به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

برنامه نویسی همروند

همروندی

- همروندی یا همزمانی [1] در یک برنامهٔ 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 می‌یابد.
- توسط برنامه نویسی همروند [2] که توسط آن قسمت‌هایی از برنامه به صورت همزمان اجرا می‌شوند، زمان پاسخ (تأخیر) [3] و توان عملیاتی [4] برنامه بهبود می‌یابد.

[1] concurrency
[2] concurrent programming
[3] response time (delay)
[4] throughput

همروندی

- 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 مرحلهٔ 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 پردازندهٔ دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر با استفاده از برنامه نویسی همروند می‌تواند پردازش شود. در این حالت می‌گوییم توان عملیاتی سیستم افزایش یافته است.

همروندی

- 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 سیستم‌عامل 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 [1] است زیرا اگر تعداد پردازنده‌ها افزایش یابد می‌تواند تعداد بیشتری داده را پردازش کند و با سرعت بیشتری اجرا شود.

[1] scalable

همروندی

- اولین کامپیوترهایی که پردازش موازی را پشتیبانی می‌کردند، در دهۀ 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 دهۀ ۱۹۶۰ سیستم عامل‌ها، برنامه‌های مختلف را به طور همزمان بر روی چندین پردازنده اجرا و زمانبندی می‌کردند.
- در اواسط دهۀ 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.

همروندی

- در آن زمان معماری سخت‌افزارهای 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 قسمت‌های مختلف داده را پردازش می‌کنند. بنابراین در این معماری که به آن معماری یک عملیات–چند داده [1] گفته می‌شود، پردازش موازی در سطح داده [2] صورت می‌گیرد. به عبارت دیگر در این معماری داده به چند قسمت تقسیم شده و یک عملیات واحد بر روی قسمت‌های مختلف داده اعمال می‌شود. پردازنده‌های گرافیکی غالباً از این نوع معماری هستند.

[1] Single Instruction, Multiple Data (SIMD)
[2] data level parallelism

همروندی

- 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.

همروندی

- 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 [1] گفته می‌شود. به عبارت دیگر چندین عملیات در یک برنامه می‌توانند به طور همزمان اجرا شوند و هر یک از عملیات بر روی یک پردازندهٔ متفاوت اجرا می‌شود. در این معماری از موازی‌سازی در سطح عملیات [2] بهره گرفته می‌شود.

[1] Multiple Instruction, Multiple Data (MIMD)
[2] task level parallelism

همروندی

- برای مثال یک سیستم‌عامل 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 می تواند مشترک باشد که در آن صورت نیاز به همگام‌سازی [1] یا هماهنگ سازی واحدهای پردازش وجود دارد تا داده‌ها به درستی خوانده و نوشته شوند. حافظه همچنین می‌تواند توزیع شده [2] باشد که در این صورت نیز نیاز که برقراری ارتباط بین پردازنده‌ها وجود دارد. به این معماری معمولاً چندپردازنده‌ای [3] نیز گفته می‌شود.

[1] synchronization
[2] distributed
[3] multiprocessor

همروندی

- از آنجایی که معماری سخت افزار دائماً در حال به‌روز 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.

همروندی

- دو دسته از واحدهای همروند وجود دارند. در دستهٔ 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 [1] گفته می‌شود. در دستهٔ دوم یک پردازنده وجود دارد و واحدهای مختلف برنامه به طور همزمان به طور قطعه قطعه شده و در هم آمیخته [2] بر روی پردازنده اجرا می‌شوند. به این دسته همروندی منطقی [3] گفته می‌شود.
- در بیشتر مواقع تعداد واحدهای همروند از تعداد پردازنده‌ها بیشتر است و بنابراین همروندی به صورت فیزیکی و منطقی اتفاق می‌افتد.

[1] physical concurrency
[2] interleaved
[3] logical concurrency

همروندی

- یک ریسمان کنترلی [1] یا ریسهٔ کنترلی به دنباله‌ای از دستورات گفته می‌شود که به طور پیوسته یکی پس از دیگری در یک واحد از برنامه اجرا می‌شوند و می‌تواند به طور مجزا توسط سیستم‌عامل زمانبندی شود.
- در همروندی فیزیکی هر پردازنده تنها می‌تواند یک ریسهٔ کنترلی را اجرا کند، اما در همروندی منطقی بیش از یک ریسهٔ کنترلی نیز می‌توانند بر روی یک پردازنده اجرا شوند.
- برنامه‌ای که در آن چند ریسهٔ کنترلی به طور همزمان اجرا می‌شوند، یک برنامه چند ریسه‌ای [2] گفته می‌شود.

