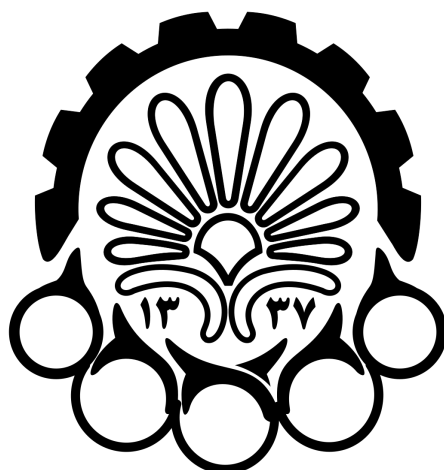


سیستم‌های عامل  
دکتر جوادی



**دانشگاه صنعتی امیرکبیر**

( پلی تکنیک تهران )

دانشکده مهندسی کامپیوتر

رضا آدینه پور ۴۰۲۱۳۱۰۵۵

تمرین سری اول

۳ آبان ۱۴۰۲

## به سوالات زیر پاسخ دهید

الف) هدف از DMA چیست؟

پاسخ

هدف اصلی استفاده از DMA افزایش سرعت انتقال داده ها بین دستگاه‌های I/O و حافظه سیستم است. با استفاده از DMA می‌توان مستقیماً و بدون نیاز به مداخله CPU به I/O و حافظه‌ها دسترسی پیدا کرد و دیتا را به صورت مستقیم انتقال داد.

ب) چگونه می‌توان سیستمی طراحی کرد که اجازه ی انتخاب یک سیستم عامل از چند سیستم عامل را هنگام بوت شدن به کاربر بدهد؟ برنامه‌ی Bootstrap برای این منظور چه کاری باید انجام دهد؟

پاسخ

برای این کار باید برنامه ای نوشته شود که در زمان Boot سیستم اجرا شود. این برنامه باید توانایی نمایش رابط کاربری مناسب (مانند منو انتخاب سیستم‌عامل) را داشته باشد و بتواند با ارتباط با I/O های سیستم ورودی کاربر که OS انتخابی آن است را دریافت کند. همچنین این برنامه باید قابلیت این را داشته باشد که پس از دریافت سیستم‌عامل انتخابی توسط کاربر، بتواند فایل‌ها و داده‌های مورد نیاز برای اجرای OS را بارگذاری کند و پس از اجرای OS باید برنامه Bootstrap کنترل سیستم را به سیستم‌عامل بدهد.

پ) توضیح دهید که تفاوت بین حالت کرنل و حالت کاربر چگونه به حفاظت و امنیت سیستم کمک میکند. کدام یک از دستورات زیر باید در حالت کرنل اجرا شوند؟

- A) Set value of timer
- B) Read the clock
- C) Clear memory
- D) Issue a trap instruction
- E) Turn off interrupts
- F) Modify entries in device-status table
- G) Switch from user to kernel mode
- H) Access I/O device

## پاسخ

حالت کرنل و حالت کاربر دو حالت اجرایی در OS هستند که تفاوت‌های مهمی در امنیت و حفاظت سیستم دارند. در حالت کرنل، برنامه‌ها و سرویس‌های سیستم با دسترسی کامل به منابع سخت‌افزاری و سیستم عامل اجرا می‌شوند و در حالت کاربر، برنامه‌ها تنها با دسترسی محدود به منابع سیستم عامل اجرا می‌شوند.

حالت کرنل به دلایل زیر به حافظت و امنیت سیستم کمک می‌کند:

۱. **محدودیت دسترسی:** در حالت کاربر، برنامه‌ها دسترسی محدودتری به منابع سیستم عامل دارند و نمی‌توانند به منابع حساس مثل حافظه سیستم یا دستگاه‌های I/O مستقیماً دسترسی داشته باشند. در حالت کرنل، سرویس‌ها و برنامه‌های سیستم عامل با دسترسی کامل به منابع سیستم عامل اجرا می‌شوند، اما این دسترسی برای برنامه‌های کاربردی محدود می‌شود.

۲. **جدا بودن فضای آدرسی:** در حالت کرنل و کاربر، فضای آدرسی برای برنامه‌ها جداگانه تعیین می‌شود. در حالت کاربر، برنامه‌ها تنها به فضای آدرسی خودشان دسترسی دارند و نمی‌توانند به فضای آدرسی برنامه‌های دیگر یا سیستم عامل دسترسی داشته باشند. این از خطرات نفوذ و دسترسی غیرمجاز جلوگیری می‌کند.

۳. **محدودیت دسترسی به سخت‌افزار:** در حالت کاربر، برنامه‌ها نمی‌توانند به دستگاه‌های سخت‌افزاری مستقیماً دسترسی داشته باشند و باید از طریق واسطه‌های سیستم عامل از آن‌ها استفاده کنند. در حالت کرنل، سرویس‌ها و برنامه‌های سیستم عامل می‌توانند به طور مستقیم با دستگاه‌های سخت‌افزاری ارتباط برقرار کنند.

