

به نام خدا

## زبان‌های برنامه‌نویسی

آرش شفیعی



## برنامه نویسی همروند

- همروندی یا همزمانی<sup>1</sup> در یک برنامه کامپیوتری به معنای توانایی واحدهای مختلف یک برنامه برای اجرای همزمان یا به عبارت دیگر اجرای موازی است به طوری که نتیجه نهایی برنامه نسبت به اجرای غیر همزمان تفاوتی نداشته باشد. با اجرای یک برنامه به صورت همروند بر روی چند پردازنده، سرعت اجرا بهبود می یابد.
- توسط برنامه نویسی همروند<sup>2</sup> که توسط آن قسمت هایی از برنامه به صورت همزمان اجرا می شوند، زمان پاسخ (تاخیر)<sup>3</sup> و توان عملیاتی<sup>4</sup> برنامه بهبود می یابد.

---

<sup>1</sup> concurrency

<sup>2</sup> concurrent programming

<sup>3</sup> response time (delay)

<sup>4</sup> 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 پردازش به پردازنده دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر با استفاده از برنامه نویسی همروندی می‌تواند پردازش شود. در این حالت می‌گوییم توان عملیاتی سیستم افزایش یافته است.

- همروندی می‌تواند در سطوح متفاوتی باشد: در سطح دستورات ماشین، در سطح دستورات زبان برنامه نویسی، در سطح زیر برنامه و یا در سطح برنامه.
- همروندی در سطح دستورات ماشین مربوط به طراحی سخت‌افزار و همروندی در سطح برنامه مربوط به طراحی سیستم عامل است. طراحان سخت افزار سازوکارهایی را برای اجرای موازی دستورات زبان ماشین فراهم می‌کنند و طراحان سیستم عامل روش‌هایی را برای اجرای موازی برنامه‌ها و زمانبندی برنامه‌ها پیاده‌سازی می‌کنند. همروندی در سطح دستورات زبان برنامه‌نویسی و زیربرنامه‌ها موضوع بحث زبان‌های برنامه نویسی است.
- در بسیاری از برنامه‌ها همروندی 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 همروند مقیاس پذیر<sup>1</sup> است زیرا اگر تعداد پردازنده‌ها افزایش یابد می‌تواند تعداد بیشتری داده را پردازش کند و با سرعت بیشتری اجرا شود.

---

<sup>1</sup> scalable

- اولین کامپیوترهایی که پردازش موازی را پشتیبانی می‌کردند، در دههٔ 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 دههٔ ۱۹۶۰ سیستم عامل‌ها، برنامه‌های مختلف را به طور همزمان بر روی چندین پردازنده اجرا و زمانبندی می‌کردند.
- در اواسط دههٔ 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 کامپیوتری 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 <sup>1</sup> گفته می شود، پردازش موازی در سطح داده <sup>2</sup> 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.

---

<sup>1</sup> Single Instructions, Multiple Data (SIMD)

<sup>2</sup> data level parallelism

- برای مثال فرض کنید می‌خواهیم عناصر یک آرایه را با یکدیگر جمع کنیم. می‌توانیم آرایه را به چند قسمت 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 قسمت 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<sup>1</sup> 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<sup>2</sup> بهره گرفته می‌شود.

---

<sup>1</sup> Multiple Instruction, Multiple Data (MIMD)

<sup>2</sup> task level parallelism

- برای مثال یک سیستم عامل یا یک مرورگر وب به طور همزمان عملیات متفاوتی را انجام می دهد که این عملیات بر روی پردازنده ها توزیع می شوند. 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 پردازنده بعدی برای اعمال عملیات دیگر انتقال می دهد و خود پردازش را با تصاویر بعدی ادامه می دهد. در این معماری حافظه می تواند مشترک باشد که در آن صورت نیاز به همگام سازی<sup>1</sup> یا هماهنگ سازی واحدهای پردازش وجود دارد تا داده ها به درستی خوانده و نوشته شوند. حافظه همچنین می تواند توزیع شده<sup>2</sup> باشد که در این صورت نیز نیاز که برقراری ارتباط بین پردازنده ها وجود دارد. به این معماری معمولاً چندپردازنده ای<sup>3</sup> نیز گفته می شود.

