یادداشت‌های درس سیستم‌های عامل - بخش سوم

در این بخش از درس به پردازه‌ها خواهیم پرداخت.

- یکی از اهداف اصلی سیستم عامل اجرای برنامه‌های کاربردی کاربران است.
- برنامه‌ها پس از اجرا به پردازه (Process) تبدیل می‌شوند.
- در این بخش، پردازه‌ها را با جزئیات بیشتری مطالعه می‌کنیم.

تاریخچه: پردازش دسته‌ای

در دهه‌های اول استفاده از کامپیوتر، بیشتر وظایف پردازشی که توسط کامپیوتر اجرا می‌شدند نیاز به تعامل مستقیم با کاربر نداشتند و ورودی‌ها را از دستگاه‌های ورودی و خروجی مثل دیسک دریافت می‌کردند و خروجی‌ها را به صورت مشابه ذخیره می‌کردند. به این وظیفه‌های پردازشی معمولا Job گفته می‌شود.

- بیشتر این Job-ها به صورت دسته‌ای از وظایف مرتبط اجرا می‌شد. به اجرای دسته‌ای این وظایف پردازش دسته‌ای (Batch Processing) گفته می‌شود (Batch در واقع دسته‌ای از Job-ها است).
- در ادامه‌ی درس چه وظیفه‌ای احتیاج به تعامل مستقیم با کاربر داشته باشد و چه نداشته باشد، از اصطلاح پردازه برای آنها استفاده می‌کنیم.

تاریخچه: چند برنامگی

در گذشته سیستم‌های عامل تنها یک پردازه را اجرا می‌کردند. وقتی پردازه‌ی در حال اجرا منتظر یک عمل ورودی یا خروجی می‌شد، پردازنده بدون استفاده باقی می‌ماند.

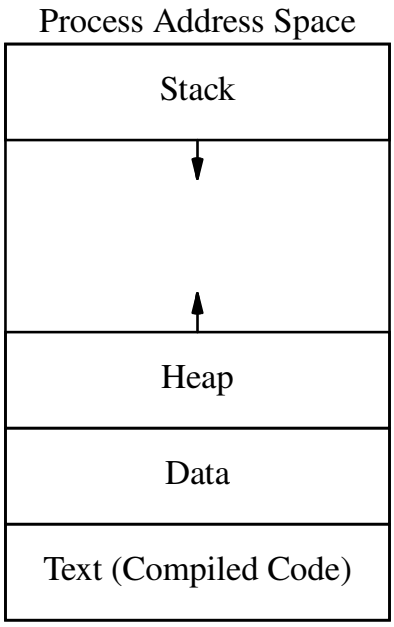
- به خاطر اینکه از پردازنده بهتر استفاده شود، سیستم‌های عامل Multi-programming به وجود آمدند؛ سیستم عامل وضعیت چند پردازنده‌ی در حال اجرا را نگه می‌دارد ولی در هر لحظه پردازنده را در اختیار یکی از این پردازه‌ها قرار می‌دهد. وقتی پردازه‌ی در حال اجرا منتظر یک عمل ورودی یا خروجی می‌شود، پردازنده به پردازه‌ی دیگری که آماده‌ی اجرا هست داده می‌شود. در این سیستم‌های عامل، سیستم عامل می‌توانست تعیین کند چند پردازه در هر لحظه در حال اجرا باشند.
- به تعداد پردازه‌های در حال اجرا درجه‌ی چند برنامگی (Degree of Multiprogramming) گفته می‌شود. هر چه درجه‌ی چند برنامگی بیشتر باشد، زمان بیکاری پردازنده کمتر می‌شود اما از سوی دیگر منابع بیشتری برای نگهداری حافظه و وضعیت پردازه‌های در حال اجرا لازم خواهد بود.

منابع پردازه‌ها

سیستم عامل به هـر پردازنده‌ی در حال اجرا منابعی را تخصیص می‌دهـد و بـرای آن اطلاعاتـی را نگـه می‌دارد. بـرای نمونه، هر پردازه به قسمت‌هایی از حافظه‌ی احتیاج دارد و تعدادی فایل باز نگه می‌دارد.

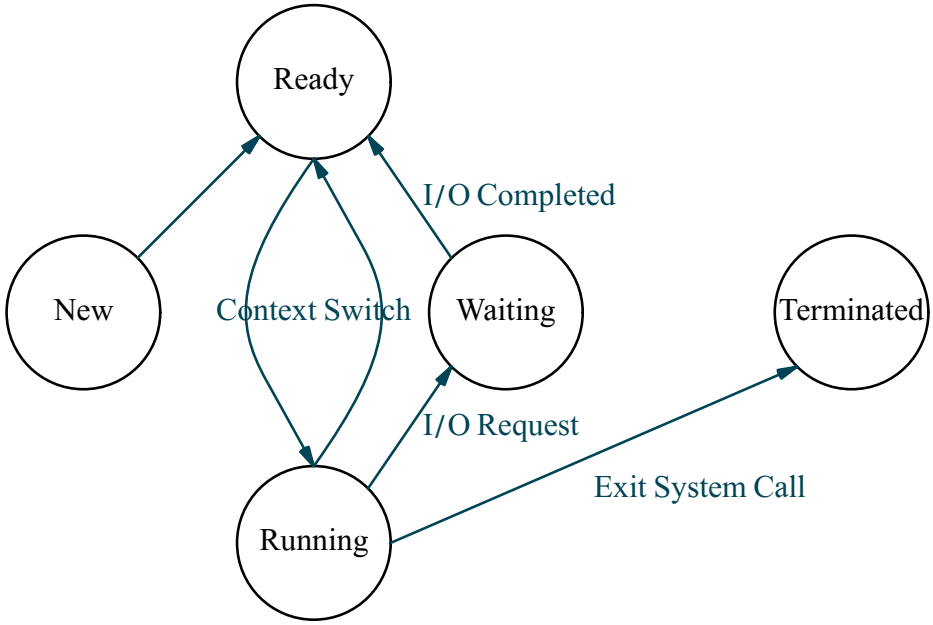
فضای آدرس

- فضای آدرس (Address Space) یک پردازه: به قسمتی از حافظه که پردازه می‌تواند آدرس آن را تولید کند و به آن دسترسی داشته باشد.
- حافظه‌ی تخصیص یافته شده به پردازه چند بخش دارد.



- قسمت Text کد ماشین قابل اجرای برنامه (که توسط کامپایلر تولید شده است) را نگه می‌دارد.
- قسمت Data داده‌های ایستای برنامه (که در زمان کامپایل اندازه یا محتویات آنها مشخص است) قرار می‌گیرد.
- قسمت Heap مربوط به قسمتی از حافظه می‌شود که در زمان اجرای برنامه به صورت پویا به آن تخصیص می‌یابد.
- قسمت Stack یا پشته برای نگهداری متغیرهای محلی توابع است؛ معمولا پشته از پایین رشد می‌کند (به سمت آدرس‌های کوچک‌تر بزرگ می‌شود).

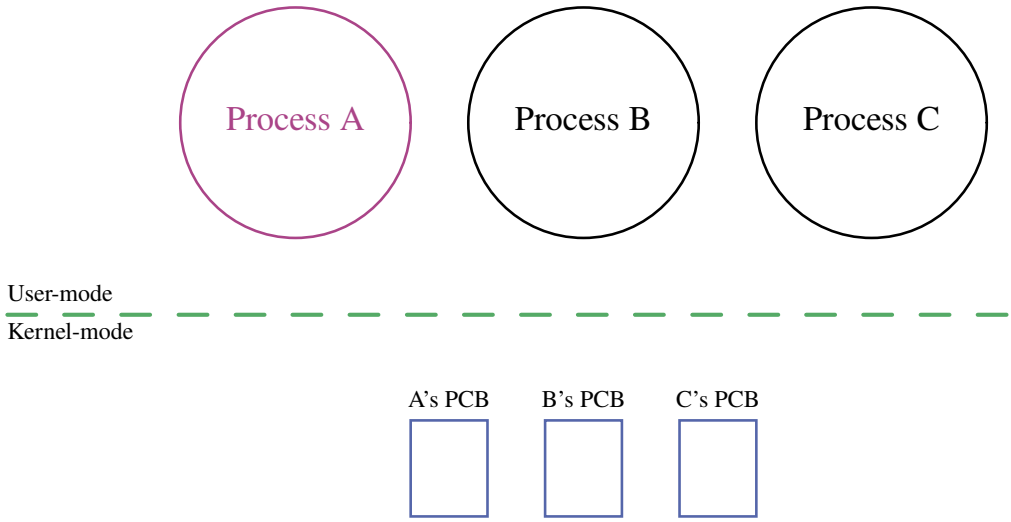
در هر لحظه یک پردازه در وضعیتی مثل وضعیت زیر قرار دارد.



