

به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی



برنامه نویسی همروند

- همروندي يا همزمانی¹ در يك برنامه کامپيوتری به معنای توانايی واحدهای مختلف يك برنامه برای اجرای همزمان يا به عبارت دیگر اجرای موازی است به طوری که نتیجه نهايی برنامه نسبت به اجرای غير همزمان تفاوتی نداشته باشد. با اجرای يك برنامه به صورت همروند بر روی چند پردازنده، سرعت اجرا بهبود می یابد.
- توسط برنامه نويسی همرونند² که توسط آن قسمت‌هایی از برنامه به صورت همزمان اجرا می‌شوند، زمان پاسخ (تأخير)³ و توان عملیاتی⁴ برنامه بهبود می یابد.

¹ concurrency

² concurrent programming

³ response time (delay)

⁴ throughput

- به طور مثال ضرب دو ماتریس را در نظر بگیرید. از آنجایی که درایه‌های ماتریس حاصلضرب مستقل از یکدیگر قابل محاسبه هستند، بنابراین هر یک از درایه‌ها در صورتی که تعداد پردازنده‌ها کافی باشد می‌تواند به طور مستقل و موازی با درایه‌های دیگر محاسبه شود. در این صورت ماتریس با تأخیر کمتر محاسبه خواهد شد.
- حال یک سیستم پردازش تصویر را در نظر بگیرید که در آن تصاویر به ترتیب از ورودی خوانده می‌شوند و پس از چند مرحله پردازش در خروجی نمایش داده می‌شوند. پس از این که اولین مرحله پردازش توسط یک پردازنده انجام شد، در صورتی که تعداد پردازنده‌ها کافی باشد، پردازنده اول می‌تواند تصویر دوم را پردازش کند و تصویر اول برای پردازنده دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر با استفاده از برنامه نویسی همواره می‌تواند پردازش شود. در این حالت می‌گوییم توان عملیاتی سیستم افزایش یافته است.

- همروندي می‌تواند در سطوح متفاوتی باشد : در سطح دستورات ماشین، در سطح دستورات زبان برنامه نویسی، در سطح زیر برنامه و یا در سطح برنامه.
- همروندي در سطح دستورات ماشين مربوط به طراحی سخت افزار و همروندي در سطح برنامه مربوط به طراحی سیستم عامل است. طراحان سخت افزار سازوکارهايی را برای اجرای موازي دستورات زبان ماشين فراهم می‌کنند و طراحان سیستم عامل روش‌هایی را برای اجرای موازي برنامه‌ها و زمانبندی برنامه‌ها پیاده‌سازی می‌کنند. همروندي در سطح دستورات زبان برنامه‌نویسی و زیر برنامه‌ها موضوع بحث زبان‌های برنامه نویسی است.
- در بسیاری از برنامه‌ها همروندي اهمیت زیادی پیدا می‌کند. برای مثال یک سرور وب به طور همزمان داده‌ها را دریافت و ارسال می‌کند و اطلاعات را به کاربر نمایش می‌دهد و به درخواست‌های کاربر واکنش نشان می‌دهد.
- یک برنامه همرونند مقیاس پذیر¹ است زیرا اگر تعداد پردازنده‌ها افزایش یابد می‌تواند تعداد بیشتری داده را پردازش کند و با سرعت بیشتری اجرا شود.

¹ scalable

- اولین کامپیوترهایی که پردازش موازی را پشتیبانی می‌کردند، در دهه ۱۹۵۰ به وجود آمدند که شامل یک پردازنده اصلی برای انجام محاسبات و یک یا چند پردازنده برای انجام عملیات ورودی و خروجی می‌شدند.
- در دهه ۱۹۶۰ سیستم عامل‌ها، برنامه‌های مختلف را به طور همزمان بر روی چندین پردازنده اجرا و زمانبندی می‌کردند.
- در اواسط دهه ۱۹۶۰ کامپیوترهایی به وجود آمدند که دستورات ماشین را به طور همزمان می‌توانستند اجرا کنند. کامپایلرها بر روی این ماشین‌ها می‌توانستند تعیین کنند چه دستوراتی به طور همزمان اجرا شوند.

- در آن زمان معماری سخت افزارهای کامپیوترا به دو دسته تقسیم شد : معماری یک عملیات چند داده و معماری چند عملیات چند داده.
- در معماری اول چندین پردازنده به طور همزمان قسمت های مختلف داده را پردازش می کنند. بنابراین در این معماری که به آن معماری یک عملیات - چند داده^۱ گفته می شود، پردازش موازی در سطح داده^۲ صورت می گیرد. به عبارت دیگر در این معماری داده به چند قسمت تقسیم شده و یک عملیات واحد بر روی قسمت های مختلف داده اعمال می شود. پردازنده های گرافیکی غالباً از این نوع معماری هستند.

¹ Single Instruction, Multiple Data (SIMD)

² data level parallelism

- برای مثال فرض کنید می‌خواهیم عناصر یک آرایه را با یکدیگر جمع کنیم. می‌توانیم آرایه را به چند قسم تقسیم کرده، و عملیات جمع را بر روی قسمت‌های مختلف آرایه اعمال کنیم. در این معماری از موازی‌سازی داده بهره گرفته می‌شود. در بسیاری از کاربردهای پردازش صدا و تصویر از موازی‌سازی داده بهره گرفته می‌شود بدین صورت که تصویر یا صدا به چند قسم تقسیم شده، سپس پردازنده‌های مختلف یک عملیات واحد را بر روی قسمت‌های مختلف صدا یا تصویر انجام می‌دهند.

- در معماری دوم چندین پردازنده به طور همزمان عملیات متفاوت را بر روی چندین داده متفاوت اعمال می‌کنند و به آن معماری چند عملیات-چند داده^۱ گفته می‌شود. به عبارت دیگر چندین عملیات در یک برنامه می‌توانند به طور همزمان اجرا شوند و هر یک از عملیات بر روی یک پردازنده متفاوت اجرا می‌شود. در این معماری از موازی‌سازی در سطح عملیات^۲ بهره گرفته می‌شود.