---

<sup>1</sup> synchronization

<sup>2</sup> distributed

<sup>3</sup> multiprocessor

- از آنجایی که معماری سخت افزار دائماً در حال به روز رسانی است کامپیوترهای جدید روش‌های همزمانی متنوعی را در سطح سخت‌افزار 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 سخت‌افزار تا حدودی می‌تواند نیازهای راندمان برنامه را برآورده کند و قسمتی از مسئولیت بهبود زمان اجرا بر عهده نرم‌افزار 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 همروندی فیزیکی<sup>1</sup> 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<sup>2</sup> بر روی پردازنده اجرا می‌شوند. به این دسته همروندی منطقی<sup>3</sup> گفته می‌شود.
- در بیشتر مواقع تعداد واحدهای همروند از تعداد پردازنده‌ها بیشتر است و بنابراین همروندی به صورت فیزیکی و منطقی اتفاق می‌افتد.

---

<sup>1</sup> physical concurrency

<sup>2</sup> interleaved

<sup>3</sup> logical concurrency

- یک ریسمان کنترلی<sup>1</sup> یا ریسۀ کنترلی به دنباله‌ای از دستورات گفته می‌شود که به طور پیوسته یکی پس از دیگری در یک واحد از برنامه اجرا می‌شوند.
- در همروندی فیزیکی هر پردازنده تنها می‌تواند یک ریسۀ کنترلی را اجرا کند، اما در همروندی منطقی بیش از یک ریسۀ کنترلی نیز می‌توانند بر روی یک پردازنده اجرا شوند.
- برنامه‌ای که در آن چند ریسۀ کنترلی به طور همزمان اجرا می‌شوند، یک برنامه چند ریسۀ<sup>2</sup> گفته می‌شود.

---

<sup>1</sup> thread of control

<sup>2</sup> multithreaded program

- همروندی در سطح دستورات معمولاً به این صورت است که دستوراتی که می‌توانند به طور موازی اجرا شوند، به صورت خودکار توسط کامپایلر تشخیص داده شده و به صورت موازی بر روی چندین پردازنده انجام شوند. برای مثال دستورات زیر را در نظر بگیرید.

---


$$\begin{array}{ll} ۱ & e = a + b \\ ۲ & f = c + d \\ ۳ & m = e * f \end{array}$$


---

- دستور سوم وابسته به دستورات اول و دوم است، اما دستورات اول و دوم از یکدیگر مستقل هستند و بنابراین می‌توانند به صورت موازی اجرا شوند. در اینصورت با تحلیل وابستگی دستورات، و اعمال موازی سازی برنامه می‌تواند با سرعت بیشتری اجرا شود.
- همروندی در سطح زیربرنامه معمولاً به این صورت است که هر زیر برنامه به یک ریشه کنترل سپرده می‌شود.

- یک واحدکار یا یک وظیفه<sup>1</sup> واحدی است از برنامه که می‌تواند به صورت همروند با واحدهای دیگر اجرا شود.
- توجه کنید که اجرای موازی<sup>2</sup> حالت خاصی از اجرای همروند<sup>3</sup> 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 بازه 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 راه‌اندازی کند.
- واحدهای کاری می‌توانند حافظه را با یکدیگر به اشتراک بگذارند و یا هر کدام حافظه مختص به خود داشته باشند.

---

<sup>1</sup> task

<sup>2</sup> parallel

<sup>3</sup> concurrent

- معمولا به واحدهای کنترلی که واحدهای کاری را بدون به اشتراک گذاری حافظه انجام می‌دهند، پردازش یا پروسه<sup>1</sup> گفته می‌شود و ریشه‌ها<sup>2</sup> مسئول پردازش واحدهای کار با به اشتراک گذاری حافظه هستند.
- ریشه‌ها نسبت به پروسه‌ها بهینه‌تر و کارآمدتر هستند، اما از طرفی چون حافظه آنها اشتراکی است نیاز به همگام‌سازی دارند.
- یک ریشه می‌تواند با ریشه‌های دیگر از طریق به اشتراک گذاری متغیرها یا از طریق کانال‌های ارتباطی و مکانیزم ارسال پیام ارتباط برقرار کند.

