leave will I : Appendix from code cleanup the runs line first The
 examples. other the of
 variables Global variables. shared contain will that type object an defines Shared
 Threads examples. these in any use won't we but threads. between shared also are
 the in local also are function a inside declared are they that sense the in local are that
 thread-specific. are they that sense
 new the prints .counter increments that loop infinite an is code child The
 seconds. Δ.° for sleeps then and value.
 children the for waits then children. two and shared creates thread parent The
 won't). they case. this in (which exit to

problem checker Mutex \.Λ

unsynchro- make children the that notice will synchronization of students Diligent
 might you program. this run you If safe! not is which .counter to updates nized
 synchronization about thing nasty The won't. probably you but errors. some see
 may testing extensive even that means which unpredictable. are they that is errors
 them. reveal not
 we case. this In search. the automate to necessary often is it errors. detect To
 .counter of values the of track keeping by errors detect can

Listing:

```

1 class Shared:
2     def __init__(self, end=10):
3         self.counter = 0
4         self.end = end
5         self.array = [0]* self.end
6
7     def child_code(shared):
8         while True:
9             if shared.counter >= shared.end: break
10            shared.array[shared.counter] += 1
11            shared.counter += 1
12
13 shared = Shared(10)
14 children = [Thread(child_code, shared) for i in range(2)]

```

```
15 | for child in children: child.join()
16 | print shared.array
```

incremented be should array the in entry each correctly. works everything If
the prints and join from returns parent the exit. children the When once. exactly
got I program. the ran I When .array of value

1] ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,1 ,[1

might we array, the of size the increase we If correct, disappointingly is which
result, the check to harder gets also it but errors, more expect

array: the in results the of histogram a making by checker the automate can We

Listing:

```

1 class Histogram(dict):
2     def __init__(self, seq=[]):
3         for item in seq:
4             self[item] = self.get(item, 0) + 1
5
6 print Histogram(shared.array)

```

get I program, the run I when Now

10} {1:

but far, so errors No expected, as times, \ appeared \ value the that means which
interesting: more get things bigger, end make we if

100} {1: ,100 = end

1000} {1: ,1000 = end

10000} {1: ,10000 = end

72439} 2: ,27561 {1: ,100000 = end

between switches context of lot a are there that enough big is end When Oops!
of *lot* a get we case, this In errors, synchronization get to start we children, the
threads where pattern recurring a into falls program the that suggests which errors.

section, critical the in interrupted consistently are

which errors, synchronization of dangers the of one demonstrates example This

a in time one occurs error an If random, not are they but rare, be may they that is

row, a in times million a happen won't it mean doesn't that million.

ac- exclusive enforce to program this to code synchronization add Puzzle:

from section this in code the download can You variables, shared the to cess

greenteapress.com/semaphores/counter.py

hint checker Mutex \.A.A

used: I Shared of version the is Here

Listing:

```
1 class Shared:
2     def __init__(self, end=10):
3         self.counter = 0
4         self.end = end
5         self.array = [0]* self.end
6         self.mutex = Semaphore(1)
```

no as come should which .mutex named Semaphore the is change only The
surprise.

solution checker Mutex ۲.۱.A

solution: my is Here

Listing:

```

1 def child_code(shared):
2     while True:
3         shared.mutex.wait()
4         if shared.counter < shared.end:
5             shared.array[shared.counter] += 1
6             shared.counter += 1
7             shared.mutex.signal()
8         else:
9             shared.mutex.signal()
10            break

```

book, this in problem synchronization difficult most the not is this Although
to easy is it particular. In right, details the get to tricky it found have might you
a cause would which loop, the of out breaking before mutex the signal to forget
deadlock.

result: following the got and ,1000000 = end with solution this ran I

1000000} {1:

start, good a to off is it but correct, is solution my mean doesn't that course. Of

problem machine coke The 2.8

removing and adding consumers and producers simulates program following The
machine: coke a from cokes