¹ Multiple Instruction, Multiple Data (MIMD)

² task level parallelism

- برای مثال یک سیستم عامل یا یک مرورگر وب به طور همزمان عملیات متفاوتی را انجام می‌دهد که این عملیات بر روی پردازنده‌ها توزیع می‌شوند. به عنوان مثال دیگر در یک برنامه پردازش تصویر، یک پردازنده عملیاتی را بر روی یک تصویر انجام داده و پس از اعمال عملیات، تصویر را به پردازندهٔ بعدی برای اعمال عملیات دیگر انتقال می‌دهد و خود پردازش را با تصاویر بعدی ادامه می‌دهد. در این معماری حافظه می‌تواند مشترک باشد که در آن صورت نیاز به همگام‌سازی^۱ یا هماهنگ‌سازی واحدهای پردازش وجود دارد تا داده‌ها به درستی خوانده و نوشته شوند. حافظه همچنین می‌تواند توزیع شده^۲ باشد که در این صورت نیز نیاز که برقراری ارتباط بین پردازنده‌ها وجود دارد. به این معماری معمولاً چندپردازنده‌ای^۳ نیز گفته می‌شود.

¹ synchronization

² distributed

³ multiprocessor

- از آنجایی که معماری سخت افزار دائماً در حال به روز رسانی است کامپیووترهای جدید روش‌های همزمانی متنوعی را در سطح سخت افزار پشتیبانی می‌کنند. برای مثال وقتی دستورات جاری در حال اجرا هستند، دستورات آینده کد گشایی و آماده اجرا می‌شوند یا دو مسیر متفاوت برای بارگیری دستورات و داده‌ها در پردازنده‌ها وجود دارد و یا بخش‌های مختلف دستورات محاسباتی ریاضی تا حد امکان به طور موازی اجرا شوند.
- همروندی در سخت افزار تا حدودی می‌تواند نیازهای راندمان برنامه را برآورده کند و قسمتی از مسئولیت بهبود زمان اجرا بر عهده نرم افزار است.

- دو دسته از واحدهای همروند وجود دارند. در دسته اول با فرض بر این که بیشتر از یک پردازنده وجود دارد، چندین واحد از برنامه به طور موازی بر روی پردازنده‌های مختلف اجرا می‌شوند. به این دسته همروندي فیزیکی^۱ گفته می‌شود. در دسته دوم یک پردازنده وجود دارد و واحدهای مختلف برنامه به طور همزمان به طور قطعه قطعه شده و در هم آمیخته^۲ بر روی پردازنده اجرا می‌شوند. به این دسته همروندي منطقی^۳ گفته می‌شود.
- در بیشتر مواقع تعداد واحدهای همروند از تعداد پردازنده‌ها بیشتر است و بنا بر این همروندي به صورت فیزیکی و منطقی اتفاق می‌افتد.

^۱ physical concurrency

^۲ interleaved

^۳ logical concurrency

- یک ریسمان کنترلی^۱ یا ریسه کنترلی به دنبالهای از دستورات گفته می شود که به طور پیوسته یکی پس از دیگری در یک واحد از برنامه اجرا می شوند و می تواند به طور مجزا توسط سیستم عامل زمانبندی شود.
- در هموندی فیزیکی هر پردازنده تنها می تواند یک ریسه کنترلی را اجرا کند، اما در هموندی منطقی بیش از یک ریسه کنترلی نیز می توانند بر روی یک پردازنده اجرا شوند.
- برنامه ای که در آن چند ریسه کنترلی به طور همزمان اجرا می شوند، یک برنامه چند ریسه ای^۲ گفته می شود.

^۱ thread of control

^۲ multithreaded program

- همروندي در سطح دستورات معمولاً به اين صورت است که دستوراتي که می‌توانند به طور موازي اجرا شوند، به صورت خودکار توسط کامپایلر تشخيص داده شده و به صورت موازي بر روی چندین پردازنده انجام شوند. برای مثال دستورات زیر را در نظر بگيريد.

۱ $e = a + b$
۲ $f = c + d$
۳ $m = e * f$

- دستور سوم وابسته به دستورات اول و دوم است، اما دستورات اول و دوم از يكديگر مستقل هستند و بنابراين می‌توانند به صورت موازي اجرا شوند. در اينصورت با تحليل وابستگي دستورات، و اعمال موازي سازی برنامه می‌تواند با سرعت بيشتری اجرا شود.
- همروندي در سطح زيربرنامه معمولاً به اين صورت است که هر زير برنامه به يك ريسه کنترل سپرده می‌شود.

- یک واحدکار یا یک وظیفه^۱ واحدی است از برنامه که می‌تواند به صورت همروند با واحدهای دیگر اجرا شود.
- توجه کنید که اجرای موازی^۲ حالت خاصی از اجرای همروند^۳ است. در اجرای همروند چندین واحد کاری به گونه‌ای اجرا می‌شوند که در بازه‌های زمانی متفاوت اجرای آنها همپوشانی داشته باشد. در اجرای موازی چندین واحد کاری در یک بازهٔ معین همزمان اجرا می‌شوند.
- معمولاً یک ریسه یک واحد کار را به عهده می‌گیرد. وقتی ریسه شروع به انجام عملیات می‌کند، برنامه‌ای که ریسه را راه اندازی کرده است نیاز ندارد منتظر اتمام عملیات ریسه بماند و می‌تواند عملیات خود را ادامه دهد یا ریسه‌های دیگر را راه اندازی کند.
- واحدهای کاری می‌توانند حافظه را با یکدیگر به اشتراک بگذارند و یا هر کدام حافظه مختص به خود داشته باشند.

¹ task

² parallel

³ concurrent

- به واحدهای کنترلی که واحدهای کاری را بدون به اشتراک گذاری حافظه پردازش می‌کنند، یعنی برای هر واحد کاری یک فضای جداگانه در حافظه در نظر می‌گیرند، پردازه یا پروسه¹ گفته می‌شود.
- ریسه‌ها² مسئول پردازش واحدهای کار با به اشتراک گذاری حافظه هستند.
- ریسه‌ها نسبت به پروسه‌ها بھینه‌تر و کارآمدتر هستند، اما از طرفی چون حافظه آنها اشتراکی است نیاز به همگام‌سازی دارند.
- یک ریسه می‌تواند با ریسه‌های دیگر از طریق به اشتراک گذاری متغیرها یا از طریق کانال‌های ارتباطی و مکانیزم ارسال پیام ارتباط برقرار کند.