- وقتی پردازه‌ای ایجاد می‌شود در وضعیت New قرار می‌گیرد.
- در حالت آماده باش (Ready) پردازه آماده‌ی اجرا است و در صف زمانبندی پردازنده قرار دارد.
- وقتی پردازنده در اختیار یک پردازه‌ی آماده‌ی اجرا قرار گیرد، وضعیت آن به در حال اجرا (Running) تغییر می‌کند. اگر پردازنده از پردازه گرفته شود، وضعیت آن دوباره به حالت آماده باش بر می‌گردد.
- اگر در حالت اجرا، پردازه یک عمل ورودی یا خروجی انجام دهد، وضعیت پردازه به در حال انتظار (Waiting) تغییر می‌کند. پس از پایان عمل ورودی و خروجی، وضعیت پردازه به آماده باش تغییر می‌کند.
- در نهایت، اگر پردازه با درخواست خودش (فراخوانی سیستم exit) به دلیل خطا یا فرستادن سیگنال خاتمه پیدا کند وضعیت آن به خاتمه (Terminated) تغییر می‌یابد. البته دقت کنید که پردازه در صورت بروز خطا یا دریافت برخی از سیگنال‌ها، از هر یک از حالت‌ها می‌تواند به حالت خاتمه انتقال یابد.

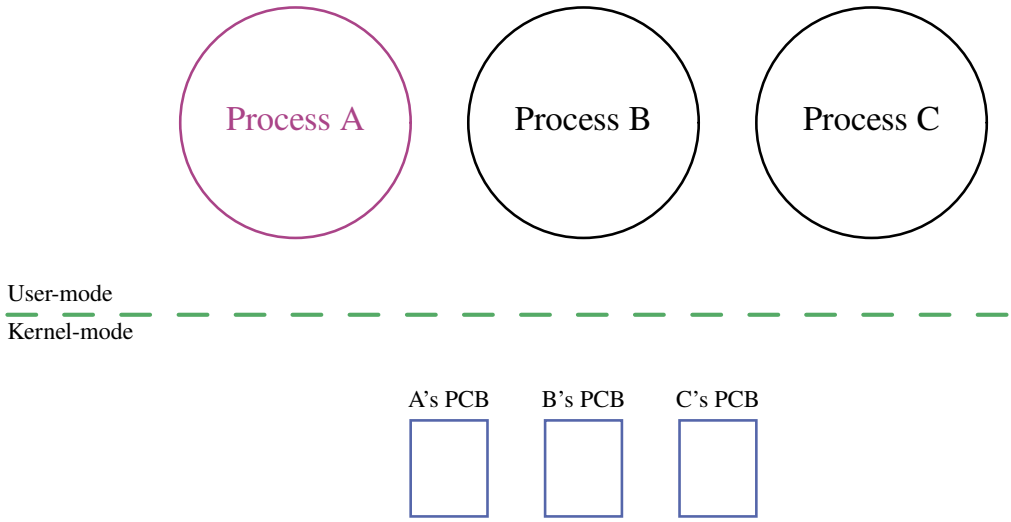
بلوک مدیریت پردازه

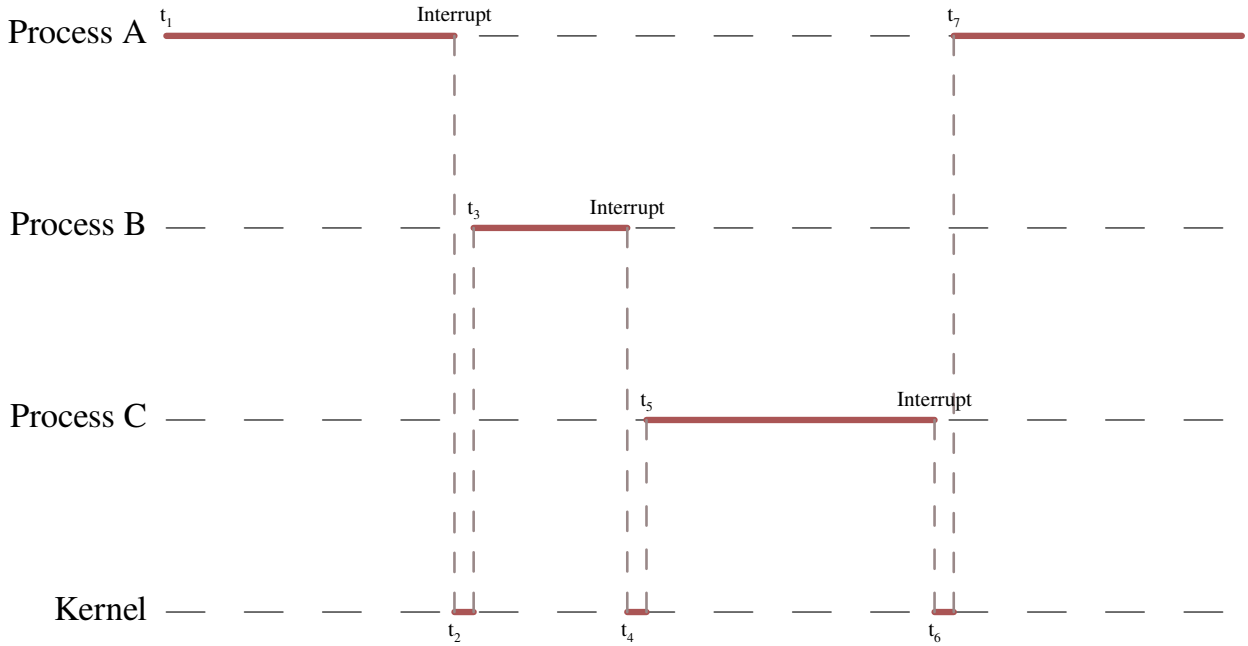
- سیستم عامل اطلاعات یک پردازه را در ساختاری به نام بلوک مدیریت پردازه (Process Control Block یا PCB) نگه می‌دارد.



- در PCB اطلاعاتی مثل، وضعیت پردازه، صاحب پردازه، فایل‌های باز، اطلاعات مربوط به مدیریت حافظه، اطلاعات مربوط به زمانبندی پردازنده، وضعیت عملیات ورودی یا خروجی قرار می‌گیرد.
- در PCB وضعیت رجیسترهای پردازنده در زمان‌هایی از اجرای پردازه نیز ذخیره می‌گردد.

به گرفتن پردازنده از یک پردازه و دادن آن به پـردازه‌ی دیگر تعویـض متن (Context Switch) گفتـه می‌شود. فـرض کنید پردازنده در اختیار پردازه‌ی A باشد و سیستم عامل قصد داشته باشد آن را به پردازه‌ی B بدهد.

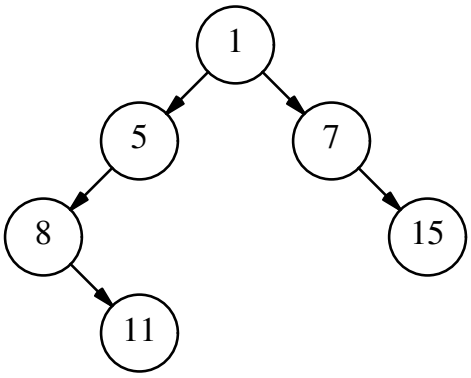




- سیستم عامل وضعیت پردازه‌ی A (از جمله وضعیت رجیسترهای پردازنده در آخرین لحظه‌ی اجرای این پـردازه) را در PCB مربوط به این پردازه ذخیره می‌کند.
- سپس، وضعیت پردازه‌ی B را از PCB مربوط به این پردازه بارگذاری می‌نماید.
- سپس با انتقال به حالت کاربری پردازنده اجرای پردازه‌ی B را ادامه می‌دهد.

هر پردازه‌ای در سیستم عامل شناسه‌ای دارد (به این شناسه Process Identifier یا به صورت مختصر PID گفته می‌شود).

