موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(fork, copy on write, stack, heap)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندور اس تنن‌بام

ارسطو خلیلی‌فر

تست‌های فصل پنجم

۹۷- با اجرای کد زیر در نهایت چند پردازه خواهیم داشت؟

(مهندسی IT- دولتی ۹۸)

```
main ()
{
        for(i = 1; i < 4; i++)
                fork();
}
```

۱) 2

۲) 4

۳) 8

۴) 16

پاسخ‌های فصل پنجم

۹۷- گزینه (۳) صحیح است.

یک فرآیند می‌تواند، چندین فرآیند جدید را از طریق یک فراخوان سیستمی ایجاد فرآیند در طول اجرا، ایجاد نماید. فرآیند ایجاد کننده، فرآیند پدر (Parent Process) و فرآیند ایجاد شده، فرآیند فرزند (Child Process) نامیده می‌شود. هر یک از این فرآیندهای جدید نیز می‌توانند فرآیندهای دیگر را بوجود آورند و **درختی** از فرآیندها را تشکیل دهند.

توجه: بیشتر سیستم عامل‌ها (یونیکس، لینوکس و ویندوز) فرآیندها را توسط یک مشخصه فرآیند(process identifier) یا pid به صورت یکتا که معمولا یک عدد صحیح است، مشخص می‌سازند.

توجه: افزون بر منابع فیزیکی و منطقی که یک فرآیند فرزند پس از ایجاد بدست می‌آورد، داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می‌شود.

توجه: سیستم عامل Unix و Linux برای ایجاد یک فرآیند فرزند (جدید) از فراخوان سیستمی fork استفاده می‌کند. در این سیستم عامل جهت پیاده‌سازی مفهوم حافظه مجازی و همچنین **صرفه‌جویی در مصرف حافظه**، تکنیک Copy-On-Write **می‌تواند** مورد استفاده قرار بگیرد.

توجه: Copy-On-Write یکی از فیلدهای جدول صفحه، به طول یک بیت است و هنگامی که بیش از یک فرآیند در یک صفحه باشد، این فیلد براساس تعداد فرآیندهای موجود در یک صفحه مقدار می‌گیرد.

توجه: هنگامی که فرآیند فرزند (جدید) ایجاد می‌شود، در مورد **زمان‌بندی پردازنده** و به تبع اجرای فرآیندهای پدر و فرزند، دو حالت ممکن است رخ دهد:

۱-فرآیند پدر بطور همروند در سیستم تک پردازنده‌ای و بطور موازی در سیستم چند پردازنده‌ای با فرآیند فرزند زمان‌بندی و به تبع اجرا شود.

۲-فرآیند پدر منتظر می‌ماند تا کار چند و یا همه فرزندانش تمام شود.

توجه: به طور کلی مستقل از اینکه فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** (تکنیک Copy-On-Write مورد استفاده قرار نگیرد) و یا **مشترک** (تکنیک Copy-On-Write مورد استفاده قرار بگیرد) **باشد، محتوای فرآیند فرزند** در ابتدا از نظر داده‌ها، مقدارها و کد شامل Stack ، Data ، Heap ، Register ، PCB و Code، یک کپی **کاملا، دقیقا و یکسان** از فرآیند پدر است، چون داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می شود. حتی مقادیر PCB فرآیند

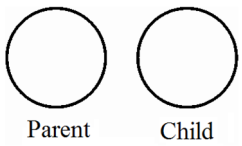
پدر در PCB فرآیند فرزند کپی می‌شـــود و تنها چیزی که در محتوای PCB فرآیند پدر و PCB فرآیند فرزند تفاوت دارد، مقدار pid است چون هر فرآیند pid مختص به خودش را دارد، در واقع pid فرآیند پدر با pid فرآیند فرزند متفاوت اســـت. همچنین دقت کنید که فضـــای آدرس PCB فرآیند پدر از فضای آدرس PCB فرآیند فرزند مستقل است، اما همانطور که گفتیم بعد از اجرای fork محتوای PCB فرآیند پدر در PCB فرآیند فرزند کپی می‌شود.

توجه: در مفهوم fork برای ایجاد یک فرآیند فرزند (جدید) می‌توان تکنیک Copy-On-Write را مورد استفاده قرار داد یا نداد. اگر تکنیک Copy-On-Write مورد استفاده قرار بگیرد، پس از دستور fork جهت صرفه‌جویی در مصرف حافظه، به جای آنکه صفحات حافظه فرآیند پدر برای فرآیند فرزند کپی شود، صفحات حافظه فرآیند پدر با فرآیند فرزند به اشتراک گذاشته می‌شود.

توجه: به تفاوت **فضـــای آدرس فرآیند** (ظرف فرآیند و محل ذخیره‌ســـازی فرآیند) و **محتوای فرآیند** (مقادیر فرآیند، داده و کد) دقت داشته باشید.

توجه: هنگامی که فرآیند فرزند (جدید) ایجاد می شود، در مورد **فضای آدرس** و **محتوای فرآیند پدر و فرزند**، چهار حالت ممکن است رخ دهد:

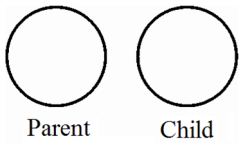
۱- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد اســتفاده قرار نگیرد، یعنی محتوای فرآیند پدر و فرآیند فرزند بســته به شــرایط، **فقط در حد تغییر مقدار متغیرها و نه تغییر برنامه و کد تغییر کند**، آنگاه فضـــای آدرس فرآیند پدر و فرآیند فرزند شــامل Stack ، Data ، Heap و Code کاملا مســتقل و جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرآیند فرزند در دو فضــای مســتقل و جدا از هم انجام می‌گردد. شــکل زیر گویای مطلب است:



توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

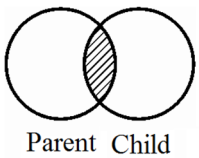
۲- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار بگیرد و یک برنامه و قطعه کد جدید به درون خود بار کند، یعنی محتوای فرآیند فرزند بسته به شرایط، **در حد تغییر ساختار و تغییر مقدار متغیرها و حتی تغییر برنامه و کد تغییر کند**، آنگاه فضــای آدرس فرآیند پدر و فرآیند فرزند شــامل Stack ، Data ، Heap و Code به تبع

اجرای دستور exec و ساخت محتوای کاملا جدید در فرآیند فرزند کاملا مستقل و جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرآیند فرزند در دو فضای مستقل و جدا از هم انجام می‌گردد. شکل زیر گویای مطلب است:



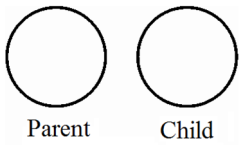
توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

۳- اگر فضای آدرس فرآیند پدر و فرآیند فرزند در ابتدا **مشترک** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار بگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار نگیرد، یعنی محتوای فرآیند پدر و فرآیند فرزند بسته به شرایط، **فقط در حد تغییر مقدار متغیرها و نه تغییر برنامه و کد تغییر کند،** آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code در ابتدا به اشتراک گذاشته می شود و جهت صرفه‌جویی در مصرف حافظه، فرآیندهای پدر و فرزند در ابتدای کار از صفحات کد و داده به صورت مشترک استفاده می‌کنند، البته تا زمانی که فقط عمل خواندن (Read) بین فرآیندها مدنظر باشد، این صفحات به صورت مشترک استفاده می‌گردد. اما بر طبق تکنیک Copy-On-Write هرگاه یکی از دو فرآیند پدر یا فرزند بخواهد محتوای صفحه‌ای از صفحات مشترک داده و نه کد را تغییر دهد (مثلاً چیزی بنویسد) یک کپی جداگانه از آن صفحه برای آن فرآیند ساخته می شود و فرآیند از آن به بعد از آن صفحه استفاده می‌کند (به جای صفحه مشترک) و فرآیندهای دیگر از صفحات اصلی (مشترک) استفاده می‌کنند. بدین ترتیب محرمانگی داده‌ها حفظ شده و تغییرات صورت گرفته توسط یک فرآیند بر روی سایر فرآیندها اثر نخواهد داشت. در تکنیک Copy-On-Write فقط صفحاتی کپی می شوند که توسط فرآیندی تغییر یابند، و تمام صفحات بدون تغییر می‌تواند بین فرآیندهای پدر و فرزند بصورت مشترک استفاده شود. به عبارت دیگر پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می شود، همه متغیرها بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. شکل زیر گویای مطلب است:



توجه: بنابراین رابطه $card(Parent \cap Child) = k$ به معنی k صفحه م شترک، برای **ف ضای آدرس مشترک** برقرار است.

۴- اگر فضـای آدرس فرآیند پدر و فرآیند فرزند در ابتدا **م شترک** باشـد یعنی تکنیک Copy-On-Write مورد استفاده قرار بگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید برای فرآیند فرزند مورد اســتفاده قرار بگیرد و یک برنامه و قطعه کد جدید به درون خود بار کند، یعنی محتوای فرآیند فرزند بسـته به شـرایط، **در حد تغییر ساختار و تغییر مقدار متغیرها و حتی تغییر برنامه و کد تغییر کند،** آنگاه فضای آدرس فرآیند پدر و فرآیند فرزند شامل Stack ، Data ، Heap و Code به تبع اجرای دستور exec و ساخت محتوای کاملا جدید در فرآیند فرزند کاملا مستقل و جدا از هم خواهد بود و تغییرات در محتوای فرآیند پدر و فرزند در دو فضای مستقل و جدا از هم انجام می‌گردد. شکل زیر گویای مطلب است:

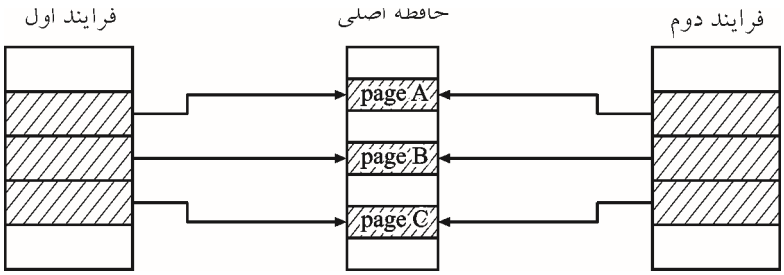


توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه م شترک، برای **ف ضای آدرس مستقل** برقرار است.