¹ process

² thread

- همگام‌سازی^۱ سازوکاری (مکانیزمی) است که توسط آن دسترسی به داده‌های مشترک کنترل می‌شود.
- به وضعیتی که در آن دو یا چند ریسه برای به دست آوردن یک منبع مشترک رقابت می‌کنند، یک وضعیت رقابتی^۲ گفته می‌شود.
- معمولاً یک برنامه سیستم عامل به نام زمانبند^۳ ریسه‌ها و پروسه‌ها را برای اجرا بر روی پردازنده‌های مختلف زمانبندی می‌کند.

¹ synchronization

² race condition

³ scheduler

- یک واحدکار می‌تواند در چند حالت مختلف باشد :
۱. جدید^۱ : واحدکار به تازگی ساخته شده و هنوز آماده انجام عملیات نیست.
 ۲. آماده^۲ : واحدکار آماده اجرا است، اما در حال اجرا نیست. زمانبند باید این واحدکار را زمانبندی کند تا به حالت اجرا درآید. همه واحدهای کار که آماده به شروع عملیات هستند در یک صف واحدهای کاری آماده^۳ قرار می‌گیرند.
 ۳. در حال اجرا^۴ : یک پردازنده به واحدکار داده شده و می‌تواند اجرا شود.

¹ new

² ready

³ task-ready queue

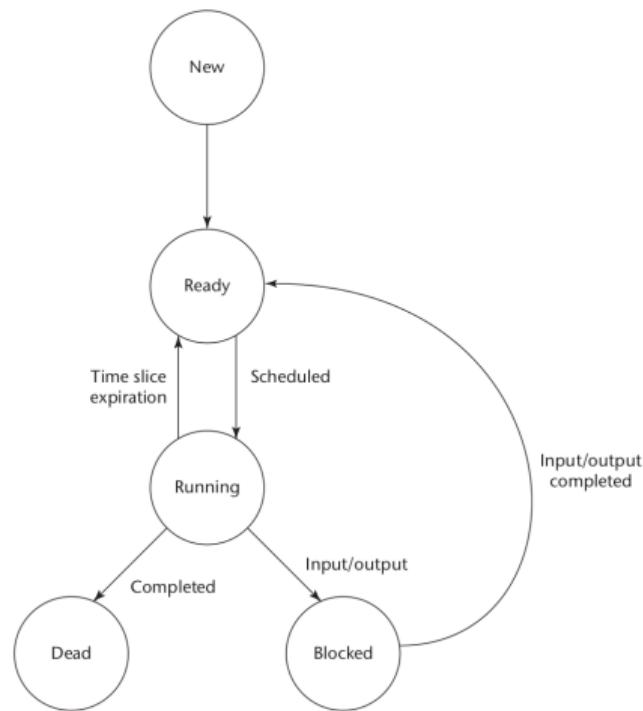
⁴ running

- ۴. مسدود^۱ : واحد کار مسدود شده است و اجرای آن متوقف شده است. دلیل توقف می‌تواند این باشد که واحد کار نیاز به عملیات ورودی خروجی داشته و یا اینکه زمانبند برای اجرای واحدهای کاری دیگر آن متوقف کرده است.
- ۵. پایان یافته^۲ : اجرای واحدکار به اتمام رسیده است. واحد کار یا با موفقیت به کار خود پایان داده است و یا به دلیل خطأ یا به طور دستی اجرای آن متوقف شده و تخربیب شده است.

¹ blocked

² dead

- در شکل زیر حالت‌های مختلف یک واحد کار نشان داده شده‌اند.



یک ویژگی برنامه‌های همروند ویژگی زنده بودن^۱ است. یک برنامه زنده برنامه‌ای است که اجرای آن تا خاتمه برنامه ادامه پیدا کند و اجرای آن متوقف نشود. یک برنامه ممکن است با بن‌بست^۲ مواجه شود بدین معنی که دو یا چند ریسه برای ادامه کار به آزادسازی منابع توسط دیگران نیاز پیدا می‌کنند و بنابراین هیچ‌کدام نمی‌توانند کار خود را ادامه دهند که در نتیجه برنامه به بن‌بست برخورد می‌کند.

^۱ liveness

^۲ deadlock

- دو نوع همگام‌سازی برای داده‌های مشترک وجود دارد : همگام‌سازی مشارکتی^۱ و همگام‌سازی رقابتی^۲.
- در همگام‌سازی مشارکتی دو واحدکاری به طور همزمان کاری را انجام می‌دهند و یکی از واحدها در نقطه‌ای متوقف می‌شود تا واحد دیگر کار خود را به اتمام برساند و با ارسال پیام به واحد کار اول اطلاع دهد که می‌تواند عملیات خود را ادامه دهد.
- در همگام‌سازی رقابتی چند واحدکاری نیاز به یک منبع مشترک دارند و آن منبع مشترک نمی‌تواند به طور همزمان مورد استفاده قرار بگیرد پس باید بر سر به دست آوردن منبع با یکدیگر رقابت کنند و اگر یک واحد کاری منبع را به دست آورده واحدکاری دیگر باید صبر کند تا منبع آزاد شود.

¹ cooperation synchronization

² competition synchronization

- در ادامه به سه روش مختلف برای اجرای همزمان واحدهای کار به صورت همروند ارائه می‌شود :
سمافور^۱ ، مانیتور^۲ و ارسال پیام^۳ .

^۱ semaphore

^۲ monitor

^۳ message passing

- یک سمافور سازوکاری ساده است که برای همگام‌سازی واحدهای کاری استفاده می‌شود.
- سمافور راهکاری قدیمی است که همچنان در زبان‌های برنامه نویسی همروند و کتابخانه‌هایی که برای پشتیبانی از همروندی طراحی شده‌اند استفاده می‌شود.
- ادسخر دایکسترا^۱ در سال ۱۹۶۵ سمافور را به عنوان راه حلی برای همگام‌سازی داده‌های مشترک بین واحدهای پردازش طراحی کرد.

^۱ Edsger Dijkstra

- برای حفاظت از منابع مشترک باید یک محافظ^۱ در اطراف منبع مشترک قرار بگیرد.
- یک محافظ در واقع وسیله‌ای است که به یک واحدکار اجازه می‌دهد که به منبعی دسترسی پیدا کند اگر شرایط آن برقرار باشد.
- سمافور در واقع پیاده سازی چنین محافظی است.
- یک سمافور ساختار داده‌ای است که از یک عدد صحیح و یک صف تشکیل شده است. این صف که صف توصیف وظایف^۲ نامیده می‌شود، اطلاعات مربوط به اجرای وظایف (واحدهای کار) را در خود نگه می‌دارد.

