# آشنایی با مدیریت فرایندها در لینوکس

## ۱.۱-هدف

آشنایی با:

- فرایند[1] و حالات مختلف آن
- انواع روش‌های ایجاد فرایند جدید در لینوکس و خاتمه آن
- اجرای فرایندها در پس‌زمینه

## ۲.۱-پیش‌آگاهی

درک عملی فرایندها در لینوکس، اضافه بر اینکه به مفهومی مهم در سیستم‌عامل عینیت می‌بخشد، سبب راهبری بهتر و بیشتر بر سیستم‌عامل لینوکس خواهد شد و به عنوان مثال در مواقعی مانند از کارافتادن برنامه، یا حتی تغییر اولویت فرایندها می‌توان از آن بهره برد. به ازای اجرای هر برنامه ممکن است یک یا بیشتر فرایند فعال شود.

### ۱.۲.۱-فرایندها در لینوکس

در زمان روشن شدن سیستم، ابتدا هسته[2] شروع به کار می‌کند. هسته، متولی آغاز اولین فرایند است که عبارتست از فرایند init. این فرآیند مسؤل تمام فرایندهای دیگر است. این فرایند فرایند دیگر را به عنوان فرایند فرزند[3] اجرا

[1] Process
[2] Kernel
[3] Child

خواهد کرد. برای مثال، mingetty از init شروع می‌شود، که مسؤل گشودن یک پوسته ورود[4] است. از mingetty، فرایند bash آغاز به کار می‌کند که اجازه کار با خط فرمان را برای کاربران لینوکس فراهم می‌کند. بنابراین یک رابطه پدر و فرزندی بین فرایندهای لینوکس وجود دارد. Init اولین فرایند است که mingetty از آن مشتق می‌شود سپس bash به عنوان فرزند از mingetty به وجود می‌آید و زین‌پس هر دستوری که در خط فرمان داده شود به عنوان فرزندی از bash به حساب می‌آید.

لینوکس یک صف برای فرایندهای آماده به اجرا نگه می‌دارد. لینوکس به‌طور پیش‌فرض به هریک از فرایندها بازه‌های زمانی مساوی اختصاص می‌دهد، اما اگر فرایندی نیاز به میزان بیشتری از زمان نسبت به سایر فرایندها داشته باشد، بایستی این کار را با فراخوانی تابعی به نام nice برای افزایش یا حتی کاهش میزان زمان مجاز برای یک فرایند فراخوانی کرد.

گاهی ممکن است نیاز باشد که یک فرایند را متوقف کنیم. به عنوان مثال ممکن است یک فرایند دیگر به هیچ سیگنالی پاسخ ندهد و نتوان با آن کار کرد یا این‌که به گونه‌ای رفتار کند که به سایر فرایندها آسیب برساند. در این صورت هسته‌ی سیستم عامل به فرایند پدر مربوطه خواهد گفت که لازم است فرزند آن را متوقف کند. در شرایط معمول، فرایند پدر تا زمان حیات فرایند فرزند، در سیستم وجود دارد، اما در شرایط غیرمعمول ممکن است فرایند فرزند در حالی در سیستم باقی مانده باشد که پدر متوقف شده باشد، در این حالت نمی‌توان فرایند فرزند را متوقف کرد و گوییم به حالت Zombie یا جادویی رفته است. بدین ترتیب دیگر نمی‌توان از خط فرمان فرایند Zombie را متوقف کرد و تنها راه، شروع مجدد[5] سیستم است.

اغلب ظهور فرایند Zombie، سایر فرایندها را نیز تحت تأثیر قرار می‌دهد. علت چنین رویدادی نیز معمولا به اشتباهات برنامه نویسی برمی‌گردد که چنین فرایندهایی را ایجاد می‌کند. سایر حالات دیگری که فرایند می‌تواند در آن‌ها قرار گیرد را می‌توان با دستور ps aux مشاهده کرد.