Listing:

```

1 import random
2
3 class Shared:
4     def __init__(self, start=5):
5         self.cokes = start
6
7     def consume(shared):
8         shared.cokes -= 1
9         print shared.cokes
10
11    def produce(shared):
12        shared.cokes += 1
13        print shared.cokes
14
15    def loop(shared, f, mu=1):
16        while True:
17            t = random.expovariate(1.0/mu)
18            time.sleep(t)
19            f(shared)
20
21    shared = Shared()
22    fs = [consume]*2 + [produce]*2
23    threads = [Thread(loop, shared, f) for f in fs]
24    for thread in threads: thread.join()

```

shared the So full. half initially is machine the that and cokes. \° is capacity The
.Δ is cokes variable
both They consumers. two and producers two threads. ¥ creates program The
These .consume invoke consumers and produce invoke producers but .loop run
no-no. a is which variable. shared a to access unsynchronized make functions
cho- duration a for sleep consumers and producers loop. the through time Each
producers two are there Since .mu mean with distribution exponential an from sen
average. on second. per machine the to added get cokes two consumers. two and
removed. get two and

vary can in run short the in but constant. is cokes of number the average on So
 value the see probably will you while. a for program the run you If widely. quite
 should these of neither course. Of .\° above climb or zero. below dip cokes of
 happen.

synchronization following the enforce to program this to code add Puzzle:

constraints:

- exclusive. mutually be should cokes to Access •
- added. is coke a until block should consumers zero. is cokes of number the If •
- removed. is coke a until block should producers ,\° is cokes of number the If •

greenteapress.com/semaphores/ from program the download can You

[coke.py](#)

hint machine Coke ۱.۲.۸

solution: my in used I variables shared the are Here

Listing:

```

1 class Shared:
2     def __init__(self, start=5, capacity=10):
3         self.cokes = Semaphore(start)
4         self.slots = Semaphore(capacity-start)
5         self.mutex = Semaphore(1)

```

tricky it makes which integer): simple a than (rather now Semaphore a is cokes
 Semaphore. a of value the access never should you course. Of value. its print to
 methods cheater the of any provide doesn't way do-gooder usual its in Python and
 implementations. some in see you
 stored is Semaphore a of value the that know to interesting it find might you But
 know. don't you case in Also. ._Semaphore__value named attribute private a in
 You attributes. private to access on restriction any enforce actually doesn't Python
 interested. be might you thought I but course. of it. access never should
 Ahem.

solution machine Coke ٢.٢.٨

with up coming trouble no had have should you book. this of rest the read you've If
this: as good as least at something

Listing:

```

1 def consume(shared):
2     shared.cokes.wait()
3     shared.mutex.wait()
4     print shared.cokes.value()
5     shared.mutex.signal()
6     shared.slots.signal()
7
8 def produce(shared):
9     shared.slots.wait()
10    shared.mutex.wait()
11    print shared.cokes._Semaphore__value
12    shared.mutex.signal()
13    shared.cokes.signal()

```

num- the that confirm to able be should you while. a for program this run you If
solution this So .\° than greater or negative never is machine the in cokes of ber
correct. be to seems
far. So

C in Synchronization

Ap- C. in program synchronized multithreaded. a write will we section this In more little a code C the make to use I code utility the of some provides ب pendix code. that on depend section this in examples The palatable.

exclusion Mutual ۱.۹

variables: shared contains that structure a defining by start We'll

Listing:

```

1 typedef struct {
2     int counter;
3     int end;
4     int *array;
5 } Shared;
6
7 Shared *make_shared (int end)
8 {
9     int i;
10    Shared *shared = check_malloc (sizeof (Shared));
11
12    shared->counter = 0;
13    shared->end = end;
14
15    shared->array = check_malloc (shared->end * sizeof(int));
16    for (i=0; i<shared->end; i++) {

```

```

17     shared->array[i] = 0;
18 }
19 return shared;
20 }

```

until threads concurrent by incremented be will that variable shared a is counter keeping by errors synchronization for check to array use will We .end reaches it increment. each after counter of value the of track

code Parent ۲.۱.۹

com- to them for wait and threads create to uses thread parent the code the is Here plete:

Listing:

```

1 int main ()
2 {
3     int i;
4     pthread_t child[NUM_CHILDREN];
5
6     Shared *shared = make_shared (100000);
7
8     for (i=0; i<NUM_CHILDREN; i++) {
9         child[i] = make_thread (entry, shared);
10    }
11
12    for (i=0; i<NUM_CHILDREN; i++) {
13        join_thread (child[i]);
14    }
15
16    check_array (shared);
17    return 0;
18 }

```

com- to them for waits loop second the threads: child the creates loop first The check to check_array invokes parent the finished, has child last the When plete.

Appendix in defined are join_thread and make_thread errors. for

code Child ۲.۱.۹

children: the of each by executed is that code the is Here

Listing:

```

1 void child_code (Shared *shared)
2 {
3     while (1) {
4         if (shared->counter >= shared->end) {
5             return;
6         }
7         shared->array[shared->counter]++;
8         shared->counter++;
9     }
10 }

```

into index an as counter use threads child the loop, the through time Each
counter increment they Then element. corresponding the increment and array
done. they're if see to check and

errors Synchronization ۳.۱.۹

incremented be should array the of element each correctly. works everything If
not are that elements of number the count just can we errors. for check to So once.
:\

Listing:

```

1 void check_array (Shared *shared)
2 {
3     int i, errors=0;
4
5     for (i=0; i<shared->end; i++) {
6         if (shared->array[i] != 1) errors++;
7     }
8     printf ("%d errors.\n", errors);
9 }

```

from code) cleanup the (including program this download can You

greenteapress.com/semaphores/counter.c

this: like output see should you program. the run and compile you If

0 counter at child Starting

10000

20000

30000

40000

50000

60000

70000

80000

90000

done. Child

100000 counter at child Starting

done. Child

Checking...

```
errors. 0
```

operating your of details on depends children the of interaction the course. Of
shown example the In computer. your on running programs other also and system
started, got thread other the before end to ° from way the all ran thread one here.

errors. synchronization no were there that surprising not is it so
children. the between switches context more are there bigger. gets end as But

and variables shared the to access exclusive enforce to semaphores use Puzzle:

errors. no are there that confirm to again program the run

hint exclusion Mutual ۴.۱.۹

solution: my in used I Shared of version the is Here

Listing:

```

1 typedef struct {
2     int counter;
3     int end;
4     int *array;
5     Semaphore *mutex;
6 } Shared;
7
8 Shared *make_shared (int end)
9 {
10     int i;
11     Shared *shared = check_malloc (sizeof (Shared));
12
13     shared->counter = 0;
14     shared->end = end;
15
16     shared->array = check_malloc (shared->end * sizeof(int));
17     for (i=0; i<shared->end; i++) {
18         shared->array[i] = 0;
19     }
20     shared->mutex = make_semaphore(1);
21     return shared;
22 }
```

the with mutex the initializes ۲۰ Line Semaphore: a as mutex declares ۵ Line

.\ value

solution exclusion Mutual ۵.۱.۹

code: child the of version synchronized the is Here

Listing:

```

1 void child_code (Shared *shared)
2 {
3     while (1) {
4         sem_wait(shared->mutex);
5         if (shared->counter >= shared->end) {
6             sem_signal(shared->mutex);
7             return;
8         }
9
10        shared->array[shared->counter]++;
11        shared->counter++;
12        sem_signal(shared->mutex);
13    }
14 }
```

to remember to is thing tricky only the here: surprising too nothing is There

statement. return the before mutex the release

greenteapress.com/semaphores/ from solution this download can You

[counter_mutex.c](#)

semaphores own your Make ٢.٩

Pthreads use that programs for tools synchronization used commonly most The these of explanation an For semaphores. not variables. condition and mutexes are