---

<sup>1</sup> process

<sup>2</sup> thread



- همگام‌سازی<sup>1</sup> 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<sup>2</sup> گفته می‌شود.
- معمولاً یک برنامه سیستم عامل به نام زمانبند<sup>3</sup> ریس‌ه‌ها و پروسه‌ها را برای اجرا بر روی پردازنده‌های مختلف زمانبندی می‌کند.

---

<sup>1</sup> synchronization

<sup>2</sup> race condition

<sup>3</sup> scheduler

- یک واحدکار می‌تواند در چند حالت مختلف باشد :
۱. جدید<sup>1</sup> : واحدکار به تازگی ساخته شده و هنوز آماده انجام عملیات نیست.
  ۲. آماده<sup>2</sup> : واحدکار آماده اجرا است، اما در حال اجرا نیست. زمانبند باید این واحدکار را زمانبندی کند تا به حالت اجرا درآید. همهٔ واحدهای کار که آماده به شروع عملیات هستند در یک صف واحدهای کاری آماده<sup>3</sup> قرار می‌گیرند.
  ۳. در حال اجرا<sup>4</sup> : یک پردازنده به واحدکار داده شده و می‌تواند اجرا شود.

---

<sup>1</sup> new

<sup>2</sup> ready

<sup>3</sup> task-ready queue

<sup>4</sup> running

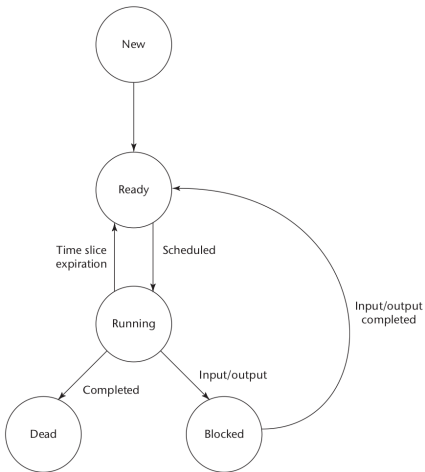
۴. مسدود<sup>۱</sup> : واحد کار مسدود شده است و اجرای آن متوقف شده است. دلیل توقف می‌تواند این باشد که واحد کار نیاز به عملیات ورودی خروجی داشته و یا اینکه زمانبند برای اجرای واحدهای کاری دیگر آن متوقف کرده است.
۵. پایان‌یافته<sup>۲</sup> : اجرای واحدکار به اتمام رسیده است. واحد کار یا با موفقیت به کار خود پایان داده است و یا به دلیل خطا یا به طور دستی اجرای آن متوقف شده و تخریب شده است.

---

<sup>۱</sup> blocked

<sup>۲</sup> dead

- در شکل زیر حالت‌های مختلف یک واحدکار نشان داده شده‌اند.



- یک ویژگی برنامه‌های همروند ویژگی زنده بودن<sup>1</sup> 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<sup>2</sup> 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.

---

<sup>1</sup> liveness

<sup>2</sup> deadlock

- دو نوع همگام‌سازی برای داده‌های مشترک وجود دارد : همگام‌سازی مشارکتی<sup>1</sup> و همگام‌سازی رقابتی<sup>2</sup>.
- 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.

---

<sup>1</sup> cooperation synchronization

<sup>2</sup> competition synchronizarion

- در ادامه به سه روش مختلف برای اجرای همزمان واحدهای کار به صورت همروند ارائه می‌شود :

سمافور<sup>1</sup> ، مانیتور<sup>2</sup> و ارسال پیام<sup>3</sup> .

---

<sup>1</sup> semaphore

<sup>2</sup> monitor

<sup>3</sup> 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<sup>1</sup> در سال ۱۹۶۵ سمافور را به عنوان راه‌حلی برای همگام‌سازی داده‌های مشترک بین واحدهای پردازش طراحی کرد.