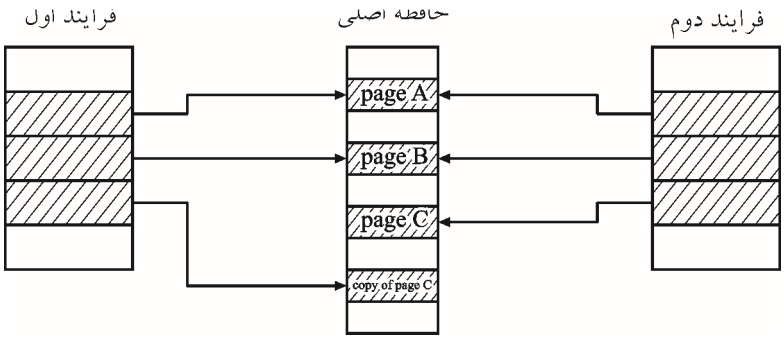
توجه: در ســیســـتم عامل Unix یک فرآیند فرزند پس از ایجاد توســط فراخوانی fork، می‌تواند بلافاصله از فراخوانی سیستمی ()exec استفاده کند و کل فضای آدرس حافظه‌ی خود را جایگزین کند و از ادامه شراکت با پدر صرفه نظر کند. بنابراین، با این کار، حافظه‌ی جداگانه‌ای برای فرآیند فرزند ایجاد می‌شـــود. با توجه به این مطلب اگر بعد از اجرای fork و ایجاد فرآیند فرزند، ابتدا فرآیند پدر اجرا گردد، ممکن است فرآیند پدر بخواهد بطور خصوصی در یکی از صفحات چیزی بنویسـد و باعث ایجاد یک کپی از صـفحه بر اسـاس تکنیک Copy-On-Write شـود. حال اگر در ادامه فرآیند فرزند اجرا گردد و در همان ابتدا از فراخوان سیستمی ()exec ا ستفاده کند و راه خود را از پدر جدا کند و شراکت را برهم زند، صفحاتی که پدر به دلیل تغییرات خود ایجاد کرده بود، سـربار به حسـاب می‌آیند و کار بیهوده تلقی می‌گردد، مانند پدری که پس از فرزنددار شـدن برای صرفه‌جویی در هزینه‌ها، از خانه‌ی خود به شکل اشتراکی استفاده می‌کند. اما این پدر بعدها به دلیل کارهای شخصی خود خانه‌ی دیگری را نیز تهیه می‌کند و بعد از تهیه خانه‌ی دوم متوجه می شود، که فرزند راه خود را جدا کرده است، و شراکت را بر هم زده است، و فرزند نیز خانه‌ای برای خود تهیه کرده اسـت، حال خانه‌ی دوم پدر برای رسـیدگی به امور شـخصـی بلااسـتفاده می‌ماند و این سربار ا ست، زیرا دیگر شراکتی در کار نیست، فرزندی نیست، همان خانه‌ی اول برای پدر کافی بود. بهتر بود پدر صبر می‌کرد، تا اول فرزند تصمیم بگیرد، سپس بر اساس تصمیم فرزند، پدر نیز تصمیم خود را می‌گرفت.

نتیجه: حال مجدداً برگردید به وادی کامپیوتر، در صورتی که اگر بعد از اجرای فراخوانی سیستمی fork، ابتدا فرآیند فرزند فراخوانی و اجرا شـــود، ممکن اســـت در همان ابتدای اجرایش دســـتور exec() را اجرا کند و در نتیجه از آن به بعد، از فضای حافظه‌ی شخصی خود استفاده کند، که در این صورت اگر فرآیند پدر بخواهد چیزی بر روی صفحات مشترک شده بنویسد، دیگر نیازی به کپی‌کردن آن صفحه نخواهد بود و این یعنی حذف سربار و افزایش کارایی.

شکل زیر گویای مطالب می‌باشد:



قبل از اینکه فرآیند اول صفحه C را تغییر دهد.



بعد از اینکه فرآیند اول صفحه C را تغییر دهد.

توجه: در این تکنیک زمانی فرآیند پدر می‌تواند خاتمه یابد که یک کپی از صـــفحات آن برای هر یک از فرزندانش ایجاد شده با شد (یعنی صفحات تمامی فرآیندهای فرزند تغییر کرده با شند، به عبارت دیگر همه فرزندان همه صـــفحه‌های مربوط به خود را تغییر داده باشـــند.) و دیگر نیازی به صفحات فرآیند پدر نباشد.

توجه: همانطور که گفتیم، سیستم عامل Unix و Linux برای ایجاد یک فرآیند فرزند (جدید) از فراخوان سیستمی fork استفاده می‌کند، بنابراین fork برای ایجاد یک فرآیند فرزند مورد استفاده قرار می‌گیرد، هدف fork ایجاد یک فرآیند فرزند برای فرآیند فراخوانی کننده آن یعنی فرآیند پدر است، به عبارت دیگر fork برای یک پدر، به شکل طبیعی یک فرزند به دنیا می‌آورد، fork متخصص زایمان است. فراخوان سیستمی fork هیچ آرگومان ورودی ندارد، اما مقدار بازگشتی دارد، fork در حالت اجرای موفق دو مقدار برمی‌گرداند که یکی برای فرآیند فرزند برابر مقدار صفر که به آن پاس داده می‌شود و یکی دیگر هم برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است، وقتی فرزندی به دنیا می‌آید، علاوه بر آینکه خودش صاحب pid می‌شود، pid آنرا تحویل پدرش هم می‌دهند. مانند وقتی که فرزندی به دنیا می‌آید، علاوه بر اینکه خودش صاحب شماره شناسنامه می‌شود، شماره شناسنامه آنرا تحویل پدرش هم می‌دهند. همچنین عدد صفر پاس داده شده به فرآیند فرزند هم به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد. نوع مقدار بازگشتی fork از جنس pid_t است که در کتابخانه sys/types.h زبان C و C++ تعریف شده است، البته به طور معمول در سایر زبان‌ها از نوع integer است. همچنین یک فرآیند پدر یا فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع ()getpid استفاده نماید.

توجه: با استفاده از مقدار حاصل از بازگشت اجرای تابع fork، می‌توان متوجه شد که در ادامه و پس از به دنیا آمدن فرزند نو رسیده، فرآیند پدر در چه مسیری اجرا شود و فرآیند فرزند در چه مسیری اجرا شود.

توجه: فراخوانی fork در حالت عدم اجرای موفق یک عدد صحیح کوچکتر از صفر برمی‌گرداند.

توجه: مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود.

توجه: اینکه فرآیند پدر یا فرآیند فرزند به چه ترتیبی اجرا شوند، بستگی به زمان‌بند پردازنده و شرایط درون خود فرآیندها دارد. بسته به شرایط ممکن است اول فرآیند پدر اجرا شود و بعد فرآیند فرزند و یا اول فرآیند فرزند اجرا شود و بعد فرآیند پدر و یا هر دو باهم به طور همروند در سیستم تک پردازنده‌ای یا موازی در سیستم چند پردازنده‌ای اجرا شوند.

توجه: فرآیند فرزند یک شماره pid یکتا دارد که با فرآیند پدر متفاوت است. همچنین فرآیند پدر نیز یک شماره pid یکتا دارد که با فرآیند فرزند متفاوت است.

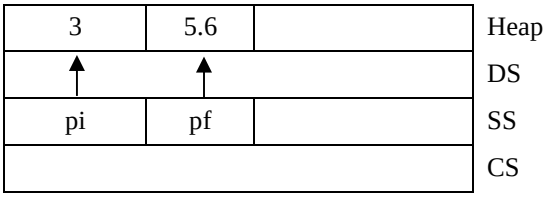
توجه: ppid نشان دهنده pid فرآیند پدر است. بنابراین شماره ppid فرآیند فرزند، برابر pid فرآیند پدر است.

توجه: هر برنامه هنگامی که در حافظه قرار می‌گیرد تا اجرا شـود، حاوی سـه قسـمت اصـلی کد (CS: Code Segment)، داده (DS: Data Segment) و پشته (SS: Stack Segment) است. مابقی حافظه که در اختیار برنامه نبوده و آزاد می‌باشـد به حافظ Heap یا حافظه پویا اختصـاص داده می شود. دستورالعمل‌های برنامه در قسمت کد، متغیرهای سراسری در قسمت داده و متغیرهای محلی در قسمت پشته ساخته می‌شوند. متغیرهای سراسری ابتدای برنامه و بیرون همه توابع ساخته شـده و تا انتهای برنامه فضـای آنها حفظ می‌گردد. متغیرهای محلی به محض ورود به زیربرنامه سـاخته شـده و هنگام اتمام زیربرنامه از بین می‌روند. به کمک مفهوم اشـاره‌گرها و حافظه Heap می‌توان متغیرهایی پویا در حافظه Heap پدید آورد (حداکثر به اندازه حافظه Heap) و همچنین هر وقت که دیگر نیازی به آنها نباشـد، می‌توان آنها را از بین برده و فضـای آنها را به Heap برگرداند. در زبان C برای گرفتن فضـا در حافظه Heap از تابع malloc و برای رهاسـازی آن از تابع free استفاده می‌شـود. معرفی این توابع در فایل stdlib.h قرار دارد. فرم کلی این توابع به صـورت زیر است:

```
void * malloc (اندازه فضای متغیر بر حسب بایت) ;
void free (void *p) ;
```

مثال: قطعه کد زیر فرم استفاده از دستورات malloc و free و نحوه تعریف متغیرهای پویا را نشان می‌دهد:

```
#include  <stdio.h> /* printf */
#include  <stdlib.h> /* malloc , free */
int main(void)
{
    int *pi;
    float *pf;
    pi = (int *) malloc (sizeof (int));
   *pi = 3;
    pf = (float *) malloc (sizeof (float));
   *pf = 5.6;
    printf("%f",*pi +*pf); /* 8.6 */
   free(pi);
   free(pf);
    return 0;
}
```



مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که یک متغیر را که مقدارش برابر با 5 است را در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include  <stdio.h> /* printf */
int main(void)
{
   int i = 5;
   printf("i=%d",i);
   return 0;
}
```

توجه: این برنامه در سیستم عامل UNIX و Linux به زبان C تحت کامپایلر gcc نوشته شده است.

خروجی نهایی برنامه به صورت زیر است:

```
i = 5
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که یک متغیر را که مقدارش برابر با 5 است را پس از اجرای دستور fork در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include <stdio.h>   /*printf */
#include <unistd.h>   /*fork */
int main ()
{
   int i = 5;
   printf("Hello!");
```

```
  /*fork a child process*/
  fork();
  printf("i=%d",i);
  return 0;
}
```

توجه: این برنامه در سیستم عامل UNIX و Linux به زبان C تحت کامپایلر gcc نوشته شده است.

توجه: در سیستم عامل ویندوز ایجاد فرآیند توسط دستور ()fork انجام نمی‌شود، در ویندوز ایجاد فرآیند توسط create process routines انجام می‌شود.

توجه: برنامه اجرا می‌شـــود و در اولین خط مقدار متغیر محلی i برابر با 5 می‌شـــود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می شود که داخل Stack Segment تعریف و مقداردهی می‌شـــود. در خط بعد کلمه‌ی !Hello توســـط دســـتور printf در خروجی نمایش داده می‌شـــود، تا اینجا همه چیز عادی و طبق روال معمول اســـت. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می شود. همانطور که گفتیم محتوای فرآیند فرزند در ابتدا از نظر داده‌ها، مقدارها و کد شـــامل Stack ، Data ، Heap ، Register ، PCB و Code، یک کپی **کاملا**، **دقیقا** و **یک سان** از فرآیند پدر است، چون داده و مقداردهی اولیه از فرآیند پدر به فرآیند فرزند کپی و پاس داده می‌شـود. حتی مقادیر PCB فرآیند پدر در PCB فرآیند فرزند کپی می‌شود و تنها چیزی که در محتوای PCB فرآیند پدر و PCB فرآیند فرزند تفاوت دارد، مقدار pid اســـت چون هر فرآیند pid مختص به خودش را دارد، در واقع pid فرآیند پدر با pid فرآیند فرزند متفاوت است. همچنین دقت کنید که فضای آدرس PCB فرآیند پدر از فضای آدرس PCB فرآیند فرزند مســتقل اســت. **در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می کنند، زیرا فیلد شمارنده برنامه PC موجود در PCB فرآیند پدر و فرزند که پس از دستور fork مقادیر یک سانی نیز دارند به دستور پس از fork اشـــاره می‌کند**. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هســـتند، مقدار متغیر i در هر دو فرآیند برابر مقدار 5 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می شود دستور printf به معنای نمایش مقدار متغیر i در خروجی است، هر کدام از فرآیندهای پدر و فرزند پس از رسیدن به دستور printf مقدار متغیر i را به طور **مستقل** در خروجی نمایش می‌دهند. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مســـیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شـــود، همه متغیرهایی که مقداردهی اولیه شـــده‌اند بعد از اجرای fork مقادیر یکســـان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد

بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشـــت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشـــت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسـان و برابر i و برابر مقدار 5 اسـت، اما فضـای آدرس فرآیند پدر و فرزند متفاوت است.

خروجی نهایی برنامه به صورت زیر است:

Parent :

```
#include< stdio.h >
#include  < unistd.h >
int main ()
{
  int i  =  5;
  printf ("Hello!");
  /*fork a child process*/
  fork();
 →printf ("i = %d",i);
  return 0;
}
```

Child :

```
#include< stdio.h >
#include  < unistd.h >
int main ()
{
  int i  =  5;
  printf ("Hello!");
  /*fork a child process*/
  fork();
  → printf ("i = %d",i);
  return 0;
}
```

```
Hello!
i = 5
i = 5
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر ا ست با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال مقدماتی فوق مسیر فرآیند پدر و فرزند به دلیل نبود شرط if از هم متمایز نشده بودند.

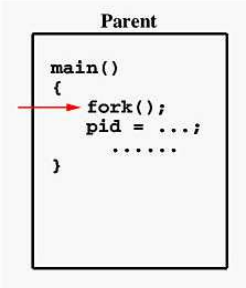
مثال: برنامه‌ی زیر یک قطعه کد به زبان C ا ست که مقدار pid را تو سط د ستور printf داخل یک حلقه‌ی for پس از اجرای دسـتور fork در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0

#include <stdio.h>   /* printf  */
```

```
#include <unistd.h>   /* fork */
#include  <sys/types.h>   /* pid_t */
int  main()
{
    pid_t  pid;
    int    i = 0;
    fork();
    pid = getpid() ;
    for (i = 1 ; i < 4 ; i++) {
            printf("pid=%d \n" , pid);
      }
    return 0;
}
```

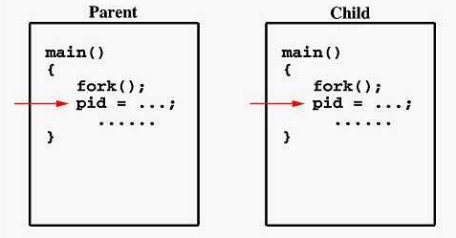
توجه: برنامه اجرا می‌شـــود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شـــود. دقت کنید که متغیر i داخل تابع main تعریف شده ا ست و یک متغیر محلی محسوب می شود که داخل Stack Segment تعریف و مقداردهی می‌شـود. در خط بعدی، دسـتور fork قرار دارد، وقتی که دسـتور fork اجرا می‌شـــود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شود.



توجه: در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دسـتور fork را اجرا می کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار اولیه متغیر i در هر دو فرآیند برابر مقدار 0 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می شود دستور pid = getpid()، یک فرآیند پدر یا فرزند می‌تواند جهت بازیابی مقدار process ID یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید. هر کدام از فرآیندهای پدر و فرزند پس از ر سیدن به د ستور pid = getpid () مقدار متغیر pid را به طور م ستقل بر ا ساس تابع getpid () مقداردهی می‌کنند. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، م سیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شـــود، همه

متغیرهایی که مقداردهی اولیه شـده‌اند بعد از اجرای fork مقادیر یکسـان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضـای آدرس مختص به خود را دارند، هرگونه تغییر، مسـتقل از سـایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشـت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر i و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور printf داخل یک حلقه for است که 3 بار تکرار می شود. هر کدام از فرآیندهای پدر و فرزند پس از رسیدن به دستور printf داخل حلقه for مقدار متغیر pid را به طور مستقل بر اساس تابع getpid () مقداردهی می‌کنند و در خروجی نمایش می‌دهند.



خروجی نهایی برنامه به صورت زیر است:

Parent :

```
#include < stdio.h >
#include < unistd.h >
#include< sys / types.h >
int main()
{
  pid _ t pid;
  int  i = 0;
  fork();
  → pid  =  getpid();
  for (i  =  1 ; i  <  4 ; i++){
        printf ("pid = %d  \n", pid);
    }
  return  0;
}
```

Child :

```
#include < stdio.h >
#include < unistd.h >
#include< sys / types.h >
int main()
{
  pid _ t pid;
  int  i = 0;
  fork();
  → pid  =  getpid();
  for (i  =  1 ; i  <  4 ; i++){
        printf ("pid = %d  \n", pid);
    }
  return  0;
}
```

```
pid = 2600
pid = 2600
pid = 2600
pid = 2603
pid = 2603
pid = 2603
```

توجه: اینکه فرآیند پدر یا فرآیند فرزند به چه ترتیبی اجرا شـــوند، بســـتگی به زمان‌بند پردازنده و شـرایط درون خود فرآیندها دارد. بسـته به شـرایط، ممکن اسـت اول فرآیند پدر اجرا شـود و بعد فرآیند فرزند و یا اول فرآیند فرزند اجرا شود و بعد فرآیند پدر و یا هر دو باهم به طور همروند در سیستم تک پردازنده‌ای یا موازی در سیستم چند پردازنده‌ای اجرا شوند. برای مثال یک فرم دیگر خروجی بر اســـاس زمان‌بندی همروند پردازنده در ســـیســـتم تک پردازنده‌ای یا زمان‌بندی موازی پردازنده در سیستم چند پردازنده‌ای می‌تواند به صورت زیر باشد:

```
pid = 2600
pid = 2603
pid = 2600
pid = 2603
pid = 2600
pid = 2603
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال مقدماتی فوق مسیر فرآیند پدر و فرزند به دلیل نبود شرط if از هم متمایز نشده بود.

توجه: این مدل اســـتفاده از fork کاربرد خاصـــی ندارد. وقتی fork و ایجاد فرآیند فرزند(جدید)، کارآمد است که بتوان بعد از اجرای fork دو محتوای متفاوت از فرآیند پدر و فرآیند فرزند را اجرا کرد و نه این که دقیقا همان کد قبلی را اجرا کرد. برای اینکه بتوان بعد از اجرای fork دو محتوای متفاوت را اجرا کرد یک راه بیشتر نداریم و آن هم استفاده از خروجی و مقدار بازگشتی دستور fork است. دستور fork در فرآیند پدر و فرآیند فرزند دو خروجی متفاوت ایجاد می‌کند. هر فرآیند

در سیستم عامل یک شماره‌ی مختص به خود دارد که سیستم عامل برای شناسایی و کار با فرآیندها از آن استفاده می‌کند که به آن process id یا pid گفته می شود. در فرآیند پدر، خروجی و مقدار بازگشتی fork شناسه‌ی pid فرآیند فرزند است، در حالی که خروجی و مقدار بازگشتی fork در فرآیند فرزند برابر با صفر است. به این ترتیب با استفاده از تفاوت خروجی و مقدار بازگشتی fork در فرآیند پدر و فرآیند فرزند، می‌توان کاری کرد که فرآیند پدر و فرآیند فرزند بعد از دستور fork کارهای متفاوتی انجام دهند، که در ادامه، مثالی از این مورد را بررسی می‌کنیم.