.[٧] *Threads POSIX with Programming* Butenhof's recommend I tools. write to them use then and variables. condition and mutexes about read Puzzle:

semaphores. of implementation an my is Here solutions. your in code utility following the use to want might You

mutexes: Pthreads for wrapper

Listing:

```

1 typedef pthread_mutex_t Mutex;
2
3 Mutex *make_mutex ()
4 {
5     Mutex *mutex = check_malloc (sizeof(Mutex));
6     int n = pthread_mutex_init (mutex, NULL);
7     if (n != 0) perror_exit ("make_lock failed");
8     return mutex;
9 }
10
11 void mutex_lock (Mutex *mutex)
12 {
13     int n = pthread_mutex_lock (mutex);
14     if (n != 0) perror_exit ("lock failed");
15 }
16
17 void mutex_unlock (Mutex *mutex)
18 {
19     int n = pthread_mutex_unlock (mutex);
20     if (n != 0) perror_exit ("unlock failed");
21 }
```

variables: condition Pthread for wrapper my And

Listing:

```
1 typedef pthread_cond_t Cond;
2
3 Cond *make_cond ()
4 {
5     Cond *cond = check_malloc (sizeof(Cond));
6     int n = pthread_cond_init (cond, NULL);
7     if (n != 0) perror_exit ("make_cond failed");
8     return cond;
9 }
10
11 void cond_wait (Cond *cond, Mutex *mutex)
12 {
13     int n = pthread_cond_wait (cond, mutex);
14     if (n != 0) perror_exit ("cond_wait failed");
15 }
16
17 void cond_signal (Cond *cond)
18 {
19     int n = pthread_cond_signal (cond);
20     if (n != 0) perror_exit ("cond_signal failed");
21 }
```


hint implementation Semaphore ۱.۲.۹

semaphores: my for used I definition structure the is Here

Listing:

```

1 typedef struct {
2     int value, wakeups;
3     Mutex *mutex;
4     Cond *cond;
5 } Semaphore;
```

pend- of number the counts wakeups semaphore. the of value the is value
yet not have but woken been have that threads of number the is, that signals: ing
semaphores our that sure make to is wakeups for reason The execution. resumed
.۳.۴ Section in described ,۳ Property have
condition the is cond ;wakeups and value to access exclusive provides mutex

semaphore. the on wait they if on wait threads variable

structure: this for code initialization the is Here

Listing:

```

1 Semaphore *make_semaphore (int value)
2 {
3     Semaphore *semaphore = check_malloc (sizeof(Semaphore));
4     semaphore->value = value;
5     semaphore->wakeups = 0;
6     semaphore->mutex = make_mutex ();
7     semaphore->cond = make_cond ();
8     return semaphore;
9 }
```


implementation Semaphore ۲.۲.۹

condition and mutexes Pthread's using semaphores of implementation my is Here
variables:

Listing:

```

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 }
```

the is tricky be might that thing only the straightforward: is this of Most
variable, condition a use to way unusual an is This .۷ Line at loop do...while
necessary. is it case this in but
loop? while a with loop do...while this replace we can't why Puzzle:

detail implementation Semaphore ٣.٢.٩

be would It .٣ Property have not would implementation this loop, while a With
 signal. own its catch and around run then and signal to thread a for possible
 one signals, thread a when that \guaranteed is it loop, do...while the With
 at mutex the gets thread another if even signal, the get will threads waiting the of
 resumes, threads waiting the of one before ٣ Line

<http://en.wikipedia.org/> (see wakeup spurious well-timed a that out turns It almost. Well,
 guarantee, this violate can ([wiki/Spurious_wakeup](http://en.wikipedia.org/wiki/Spurious_wakeup)

