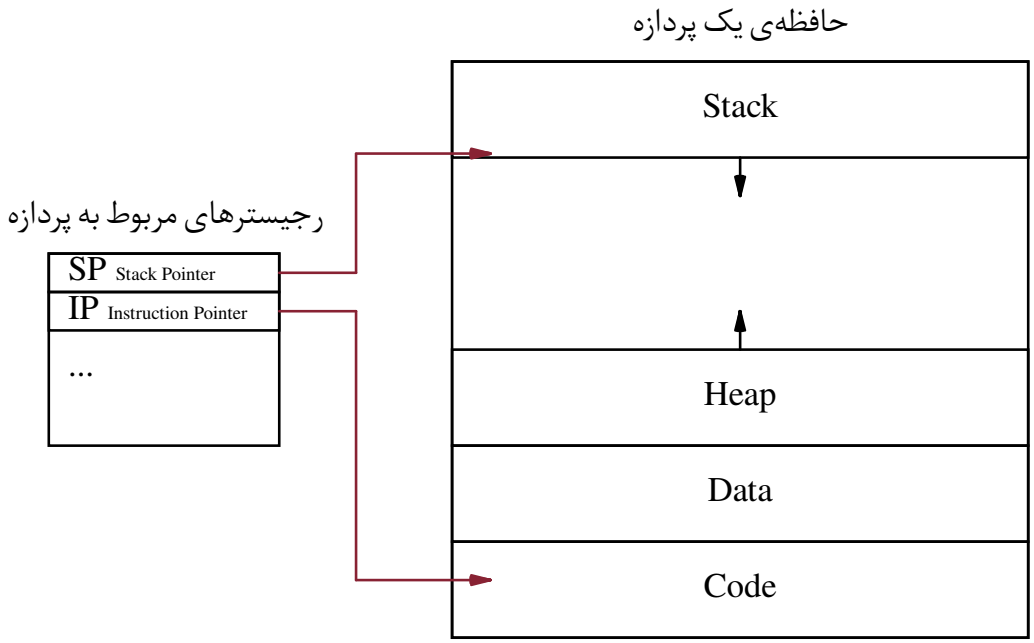
یادداشت‌های درس سیستم‌های عامل - بخش چهارم

در معرفی پردازه‌ها فرض کرده‌ایم در هر لحظه فقط یک قسمت از آن در حال اجرا هست. در این بخش خواهیم دید به کمک Threading می‌توان قسمت‌های مختلفی از حافظه‌ی یک پردازه را به صورت همزمان اجرا کرد. در فارسی کلمه‌ی Thread گاهی ریسه، گاهی ریسمان، گاهی بند و گاهی نخ ترجمه می‌شود (که هیچ یک از آنها معادل خوبی نیستند چون مفهوم جزئی از کل کلمه‌ی Thread را ندارند).

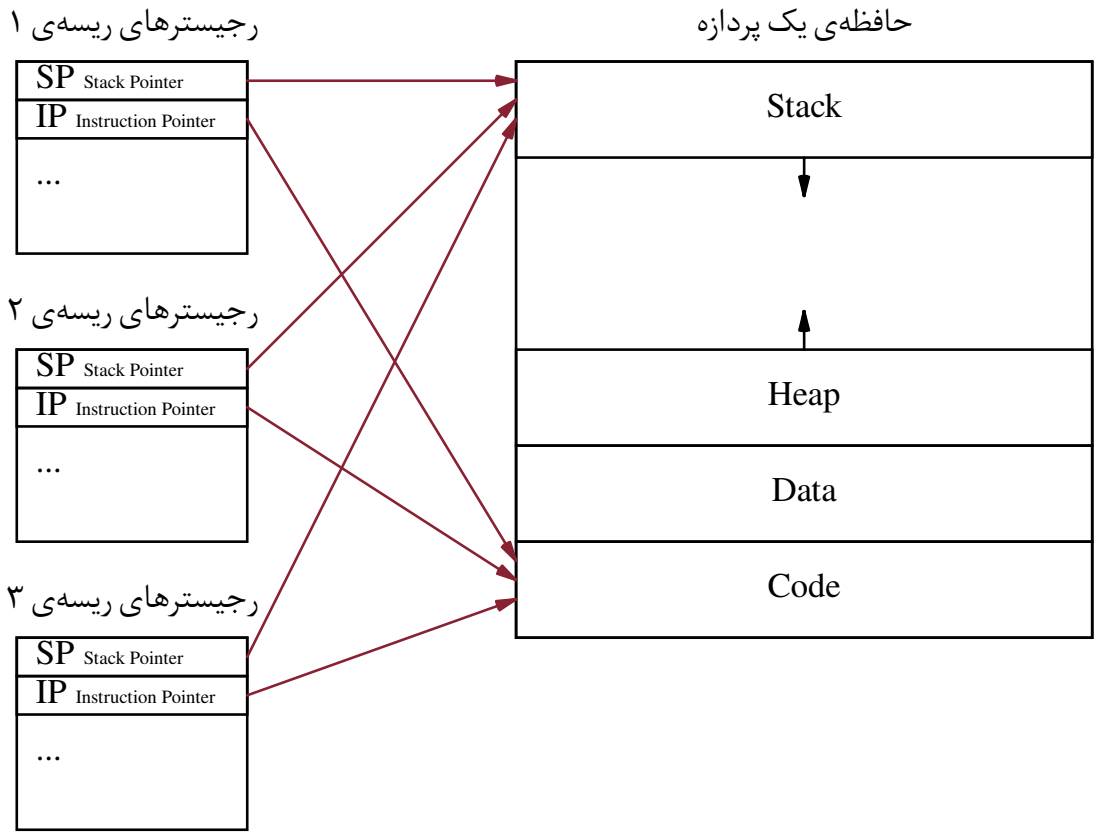
- پردازه‌هایی که تا به حال دیده‌ایم تک ریسه‌ای بوده‌اند.



