موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(حل تشریحی سوالات دولتی ۱۳۹۸)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندور اس تنن‌بام

ارسطو خلیلی‌فر

تست‌های کنکور کارشناسی ارشد سال ۱۳۹۸

۱- در سیستمی هر پردازه دارای 32 صفحه و اندازه‌ی هر صفحه 4KB است. طول آدرس فیزیکی در این سیستم 22 بیت است. طول آدرس منطقی و اندازه‌ی حافظه اصلی در این سیستم به ترتیب از راست به چپ کدام است؟ (مهندسی کامپیوتر – دولتی ۹۸)

۱) 22 بیت – 2^{12} کیلوبایت.
۲) 22 بیت – 2^{22} کیلوبایت.
۳) 17 بیت – 2^{12} کیلوبایت.
۴) 17 بیت – 2^{22} کیلوبایت.

۲- یک سیستم عامل 3 پردازه دارد که هر کدام از آنها 2 واحد از منبع R را نیاز دارند. کمترین تعداد واحدهای R چه مقداری باشد تا بن‌بست رخ ندهد؟ (مهندسی کامپیوتر – دولتی ۹۸)

۱) 3 ۲) 4 ۳) 5 ۴) 6

۳- در یک سیستم تک پردازنده، چنانچه یک برنامه که نیاز به 30 میلی‌ثانیه کار ورودی، 20 میلی‌ثانیه کار پردازش و 40 میلی‌ثانیه کار خروجی دارد، به تعداد بسیار زیاد به صورت چند برنامگی اجرا شود، در بهترین حالت، بهره‌وری CPU کدام است؟ (پردازنده‌های ورودی و خروجی از پردازنده اصلی مجزا هستند.) (مهندسی کامپیوتر – دولتی ۹۸)

۱) $\frac{1}{90}$ ۲) $\frac{20}{90}$ ۳) $\frac{20}{70}$ ۴) $\frac{20}{40}$

۴- در خصوص الگوریتم زیر، که برای پیاده‌سازی ناحیه بحرانی بین دو پردازه i و j ارائه شده است، کدام مورد درست است؟ (الگوریتم برای پردازه i است و مشابه آن برای j هم وجود دارد.)
(مهندسی کامپیوتر – دولتی ۹۸)

```
while(true){
  Flag[i] = true;
  turn = j
  while(Flag[i] && turn == j)
/* Critical Section */
Flag[i] = false
}
```

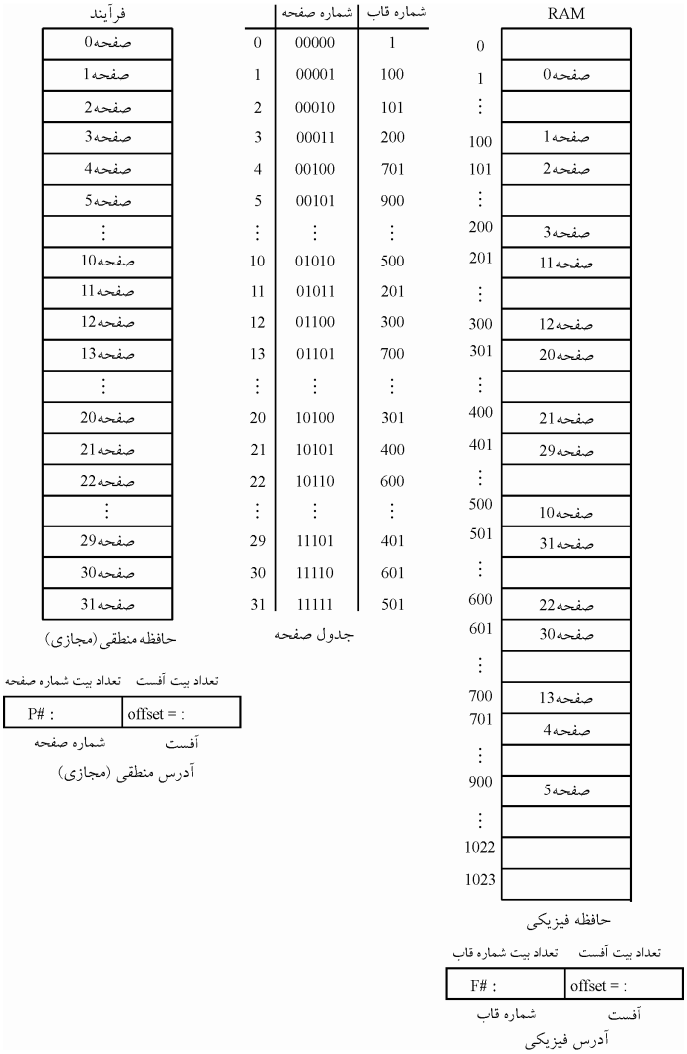
۱) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود دارد.
۲) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود ندارد.
۳) انحصار متقابل ندارد، پیشرفت دارد، انتظار محدود ندارد.
۴) انحصار متقابل ندارد، پیشرفت ندارد، انتظار محدود ندارد.

پاسخ تست‌های کنکور کارشناسی ارشد سال ۱۳۹۸

۱- گزینه (۳) صحیح است.

در صورت سوال مطرح شده است که اندازه هر پردازه یعنی فضـای آدرس منطقـی، 32 صـفحه 4 کیلوبایتی و طول آدرس فیزیکی 22 بیت است. همچنین خواسته شده است طول آدرس منطقـی و اندازه‌ی حافظه اصلی محاسبه گردد.

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بنـدی بـه صـورت زیـر است:



$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = 32$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}}$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^{32} = 5\text{bit}$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}}$$

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{4\text{KB}} = \log_2^{2^2 \times 2^{10}} = \log_2^{2^{12}} = 12\text{bit}$$

همچنین، اندازه آدرس‌های منطقی (مجازی) و فیزیکی به صورت زیر است:

طول آدرس منطقی = تعداد بیت شماره صفحه + تعداد بیت آفست $= 5\text{bit} + 12\text{bit} = 17\,\text{bit}$

طول آدرس فیزیکی = تعداد بیت شماره قاب + تعداد بیت آفست

تعداد بیت شماره قاب = تعداد بیت آدرس فیزیکی - تعداد بیت آفست $= 22\text{bit} - 12\text{bit} = 10\,\text{bit}$

همچنین داریم:

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{\text{تعداد بیت آدرس منطقی}} = 2^{\text{تعداد بیت شماره صفحه}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{17} = 2^5 \times 2^{12} = 2^{17} = 128\,\text{KB}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{\text{تعداد بیت آدرس فیزیکی}} = 2^{\text{تعداد بیت شماره قاب}} \times 2^{\text{تعداد بیت آفست}}$$

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{22} = 2^{10} \times 2^{12} = 2^{22}\,\text{bit} = 2^{12}\,\text{KB} = 4096\,\text{KB}$$

توجه: اندازه آدرس منطقی (مجازی) و فیزیکی (حقیقی) الزاماً برابر نیست.

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^{22}}{2^{12}} = 2^{10} = 1024$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{2^{10}} = 10\text{bit}$$

اندازه جدول صفحه = تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) × عرض جدول صفحه

توجه: عرض جدول صفحه برابر حاصل جمع تعداد بیت‌های کنترلی و تعداد بیت‌های شماره قاب می‌باشد، دقت کنید که تعداد بیت‌های شماره صفحه جزو عرض جدول صفحه نمی‌باشد، بلکه شماره صفحه، اندیس هر سطر جدول صفحه می‌باشد، به صورت زیر:

عرض جدول صفحه = تعداد بیت‌های شماره قاب + تعداد بیت‌های کنترلی

در سوال تعداد بیت‌های کنترلی بیان نشده است پس 0 بیت مربوط به بیت‌های کنترلی و 10 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق برابر $10\text{bit} + 0\text{bit} = 10\,\text{bit}$ می‌باشد.

همانطور که گفتیم، اندازه جدول صفحه، از رابطه زیر محاسبه می‌گردد:

عرض جدول صفحه × تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه) = اندازه جدول صفحه

که مطابق رابطه فوق داریم:

$$\text{اندازه جدول صفحه} = 32 \times 10\,\text{bit} = 320\,\text{bit} = 40\,\text{Byte}$$

۲- گزینه (۲) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

$$\sum_{i=1}^{n} \text{Request}[i] < m + n$$: مجموع درخواست‌های فرآیندها برای منابع

$$\rightarrow 2n < m + n \rightarrow n < m$$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$$n \times (r-1) = m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n \times (r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$n \times (r-1) < m$$

$$n \times r - n < m$$

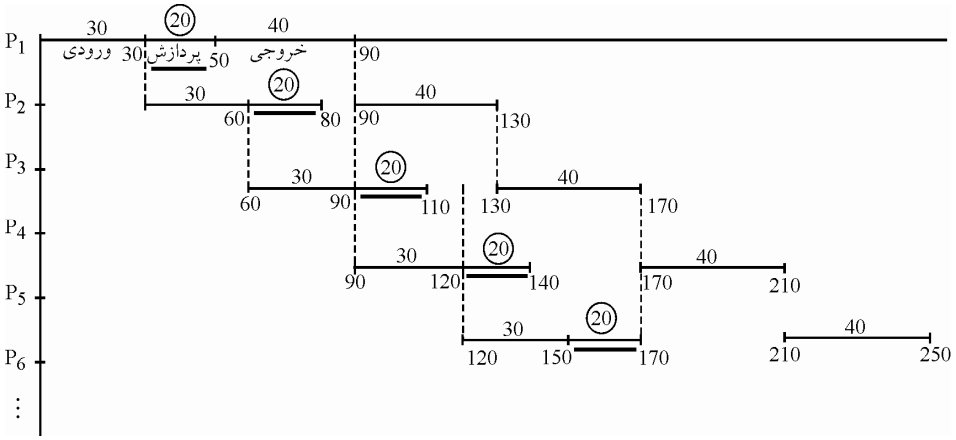
$$n \times r < m + n$$

$$\sum_{i=1}^{n} Request[i] < m + n$$

$$\longrightarrow 3 \times 2 < m + 3 \;\rightarrow\; 6 < m + 3 \;\rightarrow\; 3 < m \rightarrow R = m = 4$$

۳- گزینه (۴) صحیح است.