¹ guard

² task descriptor queue

- یک روش ساده برای پیاده سازی سمافور به اینگونه است که اطلاعات هر واحدکاری که نیاز به دسترسی به منبع مشترک دارد، در یک صفت نگهداری شده و سپس دسترسی به آنها به طور ترتیبی داده می‌شود.
- دو نوع عملیات مهم بر روی سمافور عملیات انتظار^۱ و آزادسازی^۲ یا سیگنال^۳ نامیده می‌شوند.

¹ wait

² release

³ signal

- از شمارنده سمافور می‌توان برای شمردن تعداد واحدهای کاری که می‌توانند به طور همزمان به یک منبع دسترسی داشته باشند استفاده کرد.
- یک سمافور دودویی¹ سمافوری است که شمارنده آن تنها بتواند صفر و یا یک باشد.

¹ binary semaphore

- تابع SemInit برای مقداردهی اولیه سمافور به صورت زیر عمل می‌کند.

Algorithm Semaphore Initialization

```
1: function SEMINIT(S, n)
2:     S.counter = n
3:     InitQueue(S.waitingQueue)
```

- تابع انتظار Wait به صورت زیر عمل می‌کند.

Algorithm Semaphore Wait

```
1: function WAIT(S)
2:   S.counter = S.counter - 1
3:   if S.counter < 0 then
4:     ▷ Block the caller process and add it to the waiting queue
5:     Enqueue(S.waitingQueue, process)
6:     Block(process)
    ▷ Control is transferred to some ready task.
    ▷ If the task-ready queue is empty, deadlock occurs.
```

- تابع آزادسازی Signal به صورت زیر عمل می‌کند.

Algorithm Semaphore Release

```
1: function SIGNAL(S)
2:     S.counter = S.counter + 1
3:     if S.counter ≤ 0 then
4:         ▷ Unblock a process from the waiting queue
5:         process = Dequeue(S.waitingQueue)
6:         Unblock(process)
```

- در زمان کامپایل نمی‌توان بررسی کرد که سمافورها درست استفاده شده‌اند، بنابراین ممکن است برنامه‌ای که از سمافور استفاده می‌کند با بنبست مواجه شود.
- در زبان سی++ سمافور دودویی توسط ساختار داده `mutex` پیاده سازی شده است.

- در یک برنامه تولید-صرف داده بر روی یک بافر، می‌توان از سمافور به صورت زیر استفاده کرد.

Algorithm Producer

```
1: SemInit(sem, 1)
2: SemInit(fullspots, 0)
3: SemInit(emptyspots, BUFLEN)
4: function PRODUCER
5:   while true do
6:     ▷ Produce value
7:     Wait(emptyspots)
8:     Wait(sem)
9:       ▷ Deposit value
10:    Signal(sem)
11:    Signal(fullspots)
```

Algorithm Consumer

```
- 1: function CONSUMER
  2:   while true do
  3:     Wait(fullspots)
  4:     Wait(sem)
  5:       ▷ Fetch value
  6:     Signal(sem)
  7:     Signal(emptyspots)
  8:       ▷ Consume value
```

- یک روش دیگر برای همگام‌سازی داده‌های مشترک استفاده از مانیتور^۱ است.
- مانیتور ساختار داده‌ای است که علاوه بر دربرگرفتن مکانیزم‌های لازم برای همگام‌سازی، داده را نیز در بر می‌گیرد، بنابراین داده مشترک در درون خود مانیتور است.
- از آنجایی که داده در درون مانیتور جای می‌گیرد، تنها یک دسترسی در هر لحظه به استفاده کننده می‌توان داد.
- اگر در زبان جاوا یک کلاس تعریف کنیم که همه توابع آن همزمان باشند، و از آن کلاس یک شیء بسازیم در واقع یک مانیتور ساخته‌ایم، زیرا توابع کلاس همگام سازی بر روی داده‌هایی را انجام می‌دهند که در شیء قرار دارند.

¹ monitor

ارسال پیام

- گاهی لازم است دو واحدکاری با یکدیگر ارتباط برقرار کنند. در چنین مواقعي فعالیت‌ها به پیام‌های دریافتی بستگی پیدا می‌کند.
- برای مثال فرض کنید یک واحدکاری در حال انجام محاسبات است و در همان هنگام واحدکاری دوم نیاز دارد که واحدکاری اول محاسباتی را انجام دهد. واحدکاری اول نمی‌تواند کار خود را متوقف کند، بنابراین واحدکاری دوم پیامی را در صندوق پیام‌های واحد اول ارسال می‌کند و به محض اینکه واحدکاری اول فرصت پیدا کرد، پیام را دریافت و محاسبات مورد نظر را انجام می‌دهد.
- بنابراین در این مکانیزم واحدهای کاری با یکدیگر ارتباط برقرار می‌کنند و پیام‌های خود را در صف پیام‌های یکدیگر ارسال می‌کنند.
- در زبان سی++ مکانیزم ارسال پیام توسط متغیرهای وضعیت condition variables پیاده سازی شده است.

سی++ : همروندي

- در زبان سی++ می‌توان از کتابخانه استاندارد `thread` برای ایجاد ریسه استفاده کرد.

```
1 #include <thread>
2 #include <iostream>
3 using namespace std;
4 void hello() {
5     cout << "Hello from thread!\n";
6 }
7 int main() {
8     thread t(hello); // Create thread
9     t.join(); // Wait for completion
10 }
```

- تابع `join` باعث می‌شود ریسه‌ای که یک ریسه دیگر را ساخته است، منتظر اتمام پردازش ریسه بماند.
بنابراین با استفاده از این تابع می‌توان ریسه‌ها همگام کرد.

می‌توان به یک ریسه مقدار نیز ارسال کرد.

```
1 void print_num(int n) {
2     cout << n << "\n";
3 }
4 int main() {
5     int x = 42;
6     thread t(print_num, x);
7     t.join();
8 }
```

- همچنین می توان به یک ریسه یک تابع لامبدا ارسال کرد.

```
1 int main() {  
2     int x = 10;  
3     thread t([x](){ cout << x*x << "\n"; });  
4     t.join();  
5 }
```

- حال فرض کنید دو ریسه می خواهند همزمان به یک مکان حافظه دسترسی پیدا کنند. برای مثال هر دو می خواهند کد زیر را اجرا کنند.