- به هر پردازه قسمتی از حافظه تخصیص داده می‌شود.
- سیستم عامل پردازنده را در اختیار پردازه‌های آماده‌ی اجرا قرار می‌دهد تا اجرا شوند.
- هر پردازه مجموعه‌ی رجیسترهای پردازنده‌ی خاص خودش را دارد که در هنگام تعویض متن ذخیره و بازیابی می‌شوند.
- در هر لحظه یک قسمت از حافظه‌ی پردازه در حال اجرا است (قسمتی که رجیستر Instruction Pointer پردازنده به آن اشاره می‌کند).

پردازه‌های چند ریسه‌ای

- در پردازه‌های چند ریسه‌ای این امکان وجود دارد که چند قسمت پردازه به صورت همزمان اجرا شود.



- هر پردازه یک یا چند ریسه دارد.
- هر ریسه مجموعه‌ای از رجیسترهای خاص خودش را دارد (سیستم عامل اطلاعات دیگری نیز برای هر ریسه نگه می‌دارد).
- سیستم عامل می‌تواند پردازنده را در اختیار هر یک از ریسه‌های یک پردازه قرار دهد.
- حافظه بین ریسه‌های یک پردازه مشترک است (یعنی اگر یکی از ریسه‌ها حافظه را تغییر دهد سایر ریسه‌ها حافظه‌ی تغییر داده شده را می‌خوانند).
- اگر چند پردازنده در سیستم عامل موجود باشند، سیستم عامل می‌تواند پردازنده‌ها را به صورت همزمان در اختیار ریسه‌های یک پردازه قرار دهد.

مثال برای یک پردازه‌ی چند ریسه‌ای

در مثال زیر فرض کنید create_thread() یک ریسه‌ی جدید بسازد. این ریسه تابعی را که به عنوان ورودی داده می‌شود اجرا می‌کند و با خاتمه‌ی تابع، ریسه از بین می‌رود.

```
 1  void f(void)
 2  {
 3       printf("A\n");
 4  }
 5
 6  int main(void)
 7  {
 8       create_thread(f);
 9       printf("B\n");
10       return 0;
11  }
```

- اجرای یک پردازه با یک ریسه (ریسه‌ی اصلی) شروع می‌شود. در زبان C این ریسه تابع main() را اجرا می‌کند.
- با فراخوانی create_thread() یک ریسه‌ی جدید ساخته می‌شود که تابع f را اجرا می‌کند.
- بنابراین در این وضعیت پردازه دو ریسه دارد که یکی خط‌های نهم و دهم را از تابع main() اجرا می‌کند و ریسه‌ی دیگر تابع f() را اجرا می‌کند.
- بنابراین هم A و هم B در خروجی چاپ می‌شود اما ترتیب آنها مشخص نیست و به زمانبندی سیستم عامل بستگی دارد.
- با اتمام تابع f() ریسه‌ی دوم از بین می‌رود.

مثال برای یک پردازه‌ی چند ریسه‌ای

در مثال زیر متغیر var یک متغیر سراسری است.

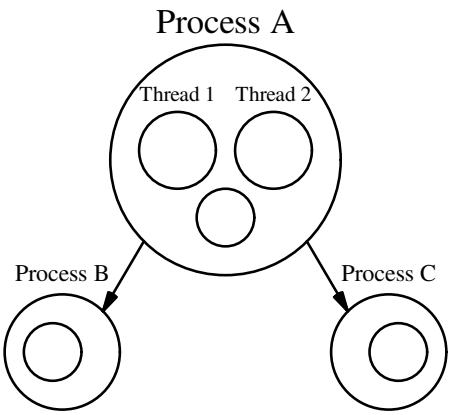
```
 1  int var = 0;
 2
 3  void f(void)
 4  {
 5       var = 1;
 6  }
 7
 8  void g(void)
 9  {
10       var = 2;
11  }
12
13  int main(void)
14  {
15       create_thread(f);
16       create_thread(g);
17       printf("var: %d\n", var);
18       return 0;
19  }
```

- پس از ساختن دو ریسه در ریسه‌ی اصلی (خط‌های پانزدهم و شانزدهم)، مقدار var چاپ می‌شود (خط هفدهم).
- چون حافظه‌ی بین ریسه‌های یک پردازه مشترک است، در ریسه‌ی اصلی ممکن است صفر، یک یا دو چاپ شود (با توجه به زمانبندی سیستم عامل).

مثال برای ترکیب پردازه‌ها و ریسه‌ها

در مثال زیر، دو پردازه ایجاد می‌شوند و پردازه‌ی پدر، دو ریسه نیز می‌سازد.

```
void f(void)
{
     printf("A: Thread 1\n");
}
void g(void)
{
     printf("A: Thread 2\n");
}
int main(void)
{
     if (fork() == 0) {
          printf("B\n");
          return 0;
     }
     if (fork() == 0) {
          printf("C\n");
          return 0;
     }
     create_thread(f);
     create_thread(g);
     return 0;
}
```