---

<sup>1</sup> Edsger Dijkstra



- برای حفاظت از منابع مشترک باید یک محافظ<sup>1</sup> 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<sup>2</sup> نامیده می‌شود، اطلاعات مربوط به اجرای وظایف (واحد‌های کار) را در خود نگه می‌دارد.

---

<sup>1</sup> guard

<sup>2</sup> 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 صف نگهداری شده و سپس دسترسی به آنها به طور ترتیبی داده می شود.
- دو نوع عملیات مهم بر روی سمافور عملیات انتظار<sup>1</sup> و آزادسازی<sup>2</sup> نامیده می شوند.

---

<sup>1</sup> wait

<sup>2</sup> release

- از شمارنده سمافور می‌توان به عنوان شمارنده برای تعداد سلول‌های یک آرایه و یا شمردن تعداد وظیفه‌هایی که می‌توانند به طور همزمان به یک منبع دسترسی داشته باشند استفاده کرد.
- یک سمافور دودویی<sup>1</sup> سمافوری است که شمارنده آن تنها بتواند صفر و یا یک باشد.

---

<sup>1</sup> binary semaphore

- توابع انتظار و آزادسازی سمافور می‌توان به صورت زیر پیاده سازی کرد.

```
۱ wait(semaphore) :  
۲     if semaphore.counter > 0 :  
۳         semaphore.counter --  
۴     else  
۵         # put the caller in semaphore's queue  
۶         # attempt to transfer control to some ready task  
۷         # (if the task-ready queue is empty, deadlock occurs)  
۸  
۹ release(semaphore) :  
۱۰     if semaphore.queue.empty() :  
۱۱         # no task is waiting  
۱۲         semaphore.counter ++  
۱۳     else  
۱۴         # put the calling task in the task-ready queue  
۱۵         # transfer control to a task from semaphore's queue
```

- 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 پیاده‌سازی شده است.

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

---

```

۱ sem = Semaphore(1)
۲ fullspots = Semaphore(0)
۳ emptyspots = Semaphore(BUFLEN)
۴
۵ def producer() :
۶     while True :
۷         # -- produce VALUE --
۸         wait(emptyspots)
۹         wait(sem)
۱۰        # DEPOSIT(VALUE)
۱۱        release(sem)
۱۲        release(fullspots)

```

---

---

```
۱ def consumer() :  
۲     while True :  
۳         wait(fullspots)  
۴         wait(sem)  
۵         # FETCH(VALUE)  
۶         release(sem)  
۷         release(emptyspots)  
۸         # -- consume VALUE --
```

---

- یک روش دیگر برای همگام‌سازی داده‌های مشترک استفاده از مانیتور<sup>1</sup> 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 یک مانیتور یک بافر قرار بگیرد همچنان وظیفه برنامه نویس است که اطمینان حاصل کند در هنگام خالی بودن به خواننده و در هنگام پر بودن بافر به نویسنده دسترسی داده نشود، وگرنه برنامه با بن‌بست مواجه می‌شود.
- 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.

---

<sup>1</sup> monitor



- گاهی لازم است دو واحدکاری با یکدیگر ارتباط برقرار کنند. در چنین مواقعی فعالیت ها به پیام‌های دریافتی بستگی پیدا می‌کند.
- برای مثال فرض کنید یک واحد کاری در حال انجام محاسبات است و در همان هنگام واحدکاری دوم نیاز دارد که واحدکاری اول محاسباتی را انجام دهد. واحد کاری 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.

- یکی 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 آنجایی که ریشه‌ها همزمان اجرا می‌شوند، هیچ تضمینی برای ترتیب اجرای آن‌ها وجود ندارد. بنابراین مشکلات متعددی در این همروندی ممکن است به وجود بیاید.
۱. وضعیت رقابتی<sup>1</sup> وقتی اتفاق می‌افتد که چندین ریشه به یک داده یا یک منبع به طور همزمان دسترسی پیدا می‌کنند، در حالی که ترتیب دسترسی آنها مشخص نیست.
۲. بن بست<sup>2</sup> وقتی اتفاق می‌افتد که چند ریشه منتظر یکدیگر بمانند و بنابراین سیستم با بن بست روبرو می‌شود.
۳. نتیجه نادرست یا دسترسی غیر مجاز وقتی اتفاق می‌افتد که به ازای یک ترتیب خاص از اجرای ریشه‌ها نتیجه مورد نظر به دست نیاید. گاهی تکرار یک سناریو با نتیجه نادرست به سادگی امکان پذیر نیست.