توجه: دقت کنید که اگر خروجی fork یک عدد منفی بود، بدین معنی است که برنامه موفق به ایجاد یک فرآیند فرزند(جدید) نشده است.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار i، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

//gcc 5.4.0

```
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include  <sys/types.h>  /* pid_t */
int  main()
{
    pid_t  pid;
    int i = 0;

       pid = fork (); /*fork a child process*/

       if (pid > 0) { /*Parent Process:*/
        /*When fork() returns a positive number, we are in the parent process*/
        /*the fork return value is the PID of the newly created child process*/

          printf ("*** Parent Process Begin *** \n");

          i = i + 1 ;
          printf ("i = %d \n" , i);
          printf ("Process id = %d \n", getpid () );
          printf ("pid = %d \n" , pid);

          printf ("*** Parent Process End *** \n");
     }
       else if (pid == 0) { /*Child Process:*/
        /*When fork() returns 0, we are in the child process.*/

          printf ("*** Child Process Begin *** \n");

          i = i - 1 ;
          printf ("i = %d \n" , i);
```

```
        printf ("Process id = %d \n", getpid () );
        printf ("pid = %d \n" , pid);

        printf ("*** Child Process End *** \n");
    }

     else { /*error occurred*/
     /*When fork() returns a negative number, an error happened*/
      printf ("fork creation failed!!! \n ");
    }

   return 0;
 }
```

توجه: برنامه اجرا می‌شـــود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شـــود. دقت کنید که متغیر i داخل تابع main تعریف شده ا ست و یک متغیر محلی محـسوب می شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دســـتور fork را اجرا می‌کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هســتند، مقدار اولیه متغیر i در هر دو فرآیند برابر مقدار 0 اســت. دســتور بعدی که توســط هر دو فرآیند پدر و فرزند اجرا می‌شود دستور if و else if است. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مســـیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شـــود، همه متغیرهایی که مقداردهی اولیه شـــده‌اند بعد از اجرای fork مقادیر یکســان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضــای آدرس فرآیند پدر و فرزند متفاوت اســـت. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضـــای آدرس مختص به خود را دارند، هرگونه تغییر، مســـتقل از ســـایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشـــت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر i و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
i = i + 1 ;
printf ("i = %d \n" , i);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 اســت و پس از اجرای دســتور i = i + 1 برابر 1 می‌شــود و این تغییر روی فرآیند فرزند اثری ندارد.

i = i + 1; **printf ("i = %d \n" , i);**	**i = 1**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
i = i - 1 ;
printf ("i = %d \n" , i);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

i = i - 1; **printf ("i = %d \n" , i);**	**i = -1**

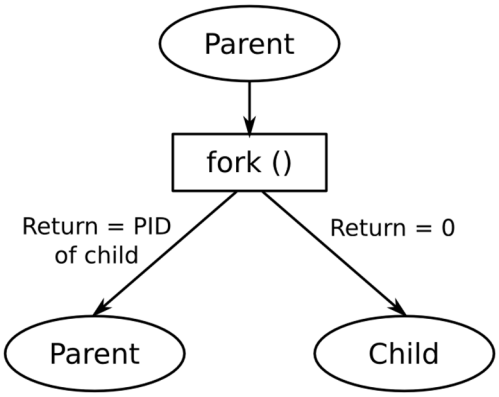
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

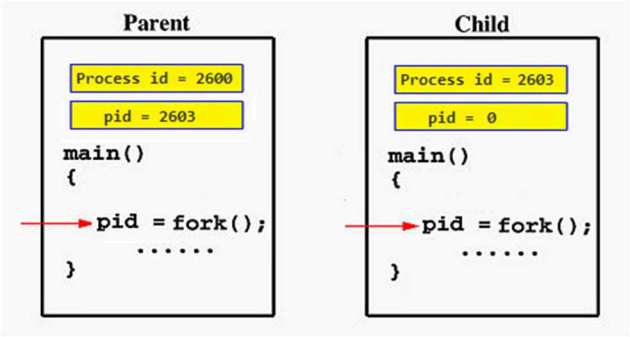
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 0**

شکل زیر گویای مطلب است:



شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
i = 1
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
i = -1
Process id = 2603
pid = 0
*** Child Process End ***
```

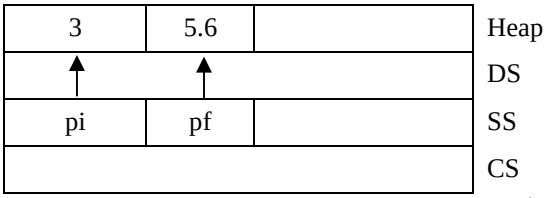
توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال فوق مسیر فرآیند پدر و فرزند به دلیل وجود شرط if از هم متمایز شده بود.

توجه: هر برنامه هنگامی که در حافظه قرار می‌گیرد تا اجرا شود، حاوی سه قسمت اصلی کد (CS: Code Segment)، داده (DS: Data Segment) و پشته (SS: Stack Segment) است. مابقی حافظه که در اختیار برنامه نبوده و آزاد می‌باشد به حافظ Heap یا حافظه پویا اختصاص داده می شود. دستورالعمل‌های برنامه در قسمت کد، متغیرهای سراسری در قسمت داده و متغیرهای محلی در قسمت پشته ساخته می‌شوند. متغیرهای سراسری ابتدای برنامه و بیرون همه توابع ساخته شده و تا انتهای برنامه فضای آنها حفظ می‌گردد. متغیرهای محلی به محض ورود به زیربرنامه ساخته شده و هنگام اتمام زیربرنامه از بین می‌روند. به کمک مفهوم اشاره‌گرها و حافظه Heap می‌توان متغیرهایی پویا در حافظه Heap پدید آورد (حداکثر به اندازه حافظه Heap) و همچنین هر وقت که دیگر نیازی به آنها نباشد، می‌توان آنها را از بین برده و فضای آنها را به Heap برگرداند. در زبان C برای گرفتن فضا در حافظه Heap از تابع malloc و برای رهاسازی آن از تابع free استفاده می‌شود. معرفی این توابع در فایل stdlib.h قرار دارد. فرم کلی این توابع به صورت زیر است:

```
void * malloc (اندازه فضای متغیر بر حسب بایت) ;
void free (void *p) ;
```

مثال: برنامه زیر دستورات و نحوه تعریف متغیرهای پویا را نشان می‌دهد:

```
#include  <stdio.h> /* printf */
#include  <stdlib.h> /* malloc , free */
int main(void)
{
   int *pi;
   float *pf;
   pi = (int *) malloc (sizeof (int));
  *pi = 3;
   pf = (float *) malloc (sizeof (float));
  *pf = 5.6;
   printf("%f",*pi +*pf); /* 8.6 */
  free(pi);
  free(pf);
   return 0;
}
```



توجه: pi* می‌گوید در آدرسی که توسط pi مشخص شده است مقدار 3 را قرار دهد یا بخواند و همچنین pf* می‌گوید در آدرسی که توسط pf مشخص شده است مقدار 5.6 را قرار دهد یا بخواند. به تفاوت pi و pi* دقت نمایید. در pi توسط دستور malloc آدرس یک متغیر پویا رزرو می‌شود، اما توسط pi* آن آدرس موجود در pi مقداردهی می‌شود و یا مقدار آن خوانده می‌شود.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار pi* تعریف شده در حافظه Heap، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0

#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include  <sys/types.h>  /* pid_t */
int  main()
{
    pid_t  pid;
    int *pi;
    pi = (int *) malloc (sizeof (int));
   *pi=0;

       pid = fork (); /*fork a child process*/

       if (pid > 0) { /*Parent Process:*/
        /*When fork() returns a positive number, we are in the parent process*/
        /*the fork return value is the PID of the newly created child process*/

          printf ("*** Parent Process Begin *** \n");

          *pi = *pi + 1 ;
          printf ("*pi = %d \n" , *pi);
          printf ("Process id = %d \n", getpid () );
          printf ("pid = %d \n" , pid);

          printf ("*** Parent Process End *** \n");
     }
       else if (pid == 0) { /*Child Process:*/
        /*When fork() returns 0, we are in the child process.*/

          printf ("*** Child Process Begin *** \n");
```

```
        *pi = *pi - 1 ;
        printf ("*pi = %d \n" , *pi);
        printf ("Process id = %d \n", getpid () );
        printf ("pid = %d \n" , pid);

        printf ("*** Child Process End *** \n");
    }

    else { /*error occurred*/
    /*When fork() returns a negative number, an error happened*/
     printf ("fork creation failed!!! \n ");
    }

  return 0;
}
```

توجه: برنامه اجرا می‌شود و در خط pi=0* مقدار متغیر پویا در آدرس pi تعریف شده در حافظه Heap برابر با 0 می شود. دقت کنید که متغیر پویا در آدرس pi داخل تابع main تعریف شده است و یک متغیر پویا محسوب می‌شود که داخل Heap تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می شود. در یک قاعده کلی، بعد از اجرای forkهر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند. از آن جایی که هر دو فرآیند از نظر محتوا کاملا کپی هم هستند، مقدار اولیه متغیر پویا pi* در هر دو فرآیند برابر مقدار 0 است. دستور بعدی که توسط هر دو فرآیند پدر و فرزند اجرا می شود دستور if و else if است. دقت کنید که پس از fork فرآیند پدر و فرآیند فرزند، مسیر جداگانه خود را پیش می‌گیرند. با توجه به اینکه داده‌های فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است. به عبارت دیگر از آنجا که فرآیند پدر و فرزند فضای آدرس مختص به خود را دارند، هرگونه تغییر، مستقل از سایرین خواهد بود. به عبارت بهتر اگر فرآیند پدر مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند فرزند اثر نخواهد داشت و همچنین اگر فرآیند فرزند مقادیر متغیرهای خودش را تغییر دهد، این تغییرات روی فرآیند پدر اثر نخواهد داشت. برای مثال هرچند که نام و مقدار متغیر در هر دو فرآیند پدر و فرزند یکسان و برابر pi* و برابر مقدار 0 است، اما فضای آدرس فرآیند پدر و فرزند متفاوت است.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
*pi = *pi + 1 ;
printf ("*pi = %d \n" , *pi);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر pi* برابر 0 است و پس از اجرای دستور *pi = *pi + 1 برابر 1 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد.