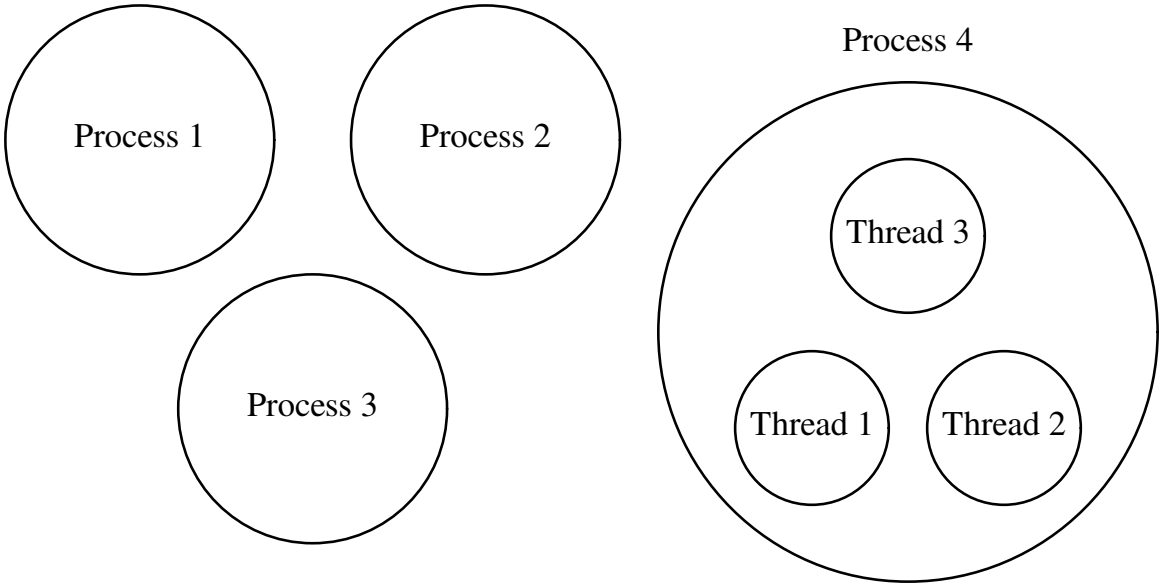


مثال‌هایی از استفاده از ریسه‌ها

برنامه‌های زیادی از ریسه‌ها استفاده می‌کنند. چند مثال:

- در یک برنامه‌ی ویرایش متن ممکن است یک ریسه متن را نمایش دهد و ریسه‌ی دیگر متن را برای خطاهای املایی بررسی کند.
- در یک مرورگر یک ریسه می‌تواند اطلاعات را از شبکه دریافت کند و ریسه‌ی دیگر آن را در صفحه نمایش دهد.
- در یک مرورگر یک ریسه‌ی مجزا می‌تواند به هر Tab یا پنجره تخصیص یابد تا پنجره‌های مختلف به صورت همزمان داده دریافت کنند یا اطلاعات را نمایش دهند.
- در یک برنامه‌ی محاوره‌ای (Interactive) یک ریسه می‌تواند داده‌ها را پردازش کند و ریسه‌ی دیگر می‌تواند دستورات کاربر را دریافت کند یا خروجی‌ها را به او نمایش دهد.

هم شکستن یک برنامه به تعدادی پردازه و هم استفاده از چند ریسه در یک پردازه این امکان را فراهم می‌کنند که بتوان چند کار را به صورت همزمان انجام داد.



- طراح یک نرم‌افزار باید تصمیم بگیرد از کدام استفاده کند.
- تفاوت مهم استفاده از چند پردازه و چند ریسه در میزان اشتراک اطلاعات بین آنها است.
- اگر اطلاعات مشترک زیاد باشد یا وابستگی زیادی موجود باشد، معمولا از ریسه‌ها استفاده می‌شود که بیشتر قسمت‌های حافظه بین آنها مشترک است.
- اگر میزان اشتراک کم باشد، می‌توان از پردازه‌ها استفاده کرد و در صورت نیاز با یکی از روش‌های انتقال اطلاعات بین پردازه‌ای اطلاعات لازم را انتقال داد.

خوبی‌های استفاده از ریسه‌ها

برخی از مزایای استفاده از پردازه‌های چند ریسه‌ای با شکستن یک برنامه به چند پردازه مشترک است:

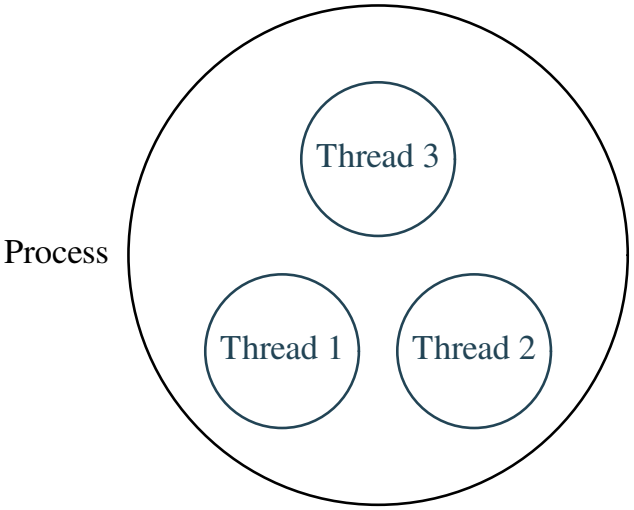
- امکان انجام چند کار به صورت همزمان بوجود می‌آید.
- می‌توان به صورت همزمان از چند پردازنده یا چند هسته‌ی یک پردازنده‌ی چند هسته‌ای استفاده کرد.
- با شکستن یک برنامه به تعدادی ریسه، Modularity می‌تواند افزایش یابد و پیاده‌سازی و نگهداری آن راحت‌تر می‌شود.
- در برنامه‌ای که برخی از قسمت‌های آن منتظر (Block) می‌شوند، در هنگام انتظار برخی از ریسه‌ها، می‌توان همچنان در ریسه‌های دیگر پردازش انجام داد.
- چند ریسه که به یک پردازه تعلق دارند منابع (مثل حافظه) کمتری نسبت به چند پردازه احتیاج دارند.
- در برنامه‌های محاوره‌ای استفاده از ریسه‌ها می‌تواند زمان پاسخ آنها را به ورودی‌های کاربر کاهش دهد.