کتاب نامه

- [۱] *Practice and Principles Programming: Concurrent* Andrews. R. Gregory [۱]
Addison-Wesley. ۱۹۹۱.
- [۲] Addison-Wesley. *Threads POSIX with Programming* Butenhof. R. David [۲]
۱۹۹۷.
- [۳] *Pro- in Reprinted* ۱۹۶۵ processes. sequential Cooperating Dijkstra. Edsger [۳]
۱۹۶۸ York New Press. Academic ed. Genuys. F. *Languages gramming*
- [۴] *Bulletin SIGCSE ACM* revisited. philosophers Dining Gingras. R. Armando [۴]
۱۹۹۰ September ، ۲۸ ، ۲۴-۲۱: (۳) ۲۲
- [۵] *Controlled Supporting Middleware: and Systems Operating* Hailperin. Max [۵]
۲۰۰۶ Technology. Course Thompson *Interaction*
- [۶] prob- exclusion mutual the to solution starvation-free A Morris. M. Joseph [۶]
۱۹۷۹ February ، ۸۰-۷۶: ۸ ، *Letters Processing Information* lem.
- [۷] without problem smokers' cigarette the to solution a On Parnas. L. David [۷]
March ، ۱۸۳-۱۸۱: ۱۸ ، *ACM the of Communications* statements. conditional
۱۹۷۵.
- [۸] for primitives semaphore Dijkstra's of capabilities and Limitations Patil. Suhas [۸]
۱۹۷۱ MIT. report. Technical processes. among coordination
- [۹] ۲۰۰۴ ، *SIGCSE ACM* In semaphores. for patterns Design Reek. A. Kenneth [۹]

Pren- .*Principles Design and Internals Systems: Operating* Stallings. William [۱۰]
.۲۰۰۰ edition, fourth Hall. tice

second Hall, Prentice .*Systems Operating Modern* Tanenbaum. S. Andrew [۱۱]
.۲۰۰۱ edition.

پیوست آ

threads Python up Cleaning

pretty are threads Python environments. threading other of lot a to Compared
fix can you Fortunately. me. annoy that features of couple a are there but good.
code. clean-up little a with them

۱.۵ Semaphore methods

which ,release and acquire called are semaphores Python for methods the First.
years. of couple a for book this on working after but choice. reasonable perfectly a is
subclassing by way my it have can I Fortunately. .wait and signal to used am I
module: threading the in Semaphore of version the

Semaphore name change

```
1 import threading
2
3 class Semaphore(threading._Semaphore):
4     wait = threading._Semaphore.acquire
5     signal = threading._Semaphore.release
```

the using Semaphores manipulate and create can you defined. is class this Once
book. this in syntax

Semaphore example

```
1 mutex = Semaphore()
```

```

2 mutex.wait()
3 mutex.signal()

```

threads Creating २.१

for interface the is me annoys that module threading the of feature other The
two and arguments keyword requires way usual The threads. starting and creating
steps:

Thread example (standard way)

```

1 import threading
2
3 def function(x, y, z):
4     print x, y, z
5
6 thread = threading.Thread(target=function, args=[1, 2, 3])
7 thread.start()

```

you when But effect. immediate no has thread the creating example. this In
argu- given the with function target the executes thread new the .start invoke
starts. it before thread the with something do to need you if great is This ments.
are args and target arguments keyword the think I Also. do. never almost I but
awkward.

code. of lines four with problems these of both solve can we Fortunately.

Cleaned-up Thread class

```

1 class Thread(threading.Thread):
2     def __init__(self, t, *args):
3         threading.Thread.__init__(self, target=t, args=args)
4         self.start()

```

automatically: start they and interface. nicer a with threads create can we Now

Thread example (my way)

```

1 thread = Thread(function, 1, 2, 3)

```

with Threads multiple create to is which like. I idiom an to itself lends also This
comprehension: list a

Multiple thread example

```
1 threads = [Thread(function, i, i, i) for i in range(10)]
```

interrupts keyboard Handling 3.1

inter- be can't Thread.join that is class threading the with problem other One
into translates Python which .SIGINT signal the generates which Ctrl-C. by rupted
KeyboardInterrupt. a

program: following the write you if So.

Unstoppable program

```

1 import threading, time
2
3 class Thread(threading.Thread):
4     def __init__(self, t, *args):
5         threading.Thread.__init__(self, target=t, args=args)
6         self.start()
7
8 def parent_code():
9     child = Thread(child_code, 10)
10    child.join()
11
12 def child_code(n=10):
13     for i in range(n):
14         print i
15         time.sleep(1)
16
17 parent_code()

```

.\SIGINT a or Ctrl-C with interrupted be cannot it that find will You works only it so ,os.wait and os.fork uses problem this for workaround My the threads. new creating before works: it how Here's Macintosh. and UNIX on and returns process new The process. new a forks which ,watcher invokes program to process child the for waits process original The program. the of rest the executes :watcher name the hence complete.