***pi = *pi + 1;** **printf ("*pi = %d \n" , *pi);**	***pi = 1**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستورات بعدی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
*pi = *pi - 1 ;
printf ("*pi = %d \n" , *pi);
printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر pi* برابر 0 است و پس از اجرای دستور *pi = *pi - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

***pi = *pi - 1;** **printf ("*pi = %d \n" , *pi);**	***pi = -1**

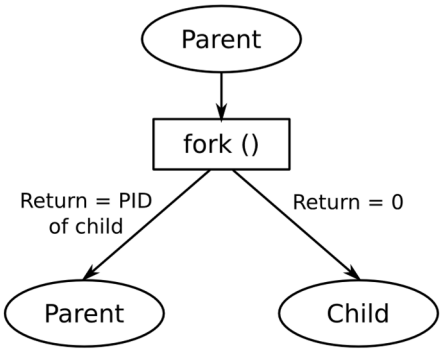
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

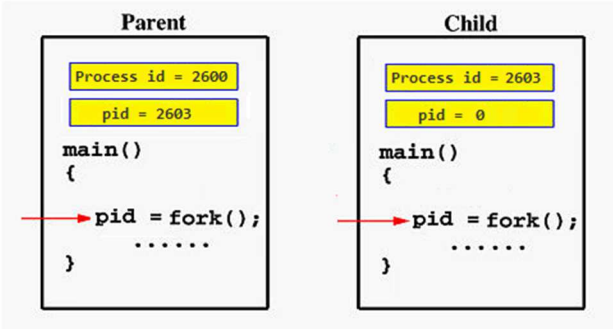
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 0**

شکل زیر گویای مطلب است:



شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
*pi = 1
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
*pi = -1
Process id = 2603
pid = 0
*** Child Process End ***
```

توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال فوق مسیر فرآیند پدر و فرزند به دلیل وجود شرط if از هم متمایز شده بود.

مثال: وقتی یک فرآیند فرزند توسط فرآیند پدر و دستور ()fork ایجاد می‌شود، کدام بخش‌های زیر مابین فرآیند پدر و فرزند به اشتراک گذاشته می‌شود؟

الف) Stack

ب) Heap

ج) Shared memory segments

پاسخ: فقط Shared memory segments مابین فرآیند پدر و فرزند به اشتراک گذاشته می‌شود، اما محتوای داده‌های Stack و Heap فرآیند پدر برای فرآیند فرزند کپی می‌شود، همه متغیرهایی که مقداردهی اولیه شده‌اند بعد از اجرای fork مقادیر یکسان دارند، اما تغییرات بعدی در هر کدام از آنها بر روی دیگری اثر ندارد، چون فضای آدرس فرآیند پدر و فرزند متفاوت است.

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، مقدار value در خطوط LINE A و LINE B برابر کدام گزینه است؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/

int value = 20;

int  main()
{
    pid_t  pid , pidChild;

        pid = fork (); /*fork a child process*/

        if (pid > 0) { /*Parent Process*/

          printf ("*** Parent Process Begin *** \n");

           pidChild = wait (NULL);
           printf ("***Child Complete*** \n");
           value = value + 15 ;
           printf ("Parent: value = %d \n",value); /*LINE A*/
           printf ("Child Process id wait = %d \n", pidChild);
           printf ("Parent Process id = %d \n", getpid () );

          printf ("*** Parent Process End *** \n");
```

```
    }
      else if (pid == 0) { /*Child Process:*/
     printf ("***Child Process Begin *** \n");
        value = value - 15 ;
        printf ("Child: value = %d \n",value); /*LINE B*/
        printf ("Child Process id = %d \n", getpid () );
    printf ("***Child Process End *** \n");
    }
     else { /*error occurred*/
      printf ("fork creation failed!!! \n ");
    }
  return 0;
}
```

۱) LINE A = 5 , LINE B = 35

۲) LINE A = 35 , LINE B = 5

۳) LINE A = 5 , LINE B = 0

۴) LINE A = 0 , LINE B = 5

پاسخ- گزینه (۲) صحیح است.

توجه: برنامه اجرا می شود و در اولین خط مقدار متغیر سرا سری value برابر با 20 می شود. دقت کنید که متغیر value بالای تابع main تعریف شده است و یک متغیر سراسری محسوب می شود که داخل Data Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که د ستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pidChild = wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

توجه: فراخوان سیستمی ()wait مقدار Process id فرآیند فرزندی که پایان یافته است را در خروجی بر می‌گرداند.

توجه: فرآیند فرزند از طریق فراخوان سیستمی ()wait می‌تواند با فرآیند پدرش ارتباط برقرار کند، به عبارت دیگر مقدار Process id فرآیند فرزند توسط فراخوان سیستمی ()wait به فرآیند پدرش پاس داده می‌شود.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
value = value - 15 ;
printf ("Child: value = %d \n",value); /*LINE B*/
printf ("Child Process id = %d \n", getpid ());
```

توجه: مقدار اولیه متغیر سراسری value برابر 20 است و پس از اجرای دستور value=value-15 برابر 5 می‌شود و این تغییر روی فرآیند پدر اثری ندارد. هر چند که متغییر value سراسری است.

value = value - 15; **printf ("Child: value = %d \n",value); /*LINE B*/**	**value = 5**

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Child Process id = %d \n", getpid ());	**Process id = 2603**

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pidChild = wait (NULL);
printf ("***Child Complete*** \n");
value = value + 15 ;
printf ("Parent: value = %d \n",value); /*LINE A*/
printf ("Child Process id wait = %d \n", pidChild);
printf ("Parent Process id = %d \n", getpid () );
```

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

توجه: دقت کنید که عمل انتساب pidChild = wait(NULL) باقی‌مانده از قبل ابتدا تکمیل می‌شود و سپس دستور زیر اجرا می‌شود.

printf ("*Child Complete \n***");**	*****Child Complete*****

توجه: مقدار اولیه متغیر سراسری value برابر 20 است و پس از اجرای دستور value=value+15 برابر 35 می‌شود و این تغییر روی فرآیند فرزند اثری ندارد. هر چند که متغییر value سراسری است. و هرچند که قبلا مقدار متغیر سراسری value توسط فرآیند فرزند برابر 5 شده است.

value = value + 15; **printf ("Parent: value = %d \n",value); /*LINE A*/**	**value = 35**

توجه: فراخوان سیستمی ()wait مقدار Process id فرآیند فرزندی که پایان یافته است را در خروجی بر می‌گرداند.

printf ("Child Process id wait = %d \n", pidChild);	**Child Process id wait = 2603**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Child Process id = %d \n", getpid ());	**Process id = 2600**

خروجی نهایی برنامه به صورت زیر است:

```
***Child Process Begin ***
Child: value = 5
Child Process id = 2603
***Child Process End ***
*** Parent Process Begin ***
***Child Complete***
Parent: value = 35
Child Process id wait = 2603
Parent Process id = 2600
*** Parent Process End ***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، مقدار pid و pid1 در خطوط LINE A، LINE B ، LINEC و LINE D برابر کدام گزینه ا ست؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/

int  main()
{
    pid_t  pid , pid1;
```

```
        pid = fork (); /*fork a child process*/

        if (pid > 0) { /*Parent Process*/

            pid1 = getpid ();
            printf ("Parent : pid = %d \n" , pid); /*LINE C*/
            printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/
        }

        else if (pid == 0) { /*Child Process*/
            pid1 = getpid ();
            printf ("Child : pid = %d  \n" , pid); /*LINE A*/
            printf ("Child : pid1 = %d  \n" , pid1); /*LINE B*/
       }
        else { /*error occurred*/
         printf ("fork creation failed!!! \n ");
{
   return 0;
}
```

۱) LINE A = 2603 , LINE B = 0 , LINE C = 2603 , LINE D = 2600

۲) LINE A = 0 , LINE B = 2603 , LINE C = 2600 , LINE D = 2603

۳) LINE A = 2600 , LINE B = 2603 , LINE C = 2603 , LINE D = 0

۴) LINE A = 0 , LINE B = 2603 , LINE C = 2603 , LINE D = 2600

پاسخ- گزینه (۴) صحیح است.

توجه: در اولین خط، د ستور fork قرار دارد، وقتی که د ستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
pid1 = getpid ();
printf ("Parent : pid = %d \n" , pid); /*LINE C*/
printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/
```

توجه: مقدار بازگ شتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

printf ("Parent : pid = %d \n" , pid); /*LINE C*/	**pid = 2603**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

pid1 = getpid (); **printf ("Parent : pid1 = %d \n" , pid1); /*LINE D*/**	**Parent : pid1 = 2600**

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
pid1 = getpid ();
printf ("Child : pid = %d \n" , pid); /*LINE A*/
printf ("Child : pid1 = %d \n" , pid1); /*LINE B*/
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

printf ("Child : pid = %d \n" , pid); /*LINE A*/	**pid = 0**

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

pid1 = getpid (); **printf ("Child : pid1 = %d \n" , pid1); /*LINE B*/**	**Child : pid1 = 2603**

خروجی نهایی برنامه به صورت زیر است:

```
Parent: pid = 2603
Parent: pid1 = 2600
Child: pid = 0
Child: pid1 = 2603
```

مثال: با اجرای قطعه کد زیر در نهایت چند فرآیند خواهیم داشت؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600، pid واقعی فرآیند فرزند اول برابر مقدار 2601، pid واقعی فرآیند فرزند دوم برابر مقدار 2602 و pid واقعی فرآیند فرزند سوم برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
```

```
int  main()
{
    if (fork())
        if(fork())
fork();
   printf ("Process id = %d \n", getpid ());
   return 0;
}
```

۱) 0　　۲) 2　　۳) 4　　۴) 8

پاسخ- گزینه (۳) صحیح است.

توجه: در اولین خط، دستور (fork()) if قرار دارد، وقتی که دستور fork() داخل دستور if اجرا می شـــود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شـود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2601)
if (fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

if (2603)	TRUE

توجه: در ابتدا با فراخوانی دستور fork() یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شـود، بنابراین حاصـل دسـتور شـرطی if (2601) برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
if (0)
if (fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می شود، بنابراین حاصل دستور شرطی (0) if برابر مقدار FALSE خواهد بود. که منجر به این می شود که دستورات fork دوم و fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند فرزند اول اجرا می‌شود.