مشکلات و سختی‌های استفاده از ریسه‌ها

برخی از مشکلات مهم استفاده از ریسه‌ها موارد زیر هستند.

- دسترسی همزمان به قسمت‌هایی از حافظه مشکلات مهمی را به همراه دارد که در بخش بعدی درس به آن می‌پردازیم.
- در صورت بروز خطا در یکی از ریسه‌ها مثل دسترسی غیر مجاز به حافظه، کل پردازه دچار مشکل می‌شود.
- آزمون برنامه‌هایی که از ریسه‌ها استفاده می‌کنند و اشکال زدایی در آنها دشوارتر است.

برنامه‌نویسی چند هسته‌ای: استفاده از هسته‌های یک پردازنده به صورت همزمان برای افزایش سرعت اجرای پردازه‌ها.



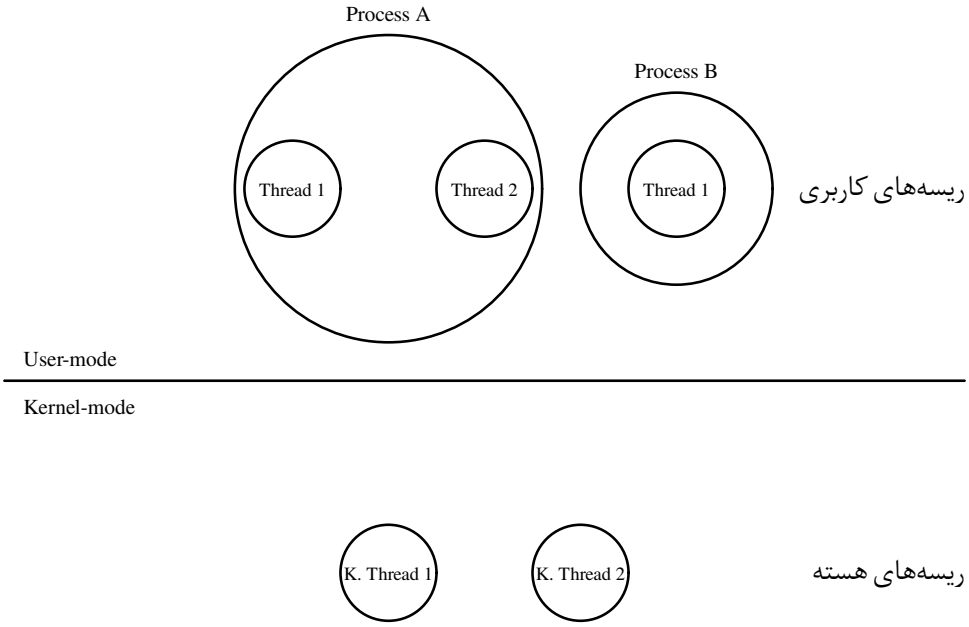
- یکی از روش‌های برنامه‌نویسی چند هسته‌ای استفاده از ریسه‌ها است.
- معمولا لازم است پردازش به قسمت‌های کوچک‌تری شکسته شود و سپس این قسمت‌ها بین ریسه‌ها تقسیم شوند.
- این قسمت‌ها باید به شکلی تقسیم شوند تا توازن بین بار پردازشی ریسه‌ها برقرار باشد.
- باید به وابستگی داده‌ها دقت کرد.
- معمولا کار سختی است و احتیاج به تجربه و مطالعه دارد. برای نمونه، فکر کنید چگونه می‌توان با استفاده از ریسه‌ها جستجوی DFS در یک گراف را به صورت همروند توسط چند ریسه انجام داد.

همروندی و توازی

دو اصطلاح همروندی (Concurrency) و توازی (Parallelism)

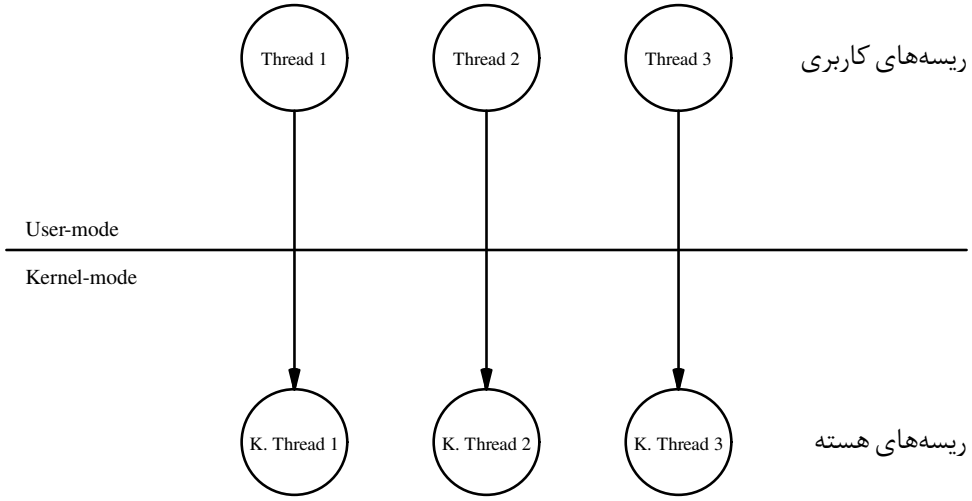
- فرض کنید در سیستم عاملی دو پردازه (یا بیشتر) پردازه (یا ریسه) موجود باشد:
- همروندی یعنی کار هر دو پردازه با گذشت زمان پیشرفت کند. سیستم عامل می‌تواند پردازنده را گاهی به پردازه‌ی اول بدهد و گاهی به پردازه‌ی دوم. به این صورت، هر دو پردازه اجرا می‌شوند. دقت کنید که لازم نیست هر دو پردازه به صورت همزمان در حال اجرا باشند.
- توازی یعنی اجرای همزمان دو پردازه. اگر دو پردازنده در اختیار سیستم عامل باشد، سیستم عامل می‌تواند هر پردازه را روی یکی از این پردازنده‌ها اجرا کند تا توازی ایجاد شود. دقت کنید که روی یک ماشین فقط با یک پردازنده‌ی تک هسته‌ای، دو پردازه نمی‌توانند به صورت موازی اجرا شوند.
- بنابراین وجود توازی، همروندی را نتیجه می‌دهد ولی عکس آن درست نیست.