### ۲.۲.۱–حالات مختلف فرآیند

همانطور که در آزمایش پیش مشاهده شد، دستور ps برای مشاهده وضعیت فرایندها استفاده می‌شود. یکی از مشخصه‌هایی که دستور ps –al برای یک فرایند نشان می‌دهد، Stat نام دارد که حالت فرایند را به صورت یک کد سه حرفی نشان می‌دهد. حرف اول این کد یکی از حروف زیر است:

- R: برای فرایندهای در حال اجرا (Running)
- S: برای فرایندهای در منتظر (Waited)
- Z: برای فرایندهای جادویی (Zombie)
- T: برای فرایندهایی که متوقف (stopped) یا ردگیری (Trace) شده‌اند (یا به اصطلاح آماده‌ی اجرا هستند)

[4] Login shell
[5] Restart

- D: برای فرایندهایی که در حالت انتظار بدون بازپس‌گیری[6] یا انتظار در حافظه‌ی جانبی هستند.

اگر حرف دوم این کد W باشد، نشان‌دهنده‌ی آن است که فرایند مورد نظر هیچ صفحه‌ای در حافظه‌ی اصلی ندارد و به‌طور کامل مبادله (Swap) شده است.

حرف سوم در مورد زمانبندی و اولویت فرایندهاست. اگر این حرف N باشد، نشان دهنده‌ی این است که ارزش مطلوب (Nice Value) برای فرایند بیشتر از صفر است (این نوع فرایند اولویت بیشتری دارد و وقت بیشتری را می‌تواند از پردازنده بگیرد).

در سیستم‌عامل لینوکس، یک فرایند می‌تواند ۹ حالت مختلف داشته باشد. این حالات در شکل و جدول زیر آمده‌اند. لازم به ذکر است که منابع مختلف، با دیدگاه‌های خلاصه و تفصیلی به حالات فرایند در لینوکس، تعدادهای کمتر و بیشتر هم برای حالات فرایند بیان کرده‌اند.

**جدول ۱-۱- حالات مختلف فرایندها**

| ردیف | نام | شرح |
|---|---|---|
| ۱ | اجرا در فضای کاربر | فرایند موقع اجرای عملیات عادی خود در این حالت قرار دارد |
| ۲ | اجرا در فضای هسته | فرایند، هنگام استفاده از سرویس‌های هسته در این حالت قرار دارد. مانند اجرای فراخوانی سیستم و روال خدماتی وقفه‌ها |
| ۳ | آماده به اجرا در حافظه‌ی اصلی | فرایند در حال اجرا نیست ولی در حافظه‌ی اصلی قرار دارد و منتظر اختصاص وقت پردازنده است تا اجرا شود |
| ۴ | متوقف (Sleep) در حافظه‌ی اصلی | فرایند در حافظه‌ی اصلی است و منتظر پایان یافتن عملی مانند ورودی- خروجی است تا دنباله‌ی اجرایش را از سربگیرد |
| ۵ | آماده به اجرا در حافظه‌ی جنبی | فرایند آماده‌ی اجراست ولی به دلیل کمبود حافظه، به حافظه‌ی جانبی منتقل شده است و قبل از آن‌که وقت پردازنده به آن تخصیص داده شود، باید به حافظه‌ی اصلی منتقل شود |
| ۶ | متوقف در حافظه‌ی جنبی | مشابه به حالت ۴ است اما فرایند به علت کمبود حافظه، توسط هسته به حافظه‌ی جانبی منتثل شده است تا حافظه‌ی کافی برای فرایندهای در حال اجرا فراهم گردد |
| ۷ | قبضه‌شده (Preempted) | فرایند از حالت اجرا در فضای کاربر به حالت اجرا در فضای هسته منتقل می‌گردد، ولی در این هنگام زمان اختصاص یافته به آن پایان می‌یابد و زمانبند فرایندها، فرایند دیگری را برای اجرا انتخاب می‌کند |
| ۸ | ایجاد | هر فرایندی که ایجاد می‌شود، نخست در این حالت قرار می‌گیرد، در این حالت نه در حال اجراست و نه در حال توقف. این حالت نقطه‌ی شروع تمام فرایندها (به‌جز فرایند با pid صفر) است |