مطابق فرض سوال، یک برنامه نیاز به 30 میلی‌ثانیه کار ورودی، 20 میلی‌ثانیه کار پردازش و 40 میلی‌ثانیه کار خروجی دارد، نمودار گانت برای n فرآیند به صورت اجرای چند برنامگی به صورت زیر است:



تعداد فرآیندها	زمان مفید CPU	زمان پایان فرآیندها
1	20	90
2	40	130
3	60	170
4	80	210
5	100	250
⋮	⋮	⋮

$$\text{بهره‌وری} = \frac{20 \times n}{90 + (n-1) \times 40} = \frac{20 \times n}{50 + 40n}$$

$$\lim_{n \to \infty} \frac{20n}{50 + 40n} = \frac{20}{40} = \frac{1}{2}$$

۴- گزینه () صحیح است.

شرایط رقابتی (مسابقه)

هرگاه دو یا چند فرآیند همزمان با هم وارد ناحیه‌ی بحرانی (منبع مشترک) شوند، شرایط رقابتی پیش می‌آید. در شرایط رقابتی، نتیجه‌ی نهایی بستگی به ترتیب دسترسی‌ها دارد. در واقع فرآیندهای همکار بر هم اثر دارند و اینکه پردازنده، به چه ترتیبی و در چه زمان‌هایی بین آنها تعویض متن انجام دهد در ایجاد پاسخ نهایی اثرگذار خواهد بود. بنابراین علت شرایط رقابت **تعویض متن پردازنده** بین فرآیندهای همکار است.

برای کنترل شرایط رقابتی، باید راه حلی ارائه شود که سه شرط زیر را به عنوان **معیارهای اخلاقی** در رقابت، رعایت کند:

۱- شرط انحصار متقابل

برای برقرای شرط انحصارمتقابل، عامل مشترک را اسکورت کنید، مانند زمانی که وارد باجه‌ی تلفن همگانی (عامل مشترک) می‌شوید، در را می‌بندید تا مانع ورود شخص دیگری گردید! در عالم انسان‌ها، هیچ دو فردی نباید به طور همزمان وارد عامل مشترک شوند. در عالم فرآیندها نیز هیچ دو فرآیندی نباید به طور همزمان وارد عامل مشترک (ناحیه بحرانی) شوند. استفاده‌ی همزمان از عامل مشترک معنا ندارد! (اخلاقی نیست) بنابراین باید راهی پیدا کنیم که از ورود همزمان دو یا چند فرآیند به ناحیه‌ی بحرانی جلوگیری کند. به عبارت دیگر، آنچه که ما به آن نیاز داریم، انحصار متقابل است که در متون فارسی به آن دو به دو ناسازگاری یا مانعه الجمعی نیز گفته می‌شود، یعنی اگر یکی از فرآیندها در حال استفاده از حافظه‌ی اشتراکی، فایل اشتراکی و یا هر عامل اشتراکی رقابت‌زاست باید مطمئن باشیم که دیگر فرآیندها، در آن زمان از انجام همان کار محروم می‌باشند. در واقع از بین تمام فرآیندها، در هر لحظه تنها یک فرآیند مجاز است، در عامل مشترک باشد. بدین معنی که اگر فرآیندی در ناحیه‌ی بحرانی است، از ورود فرآیندهای دیگر به همان ناحیه‌ی بحرانی جلوگیری شود و تا خارج شدن فرآیند اول منتظر بمانند، زیرا هیچ دو فرآیندی نباید به طور همزمان وارد ناحیه‌ی بحرانی شوند. به یاد داشته باشید که استفاده‌ی همزمان از عامل مشترک معنا ندارد!

بنابراین برای برقراری شرط انحصار متقابل باید ساختاری را طراحی کنیم که در هر لحظه فقط یک فرآیند مجوز ورود به ناحیه‌ی بحرانی را داشته باشد. لذا هر فرآیند برای ورود به بخش بحرانی‌اش باید اجازه بگیرد. بخشی از کد فرآیند که این اجازه گرفتن را پیاده‌سازی می‌کند، بخش ورودی نام دارد. بخش بحرانی می‌تواند با بخش خروجی دنبال شود. این بخش خروجی کاری می‌کند که فرآیندهای دیگر بتوانند وارد ناحیه‌ی بحرانی‌شان، شوند. بقیه‌ی کد فرآیند را بخش باقی‌مانده می‌نامند. بنابراین ساختار کلی فرآیندها برای برقراری شرط انحصار متقابل به صورت زیر می‌باشد:

```
P (int i)  {
while ( TRUE)  {
entry_section () ;  // تلاش برای کسب اجازه‌ی ورود به ناحیه‌ی بحرانی
critical_section (); // ناحیه‌ی بحرانی
exit_section () ; // اعلام خروج از ناحیه‌ی بحرانی
remainder_section () ; // ناحیه‌ی باقی مانده
   }
}
```

توجه: بدترین شرایط وقتی است که یک فرآیند بخواهد بارها و بارها وارد ناحیه بحرانی خود شود، برای اینکه سخت‌ترین شرایط بررسی شود، ناحیه بحرانی را داخل حلقه بی‌نهایت قرار می‌دهیم.

۲- شرط پیشرفت

فرآیندی که داوطلب ورود به ناحیه‌ی بحرانی نیست و نیز در ناحیه‌ی بحرانی قرار ندارد، نباید در رقابت برای ورود سایر فرآیندها به ناحیه‌ی بحرانی شرکت کند، به عبارت دیگر، نباید مانع ورود فرآیندهای دیگر به ناحیه‌ی بحرانی شود. در یک بیان ساده‌تر می‌توان گفت، فرآیندی که در ناحیه-ی باقی‌مانده قرار دارد، حق جلوگیری از ورود فرآیندهای دیگر به ناحیه‌ی بحرانی را ندارد، یعنی نباید در تصمیم‌گیری برای ورود فرآیندها به ناحیه‌ی بحرانی شرکت کند.

۳- شرط انتظار محدود

فرآیندهایی که نیاز به ورود به ناحیه‌ی بحرانی دارند، باید مدت انتظارشان محدود باشد، یعنی نباید به طور نامحدود در حالت انتظار باقی بمانند.

انتظار نامحدود به دو دسته می‌باشد: (۱) قحطی، (۲) بن‌بست، بنابراین نباید در شرایط رقابتی بین فرآیندها، قحطی یا بن‌بست رخ دهد. برای اینکه شرط انتظار محدود برقرار باشد، باید هم قحطی و هم بن‌بست رخ ندهد.

قحطی (گرسنگی)

در عالم زندگی قحطی زمانی رخ می‌دهد که عده‌ای مدام از منابع مشترک استفاده کنند، و عده‌ای دیگر قادر به استفاده از منابع مشترک نباشند. زیرا دسته‌ی اول از اختصاص منابع به دسته‌ی دوم به طور مداوم و بدون رعایت یک حد بالای مشخص جلوگیری می‌کنند. در عالم فرآیندها نیز هرگاه فرآیندی به مدت نامعلوم و بدون رعایت یک حد بالای مشخص در انتظار گرفتن یک منبع بحرانی یا دسترسی به یک عامل مشترک بماند و فرآیندی دیگر مدام در حال استفاده از منبع بحرانی باشد، در این حالت فرآیند اول دچار قحطی شده است. بنابراین در صورت اقدام یک فرآیند برای ورود

به ناحیه‌ی بحرانی، باید محدودیتی برای تعداد دفعاتی که سایر فرآیندها می‌توانند وارد ناحیه‌ی بحرانی شوند، وجود داشته باشد تا قحطی رخ ندهد.

بن‌بست

به وضعیتی که در آن مجموعه‌ای متشکل از دو یا چند فرآیند برای همیشه منتظر یکدیگر بمانند (مسدود) و به عبارت دیگر دچار سیکل انتظار ابدی شوند، بن‌بست گفته می‌شود.

توجه: به تفاوت قحطی و بن‌بست توجه کنید، در قحطی فرآیندی مدام در حال کار و فرآیندی دیگر به مدت نامعلوم در انتظار است. اما در بن‌بست، مجموعه‌ای از فرآیندها در سیکل انتظار ابدی، گرفتار شده‌اند. نه راه پس دارند و نه راه پیش.

توجه: در کنترل شرایط رقابتی، رعایت شرط انحصار متقابل، شرط لازم و رعایت شروط پیشروی و انتظار محدود، شروط کافی برای ارائه‌ی یک راه‌حل **جامع** و **اخلاقی** به شمار می‌آیند.

ابتدا کد مطرح شده در صورت سوال را برای دو فرآیند P_0 و P_1 به صورت زیر بازنویسی می‌کنیم:

```
P0:
 {
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
 }
/*critical_section*/
{
④ C[0]=FALSE;
}
/*remainder_section*/
```

```
P1:
 {
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
 }
/*critical_section*/
 {
④ C[1]=FALSE;
 }
/*remainder_section*/
```

توجه: مقادیر اولیه به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

حال شرایط رقابتی را برای این الگوریتم بررسی می‌کنیم:

شرط انحصار متقابل:

برای کنترل برقراری شرط انحصار متقابل، شرط پیشرفت و شرط انتظار محدود (گرسنگی و بن‌بست) از آزمون‌های زیر استفاده می‌کنیم:

توجه: ما نام این آزمون‌ها را به عنوان مبدع آن «**قوانین ارسطو**» نام‌گذاری کردیم، این قوانین به «**قوانین چهارگانه ارسطو**» نیز موسوم است.

قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

① C[0]= TRUE;

② turn= 1;

③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

P1:

① C[1] =TRUE;

② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:

③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:

/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون اول وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم هم تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که وارد ناحیه بحرانی خودش شود، در این حالت شرط انحصار متقابل نقض شده است، خب موفق نشد. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط اول انحصار متقابل برقرار است.

فرم ساده قانون اول ارسطو (آزمون اول شرط انحصار متقابل)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو هم جور کن که بخواد وارد باجه تلفن همگانی بشه، اگه اونم تونست وارد باجه تلفن همگانی بشه اونوقت شرط اول انحصار متقابل نقض شده. اخلاق می‌گه اگه یه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط اول انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون دوم شرط انحصار متقابل)

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است.

P0:
{
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
}
/*critical_section*/
{
④ C[0]=FALSE;
}
/*remainder_section*/

P1:
{
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
}
/*critical_section*/
{
④ C[1]=FALSE;
}
/*remainder_section*/

توجه: مقادیر اولیه به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P0:

① C[0]= TRUE;

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:

① C[1]= TRUE;

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنابراین فرآیندی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صبر پیشه کند و متغیر نوبت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود، اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:

② turn= 1;

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:

② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

P_0:

③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_0:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفق نشدند. فرآیند دوم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین شرط دوم انحصار متقابل نیز برقرار است.

فرم ساده قانون دوم ارسطو (آزمون دوم شرط انحصار متقابل)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشنن اونوقت شرط دوم انحصار متقابل نقض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشن، یعنی اگه یه نفر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگه بشه شرط دوم انحصار متقابل رو نقض کرده. اخلاق اینو می‌گه، اخلاق.

توجه: برای برقرار بودن شرط انحصار متقابل باید **قانون اول ارسطو** (آزمون اول شرط انحصار متقابل) و **قانون دوم ارسطو** (آزمون دوم شرط انحصار متقابل) هر دو باهم برقرار باشند. بنابراین شرط انحصار متقابل در سوال مطرح شده برقرار است.

قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۳)** در ادامه فرآیند دوم را داخل ناحیه بحرانی

خودش قرار بده، **(گام ۴)** سپس همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۵)** در نهایت همان فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت شرط پیشرفت برقرار است، در غیر اینصورت شرط پیشرفت برقرار نیست.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:
/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P0:
/*critical_section*/
④ C[0]= FALSE;
حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P0:
/*remainder_section*/
همانطور که در (گام ۲) گفتیم قرار شد که همان فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۳) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

(گام ۳): فرآیند دوم را داخل ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند دوم را داخل ناحیه بحرانی خودش قرار بدهیم، خب نتوانستیم قرار بدهیم. پس در ادامه آزمون سوم نمی‌توانیم وارد (گام ۴) و به تبع (گام ۵) شویم. بنابراین شرط پیشرفت برقرار نیست. شرط پیشرفت در صورتی رعایت می‌شود که هر پنج گام قانون سوم به طور کامل طی شود، در غیر اینصورت شرط پیشرفت برقرار نیست. به عبارت دیگر شرط پیشرفت در صورتی برقرار نیست که فرآیندی که داخل ناحیه باقی‌مانده قرار دارد، جلوی پیشرفت (یعنی ورود به ناحیه بحرانی) فرآیند رقیب را بگیرد.

فرم ساده قانون سوم ارسطو (آزمون شرط پیشرفت)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۲) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۳) در ادامه یه آدم دیگه رو جور کن داخل باجه تلفن همگانی قرار بده، (گام ۴) سپس همان آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، (گام ۵) در نهایت اگه همون آدم

دوباره تونست بره داخل باجه تلفن، اونوقت شرط پیشرفت برقرار است. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن نبود، وقتی از باجه تلفن اومدی بیرون می‌تونی دوباره بری داخل باجه تلفن. به عبارت دیگر اخلاق می‌گه اگه کسی قصد ورود به باجه تلفن رو نداشته باشه یعنی کسی منتظر زدن تلفن نباشه اونوقت یه شخص دیگه‌ای می‌تونه بارها و بارها داخل باجه تلفن همگانی بره. اخلاق اینو می‌گه، اخلاق.

قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، **(گام ۲)** سپس فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، **(گام ۳)** در ادامه فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، **(گام ۴)** در نهایت همان فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. و شرط انتظار محدود نقض شده است.

(گام ۱): فرآیند اول را داخل ناحیه بحرانی خودش قرار بده، یعنی:

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین فعلا نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

دو خط اول فرآیند P1 را اجرا کنید.

P1:
① C[1] =TRUE;
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

خط سوم فرآیند P0 را مجددا اجرا کنید.

P0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P0:

/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۲) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P_0 مشغول حرکت است.

(گام ۲): فرآیند دوم را پشت ناحیه بحرانی خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

فرض کنید فرآیند P1 نیز در ادامه حرکت خود قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:

③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که در (گام ۲) گفتیم قرار شد که فرآیند دوم را پشت ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۳) می‌شویم. هم اکنون پردازنده پشت ناحیه بحرانی فرآیند P1 در یک حلقه انتظار، دچار انتظار مشغول است.

(گام ۳): فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

فرض کنید فرآیند P0 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P0:

/*critical_section*/

④ C[0]=FALSE;

حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانده خودش قرار می‌گیرد، به صورت زیر:

P0:

/*remainder_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه باقی‌مانده خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون چهارم وارد (گام ۴) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P0 مشغول حرکت است.

(گام ۴): فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است. یعنی:
فرض کنید فرآیند P0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] =TRUE و turn = 1 است.

شرط حلقه TRUE است، بنابراین نمی‌توانیم فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم.
توجه: شاید در این مرحله پیش خودتان فکر کنید که خب مثل سابق دوباره سراغ فرآیند P1 می‌رویم و با turn= 0 در خط ② راه را برای ورود فرآیند P0 باز می‌کنیم، اول اینکه خب نادرست فکر کردید، دقت کنید که شما اصلا نمی‌توانید مجددا سراغ خط ② از فرآیند P1 بروید چون قبلا از آن خط عبور کرده بودید و اگر هم سراغ فرآیند P1 بروید در ادامه همان خط ③ و حلقه while را مشاهده خواهید کرد. مانند انداختن یک سکه که بعد انداختن به سمت پایین حرکت می‌کند، مگر اینکه دوباره آنرا به سمت بالا پرتاپ کنیم. دوم اینکه از قوانین تبعیت کنید و به هیچ عنوان به مراحل آن دست نزنید و فقط و فقط مطابق قوانین حرکت کنید. اینطوری موفق می‌شوید.
همانطور که در (گام ۴) گفتیم قرار شد که فرآیند اول به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیه بحرانی خودش شود، اگر موفق شود که مجددا وارد ناحیه بحرانی خودش شود، آنگاه در این حالت فرآیند دوم دچار گرسنگی شده است، خب نشد، فرآیند اول نتوانست مجددا وارد ناحیه بحرانی خودش بشود. بنابراین گرسنگی رخ نداده است.

فرم ساده قانون چهارم ارسطو (آزمون گرسنگی)

(گام ۱) یه آدم رو جور کن داخل باجه تلفن همگانی قرار بده، **(گام ۲)** سپس یه آدم دیگه رو جور کن پشت در باجه تلفن همگانی قرار بده، **(گام ۳)** در ادامه آدم داخل باجه تلفن همگانی رو از داخل باجه تلفن خارجش کن و بیارش بیرون، **(گام ۴)** در نهایت اگه همون آدم دوباره تونست بره داخل باجه تلفن، اونوقت اون یکی آدمه دچار گرسنگی شده. اخلاق می‌گه اگه یه دفعه داخل باجه تلفن همگانی رفتی و بعد بیرون کسی منتظر زدن تلفن بود، وقتی از باجه تلفن اومدی بیرون نباید دوباره بری داخل باجه تلفن چون اون موقع دوستت رو دچار گرسنگی کردی. اخلاق اینو می‌گه، اخلاق.

قانون دوم ارسطو(آزمون بن بست)

جهت بررسی بن بست از همان قانون دوم استفاده می‌شود. در واقع روال بررسی همان قانون دوم است، اما نتیجه قانون متفاوت است.

فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدید اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. به عبارت دیگر هرگاه دو فرآیند متقاضی ورود به ناحیه بحرانی به طور همزمان تا ابد منتظر ورود به ناحیه بحرانی باشند، در این شرایط هر دو فرآیند مسدود و به خواب رفته‌اند که در این حالت بن‌بست رخ داده است.

```
P0:
{
① C[0]= TRUE;
② turn= 1;
③ while(C[0] && turn = 1) do;
}
/*critical_section*/
{
④ C[0]=FALSE;
}
/*remainder_section*/
```

```
P1:
{
① C[1]= TRUE;
② turn= 0;
③ while(C[1] && turn = 0) do;
}
/*critical_section*/
{
④ C[1]=FALSE;
}
/*remainder_section*/
```

توجه: مقادیر اولیه به صورت زیر است:

turn =1, C[0] = TRUE, C[1] = TRUE

فرض کنید فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P0:
① C[0]= TRUE;
```

همچنین فرض کنید فرآیند P1 نیز به شکل همروند یا موازی با فرآیند P0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

```
P1:
① C[1]= TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین تابلوی وضعیت فرآیندها هر دو C[0] = TRUE و C[1] = TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند. اما متغیر نوبت turn نمی‌تواند در یک زمان هم صفر و هم یک باشد. زیرا پس از آنکه هر دو فرآیند، شماره‌ی فرآیند خود را در

متغیر نوبت turn ذخیره نمودند. فرآیندی که دیرتر شماره‌اش را ذخیره کند، فرآیندی است که شماره‌اش در متغیر نوبت turn باقی می‌ماند و دیگری اثرش در متغیر نوبت turn از بین می‌رود. در واقع سرنوشت ورود فرآیندها به ناحیه بحرانی به متغییر turn گره خورده است، بنابراین فرآیندی که دیرتر متغیر نوبت turn را مقداردهی کرده است، باید صبر پیشه کند و متغیر نوبت turn را نگه‌داری کند و در حلقه‌ی انتظار بچرخد. و فرآیندی که زودتر متغیر نوبت turn را مقداردهی کرده است، وارد ناحیه بحرانی می‌شود.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به مقداردهی متغیر نوبت turn کنند، بنابراین مقدار متغیر نوبت turn برابر با یک خواهد بود (turn=1)، وقتی که دو فرآیند به دستور while می‌رسند، خط ③ برای فرآیند P_0 برقرار نیست و وارد ناحیه بحرانی می‌شود اما فرآیند P1 باید در یک حلقه انتظار مشغول، مشغول باشد. پس انحصار متقابل رعایت می‌شود. به صورت زیر:

P_0:
② turn= 1;

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_1:
② turn= 0;

در ادامه پردازنده را از فرآیند P1 بگیرید و به فرآیند P0 بدهید.

P_0:
③ while(C[0] && turn = 1) do;

توجه: هم اکنون C[0] = TRUE و turn = 0 است.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_0 قرار می‌گیرد، به صورت زیر:

P_0:
/*critical_section*/

در ادامه پردازنده را از فرآیند P0 بگیرید و به فرآیند P1 بدهید.

P_0:
③ while(C[1] && turn = 0) do;

توجه: هم اکنون C[1] = TRUE و turn = 0 است.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P1 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P1 در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود می‌ماند و می‌چرخد تا مادامی که کوانتوم آن تمام شود. این پدیده به انتظار مشغول (Busy Waiting) موسوم است.

همانطور که گفتیم قرار شد که فرآیند اول و دوم را به طور همروند در سیستم تک پردازنده‌ای و یا موازی در سیستم چند پردازنده‌ای حرکت بدهیم اگر هر دو باهم نتوانستند وارد ناحیه بحرانی شوند و هردو باهم پشت ناحیه بحرانی خودشان مسدود شدند، آنگاه در این حالت بن بست رخ داده است و شرط انتظار محدود نقض شده است. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود نشدند. فرآیند اول توانست وارد ناحیه بحرانی خودش شود، اما فرآیند دوم نتوانست وارد ناحیه بحرانی خودش شود. بنابراین بن بست رخ نداده است.

فرم ساده قانون دوم ارسطو (آزمون بن بست)

دوتا آدم رو جور کن و به طور همزمان به سمت داخل باجه تلفن همگانی حرکتشون بده، اگه هر دو باهم نتونستن به طور همزمان وارد باجه تلفن همگانی بشن و هردو باهم پشت در باجه تلفن همگانی مسدود شدن اونوقت بن بست رخ داده. اخلاق دیگه اینجا چیزی نمی‌گه و سکوت می‌کنه، چون دیگه بن‌بست شده!

توجه: برای برقرار بودن شرط انتظار محدود باید **قانون دوم ارسطو** (آزمون بن بست) و **قانون چهارم ارسطو** (آزمون گرسنگی) هر دو باهم برقرار باشند. بنابراین شرط انتظار محدود در سوال مطرح شده برقرار است.

صورت سوال به این شکل است:

در خصوص الگوریتم زیر، که برای پیاده‌سازی ناحیه بحرانی بین دو پردازه i و j ارائه شده است، کدام مورد درست است؟ (الگوریتم برای پردازه i است و مشابه آن برای j هم وجود دارد.)

```
while(true){
   Flag[i] = true;
   turn = j
   while(Flag[i] && turn == j)
/*Critical Section*/
 Flag[i] = false
}
```

۱) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود دارد.

گزینه اول **نادرست** است، زیرا شرط انحصار متقابل برقرار است و برآورده می‌کند، شرط پیشرفت برقرار نیست و برآورده نمی‌کند و شرط انتظار محدود برقرار است و برآورده می‌کند.

۲) انحصار متقابل دارد، پیشرفت دارد، انتظار محدود ندارد.

گزینه دوم **نادرست** است، زیرا شرط انحصار متقابل برقرار است و برآورده می‌کند، شرط پیشرفت برقرار نیست و برآورده نمی‌کند و شرط انتظار محدود برقرار است و برآورده می‌کند.

۳) انحصار متقابل ندارد، پیشرفت دارد، انتظار محدود ندارد.

گزینه سوم **نادرست** است، زیرا شرط انحصار متقابل برقرار است و برآورده می‌کند، شرط پیشرفت

برقرار نیست و برآورده نمی‌کند و شرط انتظار محدود برقرار است و برآورده می‌کند.

۴) انحصار متقابل ندارد، پیشرفت ندارد، انتظار محدود ندارد.

گزینه چهارم **نادرست** است، زیرا شرط انحصار متقابل برقرار است و برآورده می‌کند، شرط پیشرفت برقرار نیست و برآورده نمی‌کند و شرط انتظار محدود برقرار است و برآورده می‌کند.

توجه: همانطور که واضح و مشخص هست، همه گزینه‌ها نادرست هستند و هیچ یک از گزینه‌ها نمی‌تواند پاسخ سوال باشد.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه خود، گزینه دوم را به عنوان پاسخ اعلام کرده بود. اما در کلید نهایی این سوال حذف گردید، که کار درستی بوده است.

تست‌های فصل هفتم

۹۴- در سیستمی 4 پردازه(Process) و 5 منبع یکسان وجود دارد. اگر هر پردازه حداکثر به 2 منبع نیاز داشته باشد، کدام مورد درست است؟

(مهندسی *IT* – دولتی ۹۸)

۱) حتما در این سیستم بن‌بست رخ می‌دهد.
۲) ممکن است در این سیستم بن‌بست رخ دهد.
۳) هیچگاه در این سیستم بن‌بست رخ نمی‌دهد.
۴) رخ دادن بن‌بست به ترتیب درخواست منابع بستگی دارد.

پاسخ‌های فصل هفتم

۹۴- گزینه (۳) صحیح است.

در یک مجموعه با n فرآیند و m منبع از یک نوع، اگر شرط زیر برقرار باشد، هرگز بن‌بست رخ نمی‌دهد:

مجموع درخواست‌های فرآیندها برای منابع: $\sum_{i=1}^{n} \text{Request}[i] < m + n$

$$\rightarrow 2n < m + n \rightarrow n < m$$

توجه: چنانچه فرآیندها یکی پس از دیگری و به صورت ترتیبی اجرا گردند، بدین صورت که فرآیند اول کاملاً اجرا شود و سپس فرآیند دوم اجرا گردد و بعد از اتمام، فرآیند سوم اجرا شود و به همین ترتیب ادامه پیدا کند، آنگاه در سیستم هیچگاه بن‌بست رخ نمی‌دهد.

توجه: فرض کنید هر فرآیند حداکثر به r منبع نیازمند است. اگر فرآیندی r منبع مورد نیاز خود را دریافت نماید، بعد از مدتی، اجرای فرآیند به پایان می‌رسد و منابع را آزاد می‌کند. فرآیندهای دیگر نیز یک به یک، مانند فرآیند اول، r منبع را دریافت خواهند کرد و اجرایشان به پایان می‌رسد و بدین‌ترتیب بن‌بستی در سیستم نخواهیم داشت. اما در بدترین حالت بن‌بست زمانی رخ می‌دهد که تمام فرآیندها $(r-1)$ منبع را در اختیار داشته باشند و همگی یک به یک در انتظار منبع آخر باقی بمانند. بنابراین اگر نمونه دیگری از منبع در سیستم موجود باشد، آن نمونه به یک فرآیند اختصاص می‌یابد و آن فرآیند بعد از تکمیل اجرای برنامه، تمام منابع را به سیستم برمی‌گرداند. سپس فرآیندهای دیگر یک به یک از انتظار خارج شده و بن‌بست رخ نمی‌دهد.

اگر n فرآیند در سیستم موجود باشد و هر فرآیند $(r-1)$ منبع را در اختیار داشته باشد، آنگاه شرایط ایجاد احتمال بن‌بست به صورت زیر است:

$$n \times (r-1) = m$$

حال اگر مقدار m حداقل یک واحد بیشتر از $n \times (r-1)$ شود، آنگاه سیستم دیگر دچار بن‌بست نمی‌شود، به صورت زیر:

$$n \times (r-1) < m$$

$$n \times r - n < m$$

$$n \times r < m + n$$

$$\sum_{i=1}^{n} \text{Request}[i] < m + n$$

$$\longrightarrow 4 \times 2 < 5 + 4 \rightarrow 8 < 9$$

تست‌های فصل دوم

۹۵- متوسط زمان انتظار برای پردازه‌های داده شده در حالتی که از الگوریتم «اول-کمترین-زمان» (Shortest Job First) قبضه‌شدنی استفاده می‌کنیم، کدام است؟

(مهندسی *IT* – دولتی ۹۸)

پردازه	زمان ورود	زمان مورد استفاده از CPU
P_1	2	5
P_2	3	13
P_3	0	8
P_4	5	4
P_5	1	10

۱) 10.2

۲) 10.6

۳) 12.75

۴) 18.2

پاسخ‌های فصل دوم

۹۵- گزینه (۱) صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجـرای کمتـری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و

SPT (Shortest Process Time) نیز نامیده می‌شود.

توجه: SJF یک الگوریتم **انحصاری یا غیرقبضه‌ای** (Non Preemptive) است.