- گرافی را در نظر بگیرید که در آن هر رأس آن یک پردازه موجود در سیستم عامل است. یک یال از رأس A به B نشان می‌دهد که پردازه‌ی B توسط رأس A ایجاد شده است. به این درخت، درخت پردازه‌ها (Process Tree) گفته می‌شود.



- در سیستم عامل لینوکس با دستور pstree می‌توانید درخت پردازه‌ها را مشاهده کنید.
- ریشه‌ی درخت پردازه‌ها، اولین پردازه‌ای است که ایجاد می‌شود. این پردازه توسط خود سیستم عامل ایجاد می‌شود و در Unix پردازه‌ی init نام دارد و شناسه‌ی آن یک است. همه‌ی پردازه‌های دیگر توسط یک پردازه ایجاد می‌شوند.

سیستم عامل توسط فراخوانی‌های سیستمی عملیات مختلفی را روی پردازه‌ها پیاده‌سازی می‌کند.

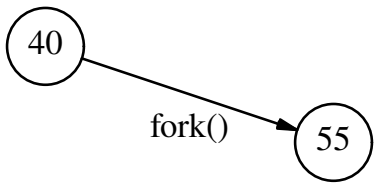
- ایجاد پردازه‌ها
- خاتمه‌ی آنها
- انتظار برای خاتمه‌ی پردازه‌ها
- اجرای برنامه‌های موجود در فایل سیستم

این عملیات را در سیستم عامل Unix و سیستم‌های عامل مشابه آن (استاندارد POSIX) بررسی می‌کنیم.

فراخوانی سیستمی fork() یک کپی از پردازه‌ی فراخوانی کننده می‌سازد.

```
1  int main(void)
2  {
3       printf("A %d\n", getpid());
4       fork();
5       printf("B %d\n", getpid());
6       return 0;
7  }
```

فرض کنید شکل زیر درخت پردازه‌ها برای مثال بالا باشد.



در این صورت، خروجی برنامه به صورت زیر است (ترتیب دو خط آخر ممکن است متفاوت باشد).

```
A 40
B 40
B 55
```

ایجاد پردازه‌ها: کد برگشتی

فراخوانی fork() در پردازه‌ی پدر شناسه‌ی پردازه‌ی فرزند (مقداری بزرگ‌تر از صفر) و در پردازه‌ی فرزند مقدار صفر را بر می‌گرداند.

```
1  int main(void)
2  {
3       if (fork() > 0)
4            printf("A\n");
5       else
6            printf("B\n");
7       return 0;
8  }
```

- ترتیب اجرای این دو پردازه (و چاپ شدن این دو خط) مشخص نیست و به زمانبند سیستم عامل بستگی دارد (این دو پردازه به صورت همروند اجرا می‌شوند).

ایجاد پردازه‌ها: حافظه‌ی مجزا

حافظه‌ی پردازه‌ی فرزند و پدر کاملا مجزا است و اگر هر یک از این دو پردازه حافظه‌ی خودش را تغییر دهد، متغیرهای پردازه‌ی دیگر تغییر نمی‌کند.

```
 1  int main(void)
 2  {
 3       int var = 0;
 4       if (fork() > 0) {
 5            var = 1;
 6       } else {
 7            printf("%d\n", var);
 8       }
 9       return 0;
10  }
```

- در شبه کد بالا، حتی اگر پردازه‌ی پدر زودتر اجرا شود، مقدار صفر در پردازه‌ی فرزند چاپ می‌شود، چون حافظه‌ی پردازه‌ی پدر و فرزند مجزا هست.

- پردازه‌ی جدیدی که پس از فراخوانی سیستمی ()fork حاصل می‌شود یک کپی از پردازه‌ی فراخوانی کننده است. برای اجرای برنامه‌ای که در فایل سیستم قرار دارد باید از یکی از توابع سیستمی خانواده‌ی exec استفاده کرد.
- در صورتی که این توابع موفق باشند، محتویات حافظه‌ی پردازه دور انداخته می‌شوند و قسمت‌های فایل اجرایی (کد و داده) در حافظه قرار می‌گیرد و اجرای آن شروع می‌شود.

```
int main(void)
{
     int var = 0;
     if (fork() == 0) {
          execlp("firefox", "firefox", NULL);
          printf("B\n", var);
          return 1;
     }
     printf("A\n", var);
     return 0;
}
```

- در مثال بالا، پس از فراخوانی تابع ()fork پردازه‌ی جدیدی ایجاد می‌شود. در پردازه‌ی پدر A چاپ می‌شود. در پردازه‌ی فرزند برنامه‌ی firefox اجرا می‌گردد. دقت کنید که با فراخوانی یکی از توابع خانواده‌ی exec (و در صورت اجرای موفق آن) حافظه‌ی پردازه کاملا عوض می‌شود و دستورات بعدی پردازه‌ی اول اجرا نمی‌گردد؛ در مثال زیر، B چاپ نمی‌شود مگر اینکه دستور execl ناموفق باشد (مثلا فایل اجرای firefox) موجود نباشد.

خاتمه‌ی پردازه‌ها

یک پردازه با فراخوانی سیستمی مناسب می‌تواند درخواست خاتمه را به سیستم عامل انتقال دهد:

- فراخوانی سیستمی مثل exit().
- با برگشت از تابع main (به صورت خودکار فراخوانی سیستمی exit() فراخوانی می‌گردد).
- وجود خطا یا درخواست پردازه‌ی مجاز دیگری می‌تواند منجر به خاتمه‌ی یک پردازه نیز شود.

خاتمه‌ی پردازه‌ها: کد برگشتی

مقداری که توسط تابع main پردازه‌ای برگشت داده می‌شود یا مقداری که به تابع exit() فرستاده می‌شود، کد برگشتی آن پردازه است. به صورت قراردادی، معمولا کد برگشتی صفر به مفهوم اجرای موفق و کد برگشتی غیر صفر به مفهوم خطا است.

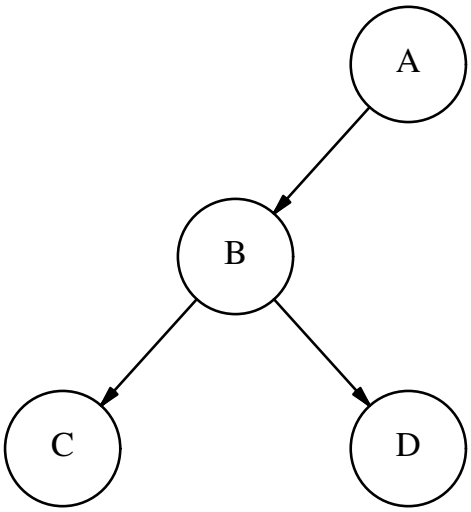
```
int main(void)
{
     if (fork() > 0) {
          exit(2);
     }
     return 1;
}
```

- در مثال بالا، کد برگشتی پردازه‌ی پدر ۲ و کد برگشتی پردازه‌ی فرزند ۱ است (چرا؟).

خاتمه‌ی پردازه‌ها: مرگ پدر

اگر پردازه‌ای خاتمه یابد، سیستم عامل باید تصمیم بگیرد با فرزندان آن پردازه چه کند.

- در برخی از سیستم‌های عامل اگر پردازه‌ای خاتمه یابد، سیستم عامل به صورت بازگشتی فرزندان آن پردازه را نیز خاتمه می‌دهد. به این رفتار خاتمه‌ی آبشاری (Cascading Termination) گفته می‌شود.



- در سیستم‌های عامل مبتنی بر Unix خاتمه‌ی آبشاری انجام نمی‌شود.

- تابع سیستمی wait() منتظر خاتمه‌ی یکی از فرزندان پردازه می‌شود و شناسه‌ی پردازه‌ی خاتمه یافته را بر می‌گرداند. اگر پردازه فرزندی نداشته باشد یا اگر این تابع با خطا متوقف شود مقدار منفی یک را بر می‌گرداند.

```
 1  int main(void)
 2  {
 3       int var = 0;
 4       if (fork() == 0) {
 5            sleep(1);
 6            printf("B\n", var);
 7            return 1;
 8       }
 9       wait(NULL);
10       printf("A\n", var);
11       return 0;
12  }
```

- به دلیل فراخوانی تابع wait() در مثال بالا ابتدا در پردازه‌ی فرزند B چاپ می‌شود و سپس در پردازه‌ی پدر (بعد از خاتمه‌ی پردازه‌ی فرزند و در نتیجه پایان فراخوانی wait()) A چاپ می‌شود.