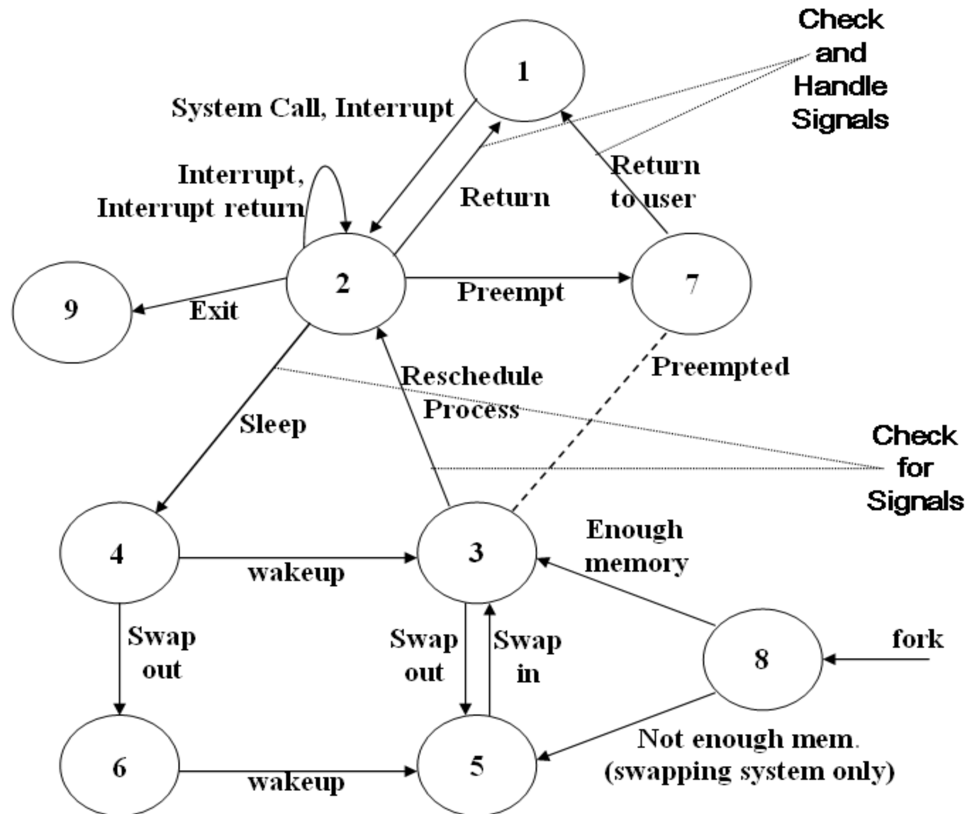
[6] Preemption

| ۹ | جادویی (Zombie) | هر فرایند توسط تابع exit، با به‌جا گذاشتن اطلاعاتی برای پدر خویش (فرایندی که آن را ایجاد کرده است)، به کار خویش خاتمه می‌دهد. هسته، تمام منابع، به‌جز کد خروج فرایند را از این نوع فرایندها می‌گیرد. این منابع شامل حافظه، پردازنده و ... است.<br>اگر فرایندی، فرایند فرزندی را ایجاد کند، مسؤلیت اتمام آن را نیز برعهده دارد. بنابراین در صورتی که فرایند فرزند، زودتر خاتمه یابد، کماکان در سیستم وجود خواهد داشت (حالت جادویی) تا اینکه یا فرایند پدر با استفاده از exit code فرایند فرزند، در یک فراخوانی سیستمی به نام wait یا waitpid، آن را مختومه کند یا اینکه فرایند پدر خود به صورت خودکار خاتمه یابد و تمام منابع اختصاصی‌اش آزاد شود. این فراخوانی‌های سیستمی باعث می‌شوند، پدر از وضعیت (کد) خروج فرزندش مطلع شود.<br>اگر پدر منتظر دریافت کد خروج فرزندش نیز نشود و کارش را به پایان برساند در حالی‌که هنوز حیات فرزند ادامه دارد، فرزند بی‌سرپرست (orphan) می‌شود. مدیریت این نوع فرایندها به عهده‌ی سیستم‌عامل است. |
|---|---|---|