دستوراتی که مستقیماً با منابع سیستم عامل یا سخت‌افزار ارتباط برقرار می‌کنند، مانند تنظیم تایمر، پاکسازی حافظه، صدا زدن دستور توقف (trap) و دسترسی به دستگاه‌های I/O، در حالت کرنل باید اجرا شوند. این دستورات نیاز به دسترسی به منابع حساس سیستم دارند که در حالت کاربر محدود می‌شود. دستوراتی که مستقیماً با منابع کاربردی برنامه‌ها ارتباط برقرار می‌کنند، مانند خواندن ساعت، تغییر I/O و مدیریت حافظه، در حالت کاربر باید اجرا شوند. این دستورات معمولاً نیاز به دسترسی مستقیم به منابع کاربران دارند و در حالت کرنل اجرا نمی‌شوند.

بنابراین فقط مورد B است که باید در حالت کاربر اجرا شود و بقیه موارد همگی در حالت کرنل اجرا می‌شوند.

(د) در یک محیط Multi programming و Time sharing چند کاربر به صورت همزمان سیستم را به اشتراک می‌گذارند و این وضعیت می‌تواند منجر به مشکلات امنیتی مختلف شود. ۲ مورد از این مشکلات را نام ببرید.

## پاسخ

۱. **کلاهبرداری از داده‌ها:** وجود چند کاربر در یک سیستم به اشتراک گذاشته شده ممکن است باعث افزایش ریسک کلاهبرداری از داده‌ها شود. اگر یک کاربر از طریق آسیب‌پذیری‌های امنیتی در سیستم، به داده‌های دیگری که توسط کاربران دیگر در حافظه سیستم قرار دارد، دسترسی پیدا کند، می‌تواند اطلاعات حساس را بدون اجازه و به طور غیرمجاز به دست آورد. این مشکل می‌تواند منجر به فاش شدن اطلاعات شخصی، رمزهای عبور، داده‌های حساس کسب و کار و سایر اطلاعات محرمانه شود.

۲. **تداخل در حافظه و منابع سیستم:** وجود چند کاربر در یک سیستم به اشتراک‌زمان، ممکن است منجر به تداخل‌های حافظه و منابع سیستم شود. زمانی که چند کاربر به صورت همزمان در حال اجرای برنامه‌ها و پردازش‌های مختلف هستند، ممکن است منابع سیستم مانند حافظه، پردازنده و I/O به طور ناهمزمان و نامتعادل مورد استفاده قرار گیرند. این موضوع می‌تواند منجر به افزایش زمان پاسخ و کندی عملکرد برنامه‌ها شود. همچنین، در مواردی که هر کاربر به منابع سیستم با دسترسی محدود دسترسی دارد، تداخل‌ها می‌توانند باعث کاهش کارایی و عملکرد کاربران شود.

## به سوالات زیر پاسخ دهید

الف) وقفه چیست؟ وقفه‌های سنکرون و آسنکرون را باهم مقایسه کنید.

### پاسخ

وقفه (Interrupt) در محیط برنامه‌نویسی به وقوع پیوستن یک رویداد ناگهانی در حین اجرای برنامه گفته می‌شود که عملکرد طبیعی برنامه را متوقف می‌کند و برنامه‌ای را به اجرای یک کد خاص یا روند دیگر تغییر می‌دهد. وقفه‌ها معمولاً توسط سخت‌افزار و سیستم عامل به منظور پاسخگویی به رویدادهای مهم مانند درخواست‌های I/O، خطاها، تایمرها و سایر رویدادها ایجاد می‌شوند. از نظر زمان وقوع، وقفه‌ها به دو دسته سنکرون و آسنکرون تقسیم می‌شوند:

#### ۱. وقفه‌های سنکرون:

- (آ) وقوع وقفه در زمانی قرار دارد که برنامه در یک نقطه مشخص خود را در حالت انتظار قرار می‌دهد و منتظر وقوع وقفه است.
- (ب) برنامه‌ای که با وقوع وقفه مواجه می‌شود، به طور مستقیم و بلافاصله وارد روند وقفه می‌شود و ادامه اجرای برنامه بعد از پایان وقفه ادامه می‌یابد.
- (ج) معمولاً وقفه‌های سنکرون توسط سخت‌افزار ایجاد می‌شوند، مانند درخواست‌های ورودی کاربر، تقاضای دستگاه‌های جانبی و غیره.

#### ۲. وقفه‌های آسنکرون:

- (آ) وقوع وقفه در زمانی قرار دارد که برنامه در حال اجرا است و به طور غیرمنتظره با یک رویداد ناگهانی مواجه می‌شود.
- (ب) وقفه آسنکرون می‌تواند در هر نقطه‌ای از اجرای برنامه رخ دهد و برنامه را به وقفه‌هایی مانند خطاها، سیگنال‌های سیستم عامل، تقاضای دیگر برنامه‌ها و غیره وصل می‌کند.
- (ج) وقفه‌های آسنکرون برنامه را از جریان اصلی آن جدا کرده و به روند وقفه منتقل می‌کنند. بعد از پایان وقفه، برنامه از جایی که متوقف شده بود، ادامه می‌یابد.
- (د) معمولاً وقفه‌های آسنکرون توسط سخت‌افزار (مانند خطاهای سخت‌افزاری) یا سیستم عامل (مانند سیگنال‌های سیستم عامل) ایجاد می‌شوند.