انتظار برای خاتمه‌ی فرزندان: علت مرگ

بعد از خاتمه‌ی یک پردازه، سیستم عامل همه‌ی اطلاعات مربوط به آن پردازه را دور نمی‌اندازد. اطلاعاتی مثل علت مرگ (خطای حافظه، خاتمه‌ی موفق یا ...) و کد برگشتی را نگه می‌دارد.

- این اطلاعات برای پدر یک پردازه اهمیت دارد. برای نمونه، اگر پردازه ناموفق باشد یا خطایی در اجرای آن رخ داده باشد، پردازه‌ی پدر شاید دوباره پردازه‌ای را ایجاد کند تا عمل را تکرار کند.
- پردازه‌ی پدر با ورودی فراخوانی سیستم wait() می‌تواند علت مرگ و کد برگشتی یک فرزند خاتمه یافته را بدست آورد.

```
 1  int main(void)
 2  {
 3       int pid, status;
 4       if (fork() == 0) {
 5            sleep(1);
 6            return 1;
 7       }
 8       pid = wait(&status);
 9       return 0;
10  }
```

- در مثال بالا، پس از فراخوانی wait()، متغیر status در پردازه‌ی پدر اطلاعات مرگ فرزند را نگه خواهد داشت.
- بعد از فراخوانی wait() توسط پدر یک پردازه‌ی خاتمه یافته، اطلاعات مربوط به آن توسط سیستم عامل دور انداخته می‌شود. در Unix، به پردازه‌ی خاتمه یافته‌ای که هنوز تابع wait() توسط پدرش برای آن پردازه فراخوانی نشده است، Zombie گفته می‌شود.
- اگر پدر پردازه‌ای بدون فراخوانی wait() برای فرزندش خاتمه یابد، وضعیت فرزندان آن پردازه به Orphan (یتیم) تغییر می‌کند؛ معمولا پردازه‌های Orphan توسط پردازه‌ی init به ارث برده می‌شوند.

در یک سیستم عامل زمانبندهای (Scheduler) مختلفی وجود دارند.

- زمانبند دراز مدت (Long Term Scheduler یا Job Scheduler):

از بین Job-های آماده‌ی اجرا، یکی را انتخاب و اجرا می‌کند. Job-های قابل اجرا در صف Job Queue قرار دارند.

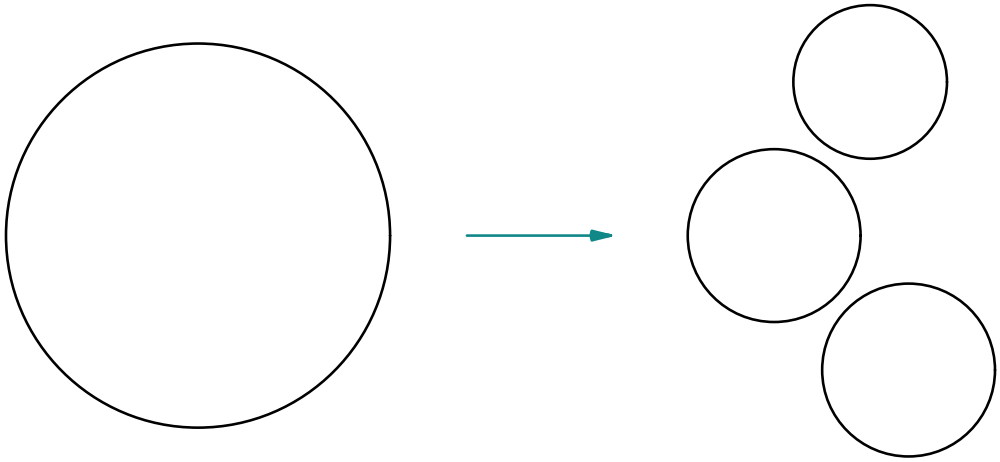
- زمابند پردازنده یا کوتاه مدت (CPU Scheduler یا Short Term Scheduler):

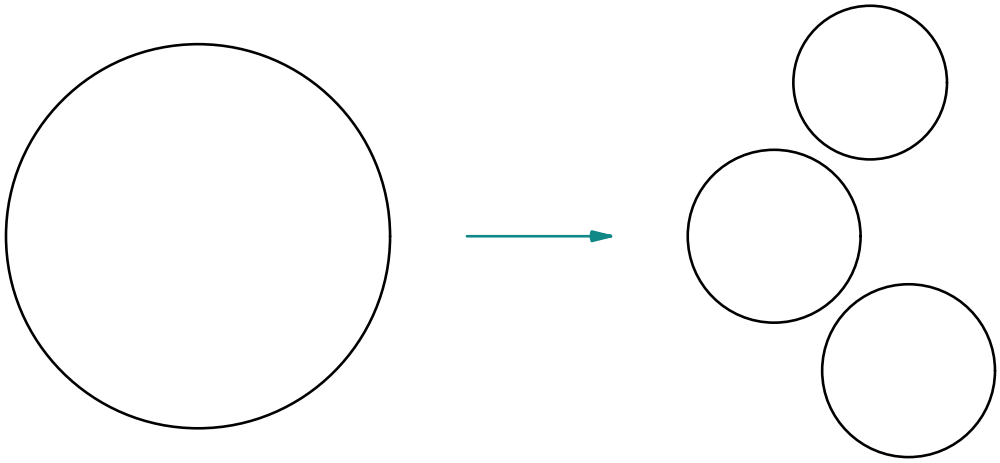
از بین پردازه‌هایی که در وضعیت آماده‌باش قرار دارند یکی را انتخاب و اجرا می‌کند. پردازه‌های آماده‌باش در صف آماده‌باش (Ready Queue) قرار دارند.

- زمانبند میان مدت (Medium Term Scheduler):

اگر منابع سیستم عامل (مثل حافظه) برای اجرای پردازه‌های در حال اجرا کافی نباشد، این زمانبندی برخی از پردازه‌ها را از حافظه‌ی اصلی به حافظه‌ی کمکی (Swap) انتقال می‌دهد تا پردازه‌های باقی مانده بدون محدودیت منابع اجرا شوند و پردازه‌های انتقال یافته بعدا اجرا گردند.

گاهی طراح یک نرم‌افزار، آن را به چند پردازه تقسیم می‌کند. دلایل مختلفی برای این کار وجود دارد:

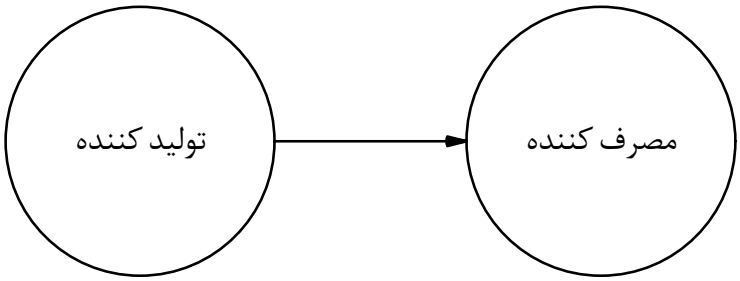




- با شکستن برنامه به قسمت‌های کوچک‌تر طراحی بهتر برنامه می‌شود (Modularity افزایش می‌یابد). در نتیجه، پیاده‌سازی، تغییر و نگهداری برنامه راحت‌تر خواهد شد.
- هر پردازه‌ی تک‌ریسه‌ای (در مورد ریسه‌ها در قسمت‌های بعد مطالبی را خواهیم دید) فقط از یک پردازنده استفاده می‌کند. با شکستن یک برنامه به چند پردازه، می‌توان از چند پردازنده (یا چند هسته‌ی یک پردازنده) به صورت همزمان استفاده کرد و در نتیجه سرعت اجرای برنامه را افزایش داد.
- برخی از عملیات ورودی یا خروجی منجر به توقف کل پردازه تا زمان خاتمه‌ی عملیات می‌شوند. با استفاده از چند پردازه، زمانی که برخی از پردازه‌ها منتظر هستند، بقیه می‌توانند از پردازنده استفاده کنند.
- استفاده از چند پردازه امکان انجام چند کار مختلف را می‌دهد.