وقتی فرایند پدر با فراخوان سیستم[۷] fork، فرایند جدیدی ایجاد می‌کند، فرایند ایجاد شده، بسته به مقدار حافظه‌ی آزاد سیستم، وارد یکی از مراحل ۳ یا ۵ خواهد شد. برای سادگی فرض کنید فرایند وارد مرحله‌ی ۳ می‌گردد. وقتی زمانبند فرایندها به صورت تصادفی آن‌را برای اجرا برگزید، فرایند وارد مرحله‌ی اجرا در فضای هسته گشته و در این هنگام کار فراخوان fork به اتمام می‌رسد. پس از این مرحله ممکن است فرایند به مرحله‌ی ۱ برود.

بعد از یک برش زمانی، به علت وقوع وقفه‌ی ساعت سیستم، فرایند به مرحله‌ی ۲ باز می‌گردد (چون تمام فرایندها این وقفه را دریافت می‌کنند). هسته از این وقفه‌ی ساعت برای زمانبندی و اولویت‌بندی فرایندها استفاده می‌کند. پس از این زمانبندی، ممکن است فرایند دیگری برای اجرا انتخاب گردد. در این هنگام فرایند قبلی وارد حالت ۷ می‌گردد و منتظر می‌ماند تا برای اجرای دوباره انتخاب گردد. اگر فرایند در حال اجرا در فضای کاربر، بخواهد از سرویس‌های هسته استفاده کند، وارد مرحله‌ی ۲ می‌گردد. اگر این سرویس، کاری مانند ورودی- خروجی باشد، فرایند وارد مرحله‌ی ۴ می‌شود و خود را متوقف می‌سازد. این توقف تا زمانی ادامه می‌یابد که به واسطه‌ی اتمام کار ورودی- خروجی، وقفه‌ای به پردازنده برسد و سرویس روتین وقفه، فرایند مربوطه را بیدار کند. این کار باعث می‌شود تا فرایند به مرحله‌ی ۳ باز گردد و منتظر شود تا هسته آن‌را زمانبندی کند.

[۷] فراخوان سیستم آن دسته از توابعی هستند که از آن‌ها برای استفاده از امکانات هسته‌ی سیستم‌عامل فراخوانی می‌کنیم. کد این برنامه‌ها در داخل هسته قرار دارد و موقع استفاده از آن‌ها، کدشان ضمیمه برنامه نمی‌شود.




جدول ۲-۱ - نمودار حالات مختلف فرایندها بر اساس شماره ردیف‌های جدول ۱

اگر فضای کافی برای فرایندهای در حال اجرا، در حافظه وجود نداشته باشد، فرایند مبادله کننده[8] (با شماره‌ی صفر) بعضی از فرایندها را بر روی دیسک منتقل می‌کند و به این ترتیب فرایند را وارد مرحله‌ی ۵ می‌کند. همین اتفاق ممکن است برای فرایندهای متوقف در حافظه (حالت ۴) نیز اتفاق بیفتد. وقتی کار فرایند به پایان رسید، وارد مرحله‌ی ۹ می‌شود.

## ۳.۲.۱-ایجاد فرایند جدید

برای ایجاد یک فرایند جدید، از یک فراخوان سیستم به نام fork استفاده می‌شود. خروجی fork از جنس pid_t (در حقیقت یک عدد صحیح) است. این خروجی همان شناسه‌ی فرایند (شماره‌ی فرایند PID) است. با احضار فراخوان سیستم fork، فرایند جدیدی ایجاد می‌شود که پدر آن، فرایندی است که fork را فراخوانده است. فرایند جدید در کنار بقیه‌ی فرایندهای موجود در سیستم قرار می‌گیرد و منتظر می‌شود تا وقت پردازنده به او اختصاص یابد. اما در این فرایند چه چیزی اجرا می‌شود؟ پاسخ این است که کد اجرایی این فرایند دقیقاً همانند پدرش است و همان چیزی را اجرا می‌کند که پدرش اجرا می‌کرده است، اما از نقطه‌ی فراخوانی به بعد. به عبارت دیگر از نقطه‌ی فراخوانی به بعد، کد پدر در دو خط موازی اجرا می‌شود، یکی توسط پدر و دیگری توسط فرایند فرزند که از آنجا به بعد با پدر یکسان است. فرایند فرزند، محیط متغییرها، پرونده‌های باز، شناسه‌های پرونده‌ی پدر را

[8] Swapper

به ارث می‌برد. البته تفاوت‌های اندکی نیز بین این دو فرایند وجود دارد؛ از جمله شماره‌ی فرایند (PID) و شماره‌ی فرایند پدر (PPID).