توجه: یک نقص عمده الگوریتمSJF این است که ممکـن اسـت باعـث قحطـی‌زدگـی فرآینـدهای طولانی شود. به این ترتیب که اگر همواره تعـدادی فرآینـد کوچـک وارد سیسـتم شـوند، اجـرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابـد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگـر دو فرآینـد مـدت زمـان اجـرای برابـر داشـته باشـند، بـر اسـاس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسـخ و میـانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآینـدها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجـرای فرآینـدها را فقـط حدس زده و به طور تقریبی بدست آورد.

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری یا قبضه‌ای** (Preemptive) الگوریتم SJF است. در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنـده را در اختیار می‌گیرد.

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،

SRTF (Shortest Remaining Time First) و

SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

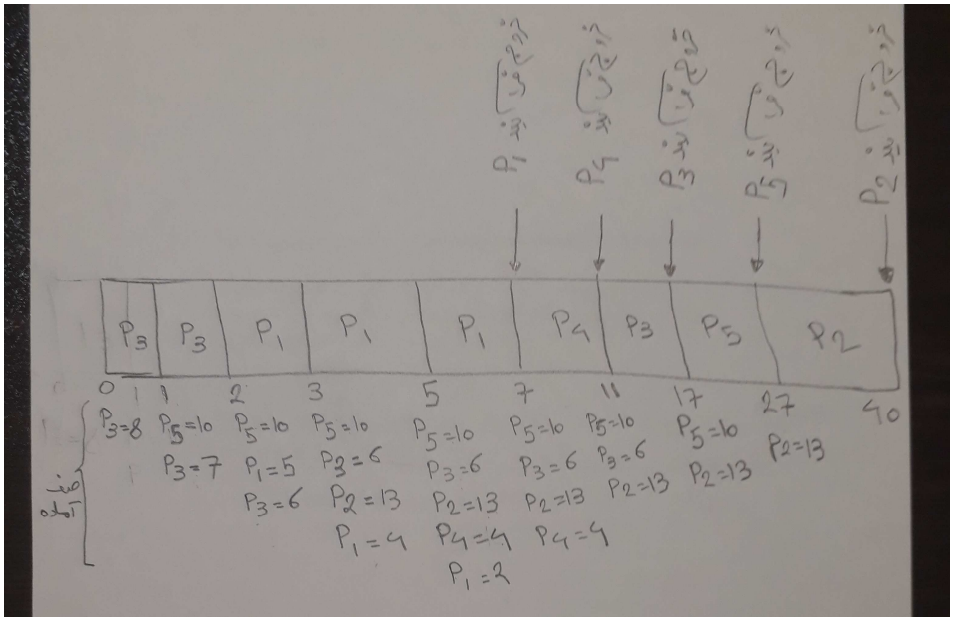
توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتمSRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	2	5		
P_2	3	13		
P_3	0	8		
P_4	5	4		
P_5	1	10		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 7-2=5$

P_2 زمان بازگشت $= 40-3=37$

P_3 زمان بازگشت $= 17-0=17$

P_4 زمان بازگشت $= 11-5=6$

P_5 زمان بازگشت $= 27 - 1 = 26$

میانگین زمان بازگشت $= ATT = \frac{5+37+17+6+26}{5} = \frac{91}{5} = 18.2$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 5 - 5 = 0$

P_2 زمان انتظار $= 37 - 13 = 24$

P_3 زمان انتظار $= 17 - 8 = 9$

P_4 زمان انتظار $= 6 - 4 = 2$

P_5 زمان انتظار $= 26 - 10 = 16$

میانگین زمان انتظار $= AWT = \frac{0+24+9+2+16}{5} = \frac{51}{5} = 10.2$

میانگین زمان اجرا $= AST = \frac{5+13+8+4+10}{5} = \frac{40}{5} = 8$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

18.2= 8 + 10.2

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	2	5	0	5
P_2	3	13	24	37
P_3	0	8	9	17
P_4	5	4	2	6
P_5	1	10	16	26

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

18.2 10.2 8

تست‌های فصل پنجم

۹۶- در یک سیستم عامل در کدام حالت فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد؟

(مهندسی *IT* – دولتی ۹۸)

۱) اندازه صفحه کوچک شود.
۲) پردازه CPU-bound باشد.
۳) پردازه IO-bound باشد.
۴) محلی بودن ارجاع‌ها در پردازه بیشتر شود.

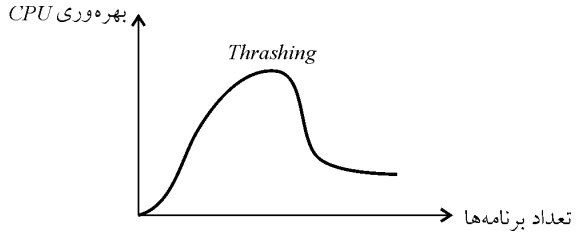
پاسخ‌های فصل پنجم

۹۶- گزینه (۴) صحیح است.

اگر حافظه تخصیص داده شده به یک فرآیند، آنقدر کوچک باشد که نتواند صفحاتی که فرآیند، زیاد با آنها سروکار دارد را در خود جای دهد، سرعت اجرای فرآیند کاهش می‌یابد، زیرا این فرآیند خطاهای نقص صفحه زیادی تولید می‌کند. در این حالت مدت زمان اجرای یک فرآیند به چندین و چند برابر حالت عادی افزایش می‌یابد. اصطلاحاً به برنامه‌ای که در هر 2 یا 3 دستور خود یک خطای نقص صفحه تولید کند، **کوبیده شده (لهیده)** گویند.

توجه: فرآیندی که در حالت Thrashing واقع است، به جای اینکه زمان CPU را به اجرا اختصاص دهد، زمان زیادی را صرف انجام عملیات صفحه‌بندی می‌کند.

توجه: نسبت میزان بهره مفیدCPU با افزایش تعداد فرآیندها به صورت زیر است:



پدیده کوبیدگی

در واقع تا یک نقطه، افزایش تعداد فرآیندها (افزایش سطح چند برنامگی)، بهره‌وری CPU را افزایش می‌دهد، اما از یک نقطه به بعد، افزایش تعداد برنامه‌ها بهره‌وری پردازنده را کاهش می‌دهد. به عنوان مثال دو فرآیند A وB را در نظر بگیرید. فرض کنید برنامه A در حال اجرا به یک صفحه نیاز دارد، بنابراین بعد از یک خطای نقص صفحه، صفحه مورد تقاضایش به حافظه آورده می‌شود و به جای یک صفحه از فرآیند B در یک قاب حافظه قرار می‌گیرد، در این لحظه نوبت به اجرای برنامه B می‌رسد و برنامه B به همان صفحه قدیمی خودش نیاز دارد، بنابراین با یک خطای نقص صفحه آن صفحه را به حافظه برمی‌گرداند و این‌بار صفحه فرآیند A باید حافظه را ترک کند و الی آخر... در واقع در این حالت صفحاتی که فرآیندها زیاد با آن سروکار دارند به طور کامل در حافظه قرار ندارد و کارایی سیستم به شدت کاهش می‌یابد.

توجه: اگر عملکرد سیستم عامل را در قبال درجه چندبرنامگی سیستم بررسی کنیم، متوجه می‌شویم تحت شرایطی خاص، پدیده Thrashing به شدت تشدید می‌شود.

سناریوی زیر را در نظر بگیرید:

می‌دانیم سیستم عامل بر بهره‌وری CPU نظارت دارد و اگر بهره‌وری CPU بسیار کم باشد،

درجه چندبرنامگی را با افزودن یک فرآیند جدید به سیستم افـزایش مـی‌دهـد تـا بهـره‌وری CPU افزایش یابد. فرض کنید در این سیستم از یک الگوریتم جایگزینی سراسری برای صفحات استفاده می‌شود که صفحات را بدون توجه به اینکه مربوط به کدام فرآیند هستند، جایگزین می‌کنـد. حـال فرض کنید فرآیندی وارد یک مرحله جدید از اجرا شده و به چند صفحه جدید نیاز دارد. بنـابراین وقفه‌های نقص صفحه برای این فرآیند آغاز می‌شوند و این فرآیند قاب‌های فرآیندهای دیگر را در اختیار می‌گیرد.

از طرفی فرآیندهایی که تعدادی از صفحات آن‌ها از حافظه خارج شده‌اند، به آن صفحات نیـاز دارند، بنابراین وقفه‌های نقص صفحه برای آن‌ها نیز شروع می‌شود و به طور مشـابه ایـن فرآینـدها قاب‌های فرآیندهای دیگر را در اختیار می‌گیرند و این مسئله برای فرآیندهای دیگر تکرار می‌شـود. بنابراین فرآیندها پی‌درپی خطای نقص صفحه را تجربه می‌کنند و صفحات درگیر، مرتباً به داخل و خارج حافظه مبادله می‌شوند. به این ترتیب بهـره‌وری CPU کـاهش مـی‌یابـد و زمانبنـد پردازنـده متوجه این کاهش بهره‌وری می‌گردد و جهت افزایش بهره‌وری CPU درجه چندبرنامگی را کـاهش می‌دهد.