توجه: دقت داشته باشید که (()fork) if دوم داخل بدنه (()fork) if اول است و ()fork سوم داخل بدنه (()fork) if دوم است. بنابراین اگر (()fork) if اول برقرار نباشد، (()fork) if دوم و ()fork سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند **(C1)** می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

توجه: دقت داشته باشید که یکی از بهترین شیوه‌های شمارش تعداد کل فرآیندها، قرار دادن یک دستور printf جهت نمایش Process id توسط دستور ()getpid در انتهای قطعه کد پایه است.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2602)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

if (2604)	TRUE

توجه: در ابتدا با فراخوانی دستور ()fork یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود، بنابراین حاصل دستور شرطی (2602) if برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
if (0)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشـتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر ا ست که به آن پاس داده می شود، بنابراین حا صل د ستور شرطی (0) if برابر مقدار FALSE خواهد بود. که منجر به این می شود که د ستور fork سوم مورد برر سی قرار نگیرد. اما از آنجا که د ستور printf م ستقل و خارج از بدنه د ستورات شرطی ا ست، در انتهای فرآیند فرزند دوم اجرا می‌شود.

توجه: دقت داشته باشید که ()fork سوم داخل بدنه ((fork()) if دوم است. بنابراین اگر ((fork()) if دوم برقرار نباشد، ()fork سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منت سب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند سوم (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

توجه: مقدار بازگ شتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: یک فرآیند پدر(P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منت سب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0
printf ("Process id = %d \n", getpid ());
```

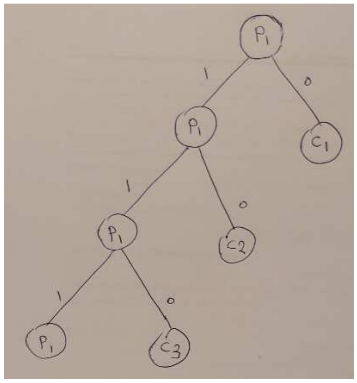
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر ا ست که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند **(C3)** می‌تواند جهت بازیابی مقدار process id یا همان pid منت سب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

نتیجه: در حالت کلی دستور fork داخل if یعنی (()if (fork باعث می‌شود که فرزند، نازا باشد، یعنی خود فرزند توسط پدر به دنیا می‌آید، اما فرزند، توان زاییدن و زاد و ولد را ندارد.

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2601
Process id = 2602
Process id = 2603
Process id = 2600
```

مثال: با اجرای قطعه کد زیر در نهایت چند فرآیند خواهیم داشـــت؟ (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600، pid واقعی فرآیند فرزند اول برابر مقدار 2601، pid واقعی فرآیند فرزند دوم برابر مقدار 2602 و pid واقعی فرآیند فرزند سوم برابر مقدار 2603 باشد.)

```
//gcc 5.4.0
#include <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include <sys/types.h>  /*pid_t*/
int main()
{
  if (fork())
     if(!fork())
fork();
   printf ("Process id = %d \n", getpid ());
```

```
    return 0;
}
```

۱) 0　　۲) 2　　۳) 4　　۴) 8

پاسخ- گزینه (۳) صحیح است.

توجه: در اولین خط، دستور (()if (fork قرار دارد، وقتی که دستور ()fork داخل دستور if اجرا می شـــود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شـود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (2601)
if (!fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

if (2601)	TRUE

توجه: در ابتدا با فراخوانی دستور ()fork یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شـود، بنابراین حاصـل دسـتور شـرطی (2601) if برابر مقدار TRUE خواهد بود. که منجر به این می‌شود که دستورات بعدی فرآیند پدر مورد بررسی قرار گیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
if (0)
if (!fork())
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

if (0)	FALSE

توجه: از آنجاییکه مقدار بازگشـــتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می شود، بنابراین حاصل دستور شرطی (0) if برابر مقدار FALSE خواهد بود. که منجر به این می شود که دستورات fork دوم و fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند فرزند اول اجرا می‌شود.

توجه: دقت داشته باشید که if (!fork()) دوم داخل بدنه if (fork()) اول است و fork() سوم داخل بدنه if (!fork()) دوم است. بنابراین اگر if (fork()) اول برقرار نباشد، if (!fork()) دوم و fork() سوم به تبع آن اجرا نخواهد شد.

توجه: یک فرآیند فرزند (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
if (!2602) = if(0)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

if (!2602) = if(0)	FALSE

توجه: در ابتدا با فراخوانی دستور fork() یک زایمان صورت می‌گیرد و از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می شود، بنابراین حاصل دستور شرطی if (!2602) = if(0) برابر مقدار FALSE خواهد بود. که منجر به این می شود که دستور fork سوم مورد بررسی قرار نگیرد. اما از آنجا که دستور printf مستقل و خارج از بدنه دستورات شرطی است، در انتهای فرآیند پدر اجرا می‌شود.

وجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
if (!0) = if(1)
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

if (!0) = if(1)	TRUE

توجه: از آنجاییکه مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می شود، بنابراین حاصل دستور شرطی if (!0) = if(1) برابر مقدار

TRUE خواهد بود. که منجر به این می‌شـود که دسـتورات بعدی فرآیند فرزند دوم مورد بررسـی قرار گیرد.

توجه: دقت بسیار زیاد داشته باشید که ()fork سوم داخل بدنه ((!fork()) if دوم است. بنابراین اگر ((!fork()) if دوم برقرار باشد، ()fork سوم به تبع آن اجرا خواهد شد.

دستوراتی که توسط فرآیند پدر(C2) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: وقتی که دسـتور()fork سـوم اجرا می‌شـود یک فرآیند فرزند سـوم (جدید) از روی فرآیند پدر(فرآیند فرزند دوم) ایجاد و متولد می‌شـود که به آن فرآیند فرزند گفته می‌شـود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر (فرآیند فرزند دوم) و فرزند (فرآیند فرزند سـوم) خط بعد از دستور fork را اجرا می کنند.

توجه: مقدار بازگشـتی fork در حالت اجرای موفق برای فرآیند پدر (فرآیند فرزند دوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر (فرآیند فرزند دوم) در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: یک فرآیند پدر (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع ()getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

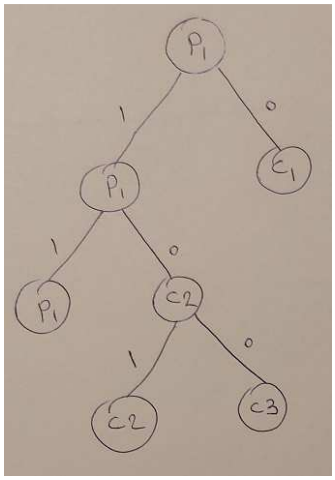
```
0
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع ()getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2601
Process id = 2600
Process id = 2602
Process id = 2603
```

با اجرای کد زیر در نهایت چند پردازه خواهیم داشت؟

```
main ()
{
      for(i = 1 ; i < 4 ; i++)
           fork();
}
```

توجه: مطابق قواعد زبان‌های برنامه‌نویسی، قطعه کد فوق معادل و هم ارز قطعه کد زیر است:

```
main ()
{
    fork();
    fork();
    fork();
}
```

توجه: دقت داشته باشید که یکی از بهترین شیوه‌های شمارش تعداد کل فرآیندها، قرار دادن یک دستور printf جهت نمایش Process id توسط دستور ()getpid در انتهای قطعه کد پایه است، به همین جهت به قطعه کد فوق یک دستور printf به صورت زیر اضافه شده است:

```
main ()
{
    fork();
    fork();
    fork();
    printf ("Process id = %d \n", getpid ());
}
```

توجه: در اولین خط، دستور ()fork قرار دارد، وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2601=fork();
fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
0
fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود.

دستورات که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2602=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
0=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0=fork();
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2604=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند چهارم است.

توجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

دستوراتی که توسط فرآیند فرزند چهارم (C4) اجرا می‌شود، به صورت زیر است:

```
0
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند چهارم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C4) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2604**

دستوراتی که توسط فرآیند پدر (C2) اجرا می‌شود، به صورت زیر است:

```
2605=fork();
```

printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشـــتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند دوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند پنجم است.

توجه: یک فرآیند پدر(C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند فرزند پنجم (C5) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند پنجم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C5) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2605

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

2606=fork();
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشـــتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند ششم است.

توجه: یک فرآیند پدر (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند فرزند ششم (C6) اجرا می‌شود، به صورت زیر است:

0
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند ششم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C6) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2606

دستوراتی که توسط فرآیند پدر (C3) اجرا می‌شود، به صورت زیر است:

2607=fork();
printf ("Process id = %d \n", getpid ());

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند سوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند هفتم است.

توجه: یک فرآیند پدر(C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

دستوراتی که توسط فرآیند فرزند هفتم (C7) اجرا می‌شود، به صورت زیر است:

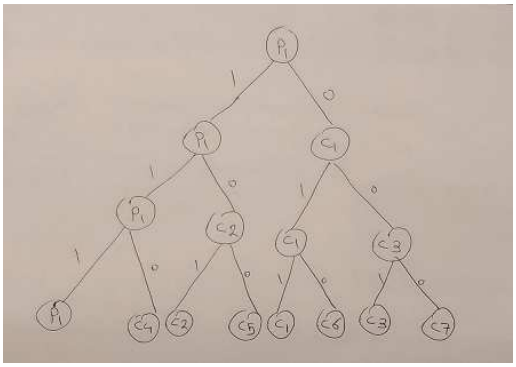
0
printf ("Process id = %d \n", getpid ());

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند هفتم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C7) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2607

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2600
Process id = 2604
```

```
Process id = 2602
Process id = 2605
Process id = 2601
Process id = 2606
Process id = 2603
Process id = 2607
```

توجه: یک کار مهم دیگری هم که می‌توان انجام داد این است که به جای اینکه در فرآیند فرزند کد متفاوتی که خودمان نوشته‌ایم اجرا شود، می‌توان کاری کرد که یک برنامه‌ی دیگر که به صورت فایل اجرایی است اجرا شود. برای این کار می‌توان از دستورات متفاوتی استفاده کرد که یکی از آن‌ها دستور execvp برای جایگزین کردن تصویر حافظه فرآیند است که از خانواده دستورات exec می‌باشد. این دستور کل فضای حافظه‌ی فرآیند فرزند را پاک می‌کند و در آن برنامه‌ی جدیدی که در execvp آمده است را می‌ریزد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند.