۱ $x = x + 1;$

- فرض کنید مقدار اولیه x برابر با ۱ است. ریسه اول مقدار x را می خواند و یک واحد به آن می افزاید، اما قبل از به روز رسانی مقدار x کنترل به ریسه دوم داده می شود و ریسه دوم همه مقدار x را که همچنان ۱ است می خواند و یک واحد به آن می افزاید. حال کنترل به ریسه اول بازگردانده می شود و مقدار x را به روز رسانی می کند و مقدار آن برابر با ۲ می شود. سپس کنترل به ریسه دوم می رسد و مقدار را به روز رسانی می کند که مقدار برابر با ۲ می شود. انتظار داشتیم هر کدام از ریسه ها یک واحد به مقدار x بیافزایند و در نهایت مقدار برابر با ۳ شود، ولی این اتفاق نیافتاد.

- این مشکل توسط mutex حل می‌شود که یک سمافور دودویی است.

```
1 #include <iostream>
2 #include <thread>
3 #include <mutex>
4 using namespace std;
5 mutex mtx;           // mutex for critical section
6 int counter = 0;     // shared resource
7
8 void increment(int id) {
9     for (int i = 0; i < 100; i++) {
10         mtx.lock();           // manually lock
11         counter = counter + 1; // critical section
12         mtx.unlock();        // manually unlock
13     }
14 }
```

```
15 int main() {
16     thread t1(increment, 1);
17     thread t2(increment, 2);
18
19     t1.join();
20     t2.join();
21
22     cout << "Final counter value: " << counter << endl;
23     return 0;
24 }
```

- با استفاده از `condition_variable` می‌توان بین دو ریسه ارتباط برقرار کرد.

```
1 #include <iostream>
2 #include <thread>
3 #include <mutex>
4 #include <condition_variable>
5 #include <queue>
6 using namespace std;
7 const int BUFFER_SIZE = 5;
8 const int TOTAL_ITEMS = 100;
9 queue<int> buffer;
10 mutex mtx;
11 condition_variable_any cv;
```

```
۱۶ void producer() {
۱۷     for (int i = 1; i <= TOTAL_ITEMS; ++i) {
۱۸         mtx.lock();
۱۹         while (buffer.size() == BUFFER_SIZE) {
۲۰             cv.wait(mtx);    // releases lock while waiting
۲۱         }
۲۲         buffer.push(i);
۲۳         cout << "Produced: " << i << endl;
۲۴         mtx.unlock();
۲۵         cv.notify_all();
۲۶     }
۲۷ }
```

```
۱۶ void consumer() {  
۱۷     for (int i = 1; i <= TOTAL_ITEMS; ++i) {  
۱۸         mtx.lock();  
۱۹         while (buffer.empty()) {  
۲۰             cv.wait(mtx);    // releases lock while waiting  
۲۱         }  
۲۲  
۲۳         int item = buffer.front();  
۲۴         buffer.pop();  
۲۵         cout << "Consumed: " << item << endl;  
۲۶  
۲۷         mtx.unlock();  
۲۸         cv.notify_all();  
۲۹     }  
۳۰ }
```

```
۲۸ int main() {  
۲۹     thread prod(producer);  
۳۰     thread cons(consumer);  
۳۱  
۳۲     prod.join();  
۳۳     cons.join();  
۳۴  
۳۵     return 0;  
۳۶ }
```

- در زبان جاوا برای ایجاد یک ریسه باید کلاسی که عملیات ریسه را پیاده سازی می‌کند یا از کلاس Thread به ارث ببرد یا رابط Runnable را پیاده سازی کند. سپس عملیات ریسه در متود run قرار داده می‌شود. ریسه با فراخوانی متود start() بر روی شیء آغاز به کار می‌کند و برای انتظار برای اتمام ریسه از متود join() استفاده می‌کند.
- سمافورها توسط کلاس Semaphore پیاده سازی شده‌اند که دو متود acquire و release برای آن تعریف شده است.
- یک متود می‌تواند با استفاده از کلمه synchronized تعریف شود. متودی که به صورت همگام شده¹ تعریف شده است باید عملیات خود را به اتمام برساند تا بتواند دوباره توسط یک ریسه فراخوانی شود.
- پیاده سازی این متودها به این صورت است که هر شیء در جاوا یک قفل دارد و برای فراخوانی یک تابع همزمان، ابتدا باید شیء قفل شود، وقتی اجرای متود به پایان رسید، قفل آزاد می‌شود.

¹ synchronized

- کلاسی که همه توابع آن به صورت همگام شده تعریف شده باشند در واقع یک مانیتور است.

- اگر بخواهیم به جای همگام‌سازی کل یک متود فقط قسمتی از آن را همگام‌سازی کنیم، می‌توانیم از کلمه `synchronized` قبل از یک بلوک این کار را انجام دهیم.

```
1 synchronized (expression) {  
2     statements  
3 }
```

- در اینجا `expression` در واقع یک شیء است که بر روی آن یک قفل گرفته می‌شود.

- هر شیء در جاوا یک صف در اختیار دارد که این صف اطلاعات ریشه‌هایی که نیاز به آن شیء دارند را در خود نگه می‌دارد.

- برای ارتباط بین ریسه‌ها، همه اشیای جاوا که از یک کلاس جد به نام کلاس Object به ارث می‌برند، متودهایی به نام `notify` ، `wait` و `notifyAll` دارند.
- متود انتظار (`wait`) موجب می‌شود یک ریسه در صف انتظار یک شی وارد شود و منتظر می‌ماند تا وقتی که یک ریسه دیگر متود اعلام (`notify`) را فراخوانی کند. وقتی متود `notify` فراخوانی می‌شود، در واقع تغییری در سیستم رخ داده است و ریسه‌ای که در صف انتظار بوده است باید بررسی کند که تغییر صورت گرفته نیازش را برآورده می‌سازد یا خیر.