---

<sup>1</sup> race condition

<sup>2</sup> deadlock

## راست : همروندی

- برای ساختن یک ریشه از تابع `spawn :: thread` استفاده می‌شود. این تابع یک تابع یا یک بستار به عنوان ورودی دریافت می‌کند.

---

```
۱ use std::thread;
۲ use std::time::Duration;
۳ fn main() {
۴     thread::spawn(|| {
۵         for i in 1..10 {
۶             println!("hi number {} from the spawned thread!", i);
۷             thread::sleep(Duration::from_millis(1));
۸         }
۹     });
۱۰ for i in 1..5 {
۱۱     println!("hi number {} from the main thread!", i);
۱۲     thread::sleep(Duration::from_millis(1));
۱۳ }
۱۴ }
```

---

- می‌توانستیم در مثال قبل به جای بستار یک تابع را به spawn ارسال کنیم. یک بستار<sup>1</sup> در زبان راست مانند یک تابع لامبدا است با این تفاوت که یک بستار از متغیرهای محیط اجرای خود نیز می‌تواند 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 }
```

---

<sup>1</sup> closure

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

---

```

۱ fn main() {
۲     let handle = thread::spawn(|| {
۳         for i in 1..10 {
۴             println!("hi number {} from the spawned thread!", i);
۵             thread::sleep(Duration::from_millis(1));
۶         }
۷     });
۸     for i in 1..5 {
۹         println!("hi number {} from the main thread!", i);
۱۰        thread::sleep(Duration::from_millis(1));
۱۱    }
۱۲    handle.join().unwrap();
۱۳ }

```

---

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



- در یک ریسه می‌توان متغیرهای محلی را نیز تسخیر<sup>1</sup> کرد. برای مثال در برنامه زیر ریسه از وکتوری که در برنامه اصلی تعریف شده استفاده می‌کند.

```
۱ use std::thread;
۲ fn main() {
۳     let v = vec![1, 2, 3];
۴     let handle = thread::spawn(|| {
۵         println!("Here's a vector: {:?}", v);
۶     }); // compile error
۷     handle.join().unwrap();
۸ }
```

---

<sup>1</sup> 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.

---

```
۱ use std::thread;
۲ fn main() {
۳     let v = vec![1, 2, 3];
۴     let handle = thread::spawn(|| {
۵         println!("Here's a vector: {:?}", v);
۶     }); // compile error (what if v is dropped while executing?)
۷     drop(v); // oh no!
۸     handle.join().unwrap();
۹ }
```

---

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

---

```
۱ fn main() {  
۲     let v = vec![1, 2, 3];  
۳     let handle = thread::spawn(move || {  
۴         println!("Here's a vector: {:?}", v);  
۵     });  
۶     handle.join().unwrap();  
۷ }
```

---

## راست : ارسال پیام

- یکی از راه‌های ارتباط بین ریشه‌ها ارسال پیام است.
- طراحان زبان راست همانند طراحان زبان گو<sup>1</sup> پیشنهاد می‌کنند برنامه نویسان به جای ایجاد ارتباط توسط حافظهٔ مشترک از مکانیزم ارسال پیام استفاده کنند.
- برای ایجاد سازوکار ارسال پیام، زبان راست کتابخانه‌ای را برای استفاده از کانال‌های<sup>2</sup> ارتباطی پیاده سازی کرده است. داده‌ها بر روی کانال‌ها بین ریشه‌ها 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.

---

<sup>1</sup> Go programming language

<sup>2</sup> channels

- یک کانال توسط تابع `mpsc::channel()` از کتابخانه استاندارد `std::sync` ساخته می‌شود. `mpsc` به معنی چند تولید کننده، یک مصرف کننده<sup>1</sup> 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.

---

<sup>1</sup> multiple producer, single consumer

- یک ریسه می‌تواند به صورت زیر بر روی کانال پیام ارسال کند.

---