ب) تفاوت‌های بین Interrupt و Trap را توضیح دهید.

پاسخ

وقفه و Trap هر دو پدیده‌هایی در محیط برنامه‌نویسی هستند که به وقوع پیوستن رویدادهای ناگهانی در حین اجرای برنامه را مشخص می‌کنند. اما تفاوت‌هایی بین این دو وجود دارد:

#### ۱. Interrupt

- (آ) وقفه‌ها معمولاً توسط سخت‌افزار یا سیستم عامل ایجاد می‌شوند و می‌توانند در هر زمانی و در هر نقطه‌ای از اجرای برنامه رخ دهند.
- (ب) هدف اصلی وقفه‌ها، متوقف کردن عادی برنامه و پاسخگویی به رویدادهای مهم است. مثال‌هایی از وقفه‌ها شامل درخواست‌های ورودی کاربر، تقاضای دستگاه‌های جانبی، خطاها و تایمرها می‌شوند.
- (ج) وقفه‌ها معمولاً باعث تغییر جریان اجرای برنامه می‌شوند. برنامه به طور مستقیم و بلافاصله به یک روند وقفه منتقل می‌شود و پس از پایان وقفه، به جریان اصلی خود بازگشت می‌کند.

#### ۲. Trap

- (آ) تریپ‌ها معمولاً توسط خود برنامه نوشته شده و قابلیت اجرای آنها وجود دارد. معمولاً در نقاط خاصی از برنامه قرار داده می‌شوند تا در صورت بروز شرایط خاص، عملیات خاصی انجام دهند.
  - (ب) هدف اصلی تریپ‌ها، نیاز به یک رفتار خاص در برنامه است و معمولاً برای انجام عملیات‌های خاص (مانند خطاها، استثناها و غیره) استفاده می‌شوند.
  - (ج) تریپ‌ها برای تعامل با سیستم عامل یا سخت‌افزار می‌توانند استفاده شوند. به عنوان مثال، یک برنامه می‌تواند تریپ برای درخواست سیستم عامل برای اختصاص حافظه یا فایل‌ها داشته باشد.
  - (د) تریپ‌ها معمولاً توسط برنامه بررسی می‌شوند و در صورت بروز شرایط، برنامه به طور دستوری به روند تریپ منتقل می‌شود. پس از انجام عملیات تریپ، برنامه به جریان اصلی خود بازگشت می‌کند.
- به طور خلاصه، وقفه‌ها معمولاً توسط سخت‌افزار یا سیستم عامل ایجاد می‌شوند و هدف اصلی آنها پاسخگویی به رویدادهای مهم است. تریپ‌ها به طور کلی توسط برنامه نوشته شده و هدف اصلی آنها انجام عملیات خاص در برنامه است.

پ) فرآیند مدیریت یک وقفه از لحظه ایجاد شدن تا اتمام آن را توضیح دهید. فرض کنید وقفه متعدد نداریم و CPU مشغول انجام برنامه کاربر است.

## پاسخ

فرآیند مدیریت یک وقفه از لحظه ایجاد شدن تا اتمام آن عموماً توسط سیستم عامل و سخت‌افزار انجام می‌شود. در زیر، مراحل اصلی مدیریت یک وقفه را توضیح می‌دهم:

۱. شناسایی وقفه: سیستم عامل و سخت‌افزار در هر لحظه وقفه‌ها را بررسی می‌کنند. این بررسی ممکن است توسط سخت‌افزار (مانند تایمرها، درخواست‌های دستگاه‌های جانبی و ...) یا سیستم عامل (مانند درخواست‌های ورودی کاربر و ...) صورت گیرد.

۲. ذخیره وضعیت فعلی: سیستم عامل اطلاعات مربوط به وضعیت فعلی برنامه را در یک مکان مناسب ذخیره می‌کند. این کار از طریق استفاده از مکانیزمی به نام Context Switch انجام می‌شود.

۳. اجرای روند وقفه: پس از ذخیره وضعیت فعلی، سیستم عامل به Interrupt Handler منتقل می‌شود. روند وقفه کدی است که توسط سیستم عامل تعریف شده است و وظیفه پاسخگویی به وقفه را دارد. در این مرحله، عملیات مربوط به وقفه انجام می‌شود (مانند پاسخ به درخواست کاربر، خواندن داده از دستگاه جانبی و ...)

۴. بازگشت به برنامه اصلی: بعد از اتمام عملیات وقفه، سیستم عامل وضعیت قبلی را بازیابی می‌کند. این شامل بازگشت به وضعیت قبلی ثبات‌های CPU، شمارنده‌ها و سایر اطلاعات مربوطه است.