- تا اینجا فرض کرده‌ایم هر ریسه‌ی فضای کاربری توسط سیستم عامل مدیریت می‌شود.
- در ادامه روش‌های بیشتری را برای طراحی ریسه‌های فضای کاربری خواهیم دید.



- برای هر زبان برنامه نویسی ممکن است یک یا چند کتابخانه برای استفاده از ریسه‌ها موجود باشد؛ مثل کتابخانه‌ی PThreads برای سیستم‌های عامل مبتنی بر POSIX.
- در این کتابخانه‌ها از یکی از مدل‌های چند ریسگی برای ریسه‌های فضای کاربری استفاده می‌شود.
- در ادامه با مدل‌های مختلف ریسه‌ها در کتابخانه‌های فضای کاربری آشنا می‌شویم.

- در مدل یک به یک، یک ریسه‌ی هسته به هر ریسه‌ی فضای کاربری تخصیص می‌یابد.

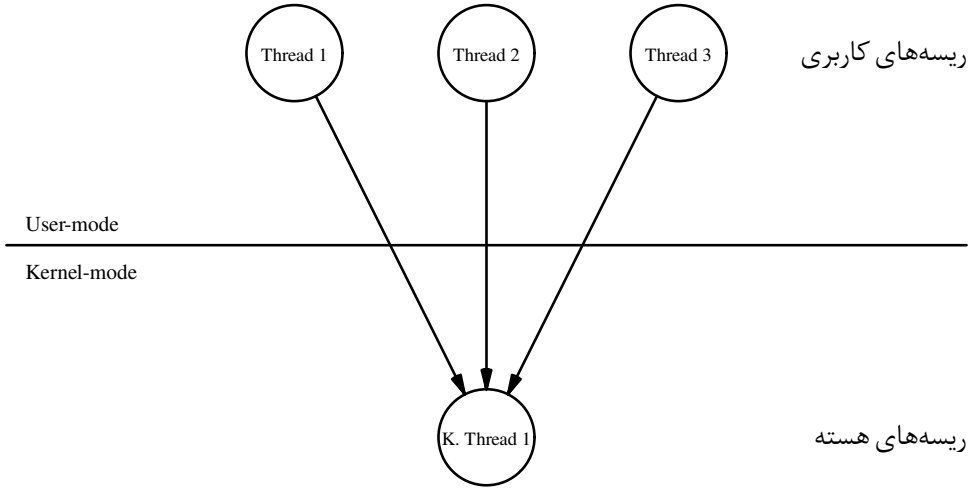


- بنابراین سیستم عامل می‌توانید هر ریسه‌ی فضای کاربری را به صورت مجزا مدیریت و زمانبندی کند.

- اگر یک ریسه‌ی یک پردازه‌ی چند ریسه‌ای منتظر (Block) شود، بقیه‌ی ریسه‌های آن می‌توانند همچنان اجرا شوند.
- اگر بیشتر از یک پردازنده موجود باشد، سیستم عامل می‌تواند ریسه‌ها را به صورت موازی اجرا نماید.
- چون سیستم عامل برای هر ریسه‌ی هسته اطلاعاتی را نگه می‌دارد و آنها را مدیریت می‌کند، تخصیص یک ریسه‌ی هسته به هر ریسه‌ی فضای کاربری منابع نسبتا زیادی را (نسبت به بقیه‌ی مدل‌ها) احتیاج دارد و پر هزینه است.

مدل چند به یک

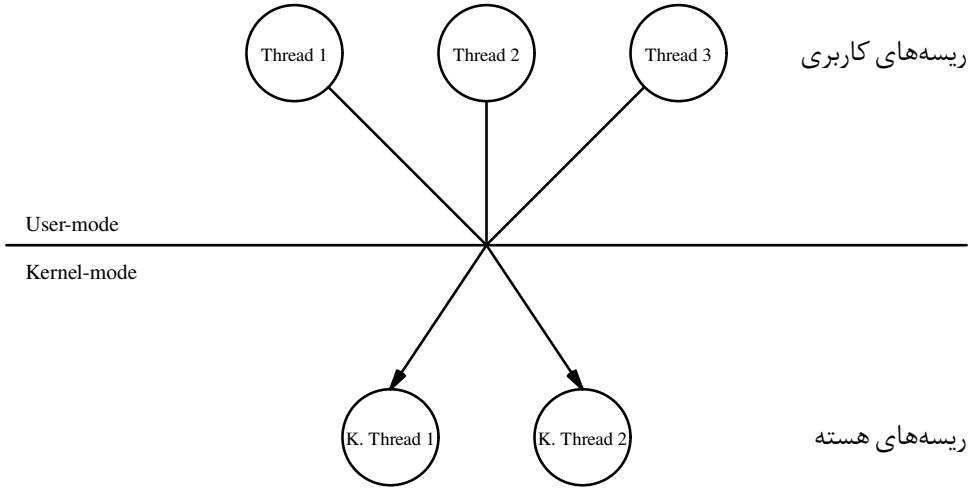
- در مدل یک به یک، همه‌ی ریسه‌های فضای کاربری یک پردازه به یک ریسه هسته نگاشت می‌شوند.