```
۱ use std::sync::mpsc;
۲ use std::thread;
۳ fn main() {
۴     let (tx, rx) = mpsc::channel();
۵     thread::spawn(move || {
۶         let val = String::from("hi");
۷         tx.send(val).unwrap();
۸     });
۹ }
```

---

## راست : ارسال پیام

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

- یک ریسه یا ریسۀ اصلی به صورت زیر می‌تواند از روی کانال پیام دریافت کند.

---

```
۱ use std::sync::mpsc;
۲ use std::thread;
۳ fn main() {
۴     let (tx, rx) = mpsc::channel();
۵     thread::spawn(move || {
۶         let val = String::from("hi");
۷         tx.send(val).unwrap();
۸     });
۹     let received = rx.recv().unwrap();
۱۰    println!("Got: {}", received);
۱۱ }
```

---



## راست : ارسال پیام

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

## راست : ارسال پیام

- مقدار بازگشتی 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 `Ok` 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 بدسن دلیل است که اطمینان حاصل می‌شود که مقدار فرستاده شده توسط دو ریشه به طور همزمان تغییر نمی‌کند.
- بنابراین برنامه زیر در زمان کامپایل پیام خطا صادر می‌کند.

```
۱ use std::sync::mpsc;
۲ use std::thread;
۳ fn main() {
۴     let (tx, rx) = mpsc::channel();
۵     thread::spawn(move || {
۶         let val = String::from("hi");
۷         tx.send(val).unwrap();
۸         println!("val is {}", val); // compile error
۹     });
۱۰ let received = rx.recv().unwrap();
۱۱ println!("Got: {}", received);
۱۲ }
```

## راست : ارسال پیام

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

```
۱ use std::sync::mpsc;
۲ use std::thread;
۳ use std::time::Duration;
۴ fn main() {
۵     let (tx, rx) = mpsc::channel();
۶     thread::spawn(move || {
۷         let vals = vec![
۸             String::from("hi"),
۹             String::from("from"),
۱۰            String::from("the"),
۱۱            String::from("thread"),
۱۲        ];
```

```
۱         for val in vals {  
۲             tx.send(val).unwrap();  
۳             thread::sleep(Duration::from_secs(1));  
۴         }  
۵     });  
۶     for received in rx {  
۷         println!("Got: {}", received);  
۸     }  
۹ }
```

## 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 ریشه 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 });
```

```
۱  thread::spawn(move || {  
۲      let vals = vec![  
۳          String::from("more"),  
۴          String::from("messages"),  
۵      ];  
۶      for val in vals {  
۷          tx.send(val).unwrap();  
۸          thread::sleep(Duration::from_secs(1));  
۹      }  
۱۰ });  
۱۱ for received in rx {  
۱۲     println!("Got: {}", received);  
۱۳ }
```

## راست : داده اشتراکی

- علاوه بر سازوکار ارسال پیام، چند ریشه می‌توانند توسط حافظه مشترک یا داده مشترک نیز با یکدیگر ارتباط برقرار کنند.
- در مکانیزم ارسال پیام وقتی پیام بر روی کانال ارسال شد، فرستنده مالک داده نیست مشکل حافظه اشتراکی این است که چند ریشه 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.



- میوتکس mutex یا قفل<sup>1</sup> مخفف کلمه ممانعت متقابل<sup>2</sup> 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 حافظهٔ 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.

---

<sup>1</sup> lock

<sup>2</sup> 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 :

```
۱ use std::sync::Mutex;
۲ fn main() {
۳     let m = Mutex::new(5);
۴     {
۵         let mut num = m.lock().unwrap();
۶         *num = 6;
۷     }
۸     println!("m = {:?}", m);
۹ }
```

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

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

---

```
۱ use std::sync::{Arc, Mutex};
۲ use std::thread;
۳ fn main() {
۴     let counter = Arc::new(Mutex::new(0));
۵     let mut handles = vec![];
۶     for _ in 0..10 {
۷         let counter = Arc::clone(&counter);
۸         let handle = thread::spawn(move || {
۹             let mut num = counter.lock().unwrap();
۱۰             *num += 1;
۱۱         });
```

---

---

```
۱      handles.push(handle);
۲  }
۳  for handle in handles {
۴      handle.join().unwrap();
۵  }
۶  println!("Result: {}", *counter.lock().unwrap());
۷ }
```

---

- 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 ریشه در انتظار یکدیگر برای آزادسازی قفل توسط طرف مقابل بمانند.