۵. ادامه اجرای برنامه: سیستم عامل بعد از بازگشت به وضعیت قبلی، اجرای برنامه را از جایی که قبل از وقفه متوقف شده بود، ادامه می‌دهد. در این مرحله، جریان اجرای برنامه به همان نقطه قبل از وقفه برمی‌گردد و برنامه از همان جایی که متوقف شده بود ادامه می‌یابد.

این فرآیند مدیریت وقفه معمولاً به صورت خودکار توسط سیستم عامل و سخت‌افزار انجام می‌شود و برنامه نویس نیازی به دخالت مستقیم در این فرآیند ندارد.

## در هر یک از موارد زیر، پردازنده از چه حالتی به چه حالت دیگری تغییر وضعیت می‌دهد؟ (منظور از حالت، وضعیت‌های مختلف پردازنده شامل Running، New، Terminated، Ready، Waiting است)

- الف) در حین اجرای پردازنده، کاربر کلیدی را فشار داده و وقفه‌ای با اولویت بالا تر در سیستم اتفاق می‌افتد.  
 ب) پردازنده در حین اجرای کد خود، به جایی می‌رسد که نیاز به دریافت داده‌ها از طریق شبکه دارد  
 پ) رویداد مربوط به یکی از دستگاه‌های ورودی/خروجی به پایان رسیده و داده‌های مورد نیاز پردازنده آماده می‌شود.  
 ت) برنامه‌ای به زبان C تابع `exit()` را فراخوانی می‌کند.  
 ث) زمان‌بندی سیستم‌عامل، پردازنده‌ای را از صف انتظار خارج کرده و به آن اجازه اجرا می‌دهد.

### پاسخ

در هر یک از موارد زیر، پردازنده از یک حالت به حالت دیگر تغییر وضعیت می‌دهد:

#### ۱. وقفه با اولویت بالا:

- (آ) وضعیت قبلی: Running  
 (ب) وضعیت جدید: Interrupted/Waiting  
 وقفه با اولویت بالا می‌تواند پردازنده‌ای که در حال اجرا است را متوقف کند و به حالت وقفه (Interrupted) یا انتظار (Waiting) برود.

#### ۲. درخواست دریافت داده‌ها از شبکه:

- (آ) وضعیت قبلی: Running  
 (ب) وضعیت جدید: Waiting  
 پردازنده در حال اجرا برای دریافت داده‌ها از شبکه نیاز به انتظار (Waiting) دارد تا داده‌های مورد نیاز آماده شوند.

#### ۳. پایان رسیدن رویداد دستگاه ورودی/خروجی:

- (آ) وضعیت قبلی: Waiting  
 (ب) وضعیت جدید: Ready  
 رویداد مربوط به دستگاه ورودی/خروجی به پایان رسیده و داده‌های مورد نیاز پردازنده آماده می‌شود، در نتیجه پردازنده از حالت انتظار (Waiting) به حالت آماده (Ready) تغییر وضعیت می‌دهد.

#### ۴. فراخوانی تابع `exit()`:

- (آ) وضعیت قبلی: Running  
 (ب) وضعیت جدید: Terminated  
 هنگامی که برنامه C تابع `exit()` را فراخوانی می‌کند، پردازنده به حالت پایانی (Terminated) می‌رود و اجرای آن به پایان می‌رسد.

#### ۵. پردازنده در هنگام اجرای کد خود، به جایی می‌رسد که نیاز به دریافت داده‌ها از شبکه دارد.

- (آ) وضعیت قبلی: Ready  
 (ب) وضعیت جدید: Waiting

## خروجی قطعه کدهای داده شده چیست؟ درخت پردازش‌های هر برنامه را رسم و راه‌حل خود را توضیح دهید.

Listing 1: Some Code

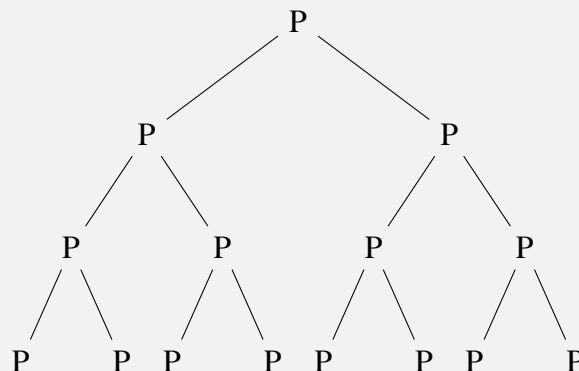
```
int (main)
{
    int i;
    for(i = 0; i < 3; ++i)
        fork();
    printf("Hello\n");
    return 0;
}
```

پاسخ

خروجی کد به صورت زیر است:

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

در این کد، یک حلقه `for` وجود دارد که سه بار تکرار می‌شود. در هر تکرار، تابع `fork()` فراخوانی می‌شود. تابع `fork()` یک پردازش جدید را ایجاد می‌کند که از پردازش فعلی تولید شده است، به طوری که حالت اجرای برنامه پس از این نقطه در هر دو پردازش ادامه می‌یابد. بنابراین، در هر تکرار حلقه، تعداد پردازش‌ها دو برابر می‌شود. در نتیجه، در این برنامه اصلی، ابتدا یک پردازش ایجاد می‌شود. در تکرار اول حلقه، یک پردازش دیگر ایجاد می‌شود و در نتیجه تعداد پردازش‌ها دو می‌شود. در تکرار دوم حلقه، هر یک از دو پردازش قبلی یک پردازش دیگر ایجاد می‌کنند و تعداد پردازش‌ها چهار می‌شود. در تکرار سوم حلقه، هر یک از چهار پردازش قبلی یک پردازش دیگر ایجاد می‌کنند و تعداد پردازش‌ها هشت می‌شود. پس از اتمام حلقه `for`، هر یک از هشت پردازش تولید شده عبارت `"Hello"` را چاپ می‌کند. بنابراین، در خروجی نهایی، عبارت `"Hello"` ۸ بار چاپ می‌شود. درخت پردازش برای این برنامه به صورت زیر است:





## Listing 2: Some Code

```

int (main)
{
    if (fork () && fork ())
    {
        fork ();
        fork ();
    }
    printf ("A");
    return 0;
}

```

## پاسخ

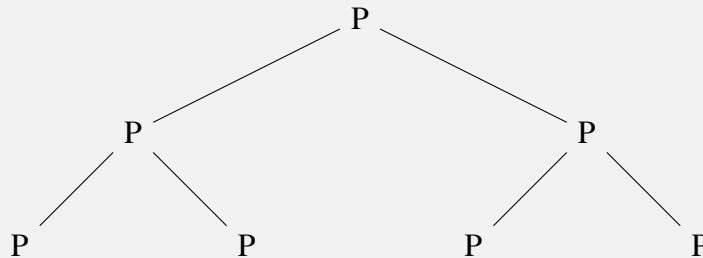
خروجی کد به صورت زیر است:

AAAA

در این برنامه، تابع `fork()` برای ایجاد پردازش‌های جدید استفاده می‌شود. با استفاده از این تابع، پردازش‌های فرزند از پردازش فعلی تولید می‌شوند. حلقه شرطی

```
if (fork() && fork())
```

باعث ایجاد چهار پردازش فرزند می‌شود. در اینجا، ابتدا یک پردازش فرزند ایجاد می‌شود و سپس دو پردازش فرزند از هر یک از پردازش‌های فرزند قبلی تولید می‌شوند. بنابراین، در نهایت، تعداد کل پردازش‌ها به چهار می‌رسد. سپس در داخل حلقه دوباره `fork()` فراخوانی می‌شود که باعث تولید چهار پردازش فرزند دیگر می‌شود. در نتیجه، تعداد کل پردازش‌ها به هشت می‌رسد. در نهایت، در همه‌ی پردازش‌ها، عبارت "A" چاپ می‌شود. بنابراین، خروجی نهایی برنامه "AAAA" خواهد بود. درخت پردازش برای این برنامه به صورت زیر است:



## Listing 3: Some Code

```

int (main)
{
    int i = 2;
    while (i > 0)
    {
        i--;
        if (fork ())
        {
            fork ();
            printf ("B");
        }
    }
}

```

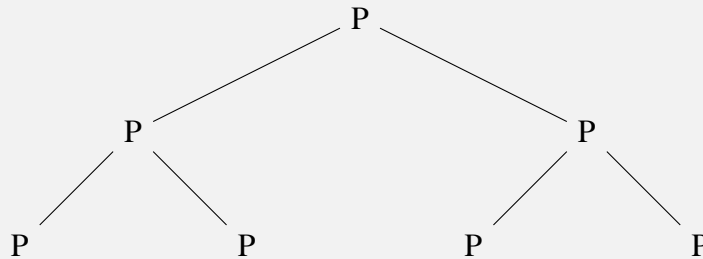
```

    }
    return 0;
}

```

پاسخ

در این برنامه حلقه بی‌نهایت وجود دارد و خروجی برنامه شامل بی‌نهایت عبارت B است. به دلیل این که هر بار که `fork()` فراخوانی می‌شود، تعداد پردازنده‌ها دو برابر می‌شود، درخت پردازنده‌ها بسیار پیچیده و حجیم خواهد شد. درخت ممکن است به طور نمایی رشد کند و نمی‌توان آن را به صورت کامل رسم کرد. در شکل زیر درخت پردازنده را صرفاً برای دور اول حلقه آورده ایم:



Listing 4: Some Code

```

int (main)
{
    if (fork () || fork () && fork ())
    {
        fork ();
        print ("C")
    }
    return 0;
}

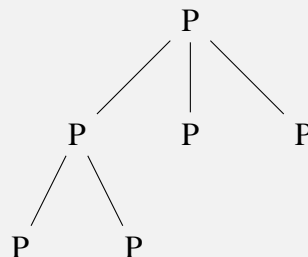
```

پاسخ

خروجی کد به صورت زیر است:

CCCC

با توجه به شرط در عبارت شرطی، اگر حداقل یکی از دو `fork()` اولیه موفق باشد (یعنی یک فرزند ایجاد شود)، عبارت داخل `if` اجرا خواهد شد. درخت پردازنده برای این برنامه به صورت زیر است:



## به سوالات زیر پاسخ کامل دهید

الف) قطعه کد زیر را در نظر بگیرید:

Listing 5: Some Code

```
int (main)
{
    int count = 0;
    pid_t ret = fork;
    if (ret == 0)
        printf("count in child = %d\n", count)
    else
        count = 1;

    return 0;
}
```

اگر والد دستور `count = 1` را قبل از اجرای فرزند برای اولین بار اجرا می‌کند، مشخص کنید مقدار چاپ شده توسط کد بالا چقدر است؟ توضیح دهید.

### پاسخ

با توجه به قطعه کد داده شده، اگر والد دستور `count = 1` را قبل از اجرای فرزند برای اولین بار اجرا کند، مقدار چاپ شده توسط کد بال به صورت `count in child = 0` خواهد بود. در ابتدا، متغیر `count` به مقدار 0 مقداردهی می‌شود. سپس با استفاده از `fork()`، یک فرآیند فرزند ایجاد می‌شود و ارزش بازگشتی `fork()` به متغیر `ret` اختصاص داده می‌شود. اگر `ret` برابر با 0 باشد، به این معنی است که کد در حال اجرا در فرآیند فرزند است. در این حالت، دستور `printf("count in child = %d\n", count)` اجرا می‌شود و مقدار `count` که در فرآیند فرزند است هنوز تغییر نکرده و برابر با 0 است. اما اگر `ret` مقدار دیگری غیر از 0 داشته باشد، به این معنی است که کد در حال اجرا در فرآیند والد است. در این حالت، دستور `count = 1` اجرا می‌شود و مقدار `count` تغییر کرده و برابر با 1 می‌شود. اما هیچ دستوری برای چاپ مقدار `count` در فرآیند والد وجود ندارد.

ب) باتوجه به پردازنده‌های `Orphan` و `Zombie` به سوالات زیر پاسخ دهید:

۱. توضیح دهید که هرکدام از این پردازنده‌ها چگونه ایجاد می‌شوند و تفاوت آنها با یکدیگر چیست؟

پاسخ

**Orphan: (آ)**

i. وقتی یک فرآیند والد خود را قبل از اتمام یا خروج از برنامه ببندد، فرزندانش به عنوان پردازش‌های orphan به شمار می‌روند. پردازش‌های orphan در واقع پردازش‌هایی هستند که والدشان قبل از اتمام خود زنده نمی‌ماند و آن‌ها بدون والد باقی می‌مانند.

ii. در این حالت، سیستم عامل پردازش‌های orphan را به عهده می‌گیرد و آن‌ها را به یک پردازش والد جدید انتقال می‌دهد. پردازش جدید به عنوان والد جایگزین برای آن‌ها عمل می‌کند.

**Zombie: (ب)**

i. وقتی یک فرآیند فرزند خود را قبل از والدش ببندد و اجرای خود را به پایان برساند، به عنوان یک پردازش zombie باقی می‌ماند.

ii. در این وضعیت، پردازش فرزند اجرا خود را به پایان رسانده است ولی والد هنوز آن را با استفاده از سیستم `wait()` یا `waitpid()` به صورت صحیح تمام نکرده است.

iii. پس از اتمام عملیات والد، سیستم عامل پردازش zombie را از جدول پردازش‌ها حذف می‌کند و منابع آن را آزاد می‌کند.

۲. باتوجه به قطعه کدهای زیر برای هر مورد مشخص کنید که بین Orphan و Zombie چه پردازش‌ای ایجاد می‌شود و جواب خود را به طور کامل توضیح دهید.

## Listing 6: Some Code

```
int (main)
{
    pid_t pid = fork;
    if(pid == 0)
    {
        printf("child_process_with_pid%d", getpid());
        exit(0);
    }
    else
    {
        printf("parent_process_with_pid%d", getpid());
        sleep(60);
    }

    return 0;
}
```

پاسخ

**Orphan: ۱.**

(آ) در این قطعه کد، پردازش فرزند `pid == 0` اجرا می‌شود و پردازش والد `pid != 0` منتظر می‌ماند. به عبارت دیگر، در صورتی که پردازش والد قبل از اتمام پردازش فرزند بسته شود، پردازش فرزند به عنوان یک پردازش orphan باقی می‌ماند.

(ب) در این حالت، پردازش فرزند پیام `"[pid] process with pid "` را چاپ می‌کند و سپس با استفاده از `exit(0)` به پایان می‌رسد.

**Zombie: ۲.**

(آ) در این قطعه کد، پردازش والد `pid != 0` در حالت انتظار `sleep(60)` قرار دارد و پردازش فرزند `pid == 0` اجرا می‌شود.

(ب) پس از اجرای پردازش فرزند، پیام `"[pid] process with pid "` چاپ می‌شود.