The watcher

```

1 import threading, time, os, signal, sys
2
3 class Thread(threading.Thread):
4     def __init__(self, t, *args):
5         threading.Thread.__init__(self, target=t, args=args)
6         self.start()
7
8 def parent_code():
9     child = Thread(child_code, 10)
10    child.join()
11

```

was it but ,\۶۷۹۳° number assigned and reported been had bug this writing: this of time the At\

.(<https://sourceforge.net/projects/python/>) unassigned and open

```

12 def child_code(n=10):
13     for i in range(n):
14         print i
15         time.sleep(1)
16
17 def watcher():
18     child = os.fork()
19     if child == 0: return
20     try:
21         os.wait()
22     except KeyboardInterrupt:
23         print 'KeyboardInterrupt '
24         os.kill(child, signal.SIGKILL)
25         sys.exit()
26
27 watcher()
28 parent_code()

```

with it interrupt to able be should you program. the of version this run you If
to delivered is SIGINT the that guaranteed is it think I but sure. not am I Ctrl-C.
deal to have threads child and parent the thing less one that's so process. watcher the
with.

can you which .threading_cleanup.py named file a in code this all keep I
greenteapress.com/semaphores/threading_cleanup.py from download
code this that understanding the with presented are Chapter in examples The
code. example the to prior executes

پیوست ب

threads POSIX up Cleaning

little a C in multithreading make to use I code utility some present I section. this In
code. this on based are ۹ Section in examples The pleasant. more
Threads. POSIX is C with used standard threading popular most the Probably
interface an and model thread a defines standard POSIX The short. for Pthreads or
imple– an provide UNIX of versions Most threads. controlling and creating for
Pthreads. of mentation

code Pthread Compiling ب.۱

libraries: C most using like is Pthreads Using

program. your of beginning the at files headers include You •

Pthreads. by defined functions calls that code write You •

library. Pthread the with it link you program. the compile you When •

headers: following the include I examples. my For

Headers

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
```

```
#include <semaphore.h>
```

for is fourth the and Pthreads for is third the standard: are two first The
option -l the use can you .gcc in library Pthread the with compile To semaphores.

line: command the on

Listing:

```
gcc -g -O2 -o array array.c -lpthread
```

the with links optimization, and info debugging with array.c compiles This
.array named executable an generates and library. Pthread
you handling, exception provides that Python like language a to used are you If
error for check to you require that C like languages with annoyed be probably will
calls function library wrapping by hassle this mitigate often I explicitly. conditions
example. For functions. own my inside code error-checking their with together
value. return the checks that malloc of version a is here

Listing:

```
void *check_malloc(int size)
{
    void *p = malloc (size);
    if (p == NULL) {
        perror ("malloc failed");
        exit (-1);
    }
    return p;
}
```

threads Creating ب.٢

my here's use: to going I'm functions Pthread the with thing same the done I've
.pthread_create for wrapper

Listing:

```
pthread_t make_thread(void *entry)(void , Shared *shared)
{
    int n;
    pthread_t thread;
```



```

5
6     n = pthread_create (&thread, NULL, entry, (void *)shared);
7     if (n != 0) {
8         perror ("pthread_create failed");
9         exit (-1);
10    }
11    return thread;
12 }

```

of think can you which ,pthread_t is pthread_create from type return The
imple- the about worry to have shouldn't You thread. new the for handle a as
a of semantics the has it that know to have do you but ,pthread_t of mentation
immutable an as handle thread a of think can you that means That .\ type primitive
this point I problems. causing without value by it pass or it copy can you so value.

minute. a in to get will I which semaphores, for true not is it because now out
of handle the returns function my and ° returns it succeeds, pthread_create If
my and code error an returns pthread_create occurs. error an If thread. new the
exits. and message error an prints function
the with Starting explaining. some take pthread_create of parameters The
fol- The variables. shared contains that structure user-defined a is Shared second.
type: new the creates statement typedef lowing

Listing:

```

1 typedef struct {
2     int counter;
3 } Shared;

```

space allocates make_shared .counter is variable shared only the case. this In
contents: the initializes and structure Shared a for

Listing:

```

1 Shared *make_shared ()
2 {
3     int i;
4     Shared *shared = check_malloc (sizeof (Shared));
5     shared->counter = 0;
6     return shared;

```

know. I implementations the all in is pthread_t a what is which example. for integer. an Like\

```
7 }

```

.pthread_create to back get let's structure. data shared a have we that Now
returns and pointer void a takes that function a to pointer a is parameter first The
are you bleed, eyes your makes type this declaring for syntax the If pointer. void a
where function the specify to is parameter this of purpose the Anyway, alone. not
named is function this convention. By begin. will thread new the of execution the
:entry

Listing:

```
1 void *entry (void *arg)
2 {
3     Shared *shared = (Shared *) arg;
4     child_code (shared);
5     pthread_exit (NULL);
6 }
```

pro- this in but pointer. void a as declared be to has entry of parameter The
it typecast can we so structure. Shared a to pointer a really is it that know we gram
work. real the does which ,child_code to along it pass then and accordingly
to used be can which pthread_exit invoke we returns. child_code When
this In thread. this with joins that parent) the (usually thread any to value a pass
.NULL pass we so say. to nothing has child the case.

threads Joining ٣.ب

invokes it complete. to thread another for wait to want thread one When
:pthread_join for wrapper my is Here .pthread_join

Listing:

```
1 void join_thread (pthread_t thread)
2 {
3     int ret = pthread_join (thread, NULL);
4     if (ret == -1) {
5         perror ("pthread_join failed");
6         exit (-1);
7     }
8 }
```

function my All for. wait to want you thread the of handle the is parameter The
result. the check and pthread_join call is does

Semaphores ٤.ب

part not is interface This semaphores. for interface an specifies standard POSIX The
semaphores. provide also Pthreads implement that UNIXes most but Pthreads. of
your make can you semaphores. without and Pthreads with yourself find you If
Section see own: ٢.٩.

the about know to have shouldn't You .sem_t type have semaphores POSIX
seman- structure has it that know to have do you but type. this of implementation
the of copy a making are you variable a to it assign you if that means which tics.
In idea. bad a certainly almost is semaphore a Copying structure. a of contents
undefined. is copy the of behavior the POSIX.
semantics. structure with types denote to letters capital use I programs. my In
wrapper a put to easy is it Fortunately. pointers. with them manipulate always I and
the and typedef the is Here object. proper a like behave it make to sem_t around
semaphores: initializes and creates that wrapper

Listing:

```

1 typedef sem_t Semaphore;
2
3 Semaphore *make_semaphore (int n)
4 {
5     Semaphore *sem = check_malloc (sizeof(Semaphore));
6     int ret = sem_init(sem, 0, n);
7     if (ret == -1) {
8         perror ("sem_init failed");
9         exit (-1);
10    }
11    return sem;
12 }
```

al- It parameter. a as semaphore the of value initial the takes make_semaphore
.Semaphore to pointer a returns and it. initializes Semaphore. a for space locates

`_` returns it that means which reporting. error UNIX old-style uses `sem_init` that is functions wrapper these about thing nice Once wrong. went something if style. reporting which use functions which remember to have don't we pro- real a like looks almost that code C write can we definitions. these With language: gramming

Listing:

```
1 Semaphore *mutex = make_semaphore(1);
2 sem_wait(mutex);
3 sem_post(mutex);
```

fix can we but .signal of instead post use semaphores POSIX Annoyingly, that:

Listing:

```
1 int sem_signal(Semaphore *sem)
2 {
3     return sem_post(sem);
4 }
```

now. for cleanup enough That's