مثال برای تعامل بین پردازه‌ها

- مثال‌های زیادی برای تعامل بین پردازه‌ای وجود دارد.
- مسئله‌ی تولید کننده و مصرف کننده در بسیاری از کاربردها ظاهر می‌شود.
- در این مسئله، یک پردازه داده‌ای را تولید می‌کند و پردازه‌ی دیگری داده‌های تولید شده را مصرف می‌کند.



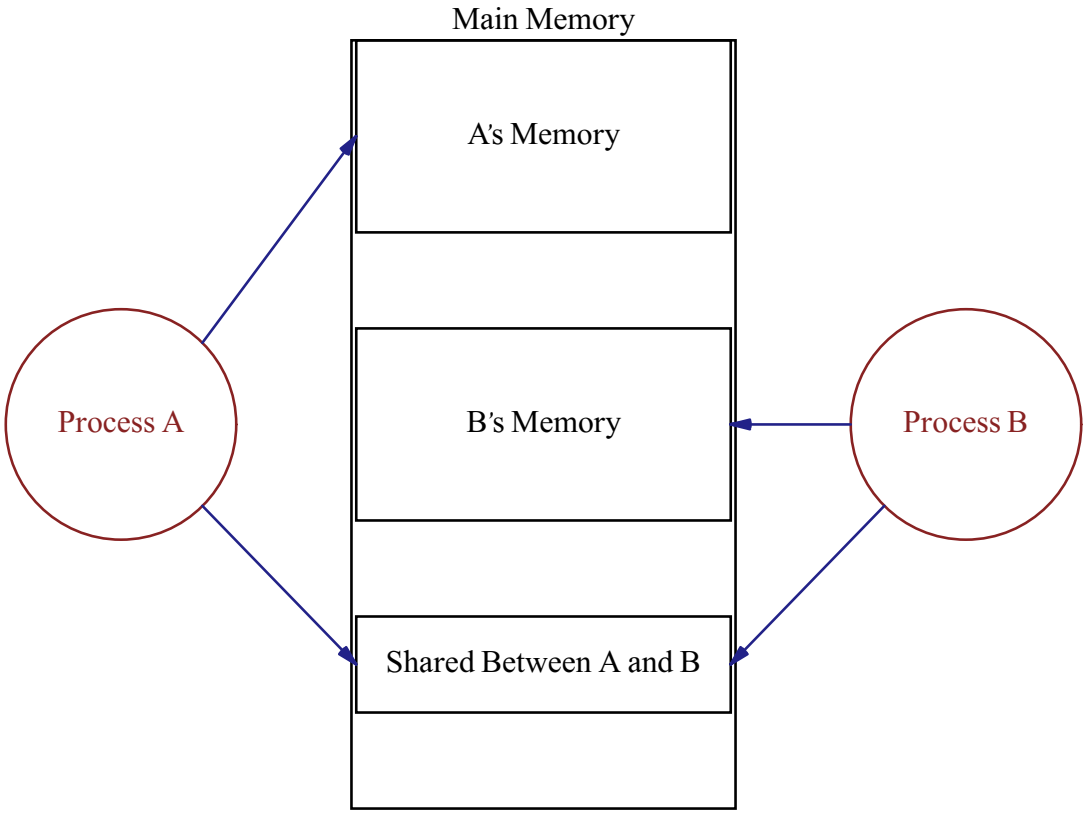
- کامپایلرها (خروجی پیش‌پردازشگر به کامپایلر و خروجی کامپایلر به اسمبلر داده می‌شود) و ابزارهای حروفچینی در یونیکس مثالی از این شیوه‌ی ارتباط بین پردازه‌ها هستند.

نیاز به انتقال اطلاعات بین پردازه‌ها

پس از شکستن یک برنامه به تعدادی پردازه، لازم است این پردازه‌ها با انتقال اطلاعات با هم ارتباط داشته باشند. به این روش ارتباط Interprocess Communication (به صورت مختصر IPC) گفته می‌شود. روش‌های مختلفی برای انتقال اطلاعات بین پردازه‌ها وجود دارد. بیشتر این روش‌ها در یکی از دو دسته‌ی زیر قرار می‌گیرند:

- حافظه‌ی مشترک
- تبادل پیغام

- روش حافظه‌ی مشترک (Shared Memory): قسمتی از حافظه‌ی دو پردازه به اشتراک گذاشته می‌شود.



- متغیرهایی که در حافظه‌ی مشترک دو پردازه قرار بگیرند توسط آن پردازه‌ها قابل خواندن و تغییر هستند.

مثالی از حافظه‌ی مشترک

در مثال زیر، حالت ساده‌ای از ارتباط بین پردازه‌ای بین یک تولید کننده و یک مصرف کننده نمایش داده شده است. فرض کنید N یک ثابت باشد و متغیرهای head و tail و آرایه‌ی Q در حافظه‌ی مشترک باشند.

```
Item Q[N];
int head;
int tail;
```

پردازه‌ی تولید کننده تابع producer را اجرا می‌کند که در آن متغیرهایی از نوع Item تولید می‌شوند. اگر آرایه‌ی Q پر باشد، حلقه‌ی while منتظر می‌ماند تا حداقل یک عنصر توسط مصرف کننده برداشته شود.

```
void producer(void)
{
     while (1) {
          Item item = produce_item();
          while ((head + 1) % N == tail)
               ;
          Q[head] = item;
          head = (head + 1) % N;
     }
}
```

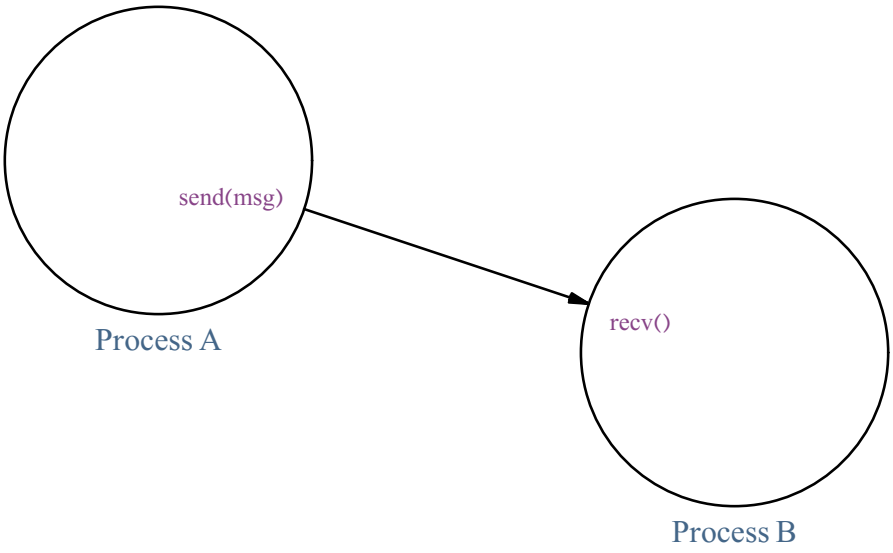
مثالی از حافظه‌ی مشترک: مصرف کننده

پردازه‌ی مصرف کننده تابع consumer را اجرا می‌کند که Item-های تولید شده توسط تولید کننده را بر می‌دارد و مصرف می‌نماید. اگر آرایه‌ی Q خالی باشد، حلقه‌ی while منتظر می‌ماند تا حداقل یک عنصر توسط تولید کننده اضافه شود.

```
void consumer(void)
{
     while (1) {
          while (head == tail)
               ;
          item = Q[tail];
          tail = (tail + 1) % N;
          consume_item(item);
     }
}
```

در تبادل پیغام ارتباط بین دو پردازه با فرستادن و دریافت پیغام انجام می‌شود. دو عمل پایه در تبادل پیغام وجود دارند.

- فرستادن (send): پیغامی را می‌فرستد.
- فرستادن (receive): پیغامی را دریافت می‌کند.



یک مثال خوب از تبادل پیغام که قبلا بررسی کرده‌ایم، ساختار میکروکرنل سیستم عامل است.