(ج) والد در حال انتظار است و در این مدت، پردازش فرزند به پایان می‌رسد و به عنوان یک پردازش zombie باقی می‌ماند.

(د) پس از گذشت زمان `sleep(60)`، والد همچنان اجرا می‌شود و پردازش zombie نهایتاً توسط سیستم عامل از جدول پردازش‌ها حذف می‌شود.

## Listing 7: Some Code

```
int (main)
{
    pid_t pid = fork;
    if(pid > 0)
        printf("parent process");
    else
    {
        sleep(60);
        printf("child process");
    }

    return 0;
}
```

پاسخ

## ۱. Orphan:

- (آ) در این قطعه کد، پردازش والد  $pid > 0$  پیام `parent process` را چاپ می‌کند و پردازش فرزند  $pid == 0$  به قسمت `else` وارد می‌شود. پس از ورود به قسمت `else`، پردازش فرزند از طریق `sleep(60)` به مدت 60 ثانیه منتظر می‌ماند و سپس پیام `child process` را چاپ می‌کند.
- (ب) در این حالت، پردازش والد به عنوان یک پردازش زنده باقی می‌ماند و پردازش فرزند نیز در زمان اجرا زنده است. بنابراین، هیچ پردازش `orphan` ایجاد نمی‌شود.

## ۲. Zombie:

- (آ) در این قطعه کد، پردازش والد  $pid > 0$  پیام `parent process` را چاپ می‌کند و پردازش فرزند  $pid == 0$  به قسمت `else` وارد می‌شود.
- (ب) پس از ورود به قسمت `else`، پردازش فرزند با استفاده از `sleep(60)` به مدت 60 ثانیه منتظر می‌ماند و سپس پیام `child process` را چاپ می‌کند.
- (ج) پس از اجرای پردازش فرزند، والد  $pid > 0$  همچنان در حال اجراست اما هیچ فعالیتی بر روی پردازش فرزند ندارد. اگر پردازش فرزند قبل از اتمام والد بسته شود، پردازش فرزند به عنوان یک پردازش `zombie` باقی می‌ماند.
- (د) در این حالت، پردازش والد پس از گذشت زمان `sleep(60)` به پایان می‌رسد و پردازش `zombie` توسط سیستم عامل از جدول پردازش‌ها حذف می‌شود.

## در این تمرین قصد داریم تا پردازنده‌های سیستم‌عامل لینوکس را مورد بررسی قرار دهیم

اولین دستوری که می‌خواهیم بررسی کنیم، دستور ps است. این دستور اطلاعات پردازنده‌ها را نمایش می‌دهد.

```

+ - ps
+ - PID TTY          TIME CMD
+ - 51041 pts/1    00:00:00 zsh
+ - 51396 pts/1    00:00:00 ps
+ - 

```

برای درک وضعیت فعلی فرآیندهای درحال اجرای سیستم خود، دستور `ps aux` بیشترین مقدار اطلاعاتی که یک کاربر معمولاً نیاز دارد را نمایش می‌دهد.

در خروجی این دستور وضعیت هر پردازنده (Stat) و شماره هر پردازنده (PID) و سایر اطلاعات موجود است. شماره پردازنده‌ها در خروجی دستور `ps` از ۱ شروع شده و می‌دانیم که یک پردازنده جدید در سیستم‌عامل با استفاده از دستورات `fork` و `exec` ساخته می‌شود. بنابراین هر پردازنده ای یک پردازنده والد دارد. برای یافتن شماره پردازنده والد، می‌توان از دستور `ps -f [pid]` استفاده کرد. این دستور PPID که شماره پردازنده والد هست را نیز نمایش می‌دهد. از این دستور استفاده کنید و پردازنده والد پردازنده ۱ را پیدا کنید. دستور `pgrep -P [pid]` همه فرزندان یک پردازنده را نمایش می‌دهد. از این دستور استفاده کنید و همه پردازنده‌هایی که پردازنده پدر مشترکی با پردازنده ۱ دارند را پیدا کنید. همچنین با استفاده از دستور `ps tree` می‌توان ارتباط بین پردازنده‌ها را نمایش داد.