توجه: یک راه حل کنترل فرکانس نقص صفحه یا در بـدترین حالـت مقابلـه بـا Trashing کنتـرل تعداد وقفه‌های نقـص صـفحه بـا اسـتفاده از الگـوریتم فرکـانس نقـص صـفحه یـا (Page Fault Frequency) PFF است. این الگوریتم زمان کاهش یا افزایش تعداد قاب صفحه تخصیص یافته بـه یک فرآیند را بیان می‌کند، اما کنترلی در مورد این که کدام صـفحه بایـد در هنگـام نقـص صـفحه جایگزین شود، انجام نمی‌دهد. این الگوریتم فقط اندازه مجموعه تخصیص را کنتـرل می‌کنـد. کـه بهتـــر اســـت، ایـــن مجموعـــه تخصـــیص برابـــر مجموعـــه کـــاری (working set) باشد. روال کار بدین صورت است که اگر تعداد نقص‌های صفحه برای یک فرآیند افزایش یافت، باید تعدادی قاب حافظه به آن اختصاص یابد و اگر تعداد نقص‌هـای صـفحه بـرای یک فرآیند از یک حد پایین کم‌تر شد، باید تعدادی قاب از آن فرآیند پس گرفته شـود. نکتـه مهـم اینجاست که اگر تعداد نقص‌های صفحه یـک فرآینـد بـالا رفـت، ولـی قـاب آزاد حافظـه وجـود نداشت، باید درجه چند برنامگی سیستم را کاهش داد و یک یا چند فرآیند را بـه حالـت معلـق در آورد تا قاب‌هایی که در اختیار دارد، آزاد شود.

مدل مجموعه کاری (Working Set Model) هر فرآیند، عبارت است از صفحاتی از آن فرآیند که اگر در حافظه قرار داشته باشند، فرآیند موردنظر کارایی و سرعت قابل قبولی دارد. در واقع اگـر مجموعه کاری یک فرآیند در حافظه باشد، فرآیند با تعداد معقول و مناسبی وقفه نقص صـفحه بـه کار خود ادامه می‌دهد. ایده مدل مجموعه کـاری در واقـع یـک رهیافـت جهـت مقابلـه بـا پدیـده Thrashing است. در این مدل (راهکار) از مفهوم مجموعه کاری به خوبی استفاده می‌شود. به طور خلاصه مدل مجموعه کاری بیان می‌کند که قبل از دادن نوبت اجرا به یک فرآینـد، بایـد مجموعـه کاری آن فرآیند به درون حافظه بار شود. می‌توان گفت با انتقال مجموعه کاری یک فرآیند قبـل از

اجرای آن به درون حافظه، نرخ خطای نقص صفحه کاهش می‌یابد.

نکته: به بار کردن صفحات یک فرآیند قبل از اجرای آن، پیش صفحه‌بندی (Prepaging) گویند.

صورت سوال به این شکل است:

در یک سیستم عامل در کدام حالت فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد؟

۱) اندازه صفحه کوچک شود.

گزینه اول پاسخ سوال نیست، زیرا اگر اندازه صفحه کوچک شود، آنگاه تعداد صفحات یک فرآیند بیشتر می‌شود و به تبع فرکانس نقص صفحه افزایش می‌یابد. آنچه در کاهش فرکانس نقص صفحه موثر است، افزایش اندازه حافظه اصلی، کاهش درجه چندبرنامگی، افزایش اندازه صفحه است. که البته پس از وقوع Trashing، فقط راه حل‌های افزایش اندازه حافظه اصلی و کاهش درجه چندبرنامگی **موثر** هستند. بالا بودن paging به دلیل کمبود حافظه اصلی و بالا بودن درجه چندبرنامگی اتفاق افتاده است. که در این حالت کوبیدگی (Trashing) رخ داده است و به تبع آن میزان بهره‌وری CPU پایین است. بنابراین CPU کنونی اغلب بی‌کار است و حتی استفاده از CPU سریعتر هم سودی نخواهد داشت. برای کاهش پدیده کوبیدگی و به تبع افزایش بهره‌وری CPU یا باید اندازه حافظه اصلی را افزایش داد و یا درجه چندبرنامگی را کاهش داد.

۲) پردازه CPU-bound باشد.

گزینه دوم پاسخ سوال نیست، زیرا CPU-bound بودن یک پردازه ارتباطی به افزایش یا کاهش فرکانس نقص صفحه ندارد. فرآیندهای CPU-bound بیشتر وقتشان صرف پردازش در CPU می‌شود. این فرآیندها محدود به محاسبه (Compute-bound) یا محدود به CPU هستند. اغلب فرآیندها به تناوب نیازمند فوران‌های (Bursts) محاسباتی و در لابه لای آن درخواست‌های IO هستند. معمولا هر فرآیند در تکه زمان‌های کوتاه به CPU نیاز دارد و در انتهای هر یک از این تکه زمان‌ها، در انتظار وقوع یک رویداد مانند تکمیل عملیات IO بلوکه می‌شود و CPU را به طور موقت رها می‌کند و پس از وقوع رویداد مورد نظر آماده می‌شود تا CPU را برای یک تکه زمانی دیگر اخذ کند. به هر یک از این تکه‌های زمانی، CPU Burst می‌گویند. در انتهای آخرین CPU Burst، فرآیند خاتمه می‌یابد.

۳) پردازه IO-bound باشد.

گزینه سوم پاسخ سوال نیست، زیرا IO-bound بودن یک پردازه ارتباطی به افزایش یا کاهش فرکانس نقص صفحه ندارد. فرآیندهای IO-bound بیشتر وقتشان صرف انتظار در IO می‌شود. این فرآیندها محدود به IO هستند. دقت کنید که آنچه که عامل تعیین کننده CPU-bound بودن یا IO-bound بودن است، CPU Burst است و نه طول زمان IO، یعنی فرآیندهای محدود به IO به این دلیل محدود به IO هستند که آنها در بین درخواست‌های IO محاسبات چندانی را انجام نمی‌دهند و علت آن این نیست که درخواست‌های IO طولانی دارند. نکته حائز اهمیت دیگر این است که

هرچه قدر سرعت CPU بیشتر شود، کار پردازش سریعتر انجام می‌شود، که در این حالت فرآیندها محدود به IO می‌شوند.

۴) محلی بودن ارجاع‌ها در پردازه بیشتر شود.

گزینه چهارم پاسخ سوال است، زیرا اگر محلی بودن ارجاع‌ها در پردازه بیشتر شود، آنگاه فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد. زیرا هر صفحه‌ای که به حافظه اصلی آورده می‌شود، به دلیل محلی بودن ارجاع‌ها اطلاعات زیادی از همان صفحه قابل استخراج است که نتیجه آن می‌شود نقص صفحه کمتر و عدم مراجعه مکرر به صفحات بعدی که نتیجه آن افزایش نقص صفحه باشد.

به طور کلی محلیت در دو نوع، (۱) محلیت مکانی (Spatial Locality) و (۲) محلیت زمانی (Temporal Locality) وجود دارد.

محلی بودن مکانی را مانند شعاع **متغیر** یک دایره در نظر بگیرید، هرچه قدر این شعاع کوچکتر شود محلی بودن مکانی در **کناره‌ها و اطراف** بیشتر و بیشتر می‌شود و هرچه قدر این شعاع بزرگتر شود محلی بودن مکانی در **کناره‌ها و اطراف** کمتر و کمتر می‌شود. شما با هم محله‌ای‌های خود محلی‌تر مکانی هستید، اما هرچه قدر از محله و مکان خود دور می‌شوید، اهالی محله‌های دیگر با شما کمتر محلی‌تر مکانی هستند. به عبارت دیگر محلی بودن مکانی خوب می‌گوید مراجعه بعدی، در **نزدیکی** همین مراجعه فعلی است.

محلی بودن زمانی را مانند شعاع **ثابت** یک دایره در نظر بگیرید، که محدوده این شعاع هیچگاه تغییر نکند. شما با خانواده خود محلی زمانی هستید. به عبارت دیگر محلی بودن زمانی **خوب** می‌گوید مراجعه بعدی، در **محل** همین مراجعه فعلی است.

مثال: درباره **ویژگی محلیت (Locality) برنامه زیر، کدام مورد درست است؟**

```
int sum (int v[n]){
    int i, sum = 0
    for(i = 0; i < n; i++)
        sum+ = v[i]
}
```

۱) متغیر sum دارای ویژگی محلیت زمانی (Temporal Locality) خوب و متغیر v دارای ویژگی محلیت زمانی بد و محلیت مکانی (Spatial Locality) خوب می‌باشد.

۲) متغیر sum دارای ویژگی محلیت زمانی (Temporal Locality) خوب و متغیر v دارای ویژگی محلیت زمانی بد و محلیت مکانی (Spatial Locality) بد می‌باشد.

۳) متغیر sum دارای ویژگی محلیت مکانی (Spatial Locality) خوب و متغیر v دارای ویژگی محلیت زمانی(Temporal Locality) بد و محلیت مکانی خوب می‌باشد.

۴) متغیر sum دارای ویژگی محلیت مکانی (Spatial Locality) خوب و متغیر v دارای ویژگی محلیت زمانی(Temporal Locality) خوب و محلیت مکانی بد می‌باشد.

پاسخ: گزینه (۱) صحیح است.

آرایه مطرح شده در صورت سوال یعنی v[i] دارای خاصیت محلی بودن مکانی **خوب** است. عناصر آرایه v[i] در داخل یک حلقه for از اندیس صفر تا n به ترتیب و پشت سرهم خوانده می‌شوند، یعنی مراجعه بعدی، مدام در نزدیکی همین مراجعه فعلی است.

متغیر مطرح شده در صورت سوال یعنی sum دارای خاصیت محلی بودن زمانی **خوب** است. متغیر sum در داخل یک حلقه for از اندیس صفر تا n خوانده می‌شود، یعنی مراجعه بعدی، مدام در محل همین مراجعه فعلی است.

همانطور که گفتیم آرایه v[i] دارای خاصیت محلی بودن مکانی **خوب** است، اما از آنجاکه به هریک از عناصر آرایه v[i] در حرکت حلقه فقط و فقط یکبار مراجعه می‌شود، بنابراین آرایه v[i] دارای خاصیت محلی بودن زمانی **بد** است. همچنین همانطور که گفتیم متغیر sum دارای خاصیت محلی بودن زمانی **خوب** است، اما از آنجاکه به کناره‌ها و اطراف متغیر sum در حرکت حلقه مراجعه نمی‌شود، بنابراین متغیر sum دارای خاصیت محلی بودن مکانی **بد** است.