قطعه کد زیر را در نظر بگیرید:

```
Pid_t child_id;
child_id = fork();
printf("forked\n");
```

با اجرای این دستور، یک فرایند جدید ایجاد خواهد شد که دارای کدی به صورت فوق بوده، شمارنده‌ی برنامه‌ی[9] آن به دستورالعمل بعد از fork اشاره می‌کند. اکنون دو فرایند وجود دارند که دستورالعمل بعدی را اجرا خواهند کرد و در نتیجه کلمه‌ی fork دو بار چاپ خواهد شد. این یک روش چند نخی در لینوکس است که مدت‌ها پیش از سیستم عامل ویندوز، آن را در اختیار برنامه‌نویسان قرار داده بود.

**نکته:** pid_t یکی از انواع تایپ‌های صحیح زبان C تحت لینوکس است که محتوی آن می‌تواند شماره‌ی یک فرایند باشد. شماره‌ی فرایند (PID) مشخصه‌ای است که سیستم عامل توسط آن یک فرایند را می‌شناسد و به آن دسترسی پیدا می‌کند.

## ۴.۲.۱-اجرای عملیاتی در فرایند فرزند متفاوت با عملیات فرایند اصلی

**تابع system:** می‌توان با کمک این تابع کتابخانه‌ای[10] برنامه‌ای را از داخل برنامه‌ی دیگر فراخوانی کرد و به این ترتیب، سبب ایجاد فرایند جدیدی مستقل از فرایند کنونی شد. شکل کلی آن به صورت زیر است:

```
int system(const char *string)
```

این تابع، دستوری را که در قالب رشته به آن داده شده است، با فراخوانی پوسته انجام می‌دهد، درست همانند اجرای یک دستور معمولی. البته بعد از انجام این تابع، کنترل اجرا به برنامه‌ی فراخوان برخواهد گشت.

**توابع exec:** فراخوان‌های exec مجموعه توابع مفیدی هستند که اجرای یک دستور سیستم عامل را (که قاعدتا از خط فرمان اجرا می‌شود) در داخل برنامه امکان‌پذیر می‌سازند. یک تابع exec، فرایند جاری را با فرایندی جدید که در آرگومان‌های path و file مشخص شده است، جایگزین می‌کند. از این امر می‌توان برای تغییر اجرای یک فرایند به فرایند دیگر استفاده کرد. به عنوان مثال، در یک برنامه ممکن است ابتدا کاربر را شناسایی کنیم و سپس به او اجازه‌ی اجرای برنامه‌ی دیگری را بدهیم. بدین ترتیب عملکرد این توابع در لینوکس همانند عملکردشان در سیستم‌عامل قدیمی DOS است؛ با این تفاوت که در DOS، اجرای exec شبیه فراخوانی یک روال معمولی است و پس از اتمام کار، به بعد از محل فراخوانی exec برمی‌گردیم. اما در لینوکس، تمام شدن اجرای exec، به اجرای برنامه (که شامل تابع main در فراخوانی exec است) خاتمه خواهد داد. توابع مختلف exec که می‌توان از آن‌ها استفاده کرد، عبارتند از:

---

[9] Program Counter
[10] Library function

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., (char *)0);
int execlp(const char *file, const char *arg0, ..., (char *)0);
int execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