- 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 کلمهٔ synchronized تعریف شود. متودی که به صورت همگام شده <sup>1</sup> 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.

---

<sup>1</sup> synchronized

- کلاسی که همهٔ 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.

---

```
۱ synchronized (expression) {  
۲     statements  
۳ }
```

---

- 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 ریسه‌هایی که نیاز به آن شیء دارند را در خود نگه می‌دارد.



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

## جاوا : همروندی

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

---

```
۱ // Queue
۲ // This class implements a circular queue for storing int
۳ // values. It includes a constructor for allocating and
۴ // initializing the queue to a specified size. It has
۵ // synchronized methods for inserting values into and
۶ // removing values from the queue.
۷ class Queue {
۸     private int [] que;
۹     private int nextIn, nextOut, filled, queSize;
۱۰    public Queue(int size) {
۱۱        que = new int [size];
۱۲        filled = 0;
۱۳        nextIn = 1;
۱۴        nextOut = 1;
```

---

```
۱      queSize = size;
۲  } /** end of Queue constructor
۳  public synchronized void deposit (int item)
۴      throws InterruptedException {
۵      try {
۶          while (filled == queSize)
۷              wait();
۸          que [nextIn] = item;
۹          nextIn = (nextIn % queSize) + 1;
۱۰         filled++;
۱۱         notifyAll();
۱۲     } /** end of try clause
۱۳     catch(InterruptedException e) {}
۱۴ } /** end of deposit method
```

```
۱    public synchronized int fetch()  
۲        throws InterruptedException {  
۳        int item = 0;  
۴        try {  
۵            while (filled == 0)  
۶                wait();  
۷            item = que [nextOut];  
۸            nextOut = (nextOut % queSize) + 1;  
۹            filled--;  
۱۰           notifyAll();  
۱۱        } /** end of try clause  
۱۲        catch(InterruptedException e) {}  
۱۳        return item;  
۱۴    } /** end of fetch method  
۱۵ } /** end of Queue class
```

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

```
۱ class Producer extends Thread {  
۲     private Queue buffer;  
۳     public Producer(Queue que) {  
۴         buffer = que;  
۵     }  
۶     public void run() {  
۷         int new_item;  
۸         while (true) {  
۹             //-- Create a new_item  
۱۰            buffer.deposit(new_item);  
۱۱        }  
۱۲    }  
۱۳ }
```

```
۱ class Consumer extends Thread {  
۲     private Queue buffer;  
۳     public Consumer(Queue que) {  
۴         buffer = que;  
۵     }  
۶     public void run() {  
۷         int stored_item;  
۸         while (true) {  
۹             stored_item = buffer.fetch();  
۱۰             //-- Consume the stored_item  
۱۱         }  
۱۲     }  
۱۳ }
```

---

```
۱ // in main :  
۲ Queue buff1 = new Queue(100);  
۳ Producer producer1 = new Producer(buff1);  
۴ Consumer consumer1 = new Consumer(buff1);  
۵ producer1.start();  
۶ consumer1.start();  
۷ producer1.join();  
۸ consumer1.join();
```

---

- در جاوا همچنین کلاس‌هایی برای دسترسی تجزیه ناپذیر<sup>1</sup> 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.

---

<sup>1</sup> atomic



- علاوه بر مکانیزم همگام‌سازی به صورت همگام‌شده 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 می‌توان دسترسی به حافظه اشتراکی را به تنها یک ریسسه محدود کرد.
- این قفل‌ها به صورت زیر استفاده می‌شوند.

```
۱ Lock lock = new ReentrantLock();  
۲ . . .  
۳ Lock.lock();  
۴ try {  
۵     // The code that accesses the shared data  
۶ } finally {  
۷     Lock.unlock();  
۸ }
```