- سیستم عامل از وجود ریسه‌های فضای کاربری اطلاعی ندارد و فقط پردازه را مدیریت و زمانبندی می‌کند. مدیریت و زمانبندی ریسه‌ها در فضای کاربری و توسط کتابخانه‌ی ریسه‌ها انجام می‌شود.

- اگر یک ریسه‌ی یک پردازه‌ی چند ریسه‌ای منتظر (Block) شود، همه‌ی ریسه‌های آن نیز منتظر می‌شوند.
- اگر بیشتر از یک پردازنده موجود باشد، سیستم عامل نمی‌تواند ریسه‌ها را به صورت موازی اجرا نماید.
- این مدل هزینه‌ی نسبتا کمی دارد.
- در سیستم‌های عاملی که از ریسه‌ها پشتیبانی نمی‌کردند، این مدل استفاده می‌شد (چون لازم نیست سیستم عامل از وجود ریسه‌ها اطلاع داشته باشد و فقط پردازه‌ها را مدیریت می‌کند).

- در مدل چند به چند، ریسه‌های یک پردازه به تعداد (معمولا کمتری) ریسه‌ی هسته نگاشت می‌شوند.



- اگر یک ریسه‌ی یک پردازه‌ی چند ریسه‌ای منتظر (Block) شود، بقیه‌ی ریسه‌های آن می‌توانند همچنان اجرا شوند (تا وقتی که همه‌ی ریسه‌های هسته‌ی اختصاص یافته به پردازه منتظر شوند).
- اگر بیشتر از یک پردازنده موجود باشد، سیستم عامل می‌تواند ریسه‌ها را به صورت موازی اجرا نماید.
- هزینه‌ی این مدل از مدل یک به یک کمتر و از مدل چند به یک بیشتر است.
- مدل دو رده‌ای حالتی از مدل چند به چند است که در آن برای برخی از ریسه‌های فضای کاربری می‌توان یک ریسه‌ی هسته‌ی مجزا در نظر گرفت.

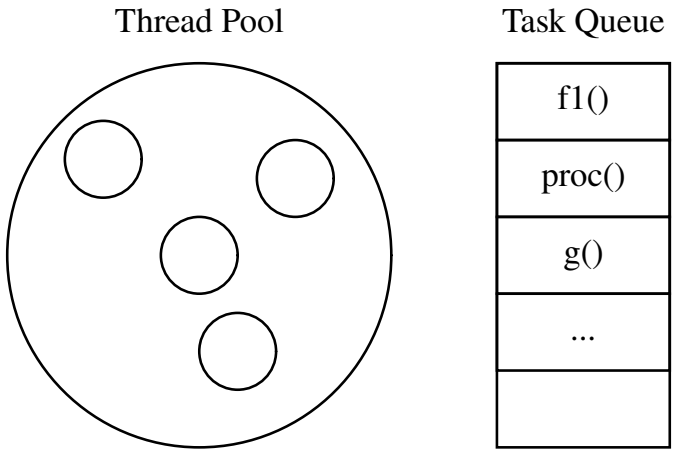
مدیریت تعداد زیادی ریسه توسط برنامه‌نویس کار راحتی نیست.

- در مدیریت ضمنی ریسه‌ها (Implicit Threading) مدیریت ریسه‌ها به کتابخانه یا محیط اجرای پردازه داده می‌شود.
- بنابراین، برنامه‌نویس می‌تواند کار انجام شده توسط ریسه‌ها را مشخص کند و ایجاد، از بین بردن و مدیریت ریسه‌ها را کتابخانه انجام دهد.
- در ادامه چند روش را برای ریسه‌های ضمنی مرور می‌کنیم.

ریسه‌های آماده‌باش

ریسه‌های آماده‌باش (Thread Pool) یکی از روش‌های مدیریت ضمنی ریسه‌ها است.

- در ریسه‌های آماده‌باش، تعداد ثابتی ریسه در ابتدا ساخته می‌شود؛ همه‌ی ریسه‌ها در شروع بیکار هستند.



- تعداد اولیه‌ی ریسه‌های آماده معمولا با توجه به منابع موجود و کاربرد تعیین می‌گردد.
- استفاده از ریسه‌های آماده‌باش چه مزیت‌هایی دارد؟

- OpenMP یک API را برای تعدادی زبان از جمله زبان C فراهم می‌کند.

```
int main(void)
{
     #pragma omp parallel
     {
          printf("A\n");
     }
     return 0;
}
```

- در قطعه کد بالا عبارت بعد از «#pragma omp parallel» به صورت موازی و یک بار در هر ریسه اجرا می‌شود.

- در مثال زیر، خط «#pragma omp parallel for» باعث می‌شود حلقه‌ی بعد از آن به صورت موازی اجرا شود.

```
int main(void)
{
     #pragma omp parallel for
     for (i = 0; i < 1000; i++)
          f(i);
     return 0;
}
```

- برای مثال، اگر دو ریسه موجود باشند، ممکن است دور صفرم تا دور ۴۹۹ توسط ریسه‌ی اول و دور ۵۰۰ تا دور ۹۹۹ در ریسه‌ی دوم انجام شود.
- برای آشنا شدن با جزئیات این کتابخانه، منابع موجود را مطالعه نمایید.

در ادامه چند نکته‌ی پراکنده در مورد ریسه‌ها را بررسی می‌کنیم.

قبلا دیده‌ایم که برای ایجاد پردازه‌های جدید از تابع ()fork استفاده می‌شود.