جاوا : همروندي

- در برنامه زیر برای همگام سازی و خواندن و نوشتن بر روی یک صف از مکانیزم انتظار و اعلام استفاده شده است.

```
1 // Queue
2 // This class implements a circular queue for storing int
3 // values. It includes a constructor for allocating and
4 // initializing the queue to a specified size. It has
5 // synchronized methods for inserting values into and
6 // removing values from the queue.
7 class Queue {
8     private int [] que;
9     private int nextIn, nextOut, filled, queSize;
10    public Queue(int size) {
11        que = new int [size];
12        filled = 0;
13        nextIn = 1;
14        nextOut = 1;
```

```
1      queSize = size;
2  } /** end of Queue constructor
3  public synchronized void deposit (int item)
4      throws InterruptedException {
5  try {
6      while (filled == queSize)
7          wait();
8      que [nextIn] = item;
9      nextIn = (nextIn % queSize) + 1;
10     filled++;
11     notifyAll();
12 } /** end of try clause
13 catch(InterruptedException e) {}
14 } /** end of deposit method
```

```
1 public synchronized int fetch()
2     throws InterruptedException {
3     int item = 0;
4     try {
5         while (filled == 0)
6             wait();
7         item = que [nextOut];
8         nextOut = (nextOut % queSize) + 1;
9         filled--;
10        notifyAll();
11    } /** end of try clause
12    catch(InterruptedException e) {}
13    return item;
14 } /** end of fetch method
15 } /** end of Queue class
```

- حال تولیدکننده و مصرفکننده صف به صورت دو ریسه به صورت زیر تعریف می‌شوند.

```
1 class Producer extends Thread {  
2     private Queue buffer;  
3     public Producer(Queue que) {  
4         buffer = que;  
5     }  
6     public void run() {  
7         int new_item;  
8         while (true) {  
9             //-- Create a new_item  
10            buffer.deposit(new_item);  
11        }  
12    }  
13 }
```

```
1 class Consumer extends Thread {  
2     private Queue buffer;  
3     public Consumer(Queue que) {  
4         buffer = que;  
5     }  
6     public void run() {  
7         int stored_item;  
8         while (true) {  
9             stored_item = buffer.fetch();  
10            //-- Consume the stored_item  
11        }  
12    }  
13 }
```

```
1 // in main :
2 Queue buff1 = new Queue(100);
3 Producer producer1 = new Producer(buff1);
4 Consumer consumer1 = new Consumer(buff1);
5 producer1.start();
6 consumer1.start();
7 producer1.join();
8 consumer1.join()
```

- در جاوا همچنین کلاس‌هایی برای دسترسی تجزیه ناپذیر¹ به داده‌ها فراهم شده است. برای مثال با استفاده از کلاس `AtomicInteger` می‌توان یک عدد صحیح تعریف کرد که ریشه‌های مختلف به آن دسترسی پیدا می‌کنند. در واقع این کلاس یک مکانیزم قفل دارد که این مکانیزم در سطح سخت افزار پیاده سازی شده است.

¹ atomic

- علاوه بر مکانیزم همگام سازی به صورت همگام شده یا synchronized ، در جاوا مکانیزم قفل نیز وجود دارد که در کلاس ReentrantLock پیاده سازی شده است توسط این قفل که در واقع همان مکانیزم ممانعت متقابل است می توان دسترسی به حافظه اشتراکی را به تنها یک ریسه محدود کرد.
- این قفل ها به صورت زیر استفاده می شوند.

```
1 Lock lock = new ReentrantLock();
2 ...
3 Lock.lock();
4 try {
5     // The code that accesses the shared data
6 } finally {
7     Lock.unlock();
8 }
```

- يکی از اهداف مهم طراحی زبان راست، افزایش قابلیت اطمینان در برنامه نویسی همروند است. توسط راست می‌توان به طور امن و کارآمد برنامه‌های همروند ایجاد کرد.
- در برنامه نویسی همروند اجزای مختلف برنامه به طور مستقل اجرا می‌شوند. معمولاً دنبال کردن این برنامه‌ها توسط برنامه نویس کار مشکلی است بدین دلیل که روند برنامه در اجرا معمولاً متفاوت از روند برنامه در هنگام دیباگ است و بنابراین پیدا کردن خطاهای برنامه نویسی و دسترسی‌های غیر مجاز در این برنامه‌ها به طور ذاتی مشکل است.
- با استفاده از قوانین مالکیت و قرض دادن بسیاری از خطاهای برنامه نویسی همروند در زمان کامپایل قابل شناسایی هستند.

- از آنجايي که ريسهها همزمان اجرا میشوند، هیچ تضميني برای ترتيب اجرای آنها وجود ندارد. بنابراین مشکلات متعددی در اين همروندي ممکن است به وجود بيايد.
- ۱. وضعیت رقابتی^۱ وقتی اتفاق میافتد که چندین ريسه به يك داده يا يك منبع به طور همزمان دسترسی پیدا میکنند، در حالی که ترتيب دسترسی آنها مشخص نیست.
 - ۲. بن بست^۲ وقتی اتفاق میافتد که چند ريسه منتظر يکديگر بمانند و بنابراین سیستم با بن بست رو برو میشود.
 - ۳. نتیجه نادرست یا دسترسی غیر مجاز وقتی اتفاق میافتد که به ازاي يك ترتيب خاص از اجرای ريسهها نتیجه مورد نظر به دست نیاید. گاهی تکرار يك سناريو با نتیجه نادرست به سادگی امکان پذير نیست.

^۱ race condition

^۲ deadlock

راست : همروندي

- برای ساختن یک ریسه از تابع `thread :: spawn` استفاده می شود. این تابع یک تابع یا یک بستار به عنوان ورودی دریافت می کند.

```
1 use std::thread;
2 use std::time::Duration;
3 fn main() {
4     thread::spawn(|| {
5         for i in 1..10 {
6             println!("hi number {} from the spawned thread!", i);
7             thread::sleep(Duration::from_millis(1));
8         }
9     });
10    for i in 1..5 {
11        println!("hi number {} from the main thread!", i);
12        thread::sleep(Duration::from_millis(1));
13    }
14 }
```

- می توانستیم در مثال قبل به جای بستار یک تابع را به `spawn` ارسال کنیم. یک بستار^۱ در زبان راست مانند یک تابع لامبدا است با این تفاوت که یک بستار از متغیرهای محیط اجرای خود نیز می تواند استفاده کند.

```
1 fn print_th() {
2     for i in 1..10 {
3         println!("hi number {} from spawned print_th", i);
4         thread::sleep(Duration::from_millis(1));
5     }
6 }
7 fn main() {
8     thread::spawn(print_th);
9 }
```

^۱ closure

راست : همروندي

- وقتی یک ریسه در یک برنامه به همراه برنامه اصلی اجرا شود، ممکن است برنامه اصلی قبل از ریسه به اتمام برسد، و بنابراین ممکن است ریسه کار خود را به اتمام نرساند.
- بدین منظور گاهی نیاز داریم یک ریسه برای یک ریسه دیگر صبر کند. این کار با متود `join()` امکان پذیر است.

```
1 fn main() {
2     let handle = thread::spawn(|| {
3         for i in 1..10 {
4             println!("hi number {} from the spawned thread!", i);
5             thread::sleep(Duration::from_millis(1));
6         }
7     });
8     for i in 1..5 {
9         println!("hi number {} from the main thread!", i);
10        thread::sleep(Duration::from_millis(1));
11    }
12    handle.join().unwrap();
13 }
```

- در مثال قبل از متود `unwrap` استفاده کردیم. با استفاده از این متود، اگر خطایی در هنگام پیوستن^۱ ریسه اتفاق بیافتد، آن خطا به کاربر نمایش داده می‌شود.