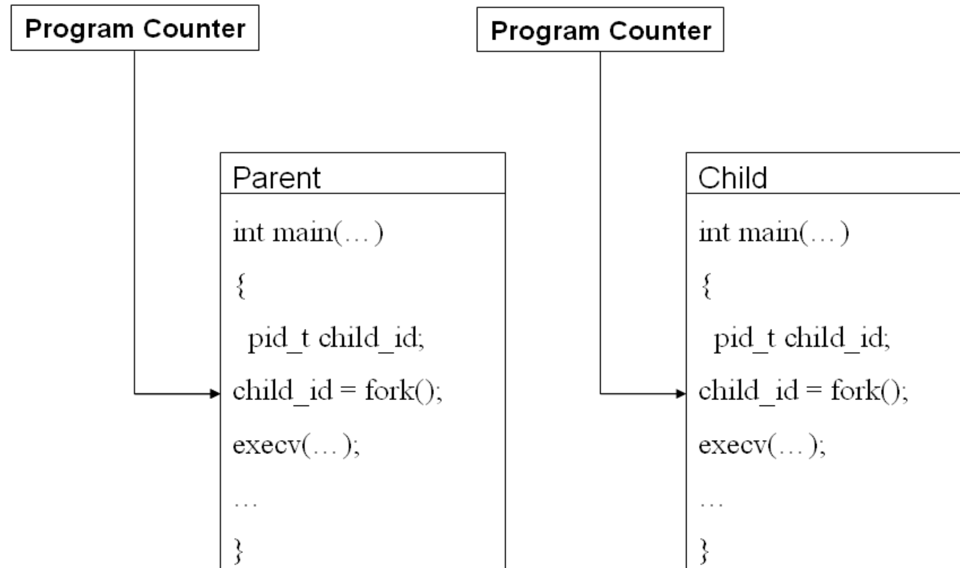
برای اطلاعات بیشتر در مورد این توابع، می‌توان به کتاب‌های آموزشی زبان C مراجعه کرد، اما برای آشنایی بیشتر، به مثال زیر توجه کنید:

```
/* Example of an argument list */
/* Note that we need a program name for argv[0] */
char *const ps_argv[] = {"ps", "ax", 0};
/* Example environment, not terribly useful */
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};
/* Possible calls to exec functions */
execl("/bin/ps", "ps", "ax", 0); /* assumes ps is in /bin */
execlp("ps", "ps", "ax", 0); /* assumes /bin is in PATH */
execle("/bin/ps", "ps", "ax", 0, ps_envp); /* passes own environment */
execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

در اینجا به نظر می‌رسد اگر بتوان تابع exec را در مسیری جدا از مسیربرنامه‌ی اصلی اجرا کرد، مشکل اجرای فرایندی متفاوت با فرایند پدر در فرایند فرزند، قابل حل خواهد بود. اگر برای ایجاد این مسیر جدید، بعد از fork یک فراخوانی exec قرار دهیم (مطابق شکل ۲)، این تابع در فرایند پدر نیز اجرا خواهد شد که مطلوب نیست (مانند دستور printf در قطعه کد بالا داشتیم).

نکته رفع مشکل فوق توجه به تفاوت بین مقدار child_id در پدر و فرزند ایجاد شده توسط دستور fork است. زیرا عبارت child_id=fork()، در صورت ایجاد موفقیت‌آمیز فرایند فرزند، در هر دو فرایند پدر و فرزند مقدار خواهد گرفت. در فرایند پدر مقداری بزرگتر از صفر (به عنوان شماره فرزند ایجاد شده) در child_id قرار خواهد گرفت و در فرایند فرزند، مقدار صفر در این متغییر ذخیره خواهد شد.

اگر اشکالی در ایجاد فرایند فرزند به وجود آید، fork مقدار ۱- را برخواهد گرداند که به این ترتیب چون فرایند فرزند ایجاد نشده در child_id از فرایند پدر ذخیره خواهد شد.




**شکل ۱-۱- وضعیت فرایند اصلی (پدر) و فرایند ایجاد شده (فرزند) بلافاصله بعد از fork**

با توجه با مطالب بالا می‌توان بعد از fork:قسمت‌هاي اختصاصي پدر و فرزند را به شکل زیر مشخص کرد ،