[1] thread of control
[2] multithreaded program

همروندی

- همروندی در سطح دستورات معمولا 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.

```
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 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 ریسه کنترل سپرده می‌شود.

همروندی

- یک واحدکار یا یک وظیفه [1] واحدی است از برنامه که می‌تواند به صورت همروند با واحدهای دیگر اجرا شود.
- توجه کنید که اجرای موازی [2] حالت خاصی از اجرای همروند [3] 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 راه‌اندازی 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

همروندی

- 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 پروسه [1] گفته می‌شود.
- ریسه‌ها [2] مسئول پردازش واحدهای‌کار با به اشتراک گذاری حافظه هستند.
- ریسه‌ها نسبت به پروسه‌ها بهینه‌تر 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

همروندی

- همگام‌سازی [1] 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 [2] گفته می‌شود.
- معمولاً یک برنامهٔ سیستم عامل به نام زمانبند [3] ریسه‌ها و پروسه‌ها را برای اجرا بر روی پردازنده‌های مختلف زمانبندی می‌کند.

[1] synchronization
[2] race condition
[3] scheduler

همروندی

- یک واحدکار می‌تواند در چند حالت مختلف باشد :

۱. جدید [1] : واحدکار به تازگی ساخته شده و هنوز آماده انجام عملیات نیست.

۲. آماده [2] : 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 [3] قرار می‌گیرند.

۳. در حال اجرا [4] : یک پردازنده به واحد کار داده شده و می‌تواند اجرا شود.

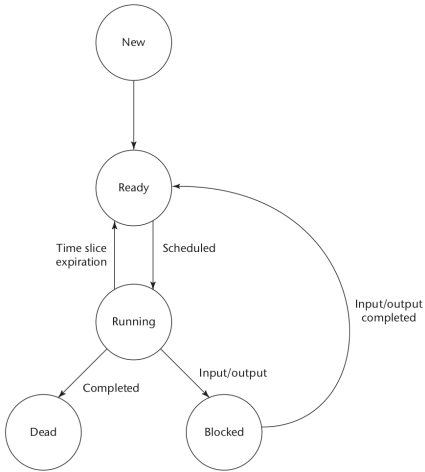
[1] new
[2] ready
[3] task-ready queue
[4] running

- ۴. مسدود [1] : 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 [2] : 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.

[1] blocked
[2] dead

همروندی

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



همروندی

- یک ویژگی برنامه‌های همروند ویژگی زنده بودن [1] 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 [2] 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.

[1] liveness
[2] deadlock

همروندی

- دو نوع همگام‌سازی برای داده‌های مشترک وجود دارد : همگام‌سازی مشارکتی [1] و همگام‌سازی رقابتی [2].
- 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 synchronizarion

همروندی

- در ادامه به سه روش مختلف برای اجرای همزمان واحدهای کار به صورت همروند ارائه می‌شود : سمافور [1] ، مانیتور [2] و ارسال پیام [3] .

[1] semaphore
[2] monitor
[3] message passing

سمافور

- 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 [1] در سال ۱۹۶۵ سمافور را به عنوان راه‌حلی برای همگام‌سازی داده‌های مشترک بین واحدهای پردازش طراحی کرد.

[1] Edsger Dijkstra

سمافور

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

[1] guard
[2] task descriptor queue

سمافور

- 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 می‌شود.
- دو نوع عملیات مهم بر روی سمافور عملیات انتظار [1] و آزادسازی [2] نامیده می‌شوند.

[1] wait
[2] release

سمافور

- از شمارنده سمافور می‌توان به عنوان شمارنده برای شمردن تعداد واحدهای کاری که می‌توانند به طور همزمان به یک منبع دسترسی داشته باشند استفاده کرد.
- یک سمافور دودویی [1] سمافوری است که شمارندهٔ آن تنها بتواند صفر و یا یک باشد.

[1] binary semaphore

سمافور

- 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          # enqueue the task (caller) in semaphore's queue and block it
۶          # control is transferred 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         # dequeue a task T from semaphore's queue
۱۵         # control is transferred to the task T
```

سمافور

- 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.

سمافور

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

```
sem = Semaphore(1)
fullspots = Semaphore(0)
emptyspots = Semaphore(BUFLEN)

def producer() :
    while True :
        # -- produce VALUE --
        wait(emptyspots)
        wait(sem)
        # DEPOSIT(VALUE)
        release(sem)
        release(fullspots)
```

سمافور

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

مانیتور

- یک روش دیگر برای همگام‌سازی داده‌های مشترک استفاده از مانیتور [1] 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 همهٔ 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 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.