- فرض کنید یک پردازه که چند ریسه دارد ()fork را فراخوانی کنید.

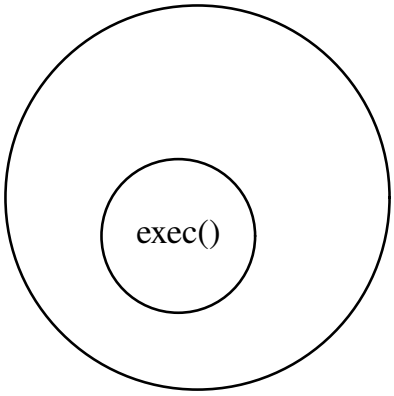
```
static void f(void)
{
     fork();
     printf("B: %d\n", getpid());
}

int main(void)
{
     create_thread(f);
     sleep(1);
     printf("A: %d\n", getpid());
     return 0;
}
```

- دقت کنید که از بین ریسه‌های پردازه‌ی پدر، فقط یکی از آنها ()fork را فراخوانی کرده است.
- در بسیاری از سیستم‌های عامل (از جمله آنهایی که مبتنی بـر استاندارد POSIX هستند) پـردازه‌ی فرزند فقـط یک ریسه دارد که مشابه ریسه‌ای است که در پردازه‌ی پدر ()fork را فراخوانی کرده است.

قبلا دیده‌ایم که برای اجرای برنامه‌ها از یکی از توابع خانواده‌ی exec استفاده می‌شود.

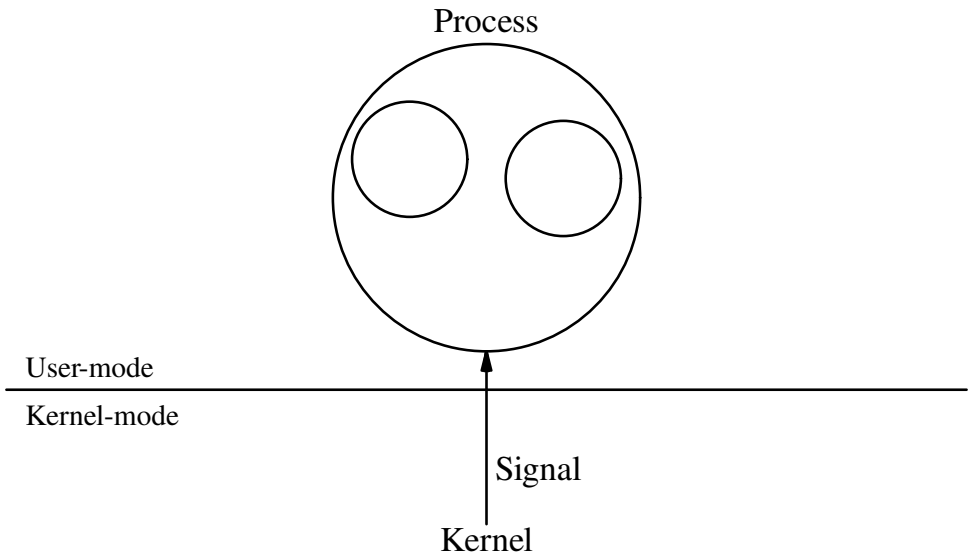
- فرض کنید پردازه‌ای که چند ریسه دارد یکی از توابع خانواده‌ی exec را فراخوانی کند.



- در صورتی که فراخوانی موفق باشد، همه‌ی ریسه‌های پردازه به جز یکی از بین می‌روند.
- برنامه‌ی جدید، پس از انتقال به حافظه فقط با یک ریسه اجرا می‌شود.

قبلا دیده‌ایم که با سیگنال‌ها سیستم عامل پردازه را از رخدادی آگاه می‌کند.

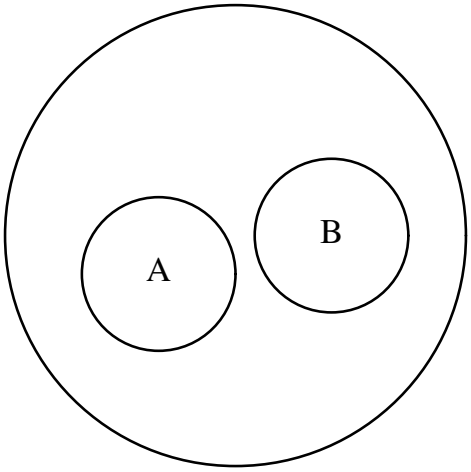
- در صورت بروز سیگنال، تابعی در فضای کاربری می‌تواند اجرا شود.
- اگر پردازه چند ریسه داشته باشد، باید مشخص شود کدام یک از ریسه‌ها به سیگنال پاسخ می‌دهد.



سیستم‌های عامل به شکل‌های متفاوتی ریسه‌ی پاسخگو به سیگنال را انتخاب می‌کنند، مثل موارد زیر:

- همه‌ی سیگنال‌ها توسط یک ریسه‌ی مشخص پاسخ داده شود.
- همه‌ی ریسه‌ها به همه‌ی سیگنال‌ها پاسخ دهند.
- اگر سیگنال توسط یک ریسه ایجاد شده است (مثل خطا در کدی که ریسه اجرا کرده است)، سیگنال باید توسط همان ریسه پاسخ داده شود.
- هر نوع سیگنال به ریسه‌ی خاصی داده شود.

برای لغو (Cancellation یا خاتمه) یک ریسه توسط یک ریسه‌ی دیگر روش‌های متفاوتی موجود است.