مثالی از تبادل پیغام

در قطعه کد زیر برای مسئله‌ی تولید کننده و مصرف کننده از تبادل پیغام استفاده شده است. در این مثال تابع send() در تولید کننده پیغامی را می‌فرستد و تابع recv() در مصرف کننده پیغامی که تولید کننده فرستاده است را دریافت می‌کند.

```
void producer(void)
{
     while (1) {
          Item item = produce_item();
          send(item, dst);
     }
}
```

```
void consumer(void)
{
     while (1) {
          Item item = recv(src);
          consume_item(item);
     }
}
```

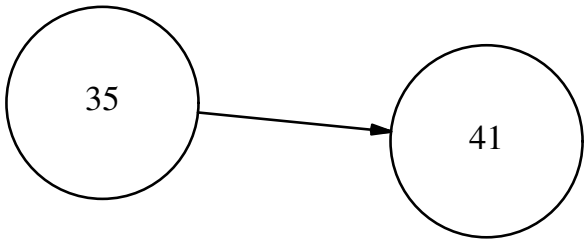
دقت کنید که در این مثال حافظه‌ی پردازه‌ی تولید کننده و مصرف کننده اشتراکی ندارد.

اگر تعداد پردازنده‌ها کم باشد، حافظه‌ی مشترک معمولا سریع‌تر از تبادل پیغام است.

- اگر چه ظاهرا در حافظه‌ی مشترک نیازی به اضافه کردن عملیات جدیدی به برنامه‌ها نیست، حافظه‌ی مشترک به علت مشکلات همگام‌سازی (که در آینده مطالعه خواهیم کرد) پیچیدگی‌هایی دارد.
- لینک ارتباطی بین پردازه‌ها در تبادل پیغام را می‌توان به روش‌های مختلفی پیاده‌سازی کرد، مثل حافظه‌ی مشترک، شبکه، سخت‌افزار.

- کتابخانه‌های تبادل پیغام با توجه به ویژگی‌هایشان به چند دسته تقسیم می‌شوند که در ادامه معرفی می‌گردند.
- فرستادن و دریافت پیغام‌ها می‌تواند در دو حالت مستقیم یا غیر مستقیم باشد.

- ارتباط مستقیم (Direct Communication) دو حالت متقارن (Symmetric) و نامتقارن (Asymmetric) را دارد.
- در حالت متقارن فرستنده و گیرنده باید پردازه‌ی مقصد و مبدأ پیغام را مشخص کنند.



برای ارسال پیغام یک پیغام از پردازه‌ی ۳۵ به پردازه‌ی ۴۱، در عمل فرستادن باید شناسه‌ی پردازه‌ی گیرنده مشخص شود.

```
send(41, msg);
```

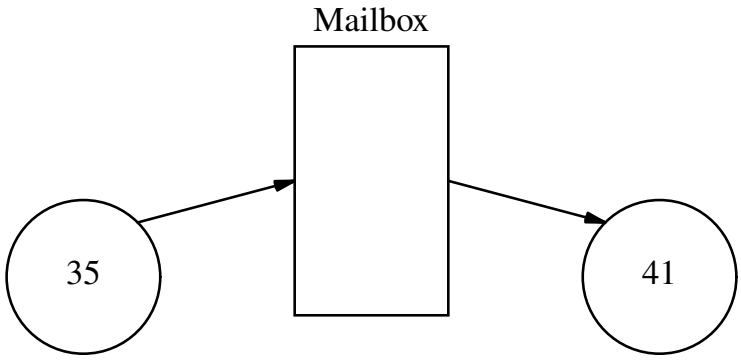
در حالت متقارن، گیرنده نیز باید مشخص کند از چه پردازه‌ای پیغام دریافت می‌شود.

```
recv(35, &msg);
```

در حالت نامتقارن، فرستنده مثل قبل مشخص می‌کند به چه پردازه‌ای پیغام را بفرستد اما گیرنده یک پیغام را از هر مقصدی دریافت می‌کند.

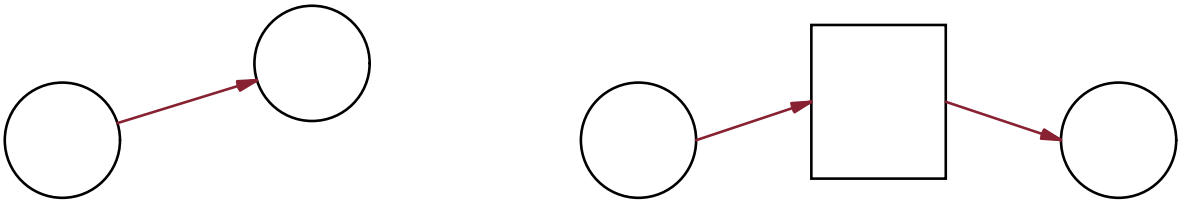
```
recv(&pid, &msg);
```

- در حالت غیر مستقیم، فرستنده‌ها پیغام‌ها را برای صندوق‌های پستی (Mailbox) یا درگاه‌ها (Port) می‌فرستند و گیرنده‌ها نیز این همین صندوق‌های پستی پیغام‌ها را دریافت می‌کنند.



- بنابراین هم فرستنده و هم گیرنده به جای اینکه شناسه‌ی پردازه‌ی طرف دیگر را مشخص کنند، فقط شناسه‌ی درگاه یا صندوق پستی را مشخص می‌نمایند.

مقایسه‌ی ارتباط مستقیم و غیر مستقیم

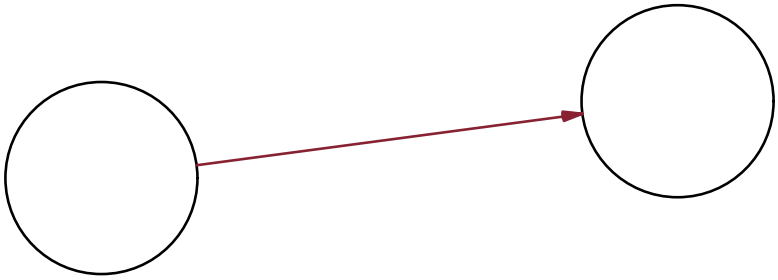


- در حالت مستقیم فقط یک راه ارتباطی بین دو پردازه می‌تواند وجود داشته باشد. اما در حالت غیر مستقیم بین دو پردازه می‌توان بیش از یک صندوق برای ارتباط استفاده کرد (مثلا برای یکی برای انتقال داده‌ها، یک برای دستورات، یکی برای داده‌های مهم و یکی برای خطاها).
- در حالت مستقیم، هر لینک ارتباط فقط از یک فرستنده و به یک گیرنده است. اما در حالت غیر مستقیم چند فرستنده می‌توانند به یک صندوق پستی پیغام بفرستند و چند گیرنده از آن پیغام دریافت کنند.
- در حالت مستقیم باید پردازه‌ی دریافت کننده موجود باشد و فرستنده باید از شناسه‌ی پردازه‌ی دریافت کننده آگاه باشد. اما در حالت غیر مستقیم پردازه‌های فرستنده پیغام‌ها را به صندوق‌های پستی می‌فرستند و لازم نیست از شناسه‌ی پردازه‌های دریافت کننده را بدانند (حتی ممکن است هنوز پردازه‌های دریافت کننده موجود نباشند).

حالت انتظار و بدون انتظار

عملیات فرستادن و دریافت می‌توانند در دو حالت با انتظار (Blocking) و بدون انتظار (Non-blocking) باشند.

- در فرستادن با انتظار، عمل send() تا وقتی که پیغام توسط گیرنده انجام شود منتظر می‌ماند.
- در فرستادن بدون انتظار، پیغام فرستاده شده در صف پیغام‌ها قرار می‌گیرد و فرستنده منتظر دریافت پیغام توسط گیرنده نمی‌شود.