## خروجی دستور ps aux

```

ps aux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root            1  0.0  0.1 168928 13660 ?        Ss   09:41   0:04 /sbin/init splash
root            2  0.0  0.0      0     0 ?        S    09:41   0:00 [kthreadd]
root            3  0.0  0.0      0     0 ?        I<   09:41   0:00 [rcu_gp]
root            4  0.0  0.0      0     0 ?        I<   09:41   0:00 [rcu_par_gp]
root            5  0.0  0.0      0     0 ?        I<   09:41   0:00 [slub_flushwq]
root            6  0.0  0.0      0     0 ?        I<   09:41   0:00 [netns]
root            8  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/0:0H-events_highpri]
root           10  0.0  0.0      0     0 ?        I<   09:41   0:00 [mm_percpu_wq]
root           11  0.0  0.0      0     0 ?        I    09:41   0:00 [rcu_tasks_kthread]
root           12  0.0  0.0      0     0 ?        I    09:41   0:00 [rcu_tasks_rude_kthread]
root           13  0.0  0.0      0     0 ?        I    09:41   0:00 [rcu_tasks_trace_kthread]
root           14  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/0]
root           15  0.0  0.0      0     0 ?        I    09:41   0:21 [rcu_preempt]
root           16  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/0]
root           17  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/0]
root           19  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/0]
root           20  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/1]
root           21  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/1]
root           22  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/1]
root           23  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/1]
root           25  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/1:0H-events_highpri]
root           26  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/2]
root           27  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/2]
root           28  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/2]
root           29  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/2]
root           31  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/2:0H-events_highpri]
root           32  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/3]
root           33  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/3]
root           34  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/3]
root           35  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/3]
root           37  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/3:0H-events_highpri]
root           38  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/4]
root           39  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/4]
root           40  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/4]
root           41  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/4]
root           43  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/4:0H-events_highpri]
root           44  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/5]
root           45  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/5]
root           46  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/5]
root           47  0.1  0.0      0     0 ?        S    09:41   0:49 [ksoftirqd/5]
root           49  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/5:0H-events_highpri]
root           50  0.0  0.0      0     0 ?        S    09:41   0:00 [cpuhp/6]
root           51  0.0  0.0      0     0 ?        S    09:41   0:00 [idle_inject/6]
root           52  0.0  0.0      0     0 ?        S    09:41   0:00 [migration/6]
root           53  0.0  0.0      0     0 ?        S    09:41   0:00 [ksoftirqd/6]
root           55  0.0  0.0      0     0 ?        I<   09:41   0:00 [kworker/6:0H-events_highpri]

```

## خروجی دستور ps -f [pid]

```

~ ps -f 1
UID PID PPID C STIME TTY STAT TIME CMD
root 1 0 0 09:41 ? Ss 0:04 /sbin/init splash
~

```



خروجی دستور `pgrep -P [pid]`

```

reza@r324:~$ pgrep -P 1
868
985
1522
1524
1525
1619
1621
1622
1623
1624
1638
1631
1634
1643
1647
1652
1653
1654
1708
1721
1725
1731
1769
1762
1763
1766
1792
1793
1795
1798
1839
2043
2098
2257
2540
49547

```

خروجی دستور `pstree`

```

reza@r324:~$ pstree
systemd
├── ModemManager─┬─3*[{ModemManager}]
├── NetworkManager─3*[{NetworkManager}]
├── accounts-daemon─3*[{accounts-daemon}]
├── avahi-daemon─avahi-daemon
├── bluetoothd
├── colord─3*[{colord}]
├── cron
├── cups-browsed─3*[{cups-browsed}]
├── cupsd─2*[{dbus}]
├── dbus-daemon
├── fwupd─5*[{fwupd}]
├── gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─gnome-session-b─3*[{gnome-session-b}]
│       │                 │                 │                 │3*[{gdm-wayland-ses}]
│       │                 └─3*[{gdm-session-wor}]
│       └─3*[{gdm3}]
├── irqbalance─{irqbalance}
├── 2*[kerneloops]
├── polkitd─3*[{polkitd}]
├── power-profiles─3*[{power-profiles-}]
├── rsyslogd─3*[{rsyslogd}]
├── rtkit-daemon─2*[{rtkit-daemon}]
├── run-cups-browse─run-cups-browse─sleep
├── run-cupsd─cupsd
├── snapd─18*[{snapd}]
├── switcheroo-cont─3*[{switcheroo-cont}]
├── systemd─┬─(sd-pam)
│           ├── at-spi2-registr─3*[{at-spi2-registr}]
│           ├── dbus-daemon
│           ├── dconf-service─3*[{dconf-service}]
│           ├── evince─5*[{evince}]
│           ├── evince-d─3*[{evince-d}]
│           ├── evolution-addre─6*[{evolution-addre}]
│           ├── evolution-calen─9*[{evolution-calen}]
│           ├── evolution-sourc─4*[{evolution-sourc}]
│           ├── gcr-ssh-agent─2*[{gcr-ssh-agent}]
│           ├── 2*[{gjs}─11*[{gjs}]]
│           ├── gnome-keyring-d─4*[{gnome-keyring-d}]
│           ├── gnome-session-b─┬─at-spi-bus-laun─┬─dbus-daemon
│                               │                 │4*[{at-spi-bus-laun}]
│                               │                 └─evolution-alarm─6*[{evolution-alarm}]
│                               └─gsd-disk-utilit─3*[{gsd-disk-utilit}]
│                                   └─update-notifier─5*[{update-notifier}]
│                                       └─4*[{gnome-session-b}]
│                                           └─gnome-session-c─{gnome-session-c}
├── gnome-shell─┬─Xwayland─12*[{Xwayland}]
│               ├── Firefox─4*[{Isolated Web Co─27*[{Isolated Web Co}]]
│               └─31*[{Isolated Web Co─25*[{Isolated Web Co}]]

```