سی++ : همروندی

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

```
۱  #include <thread>
۲  #include <iostream>
۳  using namespace std;
۴  void hello() {
۵      cout << "Hello from thread!\n";
۶  }
۷  int main() {
۸      thread t(hello); // Create thread
۹      t.join(); // Wait for completion
۱۰ }
```

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

سی++ : همروندی

- می‌توان به یک ریسه مقدار نیز ارسال کرد.

```
۱ void print_num(int n) {
۲     cout << n << "\n";
۳ }
۴ int main() {
۵     int x = 42;
۶     thread t(print_num, x);
۷     t.join();
۸ }
```

سی++ : همروندی

- همچنین می‌توان به یک ریسه یک تابع لامبدا ارسال کرد.

```
۱ int main() {
۲     int x = 10;
۳     thread t([x](){ cout << x*x << "\n"; });
۴     t.join();
۵ }
```

سی++ : همروندی

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

```
1  x = x + 1;
```

- فرض کنید مقدار اولیه x برابر با ۱ است. ریسهٔ اول مقدار x را می‌خواند و یک واحد به آن می‌افزاید، اما قبل از به روز رسانی مقدار x کنترل به ریسهٔ دوم داده می‌شود و ریسه دوم همه مقدار x را که همچنان ۱ است می‌خواند و یک واحد به آن می‌افزاید. حال کنترل به ریسه اول بازگردانده می‌شود و مقدار x را به روز رسانی می‌کند و مقدار آن برابر با ۲ می‌شود. سپس کنترل به ریسهٔ دوم می‌رسد و مقدار را به روز رسانی می‌کند که مقدار برابر با ۲ می‌شود. انتظار داشتیم هر کدام از ریسه‌ها یک واحد به مقدار x بیافزایند و در نهایت مقدار برابر با ۳ شود، ولی این اتفاق نیافتاد.

سی++ : همروندی

- این مشکل توسط mutex حل می‌شود.

```
۱  #include <iostream>
۲  #include <thread>
۳  #include <mutex>
۴  using namespace std;
۵  mutex mtx;        // mutex for critical section
۶  int counter = 0;       // shared resource
۷
۸  void increment(int id) {
۹      for (int i = 0; i < 100; i++) {
۱۰         mtx.lock();          // manually lock
۱۱         counter = counter + 1;          // critical section
۱۲         mtx.unlock();        // manually unlock
۱۳     }
۱۴ }
```

```
۱۵ int main() {
۱۶     thread t1(increment, 1);
۱۷     thread t2(increment, 2);
۱۸
۱۹     t1.join();
۲۰     t2.join();
۲۱
۲۲     cout << "Final counter value: " << counter << endl;
۲۳     return 0;
۲۴ }
```

سی++ : همروندی

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

```
۱  #include <iostream>
۲  #include <thread>
۳  #include <queue>
۴  #include <mutex>
۵  #include <condition_variable>
۶  std::queue<int> q;
۷  std::mutex mtx;
۸  std::condition_variable cv;
۹  void producer() {
۱۰     mtx.lock();
۱۱     q.push(42);  // produce one value
۱۲     std::cout << "Produced: 42" << std::endl;
۱۳     mtx.unlock();
۱۴     cv.notify_one(); // notify consumer
۱۵ }
```

سی++ : همروندی

```
۱۶ void consumer() {
۱۷     mtx.lock();
۱۸     if (q.empty()) {
۱۹         cv.wait(mtx); // wait for producer
۲۰     }
۲۱
۲۲     int value = q.front();
۲۳     q.pop();
۲۴     mtx.unlock();
۲۵
۲۶     cout << "Consumed: " << value << endl;
۲۷ }
```

سی++ : همروندی

```
۲۸ int main() {
۲۹     thread t1(producer);
۳۰     thread t2(consumer);
۳۱
۳۲     t1.join();
۳۳     t2.join();
۳۴ }
```

راست : همروندی

- یکی از اهداف مهم طراحی زبان راست، افزایش قابلیت اطمینان در برنامه نویسی همروند است. توسط راست می‌توان به طور امن و کارامد برنامه‌های همروند ایجاد کرد.
- در برنامه نویسی همروند اجزای مختلف برنامه به طور مستقل اجرا می‌شوند. معمولا دنبال کردن این برنامه‌ها توسط برنامه نویس کار مشکلی است بدین دلیل که روند برنامه در اجرا معمولا 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.

راست : همروندی

- از آنجایی که ریسه‌ها 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 [1] وقتی اتفاق می‌افتد که چندین ریسه به یک داده یا یک منبع به طور همزمان دسترسی پیدا می‌کنند، در حالی که ترتیب دسترسی آنها مشخص نیست.

۲. بن بست [2] وقتی اتفاق می‌افتد که چند ریسه منتظر یکدیگر بمانند و بنابراین سیستم با بن‌بست روبرو می‌شود.

۳. نتیجه نادرست یا دسترسی غیر مجاز وقتی اتفاق می‌افتد که به ازای یک ترتیب خاص از اجرای ریسه‌ها نتیجه مورد نظر به دست نیاید. گاهی تکرار یک سناریو با نتیجهٔ نادرست به سادگی امکان پذیر نیست.