```
child_id = fork();
if (child_id = = -1) {
	/* خطا در ایجاد فرایند فرزند:
	معمولا در این‌حالت، تابع با یک کد خطا ترک می‌شود.*/
}
if (child_id = = 0) {
	/* قسمت اختصاصی فرزند:
	کارهای مربوط به فرایند فرزند در این قسمت انجام می‌شود.
	این بخش با اینکه در فرایند پدر نیز وجود دارد،
	هرگز در فرایند پدر اجرا نمی‌شود. به این علت که در صورت موفقیت
	fork، child_id در فرایند پدر، مقـــدار بزرگتر از صفر به
	خود می‌گیرد.
	*/
}
else {
	/* قسمت اختصاصی پدر:
	کارهای اختصاصی فرایند پدر نیز در این قسمت انجام می‌گیرد. زیرا
	در صورت موفقیت fork، child_id در فرایند فرزند مخالف صفر
	خواهد شد.
	*/
```

قسمت‌هایی که مقدار برگشتی از fork، ۱- نبوده است، می‌تواند به عنوان قسمت اشتراکی بین هر دو فرایند نیز در نظر گرفته شود. البته مانند کد بالا می‌توان فرایندهای پدر و فرزند را از هم جدا کرد. یک روش دیگر برای نوشتن کد بالا، مانند زیر است:

```
child_pid = fork ();
switch (child_pid)
{
        case -1:        /* خطا در ایجاد فرایند */
        case 0:         /* فرایند فرزند */
        default:        /* فرایند پدر */
}
```

### ۵.۲.۱-متوقف کردن یک فرایند

اگر فرایندی با استفاده از fork، فرایند جدیدی را به وجود آورد. به محض ایجاد فرایند فرزند، دو فرایند پدر و فرزند به موازات هم اجرا خواهند شد. ولی گاهی این عمل مد نظر نیست، بلکه مثلا فرایند پدر باید منتظر فرایند فرزند بماند تا کار او تمام شود، آنگاه به کار خود ادامه دهد. برای این‌کار فرایند پدر باید از فراخوان سیستم wait استفاده کند. این تابع به صورت زیر تعریف شده است:

```
int wait (int* status)
```

فرایندی که این تابع را احضار کرده باشد، تا اتمام کار یکی از فرزندانش صبر خواهد کرد (مقدار خروجی این تابع همان PID فرایند فرزندی است که کارش تمام شده است و کد خروج این فرزند در پارامتر status ظاهر خواهد شد). اگر فرایند احضار کننده، هیچ فرزندی نداشته باشد، این تابع را بلافاصله بازگشت می‌کند.

فراخوان سودمند دیگر برای متوقف کردن یک فرایند، waitpid است. در مورد آن می‌توان از صفحات راهنمای موجود در لینوکس کمک گرفت. این تابع، فرایند جاری را قادر می‌سازد تا منتظر اتمام کار فرایند فرزندی شود که شماره‌ی آن فرایند فرزند را به عنوان پارامتر به تابع waitpid ارائه کرده‌ایم.

### ۶.۲.۱-اجرای فرایندها در پس‌زمینه[11]

با افزودن علامت & به آخر هر فرمان یا برنامه، می‌توان آن را در پس‌زمینه اجرا کرد. در این حالت، لینوکس علاوه بر اجرای برنامه‌ی ذکر شده، با دادن اعلان خط فرمان، آمادگی خود را جهت گرفتن فرمان، یا برنامه‌ای دیگر اعلام می‌کند (بدون اینکه اجرای برنامه‌ی قبلی پایان یافته باشد). بدین ترتیب دو یا چند فرمان را می‌توان با هم اجرا کرد و شیوه‌ای دیگر از چندوظیفه‌ای را پیاده‌سازی نمود.

**مثال:**