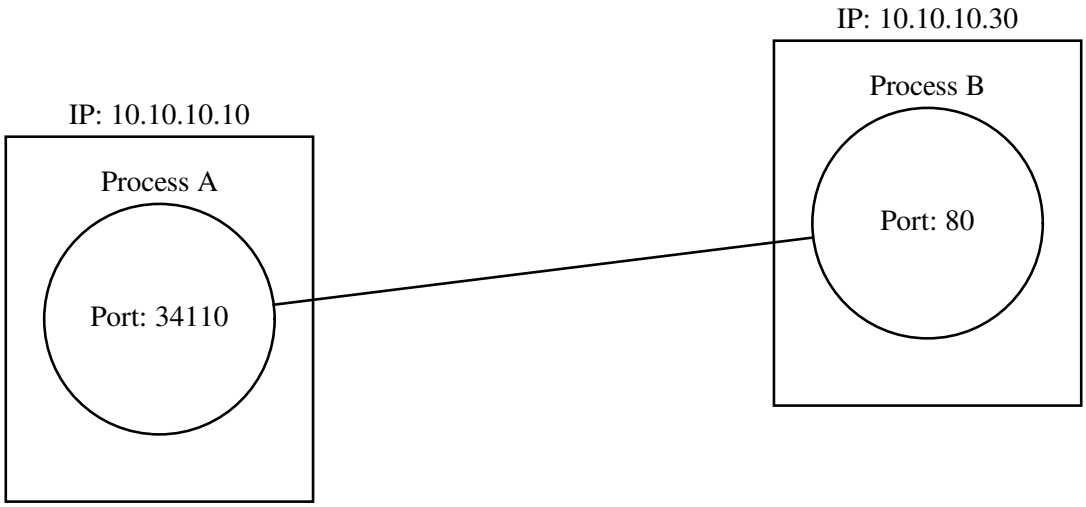
به صورت مشابه برای عمل دریافت نیز داریم:

- در دریافت با انتظار، عمل recv() منتظر می‌شود یک پیغام دریافت شود (اگر پیغامی قبلا فرستاده شده باشد و در صف پیغام‌ها قرار داشته باشد آن پیغام به دریافت کننده داده می‌شود و انتظار خاتمه می‌یابد).
- در دریافت بدون انتظار، اگر پیغامی در صف پیغام‌ها باشد یا فرستنده‌ای منتظر دریافت پیغامش باشد، پیغام به گیرنده داده می‌شود. در غیر این صورت (پیغامی موجود نباشد)، گیرنده منتظر نمی‌ماند و با خطا خاتمه می‌یابد.

در ادامه چند روش برای انتقال اطلاعات بین پردازه‌ها را خواهیم دید:

- ساکت‌های شبکه
- فراخوانی توابع از راه دور
- لوله
- سیگنال‌ها

با استفاده از ساکت‌های شبکه (Network Sockets) می‌توان بین دو پردازه در کامپیوترهای مجزا در شبکه اطلاعات را انتقال داد.



- در شبکه‌ی اینترنت هر کامپیوتری که به شبکه متصل است شناسه‌ای به نام آدرس IP (IP Address) دارد.
- هر کامپیوتری تعدادی درگاه دارد؛ هر درگاه با یک عدد بین صفر تا 65536 مشخص می‌شود.
- یک اتصال بین دو پردازه در دو ماشین در شبکه‌ی اینترنت با پنج شناسه مشخص می‌شود: آدرس IP مبدأ، آدرس IP مقصد، درگاه مبدأ، درگاه مقصد و پروتکل.
- با یک اتصال می‌توان داده‌ها را در قالب تعدادی بسته بین دو سر اتصال انتقال داد.

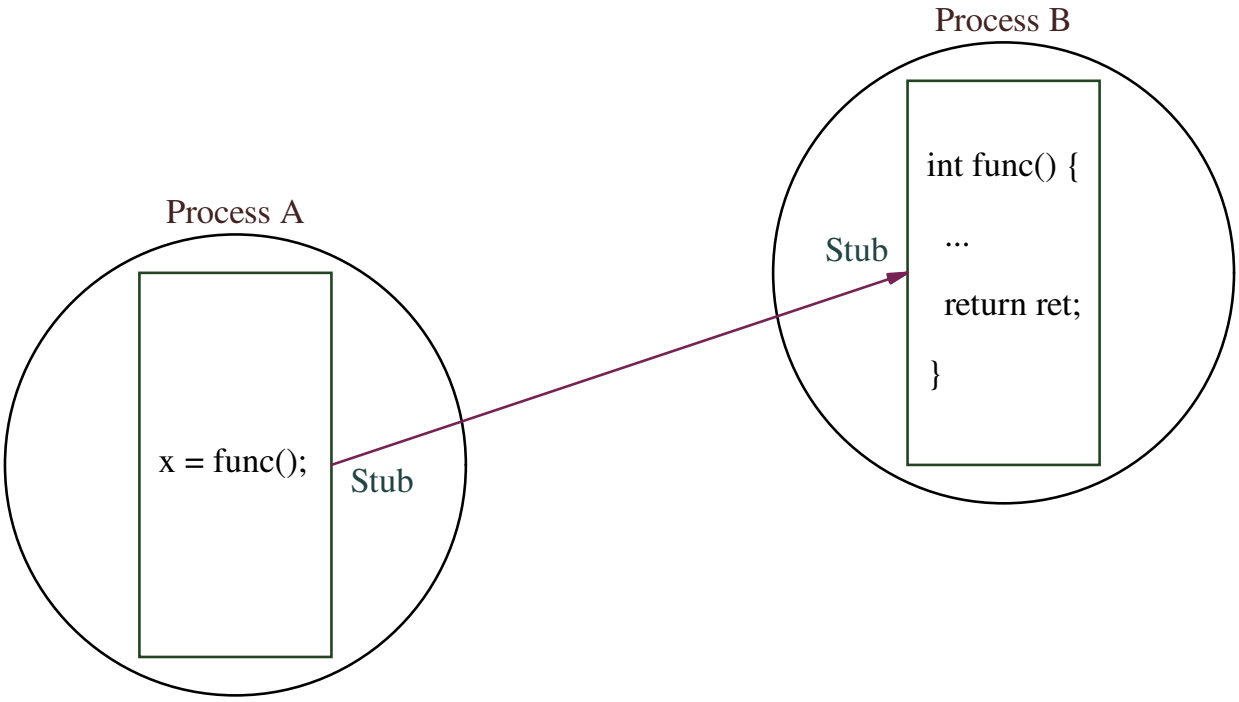
پروتکل‌های (لایه‌ی انتقال) اینترنت در دو دسته قرار می‌گیرند.

- در پروتکل‌های بدون اتصال (Connection-less)، بسته‌ها ممکن است گم شوند، با ترتیب متفاوت یا چند بار دریافت شوند. مثال پروتکل UDP (پروتکل DNS یک مثال از این پروتکل است).
- پروتکل‌های اتصال گرا، بسته‌ها دقیقا یک بار و با ترتیب ارسال دریافت می‌شوند. مثال پروتکل TCP (پروتکل HTTP یک مثال از این پروتکل است).

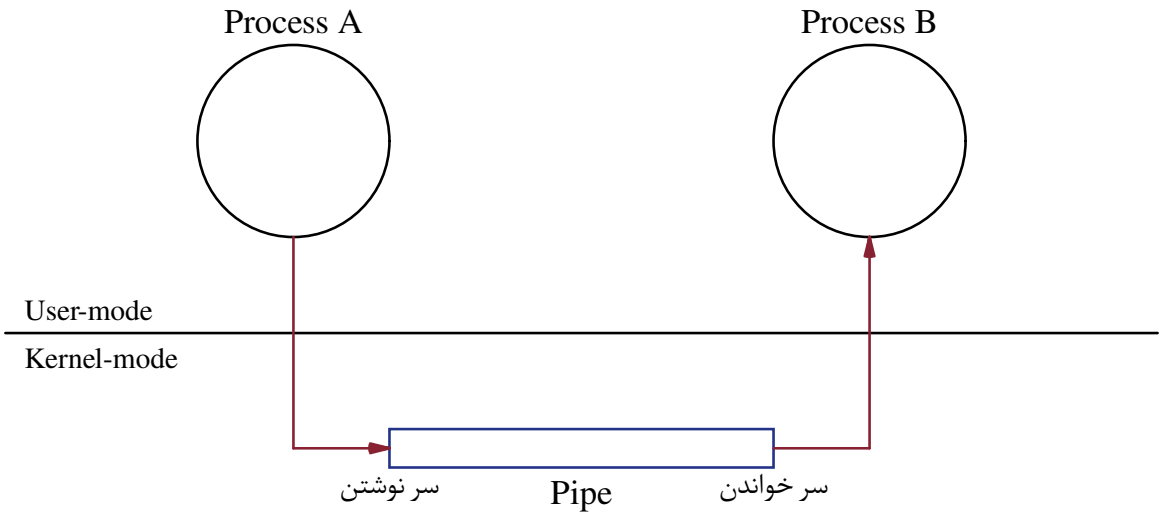
فراخوانی توابع از راه دور

در فراخوانی توابع از راه دور (Remote Procedure Call یا به صورت مختصر RPC) با فراخوانی تابعی در پردازه‌ای در کامپیوتر مبدأ، پیاده‌سازی آن تابع در پردازه‌ی دیگری در کامپیوتر دیگری اجرا می‌شود و خروجی آن برگشت داده می‌شود.