[1] race condition
[2] deadlock

راست : همروندی

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

```
use std::thread;
use std::time::Duration;
fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

راست : همروندی

- می‌توانستیم در مثال قبل به جای بستار یک تابع را به spawn ارسال کنیم. یک بستار [1] در زبان راست مانند یک تابع لامبدا است با این تفاوت که یک بستار از متغیرهای محیط اجرای خود نیز می‌تواند استفاده کند.

```
fn print_th() {
    for i in 1..10 {
        println!("hi number {} fron spawned print_th", i);
        thread::sleep(Duration::from_millis(1));
    }
}
fn main() {
    thread::spawn(print_th);
}
```

[1] closure

راست : همروندی

- 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.

راست : همروندی

```rust
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```

راست : همروندی

- در مثال قبل از متود `unwrap` استفاده کردیم. با استفاده از این متود، اگر خطایی در هنگام پیوستن [1] ریسه اتفاق بیافتد، آن خطا به کاربر نمایش داده می‌شود.

[1] joining

راست : همروندی

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

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

[1] capture

راست : همروندی

- 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 }
```

راست : همروندی

- بنابراین مالکیت متغیر v باید به ریسه انتقال پیدا کند. برای این کار از کلمهٔ 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 ریسه‌ها ارسال پیام است.
- طراحان زبان راست همانند طراحان زبان گو [1] 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 [2] 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.

[1] Go programming language
[2] channels

راست : ارسال پیام

- یک کانال توسط تابع `mpsc::channel()` از کتابخانهٔ استاندارد `std::sync` ساخته می‌شود. `mpsc` به معنی چند تولید کننده، یک مصرف کننده [1] 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 فرستندهٔ کانال است و عنصر دوم آن گیرندهٔ کانال.

[1] multiple producer, single consumer

راست : ارسال پیام

- 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

- ریسه نیاز دارد مالکیت فرستندهٔ 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 ریسهٔ 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` ودر 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 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         ];
```

راست : ارسال پیام

```
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
    for received in rx {
        println!("Got: {}", received);
    }
}
```

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.

```
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("there"),
    ];
    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
```

راست : ارسال پیام

```
    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
        ];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
    for received in rx {
        println!("Got: {}", received);
    }
```

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 کانال 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 [1] مخفف کلمه ممانعت متقابل [2] 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.

[1] lock
[2] mutual exclusion

راست : داده اشتراکی

- نوع Mutex<T> یک قفل ایجاد می‌کند که دادهٔ 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 :

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

راست : داده اشتراکی

- برای ارتباط چند ریسه توسط سازوکار قفل باید همهٔ ریسه‌ها بتوانند مالکیت قفل را در اختیار بگیرند. برای دادن مالکیت اشتراکی از اشاره‌گرهای هوشمند استفاده می‌کنیم. نوع `Arc<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.

```
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         });
```

راست : داده اشتراکی

```
            handles.push(handle);
        }
        for handle in handles {
            handle.join().unwrap();
        }
        println!("Result: {}", *counter.lock().unwrap());
}
```

راست : داده اشتراکی

- دقت کنید که قفل را به صورت غیر قابل تغییر تعریف کردیم ولی دادهٔ 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.

جاوا : همروندی

- 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 [1] تعریف شده است باید عملیات خود را به اتمام برساند تا بتواند دوباره توسط یک ریسه فراخوانی شود.
- پیاده سازی این متودها به این‌صورت 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

جاوا : همروندی

- 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.

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

جاوا : همروندی

- برای ارتباط بین ریسه‌ها، همهٔ اشیای جاوا که از یک کلاس جد به نام کلاس `Object` به ارث می‌برند، متودهایی به نام `wait` ، `notify` و `notifyAll` دارند.
- متود انتظار (`wait`) موجب می‌شود یک ریسه در صف انتظار یک شی وارد شود و منتظر می‌ماند تا وقتی که یک ریسهٔ 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 : همروندی

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

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.
class Queue {
    private int [] que;
    private int nextIn, nextOut, filled, queSize;
    public Queue(int size) {
        que = new int [size];
        filled = 0;
        nextIn = 1;
        nextOut = 1;
```

جاوا : همروندی

```
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
```

جاوا : همروندی

```
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
```

جاوا : همروندی

- حال تولیدکننده و مصرف‌کنندهٔ صف به صورت دو ریسه 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 }
```

جاوا : همروندی

```
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 }
```

جاوا : همروندی

```
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()
```

جاوا : همروندی

- در جاوا همچنین کلاس‌هایی برای دسترسی تجزیه ناپذیر [1] به داده‌ها فراهم شده است. برای مثال با استفاده از کلاس `AtomicInteger` می‌توان یک عدد صحیح تعریف کرد که ریسه‌های 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

جاوا : همروندی

- 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 ریسه 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 }
```