```
ls –l &
```

[11] Background

## ۳.۱-مراجع


1. Neil Matthew and Richard Stones, **Beginning Linux Programming**, 4'th Ed., Wiley Publishing, Inc., 2007.
2. Keir Thomas, **Beginning Ubuntu Linux From Novice to Professional**, Apress, 2006.
3. Sander van Vugt, **Beginning the Linux Command Line**, Apress, 2009.
4. Matthias Kalle Dalheimer, Terry Dawson, Lar Kaufman and Matt Welsh, **Running Linux**, 4'th Ed., O'Reilly, 2002.

۵. صدیقی مشکنانی، محسن. **دستورکار آزمایشگاه سیستم‌عامل**. دانشکده برق و کامپیوتر دانشگاه صنعتی اصفهان. زمستان ۱۳۷۷.

## ۴.۱-دستورکار

**توجه:** برای افزایش سرعت نوشتن برنامه‌ها، می‌توانید از Text Editor موجود در Accessories، واقع در Application استفاده کنید.

سیستم را روشن کنید و پس از راه‌اندازی Linux، وارد حساب کاربری خود شوید. برای این کار نام کاربری و کلمه‌ی عبور خود را از مربی آزمایشگاه دریافت کنید. از Application به Accessories بروید و گزینه Terminal را انتخاب کنید تا اعلان سیستم‌عامل در خط فرمان را مشاهده کنید. سپس آزمایش‌های زیر را انجام دهید:

۱. دستور pstree را اجرا نموده و علاوه برذکر عنصر ریشه‌ی آن، دو مورد آشنای داخلی را نیز یادداشت نمایید.

۲. برنامه‌ای مانند ماشین حساب را باز کنید و با دستور kill به اجرای آن خاتمه دهید. نحوه یافتن آن برنامه و نحوه استفاده از دستور kill را در کاربرگ بنویسید.

۳. برنامه زیر را در فایلی به نام first.c ذخیره و اجرا کنید. خروجی را در کاربرگ خود یادداشت کنید. آیا انتظار می‌رود با اجرای مجدد آن، نتایج متفاوتی حاصل شود؟ چرا؟

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
	pid_t pid;
	char *message;
	int n;

	printf("fork program starting\n");
	pid = fork();
	switch(pid)
	{
	case -1:
		perror("fork failed");
		exit(1);
	case 0:
		message = "This is the child";
		n = 5;
		break;
```

```
        default:
                message = "This is the parent";
                n = 3;
                break;
        }

        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
exit(0);
}
```

۴. قطعه برنامه‌ی زیر را با نام execp.c ذخیره کنید و اجرا نمایید و بیان کنید که آیا خروجی دستور printf("Done.\n"); نیز قابل رویت است؟ چرا؟

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        printf("Running ps with execlp\n");
        execlp("ps", "ps", "ax", 0);
        printf("Done.\n");
        exit(0);
}
```

۵. با استفاده از تابع system، سطر execlp("ps", "ps", "ax", 0); را از تمرین قبل جایگزین کرده و خروجی را مقایسه کنید و علاوه بر دستور system مورد استفاده، تفاوت خروجی را همراه با دلیل آن شرح دهید.

۶. با استفاده از گزینه‌ی & برای فرستادن برنامه به پس‌زمینه و با استفاده از دستور sleep(i) که در آزمایش ۲ فراگرفتید و برای ایجاد درنگ در برنامه به مدت i ثانیه است، دو برنامه‌ی ساده به نام‌های 11.c و 12.c بنویسید که اولی خروجی First Program Finished را بعد از بیست ثانیه، ظاهر کند و دومی به مدت بیست ثانیه در هر یک ثانیه، پیام Second Program running را چاپ نماید. ابتدا هر دو را کامپایل کنید، سپس برنامه‌ی 11 را به اجرا در پس‌زمینه بفرستید، سپس برنامه‌ی 12 را اجرا کنید و در نهایت خروجی را به مدرس نشان دهید. متن برنامه‌ها را در کاربرگ قید کنید.

۷. برنامه‌ی آزمایش ۳ را طوری تغییر دهید که برنامه‌ی فرزند پیش از پدر خاتمه یابد. آنگاه پس از اجرای برنامه‌ی فرزند، در خط فرمان دستور ps –al را بزنید و با توجه به ستون وضعیت فرایندها (S)، شناسه‌ی فرزند را بیابید. چطور می‌توان ثابت نمود که این شناسه حتما متعلق به فرزند است؟