توجه: رعایت اصول ساده برنامه‌نویسی می‌تواند در افزایش سرعت اجرای فرآیندها و کاهش تعداد نقص صفحه تأثیر مستقیم داشته باشد. به عنوان مثال قطعه برنامه زیر را در نظر بگیرید که سعی دارد همه عناصر یک آرایه دو بعدی 100× 100 را با صفر پر کند:

```
for i := 1 to 100 do
    for j := 1 to 100  do
        a[j,i] := 0;
```

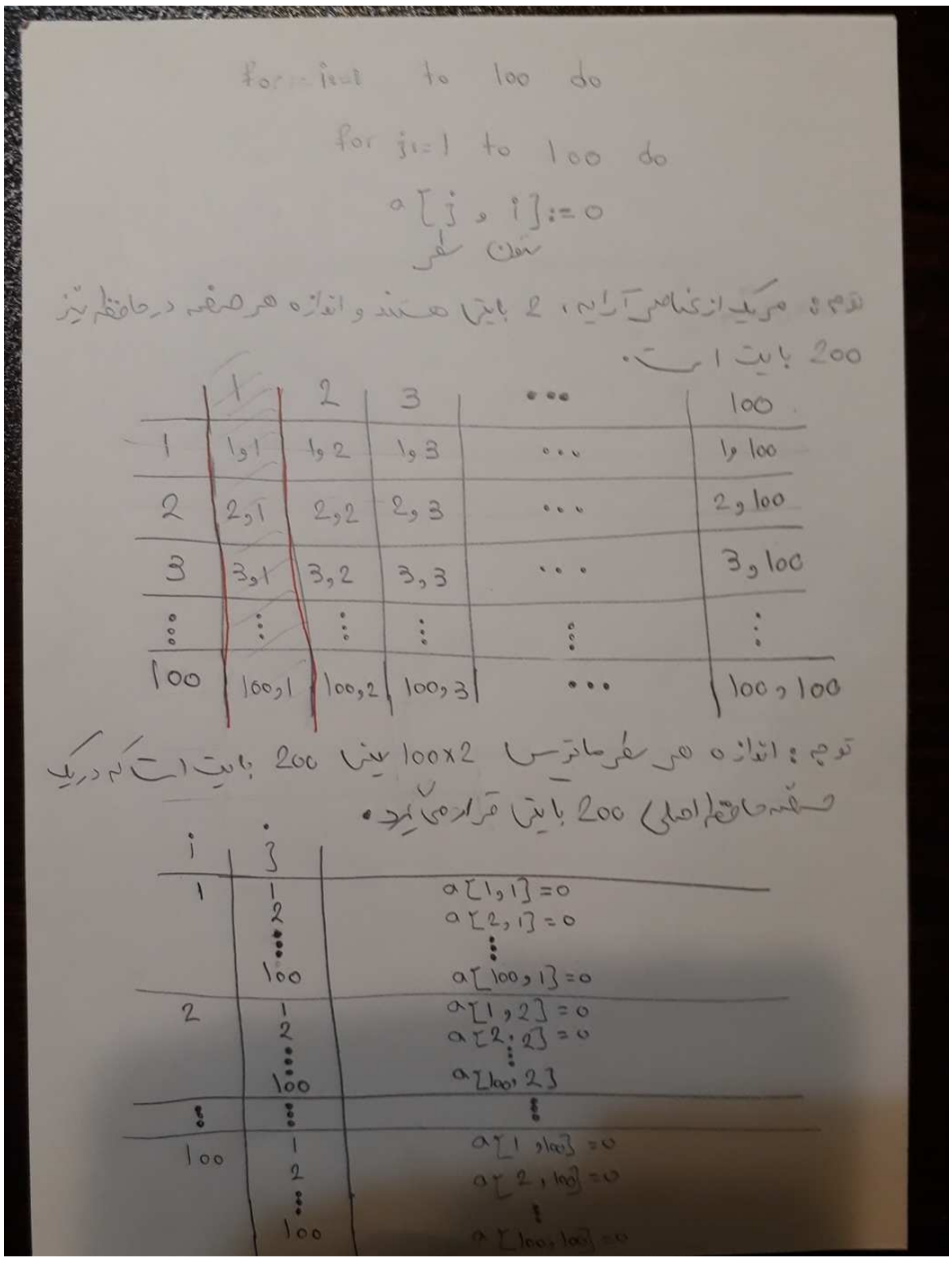
فرض کنید هریک از عناصر این آرایه، 2 بایتی هستند و اندازه هر صفحه در حافظه نیز 200 بایت باشد. در این صورت این آرایه 1000 عنصری، جمعاً 100 صفحه را اشغال می‌کند و از آنجا که عناصر آرایه در زبان‌های برنامه‌نویسی C و پاسکال به صورت سطری در حافظه ذخیره می شوند، در واقع هر سطر این آرایه در یک صفحه قرار می‌گیرد. یعنی در صفحه اول، عناصر a[1,1] تا a[1,100]، در صفحه دوم عناصر a[2,1] تا a[2,100]، تا صفحه صدم که عناصر a[100,1] تا a[100,100] قرار می‌گیرند.

با دقت در قطعه برنامه نوشته شده، مشاهده می‌شود که پردازش به صورت ستونی انجام می شود زیرا از اندیس حلقه بیرونی به عنوان اندیس ستون در آرایه استفاده کرده است (اندیس i). به این ترتیب اگر فقط یک صفحه برای داده‌ها در اختیار این برنامه باشد، برای انجام عملیات خود، دقیقاً 10000 خطای نقص صفحه رخ می‌دهد. زیرا هنگامی که یک صفحه به حافظه آورده شد، فقط یکی از عناصر آن پردازش می‌شود و برای عنصر بعدی یک خطای نقص صفحه رخ می‌دهد،

زیرا این عنصر در صفحه بعدی قرار دارد.

به این ترتیب به ازای هر عنصر، یک خطای نقص صفحه رخ می‌دهد.

شکل زیر گویای مطلب است:

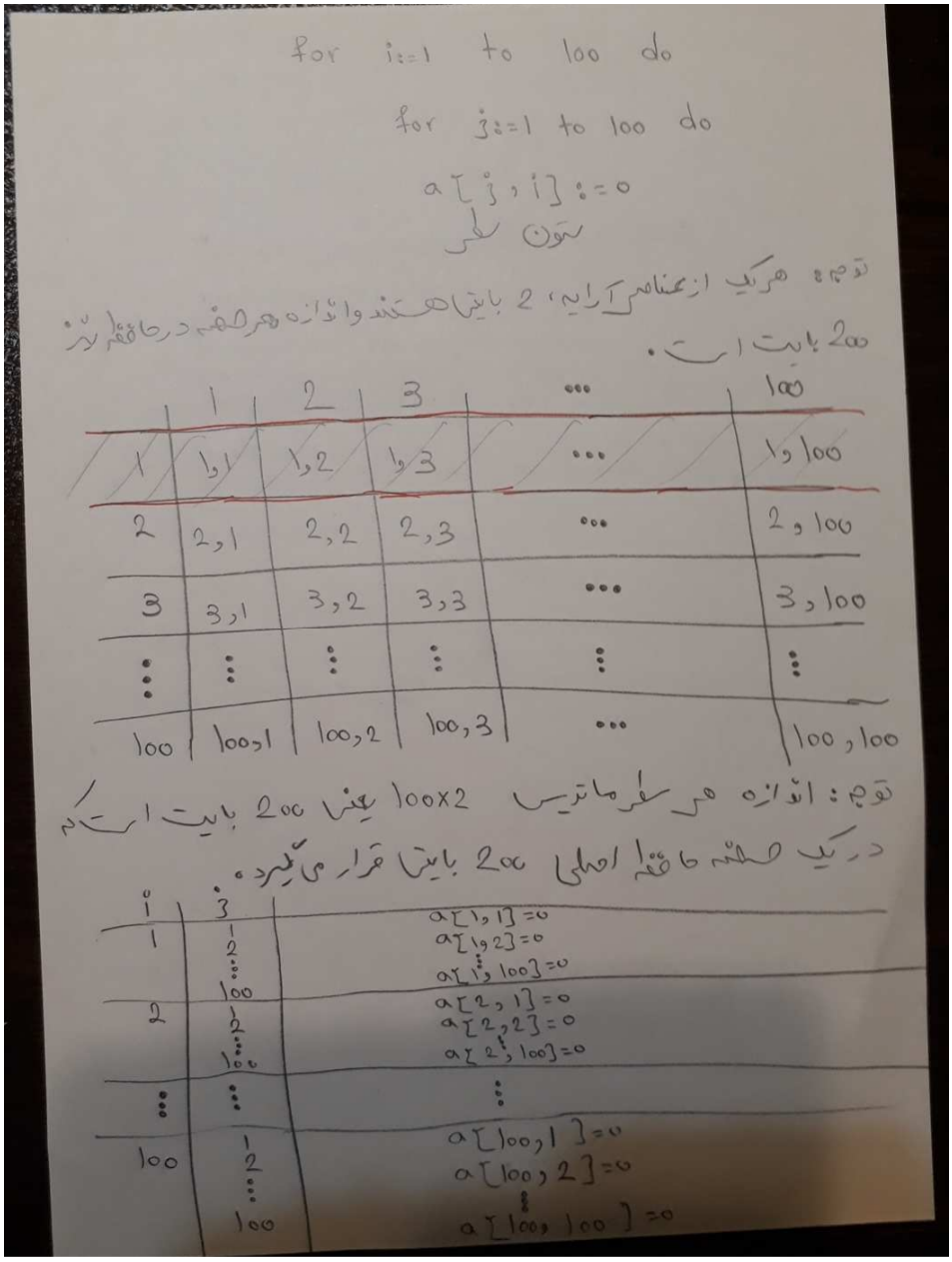


اما اگر برنامه به صورت زیر نوشته شود، شرایط تغییر می‌کند:

```
for i := 1 to 100 do
    for j := 1 to 100 do
        a[i, j] := 0;
```

در این حالت هنگامی که یک صفحه به حافظه آورده می‌شود، هر 100 عنصر آن به ترتیب پردازش می‌شوند و نیازی به نقص صفحه نیست. در واقع جمعاً 100 نقص صفحه رخ می‌دهد به ازای 100 سطر آرایه.