¹ joining

- در یک ریسه می‌توان متغیرهای محلی را نیز تسخیر^۱ کرد. برای مثال در برنامه زیر ریسه از وکتوری که در برنامه اصلی تعریف شده استفاده می‌کند.
-

```

1 use std::thread;
2 fn main() {
3     let v = vec![1, 2, 3];
4     let handle = thread::spawn(|| {
5         println!("Here's a vector: {:?}", v);
6     }); // compile error
7     handle.join().unwrap();
8 }
```

^۱ capture

- اما کامپایلر پیام خطا صادر می‌کند. در اینجا ریسه در واقع متغیر `v` را قرض گرفته است، اما کامپایلر نمی‌تواند اطمینان حاصل کند که قبل از اتمام اجرای ریسه متغیر `v` معتبر می‌ماند. برای مثال فرض کنید قبل از اتمام ریسه، متغیر `v` را توسط `drop` به صورت زیر تخریب کنیم.

```

1 use std::thread;
2 fn main() {
3     let v = vec![1, 2, 3];
4     let handle = thread::spawn(|| {
5         println!("Here's a vector: {:?}", v);
6     }); // compile error (what if v is dropped while executing?)
7     drop(v); // oh no!
8     handle.join().unwrap();
9 }
```

- بنابراین مالکیت متغیر v باید به ریسه انتقال پیدا کند. برای این کار از کلمه move استفاده می‌کنیم.

```
1 fn main() {  
2     let v = vec![1, 2, 3];  
3     let handle = thread::spawn(move || {  
4         println!("Here's a vector: {:?}", v);  
5     });  
6     handle.join().unwrap();  
7 }
```

راست : ارسال پیام

- یکی از راههای ارتباط بین ریسه‌ها ارسال پیام است.
- طراحان زبان راست همانند طراحان زبان گو^۱ پیشنهاد می‌کنند برنامه نویسان به جای ایجاد ارتباط توسط حافظه مشترک از مکانیزم ارسال پیام استفاده کنند.
- برای ایجاد سازوکار ارسال پیام، زبان راست کتابخانه‌ای را برای استفاده از کانال‌های^۲ ارتباطی پیاده سازی کرده است. داده‌ها بر روی کانال‌ها بین ریسه‌ها مبادله می‌شوند.
- یک کانال از دو بخش تشکیل شده است: یک فرستنده و یک گیرنده. فرستنده پیام را بر روی کانال ارسال می‌کند و گیرنده آن را دریافت می‌کند.
- وقتی فرستنده و گیرنده از بین بروند کانال بسته می‌شود.

^۱ Go programming language

^۲ channels

راست : ارسال پیام

- یک کانال توسط تابع ()`mpsc::channel` از کتابخانه استاندارد `std::sync` ساخته می‌شود. به معنی چند تولید کننده، یک مصرف کننده^۱ است. در واقع در پیاده سازی کانال در کتابخانه استاندارد، در یک کانال چندین تولید کننده می‌توانند پیام ارسال کنند و تنها یک مصرف کننده می‌تواند پیام‌ها را دریافت کند.
- تابع ()`mpsc::channel` یک دوتایی تولید می‌کند که عنصر اول آن فرستنده کانال است و عنصر دوم آن گیرنده کانال.

^۱ multiple producer, single consumer

راست : ارسال پیام

یک ریسنه می تواند به صورت زیر بر روی کانال پیام ارسال کند.

```
1 use std::sync::mpsc;
2 use std::thread;
3 fn main() {
4     let (tx, rx) = mpsc::channel();
5     thread::spawn(move || {
6         let val = String::from("hi");
7         tx.send(val).unwrap();
8     });
9 }
```

راست : ارسال پیام

- ریسه نیاز دارد مالکیت فرستنده کانال را در اختیار بگیرد تا بتواند پیام ارسال کند. متود `send` نوع `Result<T,F>` را بازمی‌گرداند و در صورتی که گیرندهٔ پیام از بین رفته باشد پیام خطا باز می‌گرداند. متود `unwrap` در صورت عدم موفقیت پیام خطا تولید می‌کند.

راست : ارسال پیام

- یک ریسه یا ریسه اصلی به صورت زیر می تواند از روی کانال پیام دریافت کند.

```
1 use std::sync::mpsc;
2 use std::thread;
3 fn main() {
4     let (tx, rx) = mpsc::channel();
5     thread::spawn(move || {
6         let val = String::from("hi");
7         tx.send(val).unwrap();
8     });
9     let received = rx.recv().unwrap();
10    println!("Got: {}", received);
11 }
```

راست : ارسال پیام

- گیرنده دو متود برای دریافت پیام دارد. متود `recv` ریسه را متوقف می‌کند و منتظر می‌ماند تا پیامی از روی کanal دریافت کند.

راست : ارسال پیام

- مقدار بازگشته متود از نوع `Result<T, F>` است. که در صورت موفقیت پیام را دریافت و در صورت عدم موفقیت (چنانچه فرستنده متوقف شده باشد) پیام خطا ارسال می‌کند.
- متود `try-recv` ریسه را متوقف نمی‌کند. در صورتی که کانال باز باشد ولی پیامی ارسال شده باشد مقدار `0k` و در غیر اینصورت مقدار `Err` را توسط نوع داده شمارشی باز می‌گرداند. سپس ریسه می‌تواند کارهای دیگر خود را انجام می‌دهد و در یک حلقه کانال را دوباره بررسی کند.