- در مثال زیر، پیاده‌سازی تابع func در پردازه‌ی B قرار دارد.



- با فراخوانی پردازه‌ی A، ورودی‌های تابع به شکلی تبدیل می‌شوند که از شبکه قابل انتقال باشد؛ به این کار Marshal کردن می‌گویند و این کار توسط قسمتی از کد به نام Stub انجام می‌شود.
- ورودی‌های تابع توسط Stub در پردازه‌ی B دریافت می‌شود و به حالت اولیه‌ی خود برگردانده می‌شود (به این کار Unmarshal کردن می‌گویند).
- سپس تابع در B فراخوانی می‌شود و پس از اتمام آن، خروجی تابع توسط Stub در پردازه‌ی B Marshal می‌شود.
- خروجی تابع در پردازه‌ی A دریافت می‌شود و توسط Stub به حالت اول برگردانده می‌شود و فراخوانی تابع func خاتمه می‌یابد.
- روش RPC در دنیای تجاری زیاد استفاده می‌شود. نمونه‌هایی از پروتکل‌های RPC موجود Java RMI، COBRA، JSON-RPC و SOAP هستند.



- لوله (Pipe) بافری (Buffer) در هسته است که با کمک آن دو پردازه می‌توانند با هم ارتباط داشته باشند.
- اطلاعاتی که به سر نوشتن نوشته می‌شود از سر خواندن خوانده می‌شود.

انتظار در لوله

پردازه‌ی نویسنده به لوله و پردازه‌ی خواننده از لوله در حالت‌هایی منتظر (Block) می‌شوند.

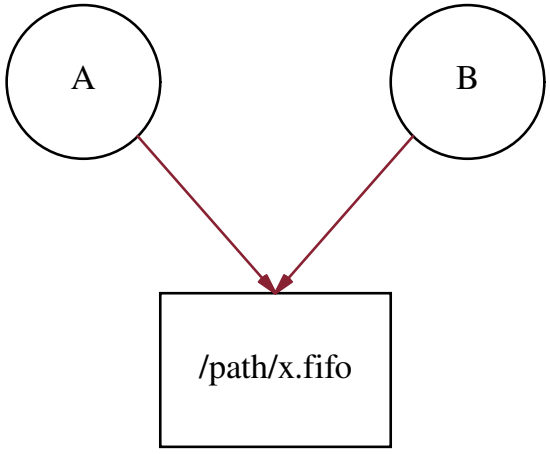
- حافظه‌ی لوله محدود است و اگر لوله پر شود نویسنده توسط سیستم عامل Block می‌شود تا وقتی که مقداری از اطلاعات لوله توسط خواننده خوانده شود و فضای خالی در آن ایجاد شود.
- در صورتی که لوله خالی باشد، گیرنده Block می‌شود تا داده‌ای به لوله نوشته شود.

مقایسه‌ی لوله با فایل‌های عادی

در مقایسه با استفاده از فایل‌های عادی، لوله مزیت‌هایی برای ارتباط بین پردازه‌ای دارد.

- چون داده‌ها همانطور که تولید می‌شوند، مصرف هم می‌شوند لازم نیست اطلاعات در دیسک ذخیره شود.
- به همین دلیل تأخیر پردازش اطلاعات تولید شده کمتر است و لازم نیست کل اطلاعات تولید شود و بعدا پردازش آن انجام شود.

لوله‌های با نام (Named Pipe یا FIFO) فایل‌هایی در فایل سیستم هستند.

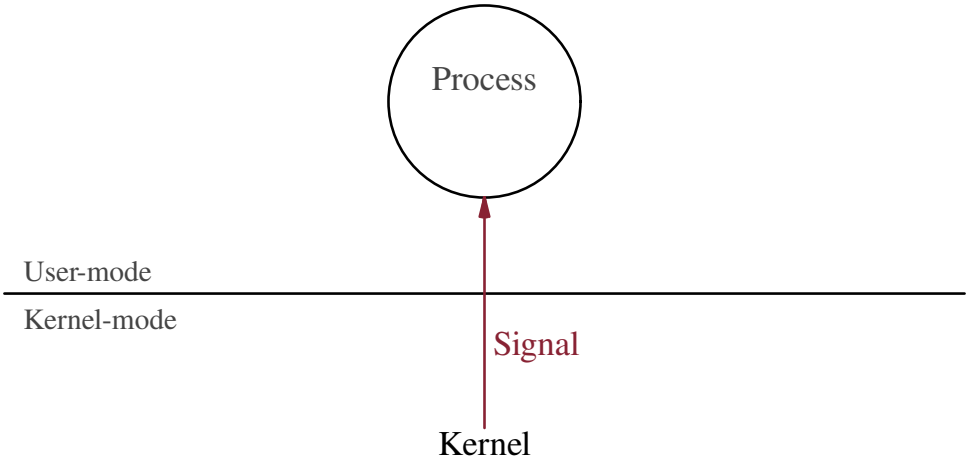


- سیستم عامل برای هر FIFO یک لوله تخصیص می‌دهد.

نسبت به لوله‌ی معمولی، لوله‌ی با نام تفاوت‌های زیر را دارند.

- با نام بین هر دو پردازه‌ای که اجازه‌ی دسترسی به فایل FIFO را دارند قابل استفاده است.
- در لوله‌های با نام ممکن است بیش از یک نویسنده یا خواننده از لوله استفاده کنند.

سیگنال‌ها (Signal) روشی برای آگاه کردن یک پردازه از رخدادهای خارجی است که می‌تواند برای IPC نیز استفاده شود.



- هر سیگنال با یک شناسه‌ی عددی دارد و با این شناسه، سیگنال‌ها از هم متمایز می‌شوند.
- سیگنال‌ها برای کاربردهای مختلفی استفاده می‌شوند مثال‌هایی بیان می‌شوند:

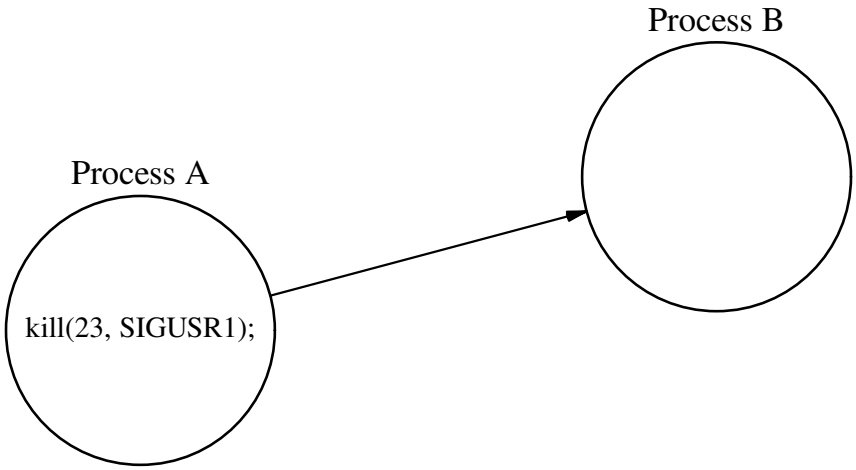
  سیگنال SIGCHLD.

  سیگنال‌های SIGINT، SIGTERM، SIGKILL.

  سیگنال SIGSEGV.

  سیگنال SIGALRM (در کنار تابع alarm()).

پردازه‌ها می‌توانند از سیگنال‌ها استفاده کنند تا پردازه‌ای را از رخدادی آگاه کنند.



- سیگنال‌هایی مثل SIGUSR1 و SIGUSR2.
- می‌توان از دستور kill یا در زبان برنامه نویسی C با تابع kill به پردازه‌ای یک سیگنال فرستاد.
- با فراخوانی‌های سیستمی مناسب، هر پردازه می‌تواند برای هر سیگنال مشخص کند در صورت رخداد چه عملی انجام شود: عمل پیش‌فرض، خاتمه‌ی پردازه، نادیده گرفتن یا فراخوانی یک تابع.