مثال: در کد زیر یک نمونه از استفاده‌ی execvp() را می‌بینید که در آن برنامه‌ی ls لینوکس را اجرا می‌کند، که فایل‌های موجود در یک دایرکتوری را نمایش می‌دهد و همین طور فرآیند پدر اعداد 1 تا 99 را در خروجی نمایش می‌دهد:

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
int  main()
{
    pid_t  pid;

/*fork a child process*/
    pid =  fork();
    if (pid < 0) {
        printf("Fork Faild!");
}
   else if(pid == 0){  /*child process/*
      char *arg[] = {"/bin/ls" , "-l","-a", NULL};
      execvp(arg[0],arg);
```

```
}
   else { /*parent process/*
        wait(NULL);
        printf ("***Child Complete*** \n");

       for (int i = 0; i < 5; ++i) {
          printf ("Parent process counter :%d \n",i);
}

}
   return 0;
}
```

توجه: در اولین خط، د ستور fork قرار دارد، وقتی که د ستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
char *arg[] = {"/bin/ls" , "-l","-a", NULL};
execvp(arg[0],arg);
```

توجه: پس از فراخوانی تابع ()execvp، کل فضای آدرس فرآیند فرزند پاک می‌شود و در آن برنامه ی جدیدی که در ()execvp آمده است قرار می‌گیرد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضـــای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند. بنابراین پس از اجرای تابع ()execvp، قطعه کد و برنامه جدید اجرا می‌شود و کنترل اجرای برنامه دیگر هیچ وقت به دستور قبل و بعد تابع ()execlp باز نمی‌گردد.

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
printf ("***Child Complete*** \n");
```

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("***Child Complete \n***");	***Child Complete***

توجه: مقادیر متغیر محلی i در فرآیند پدر توســـط حلقه for از 0 تا 4 در خروجی نمایش داده می‌شود.

```
for (int i = 0; i < 5; ++i) {
        printf ("Parent process counter :%d \n",i);
}
```

printf ("Parent process counter :%d \n",i);	**Parent process counter :0** **Parent process counter :1** **Parent process counter :2** **Parent process counter :3** **Parent process counter :4**

خروجی نهایی برنامه به صورت زیر است:

```
total 88
drwxr-xr-x 22 root              root              4096 Jun 27 18:52 .
drwxrwxr-x  3 root              ren               4096 May 21 16:27 ..
drwxr-xr-x  2 rextester_user31  rextester_user31  4096 Jun 22 05:52 1311737820
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 1358939562
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1422882116
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 1471488187
drwxr-xr-x  2 rextester_user127 rextester_user127 4096 Jun 27 18:52 1516783416
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1521451137
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 1642473737
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 1790473723
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 1857273623
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 2098476182
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 213214471
drwxr-xr-x  2 rextester_user295 rextester_user295 4096 Jun 27 18:52 404136654
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 521610754
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 703843859
drwxr-xr-x  2 root              root              4096 Jun 21 02:40 7215752
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 793881491
drwxr-xr-x  2 root              root              4096 Jun 22 05:50 815181955
drwxr-xr-x  2 rextester_user184 rextester_user184 4096 Jun 21 18:53 861836283
drwxr-xr-x  2 root              root              4096 Jun 22 05:52 945697616
drwxr-xr-x  2 root              root              4096 Jun 22 05:51 963143096
Parent process counter :0
Parent process counter :1
Parent process counter :2
```

```
Parent process counter :3
Parent process counter :4
***Child Complete***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C اســت، خط printf("LINE J") پس از اجرای دســتور execlp() چندبار اجرا می‌شود؟

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
int  main()
{
     pid_t  pid;

/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}
else if (pid == 0) { /* child process*/

     execlp("/bin/ls" , "ls" , NULL);
     printf("LINE J");
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf ("***Child Complete*** \n");
 }
  return 0;
}
```

۱) 0 بار ۲) 1 بار ۳) 2 بار ۴) 3 بار

پاسخ- گزینه (۱) صحیح است.

توجه: در اولین خط، د ستور fork قرار دارد، وقتی که د ستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شـــود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
execlp("/bin/ls" , "ls" , NULL);
printf("LINE J");
```

توجه: پس از فراخوانی تابع ()execlp، کل فضای آدرس فرآیند فرزند پاک می شود و در آن برنامه ی جدیدی که در ()execlp آمده است قرار می‌گیرد و هیچ اثری از کدها و داده‌های قبلی که در آن بوده نمی‌ماند. این تابع نام فایل اجرایی و آرگومان‌هایی که باید به برنامه پاس داده شود را می‌گیرد و آن را در فضـــای حافظه‌ی فرآیند فرزند بارگذاری می‌کند و آن را اجرا می‌کند. بنابراین پس از اجرای تابع ()execlp، قطعه کد و برنامه جدید اجرا می شود و کنترل اجرای برنامه دیگر هیچ وقت به دســتور قبل و بعد تابع ()execlp باز نمی‌گردد، در این حالت پس از اجرای تابع ()execlp، خط printf("LINE J") دیده نخواهد شـــد و به تبع هرگز اجرا هم نخواهد شـــد. اما اگر اجرای تابع ()execlp موفقیت آمیز نباشـــد، آنگاه کنترل برنامه به خط بعد از تابع ()execlp باز می‌گردد و خط printf("LINE J") اجرا و چاپ می‌شود.

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
printf ("***Child Complete*** \n");
```

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی ()wait منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی ()wait را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("***Child Complete \n***");	***Child Complete***

خروجی نهایی برنامه به صورت زیر است:

```
7215752
793881491
815181955
861836283
945697616
963143096
***Child Complete***
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C ا ست، مقدار خروجی در خطوط LINE X و LINE Y برابر کدام گزینه است؟

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int  main()
{
int i;
pid_t  pid;


   pid = fork();

   if (pid == 0) {
        for (i = 0 ; i < SIZE ; i++){
           nums[i] *= -i;
           printf("CHILD: %d \n" , nums[i]); /* LINE X*/
     }
}
else if (pid > 0) {
    wait(NULL);
    printf ("***Child Complete*** \n");
    for (i = 0; i < SIZE; i++)
         printf("PARENT: %d \n", nums[i]); /* LINE Y*/
  }
   return 0;
}
```

۱) LINE X = 0, -1, -4, -9, -16 , LINE Y = 0, 1, 2, 3, 4

۲) LINE X = -16, -9, -4, -1, 0 , LINE Y = 0, 1, 2, 3, 4

۳) LINE X = 0, -1, -4, -9, -16 , LINE Y = 4, 3, 2, 1, 0

۴) LINE X = -16, -9, -4, -1, 0 , LINE Y = 4, 3, 2, 1, 0

پاسخ- گزینه (۱) صحیح است.

توجه: برنامه اجرا می‌شود و در اولین خط مقادیر آرایه سراسری nums[5] برابر با مقادیر {0,1,2,3,4} می‌شود. دقت کنید که آرایه سراسری nums[5] بالای تابع main تعریف شده است و یک آرایه سراسری محسوب می‌شود که داخل Data Segment تعریف و مقداردهی می‌شود. همچنین دقت

کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می‌شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
wait (NULL);
```

توجه: فرآیند پدر با استفاده از فراخوان سیستمی wait() منتظر تکمیل فرآیند فرزند می‌ماند، در واقع فرآیند پدر منتظر می‌ماند تا کار فرآیند فرزند تمام شود. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی wait() را فراخوانی کرده است، شروع به ادامه کار می‌کند.

دستوراتی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
for (i = 0 ; i < 5 ; i++){
    nums[i] *= -i;
    printf("CHILD: %d \n" , nums[i]); /* LINE X*/
 }
```

توجه: مقدار اولیه متغیر i برابر 0 اســـت و پس از اجرای دســـتور i++ برابر مقادیر {0,1,2,3,4} می‌شود و این تغییرات روی فرآیند پدر اثری ندارد.

توجه: مطابق قوانین کوتاه‌نویسی د ستورات در زبان برنامه‌نویسی C د ستور nums[i] *= -i معادل دستور nums[i] =nums[i] *(-i) است.

توجه: مقادیر اولیه آرایه ســـراســـری nums[5] برابر {0,1,2,3,4} اســـت و پس از اجرای دســـتور nums[i] =nums[i] *(-i) برابر مقادیر {0,-1,-4,-9,-16} می‌شود و این تغییر روی فرآیند پدر اثری ندارد. هر چند که آرایه nums[5] سراسری است.

nums[i] =nums[i] *(-i) **printf("CHILD: %d \n" , nums[i]); /* LINE X*/**	**CHILD: 0** **CHILD: -1** **CHILD: -4** **CHILD: -9** **CHILD: -16**

دستورات بعدی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
for (i = 0 ; i < 5 ; i++){
    printf("PARENT: %d \n" , nums[i]); /* LINE Y*/
}
```

توجه: مقدار اولیه متغیر i برابر 0 اســـت و پس از اجرای دســـتور i++ برابر مقادیر {0,1,2,3,4} می‌شود و این تغییرات روی فرآیند فرزند اثری ندارد.