- در لغو ناهمگام (Asynchronous Cancellation) یک درخواست لغو ریسه‌ی مشخص شده را بدون تأخیر از بین می‌برد.
- در لغو با تأخیر (Deferred Cancellation) ریسه به صورت متناوب بررسی می‌کند که آیا باید لغو شود یا خیر.
- مزیت لغو با تأخیر چه هست؟

حافظه‌ی محلی ریسه‌ها (Thread Local Storage) یا به صورت مخفف TLS حافظه‌ای برای متغیرهای سراسری است که بین ریسه‌ها متشرک نیست.

- متغیرهای سراسری در زبان‌هایی مثل C بین همه‌ی ریسه‌های یک پردازه مشترک هستند.
- برای مثال، در شبه کد زیر در ریسه‌ی اصلی مقدار یک چاپ می‌شود چون در ریسه‌ی دوم مقدار این متغیر سراسری تغییر می‌کند.

```
int var = 0;
void f(void)
{
     var = 1;
}
int main(void)
{
     create_thread(f);
     sleep(1);
     printf("%d\n", var);
     return 0;
}
```

- گاهی لازم است یک نسخه از متغیری سراسری برای هر ریسه موجود باشد.
- سیستم عامل حافظه‌ای برای این متغیرها در اختیار پردازه‌ها قرار می‌دهد که به آن حافظه‌ی محلی ریسه یا TLS می‌گویند.
- اگر یکی از ریسه‌ها یک متغیر در TLS را تغییر دهد، مقدار آن متغیر در سایر ریسه‌ها تغییر نمی‌کند.

- در استاندارد سال ۲۰۱۱ زبان C با کلمه‌ی رزرو شده‌ی thread_local می‌توان متغیر محلی ریسه تعریف کرد.

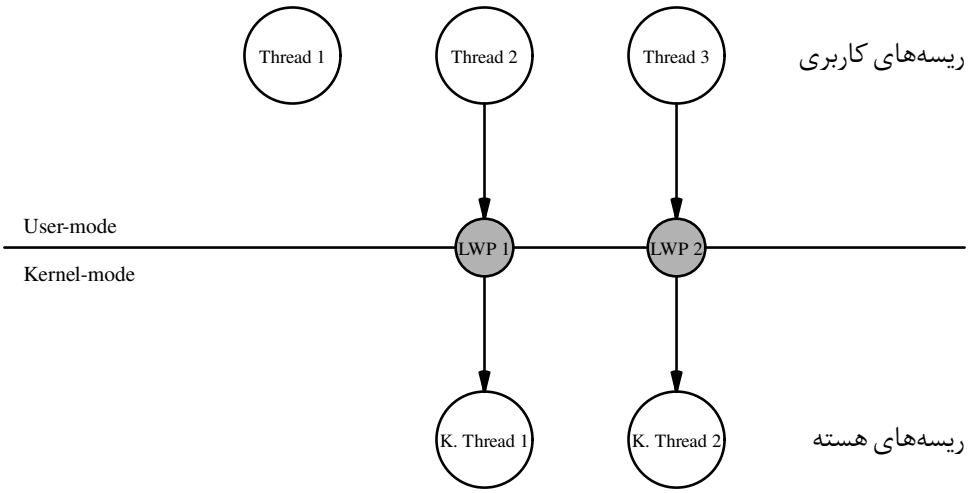
```
thread_local int var;
```

- در شبه کد صفحه‌ی قبل، اگر متغیر var محلی ریسه تعریف شود، خروجی شبه کد همواره صفر است.

پردازه‌های سبک وزن (Light-Weight Process یا به صورت مخفف LWP) یک روش برای پیاده‌سازی مدل ریسه‌ی چند به چند در سیستم عامل Solaris است.

- فرض کنید که از مدل ریسه‌ی چند به چند استفاده شود.
- فرض کنید n ریسه‌ی فضای کاربری یک پردازه به m ریسه‌ی فضای هسته نگاشت شده باشند به صورتی که n باشد.
- در این صورت اگر به تعداد m ریسه‌ی کاربری Block شوند چه اتفاقی می‌افتد؟
- به کمک LWP این مشکل حل می‌شود.

- LWP یک ساختمان داده‌ی مشترک بین فضای هسته و کاربر است.
- به هر LWP یک ریسه‌ی هسته تخصیص می‌یابد.



- در فضای کاربری نیز یک ریسه به LWP نگاشت می‌شود تا اجرا شود. این نگاشت توسط قسمتی از کتابخانه‌ی ریسه‌ها انجام می‌شود که مسئولیت زمانبندی ریسه‌های کاربری را در اختیار دارد.
- وقتی که یک LWP منتظر می‌شود، هسته به کمک یک فراخوانی Upcall فضای کاربری (زمانبند ریسه‌های کاربری) را مطلع می‌سازد.
- زمانبند نیز ریسه‌ی کاربری دیگری را به LWP نگاشت می‌کند تا اجرا شود.
- بنابراین، اگر یک ریسه‌ی کاربری Block شود، ریسه‌ی دیگری جای آن را می‌گیرد.
- چون LWP موجب فعال شدن زمانبند ریسه‌های کاربری می‌شود، به آن اصطلاحا فعال‌سازی زمانبند (Scheduler Activation) هم گفته می‌شود.

- کتابخانه‌های متفاوتی برای استفاده از ریسه‌ها وجود دارند.
- بسیاری از زبان‌های سطح بالا مثل Java در کتابخانه‌ی استانداردشان قسمتی برای ریسه‌ها و همروندی دارند.
- در سیستم‌های عاملی که منطبق بر POSIX هستند، می‌توان از کتابخانه‌ی PThreads (مخفف POSIX Threads) استفاده کرد.
- در درس آزمایشگاه سیستم‌های عامل با این کتابخانه آشنا می‌شوید.