راست : ارسال پیام

- وقتی از متود `send()` استفاده می‌کنیم، مالکیت مقدار فرستاده شده منتقل می‌شود. این بدین دلیل است که اطمینان حاصل شود که مقدار فرستاده شده توسط دو ریسه به طور همزمان تغییر نمی‌کند.
- بنابراین برنامه زیر در زمان کامپایل پیام خطأ صادر می‌کند.

```
1 use std::sync::mpsc;
2 use std::thread;
3 fn main() {
4     let (tx, rx) = mpsc::channel();
5     thread::spawn(move || {
6         let val = String::from("hi");
7         tx.send(val).unwrap();
8         println!("val is {}", val); // compile error
9     });
10    let received = rx.recv().unwrap();
11    println!("Got: {}", received);
12 }
```

راست : ارسال پیام

- بر روی یک کانال می‌توان تعداد متعددی پیام به صورت زیر ارسال کرد.

```
1 use std::sync::mpsc;
2 use std::thread;
3 use std::time::Duration;
4 fn main() {
5     let (tx, rx) = mpsc::channel();
6     thread::spawn(move || {
7         let vals = vec![
8             String::from("hi"),
9             String::from("from"),
10            String::from("the"),
11            String::from("thread"),
12        ];
13    });
14 }
```

```
1   for val in vals {
2       tx.send(val).unwrap();
3       thread::sleep(Duration::from_secs(1));
4   }
5 );
6 for received in rx {
7     println!("Got: {}", received);
8 }
9 }
```

راست : ارسال پیام

در صورتی که بخواهیم بر روی یک کانال بیش از یک فرستنده داشته باشیم، باید فرستنده کانال را توسط `clone()` کپی عمیق کنیم، زیرا مالکیت فرستنده کانال در ارسال پیام باید به ریسه انتقال پیدا کند.

```
1  let (tx, rx) = mpsc::channel();  
2  
3  let tx1 = tx.clone();  
4  thread::spawn(move || {  
5      let vals = vec![  
6          String::from("hi"),  
7          String::from("there"),  
8      ];  
9      for val in vals {  
10          tx1.send(val).unwrap();  
11          thread::sleep(Duration::from_secs(1));  
12      }  
13  });
```

```
1  thread::spawn(move || {
2      let vals = vec![
3          String::from("more"),
4          String::from("messages"),
5      ];
6      for val in vals {
7          tx.send(val).unwrap();
8          thread::sleep(Duration::from_secs(1));
9      }
10 });
11 for received in rx {
12     println!("Got: {}", received);
13 }
```

راست : داده اشتراکی

- علاوه بر سازوکار ارسال پیام، چند ریسه می‌توانند توسط حافظه مشترک یا داده مشترک نیز با یکدیگر ارتباط برقرار کنند.
- در مکانیزم ارسال پیام وقتی پیام بر روی کanal ارسال شد، فرستنده مالک داده نیست. مشکل حافظه اشتراکی این است که چند ریسه می‌خواهند همزمان مالکیت حافظه را به دست بیاورند که ممکن است باعث ایجاد خطا شود. راست در زمان کامپایل اطمینان حاصل می‌کند که این اتفاق نمی‌افتد.

راست : داده اشتراکی

- میوتکس `mutex` یا قفل^۱ مخفف کلمه ممانعت متقابل^۲ است، بدین معنی که با استفاده از مکانیزم قفل تنها یک ریسه می‌تواند در یک زمان به داده دسترسی داشته باشد.
- برای دسترسی به یک حافظه اشتراکی ریسه باید ابتدا کلید قفل را به دست آورد. قفل در واقع یک ساختار داده است که ریسه‌هایی که نیاز به دسترسی به حافظه مشترک را دارند را در یک صف انتظار قرار می‌دهد. سپس به ترتیب دسترسی به ریسه‌ها داده می‌شود و هر ریسه‌ای که دسترسی را به دست آورد ورودی را قفل می‌کند و در هنگام اتمام کار قفل را آزاد می‌کند.

^۱ lock

^۲ mutual exclusion

راست : داده اشتراکی

- نوع `Mutex<T>` یک قفل ایجاد می‌کند که داده اشتراکی آن از نوع `T` است. دقت کنید که داده در درون قفل قرار دارد و می‌توان گفت که قفل‌ها در راست به صورت مانیتور پیاده سازی شده‌اند.
- برای مثال :

```
1 use std::sync::Mutex;
2 fn main() {
3     let m = Mutex::new(5);
4     {
5         let mut num = m.lock().unwrap();
6         *num = 6;
7     }
8     println!("m = {:?}", m);
9 }
```

راست : داده اشتراکی

- برای ارتباط چند ریسه توسط سازوکار قفل باید همه ریسه‌ها بتوانند مالکیت قفل را در اختیار بگیرند. برای دادن مالکیت اشتراکی از اشاره‌گرهای هوشمند استفاده می‌کنیم. نوع $\text{Arc} < \text{T} >$ تعداد دسترسی‌ها به یک متغیر را شمارش می‌کند، بنابراین می‌تواند هنگامی که هیچ دسترسی به متغیر وجود ندارد فضای حافظه آن را آزاد کند.

راست : داده اشتراکی

- بنابراین برای دسترسی چند ریسه به یک قفل به طور همزمان به صورت زیر عمل می‌کنیم.

```
1 use std::sync::{Arc, Mutex};  
2 use std::thread;  
3 fn main() {  
4     let counter = Arc::new(Mutex::new(0));  
5     let mut handles = vec![];  
6     for _ in 0..10 {  
7         let counter = Arc::clone(&counter);  
8         let handle = thread::spawn(move || {  
9             let mut num = counter.lock().unwrap();  
10            *num += 1;  
11        });  
12    }  
13 }
```

```
1     handles.push(handle);
2 }
3 for handle in handles {
4     handle.join().unwrap();
5 }
6 println!("Result: {}", *counter.lock().unwrap());
7 }
```

راست : داده اشتراکی

- دقت کنید که قفل را به صورت غیر قابل تغییر تعریف کردیم ولی داده درون آن قابل تغییر است.
- قفل‌ها ممکن است به نحوی استفاده شوند که باعث ایجاد بن بست شود. راست نمی‌تواند از این خطای احتمالی در زمان کامپایل جلوگیری کند. بن بست وقتی رخ می‌دهد که دو ریسه در انتظار یکدیگر برای آزادسازی قفل توسط طرف مقابل بمانند.