توجه: همانطور که گفتیم، فرآیند پدر با استفاده از فراخوان سیستمی wait() منتظر تکمیل فرآیند فرزند می‌ماند. هنگامی که فرآیند فرزند تکمیل شد، فرآیند پدر از جاییکه فراخوان سیستمی wait()

را فراخوانی کرده است، شروع به ادامه کار می‌کند. در یک قاعده کلی، بعد از اجرای ()wait فرآیند پدر خط بعد از دستور ()wait را اجرا می کند.

printf ("*Child Complete \n***");**	*****Child Complete*****

توجه: مقادیر اولیه آرایه سراسری nums[5] در فرآیند پدر برابر{0,1,2,3,4} است. هرچند که قبلا مقادیر آرایه سراسری nums[5] توسط فرآیند فرزند برابر مقادیر {0,-1,-4,-9,-16} شده است.

printf("PARENT: %d \n" , nums[i]); /* LINE Y*/	**PARENT:0** **PARENT:1** **PARENT:2** **PARENT:3** **PARENT:4**

خروجی نهایی برنامه به صورت زیر است:

```
CHILD:0
CHILD:-1
CHILD:-4
CHILD:-9
CHILD:-16
PARENT:0
PARENT:1
PARENT:2
PARENT:3
PARENT:4
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است، با اجرای کد زیر در نهایت چند فرآیند و چند نخ خواهیم داشت؟

```
//gcc 5.4.0
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork*/
#include  <sys/types.h>  /*pid_t*/
int  main()
{
    pid_t  pid;

   pid = fork();
```

```
   if (pid == 0) { /* child process*/

        fork ();
        thread_create(…);
}

fork();

printf ("Process id = %d \n", getpid ());

    return 0;
}
```

۱) 2 فرآیند و 6 نخ
۲) 6 فرآیند و 0 نخ
۳) 6 فرآیند و 2 نخ
۴) 0 فرآیند و 2 نخ

پاسخ- گزینه (۳) صحیح است.

توجه: در اولین خط، دستور ()fork قرار دارد، وقتی که دستور()fork اجرا می‌شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می‌شود که به آن فرآیند فرزند گفته می‌شود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند خط بعد از دستور fork را اجرا می کنند.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2601=pid=fork();
if (pid == 0) { /* child process*/

        fork ();
        thread_create(…);
}

fork();

printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند اول است. با توجه به شرط (pid == 0) if در فرآیند پدر (P1) باعث می‌شود ()fork دوم و (…)thread_create اجرا نشود و فقط در ادامه ()fork سوم مورد بررسی قرار بگیرد.

دستوراتی که توسط فرآیند فرزند اول (C1) اجرا می‌شود، به صورت زیر است:

```
0=pid=fork();
if (pid == 0) { /* child process*/
```

```
        fork ();
        thread_create(…);
}

fork();

printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند اول برابر مقدار صفر است که به آن پاس داده می‌شود. و با توجه به شرط if (pid == 0) در فرآیند فرزند اول (C1) باعث می‌شود fork() دوم و thread_create(…) اجرا شود و همچنین در ادامه fork() سوم هم مورد بررسی قرار بگیرد.

دستوراتی که توسط فرآیند پدر (P1) اجرا می‌شود، به صورت زیر است:

```
2602=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند دوم است.

توجه: یک فرآیند پدر (P1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2600

دستوراتی که توسط فرآیند فرزند دوم (C2) اجرا می‌شود، به صورت زیر است:

```
0=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند دوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C2) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2602

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2603=fork();
thread_create(…);
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند اول) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند سوم است.

توجه: توسط دستور thread_create(…) نخ اول ایجاد می‌شود.

دستوراتی که توسط فرآیند فرزند سوم (C3) اجرا می‌شود، به صورت زیر است:

```
0=fork();
thread_create(…);
fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: توسط دستور thread_create(…) نخ دوم ایجاد می‌شود.

دستوراتی که توسط فرآیند پدر (C1) اجرا می‌شود، به صورت زیر است:

```
2604=fork();
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند چهارم است.

توجه: یک فرآیند پدر (C1) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2601

دستوراتی که توسط فرآیند فرزند چهارم (C4) اجرا می‌شود، به صورت زیر است:

```
0
printf ("Process id = %d \n", getpid ());
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند چهارم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C4) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2604

دستوراتی که توسط فرآیند پدر (C3) اجرا می‌شود، به صورت زیر است:

```
2605=fork();
printf ("Process id = %d \n", getpid () )
```

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر(فرآیند فرزند سوم) برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند پنجم است.

توجه: یک فرآیند پدر(C3) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2603

دستوراتی که توسط فرآیند فرزند پنجم (C5) اجرا می‌شود، به صورت زیر است:

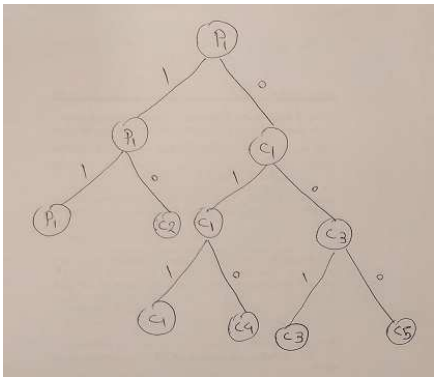
0
printf ("Process id = %d \n", getpid ())

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند پنجم برابر مقدار صفر است که به آن پاس داده می‌شود.

توجه: یک فرآیند فرزند (C5) می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	Process id = 2605

شکل زیر گویای مطلب است:



خروجی نهایی برنامه به صورت زیر است:

```
Process id = 2600
Process id = 2602
Process id = 2601
Process id = 2604
Process id = 2603
Process id = 2605
```

مثال: برنامه‌ی زیر یک قطعه کد به زبان C است که مقدار i، مقدار pid حاصل از بازگشت fork و مقدار Process id حاصل از بازگشت getpid را در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

//gcc 5.4.0

```
#include  <stdio.h>   /* printf */
#include <unistd.h>  /*fork */
#include  <sys/types.h>  /* pid_t */
int  main()
{
     pid_t  pid;
     int i = 0;

         pid = fork (); /*fork a child process*/

         if (pid > 0) { /*Parent Process:*/

            printf ("*** Parent Process Begin *** \n");

            printf ("&i = %d \n" , &i);

            i = i + 1 ;
            printf ("i = %d \n" , i);

            printf ("&i = %d \n" , &i);

            printf ("Process id = %d \n", getpid () );
            printf ("pid = %d \n" , pid);

            printf ("*** Parent Process End *** \n");
       }
         else if (pid == 0) { /*Child Process:*/

            printf ("*** Child Process Begin *** \n");

            printf ("&i = %d \n" , &i);

            i = i - 1 ;
            printf ("i = %d \n" , i);

            printf ("&i = %d \n" , &i);

             printf ("Process id = %d \n", getpid () );
             printf ("pid = %d \n" , pid);

             printf ("*** Child Process End *** \n");
        }

         else { /*error occurred*/
          printf ("fork creation failed!!! \n ");
        }

    return 0;
 }
```

توجه: برنامه اجرا می‌شـــود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شـــود. دقت کنید که متغیر i داخل تابع main تعریف شده است و یک متغیر محلی محسوب می شود که داخل Stack Segment تعریف و مقداردهی می‌شود. در خط بعدی، دستور fork قرار دارد، وقتی که دستور fork اجرا می شود یک فرآیند فرزند (جدید) از روی فرآیند پدر ایجاد و متولد می شود که به آن فرآیند فرزند گفته می‌شـــود. در یک قاعده کلی، بعد از اجرای fork هر دو فرآیند پدر و فرزند دقیقا خط بعد از دستور fork را اجرا می‌کنند.

دستوراتی که توسط فرآیند پدر اجرا می‌شود، به صورت زیر است:

```
printf ("&i = %d \n" , &i);

i = i + 1 ;
printf ("i = %d \n" , i);

printf ("&i = %d \n" , &i);

printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 اســـت و پس از اجرای دســـتور i = i + 1 برابر 1 می‌شـــود و این تغییر روی فرآیند فرزند اثری ندارد.

توجه: به تفاوت آدرس مجازی و آدرس فیزیکی در این قطعه کد توجه نمایید، عبارت &i دســـترســـی به آدرس مجازی را ایجاد می‌کند، و آدرس مجازی فرآیند پدر و فرآیند فرزند کاملا **یکسان** است. اما آدرس فیزیکی متغیر i در فرآیند پدر و فرآیند فرزند کاملا **متفاوت** است.

printf ("&i = %d \n" , &i); **i = i + 1;** **printf ("i = %d \n" , i);** **printf ("&i = %d \n" , &i);**	**&i = 10000** **i=1** **&i = 10000**

توجه: یک فرآیند پدر می‌تواند جهت بازیابی مقدار process id یا همان pid منتســـب شـــده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2600**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند پدر برابر یک عدد صحیح بزرگتر از صفر است که به آن پاس داده می‌شود. این عدد صحیح بزرگتر از صفر پاس داده شده به فرآیند پدر در واقع pid یعنی Process id فرآیند فرزند است.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 2603**

دستورات بعدی که توسط فرآیند فرزند اجرا می‌شود، به صورت زیر است:

```
printf ("&i = %d \n" , &i);

i = i - 1 ;
printf ("i = %d \n" , i);

printf ("&i = %d \n" , &i);

printf ("Process id = %d \n", getpid () );
printf ("pid = %d \n" , pid);
```

توجه: مقدار اولیه متغیر i برابر 0 است و پس از اجرای دستور i = i - 1 برابر 1- می‌شود و این تغییر روی فرآیند پدر اثری ندارد.

توجه: به تفاوت آدرس مجازی و آدرس فیزیکی در این قطعه کد توجه نمایید، عبارت &i دسترسی به آدرس مجازی را ایجاد می‌کند، و آدرس مجازی فرآیند پدر و فرآیند فرزند کاملا **یکسان** است. اما آدرس فیزیکی متغیر i در فرآیند پدر و فرآیند فرزند کاملا **متفاوت** است.

printf ("&i = %d \n" , &i); **i = i + 1;** **printf ("i = %d \n" , i);** **printf ("&i = %d \n" , &i);**	**&i = 10000** **i=-1** **&i = 10000**

توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

printf ("Process id = %d \n", getpid ());	**Process id = 2603**

توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

pid = fork; **printf ("pid = %d \n" , pid);**	**pid = 0**

خروجی نهایی برنامه به صورت زیر است:

```
*** Parent Process Begin ***
&i = 10000
i = 1
&i = 10000
```

```
Process id = 2600
pid = 2603
*** Parent Process End ***
*** Child Process Begin ***
&i = 10000
i = -1
&i = 10000
Process id = 2603
pid = 0
*** Child Process End ***
```