شکل زیر گویای مطلب است:



بنابراین همانطور که گفتیم اگر محلی بودن ارجاع‌ها در پردازه بیشتر شود، آنگاه فرکانس نقص صفحه (page fault frequency) کاهش می‌یابد. زیرا هر صفحه‌ای که به حافظه اصلی آورده می‌شود، به دلیل محلی بودن ارجاع‌ها اطلاعات زیادی از همان صفحه قابل استخراج است که

نتیجه آن می‌شود نقص صفحه کمتر و عدم مراجعه مکرر به صفحات بعدی که نتیجـه آن افـزایش نقص صفحه باشد. مثال اخیر محلیت مکانی (Spatial Locality) خوب بود.

تست‌های فصل اول

۹۸- عامل اصلی برای استفاده سیستم عامل از وقفه کدام است؟

(مهندسی *IT* – دولتی ۹۸)

۱) افزایش بهره‌وری
۲) سادگی در پیاده‌سازی
۳) کاهش زمان ارتباطات
۴) جلوگیری از اتلاف در IO

پاسخ‌های فصل اول

۹۸- گزینه (۱) صحیح است.

به طور کلی برای تشخیص اتمام انتقال داده‌ها دو روش در کامپیوترها قابل استفاده است. یکی روش سرکشی یا نمونه‌برداری (Polling) و دیگری روش وقفه (Interupt) است.

در روش سرکشی (Polling) هر دستگاه دارای ثباتی کنترلی است که CPU با بررسی آن می‌تواند متوجه شود که آیا عملیات انتقال آن دستگاه تمام شده است یا خیر. لذا در این تکنیک CPU می‌بایست در یک پریود زمانی، مرتبا ثبات‌های کنترلی دستگاه‌های مختلف را به صورت سرکشی بررسی کند که باعث اتلاف وقت CPU و به تبع کاهش بهره‌وری CPU است. با آنکه پیاده‌سازی این روش ساده است و به امکانات سخت‌افزاری خاصی نیاز ندارد ولی سرعت آن پایین است. این روش مانند کلاس درسی است که در آن استاد هر چند دقیقه یکبار از تک تک دانشجویان به ترتیب بپرسد که آیا سوالی دارند یا خیر، همچنین روش سرکشی باعث اتلاف وقت دستگاه‌های IO و به تبع کاهش بهره‌وری دستگاه‌های IO می‌شود، چون برای مثال اگر کار IO یک دستگاه ورودی و خروجی تمام شود باید آنقدر معطل بماند تا نوبت سرکشی آن توسط CPU فرا برسد. اما در روش وقفه (Interupt) هر دستگاه دارای سیگنال کنترلی مخصوص به خود است که به نحوی به CPU ارتباط دارد. هرگاه انتقال داده‌ها توسط آن دستگاه تمام شود، سیگنالی را به نام وقفه به سمت CPU می‌فرستد تا آن را از این موضوع مطلع سازد. در این حالت پردازنده، پردازش جاری را متوقف ساخته و به سرویس‌دهی این وقفه می‌پردازد. این روش مانند کلاس درسی است که در آن هر دانشجویی که سوال دارد در هر زمان با بالا بردن دست خود این موضوع را به اطلاع استاد می‌رساند. در این حالت استاد در اولین زمان مناسب صحبت‌های جاری خود را قطع کرده و پاسخ آن دانشجو را می‌دهد. سرعت این روش از روش سرکشی بیشتر است. همچنین روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود. تمام کامپیوترها راهکاری را فراهم می‌کنند تا قسمت‌های مختلف کامپیوتر (مانند ورودی و خروجی) روند اجرای دستورالعمل‌ها توسط پردازنده را جهت سرویس‌دهی‌های دیگری قطع کنند. در واقع مکانیزمی را فراهم می‌کند تا اجرای دستورالعمل‌های جاری پردازنده موقتاً متوقف شده و دستورات سرویس‌دهی دیگری اجرا شوند، پس از آن دوباره کنترل به همان برنامه باز می‌گردد. سیستم عامل جهت تسریع در پاسخ دادن به وقفه‌ها، آدرس روال‌های سرویس‌دهنده به آنها را در جدولی به نام جدول توصیف وقفه **(Interrupt Descriptor Table)** نگهداری می‌کند که به ازای هر وقفه یک درایه در این جدول وجود دارد که به آن بردار وقفه **(Interrupt Vector)** می‌گویند.

صورت سوال به این شکل است:

عامل اصلی برای استفاده سیستم عامل از وقفه کدام است؟

۱) افزایش بهره‌وری

گزینه اول پاسخ سوال است، زیرا عامل اصلی برای استفاده سیستم عامل از وقفه، افزایش بهره‌وری است. روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود.

۲) سادگی در پیاده‌سازی

گزینه دوم پاسخ سوال نیست، زیرا روش سرکشی پیاده‌سازی ساده‌تری نسبت به روش وقفه دارد.

۳) کاهش زمان ارتباطات

گزینه سوم پاسخ سوال نیست، زیرا کاهش زمان ارتباطات مرتبط با بحث شبکه‌های کامپیوتری یعنی کاهش زمان تاخیر پردازش(T_F)، کاهش زمان تاخیر انتشار(T_{Prob})، کاهش زمان تاخیر صف (T_{queue}) و کاهش زمان تاخیر پردازش ($T_{Process}$) است.

۴) جلوگیری از اتلاف در IO

گزینه چهارم پاسخ سوال نیست، زیرا روش وقفه باعث افزایش بهره‌وری CPU و دستگاه‌های IO می‌شود، اما باعث جلوگیری 100 درصد از اتلاف و به تبع بهره‌وری 100 درصد در CPU و دستگاه‌های IO نمی‌شود.

تست‌های فصل دوم

۹۹- در زمان‌بند غیرقبضه‌ای «بعدی-بیشترین-نسبت-زمان پاسخ» (HRRN) پردازه‌ای برای اجرا انتخاب می‌شود که بیشترین نسبت $\text{Ratio} = \frac{\text{waiting time}}{\text{CPU Burst Time}}$ را داشته باشد. چنانچه 4 پردازه به صورت جدول زیر داشته باشیم، در مقایسه با زمان‌بند غیرقبضه‌ای «اول-کمترین-زمان» (Shortest Job First) کدام مورد در خصوص متوسط زمان پاسخ درست است؟

(مهندسی IT – دولتی ۹۸)

پردازه	زمان ورود	زمان مورد استفاده از CPU (CPU Burst Time)
P_1	0	8
P_2	1	9
P_3	2	5
P_4	3	4

۱) متوسط زمان انتظار روش HRRN برابر روش SJF است.
۲) متوسط زمان انتظار روش HRRN کمتر از روش SJF است.
۳) متوسط زمان انتظار روش HRRN بیشتر از روش SJF است.
۴) متوسط زمان انتظار روش HRRN قابل محاسبه نیست.

پاسخ‌های فصل دوم

۹۹- گزینه (۱) صحیح است.

الگوریتم SJF (Shortest Job First)

در این روش ابتدا کاری برای اجرا انتخاب می‌شود که از همه کوتاهتر باشد (زمان اجـرای کمتـری داشته باشد).

توجه: این الگوریتم، **SPN** (Shortest Process Next) و

SPT (Shortest Process Time) نیز نامیده می‌شود.

توجه: SJF یک الگوریتم **انحصاری** (Non Preemptive) است. در سایر متون فارسی به الگـوریتم انحصاری، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یـا «**غیرقابل تخلیه پیش هنگـام**» نیـز گفته می‌شود.

توجه: یک نقص عمده الگوریتمSJF این است که ممکـن اسـت باعـث قحطـی‌زدگـی فرآینـدهای طولانی شود. به این ترتیب که اگر همواره تعـدادی فرآینـد کوچـک وارد سیسـتم شـوند، اجـرای فرآیندهای بزرگ به طور متناوب به تعویق می‌افتد. این روال حتی می‌تواند تا بینهایت ادامه یابـد و هیچگاه نوبت به فرآیندهای بزرگ نرسد!!!!!

توجه: در این روش اگـر دو فرآینـد مـدت زمـان اجـرای برابـر داشـته باشـند، بـر اسـاس FCFS زمان‌بندی می‌شوند.

توجه: هدف الگوریتم SJF به حداقل رساندن میانگین زمان انتظار، میانگین زمان پاسـخ و میـانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآینـدها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجـرای فرآینـدها را فقـط حدس زده و به طور تقریبی بدست آورد.

الگوریتم SRT (Shortest Remaining Time)

این الگوریتم نسخه **غیرانحصاری** (Preemptive) الگوریتم SJF است. در سایر متون فارسی به الگوریتم غیرانحصاری، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

در این الگوریتم اگر حین اجرای یک فرآیند، فرآیندی وارد شود که زمان اجرای کوتاه‌تری داشته باشد، پردازنده را در اختیار می‌گیرد.

توجه: این الگوریتم، **SRPT** (Shortest Remaining Processing Time)،

SRTF (Shortest Remaining Time First) و

SRTN (Shortest Remaining Time Next) نیز نامیده می‌شود.

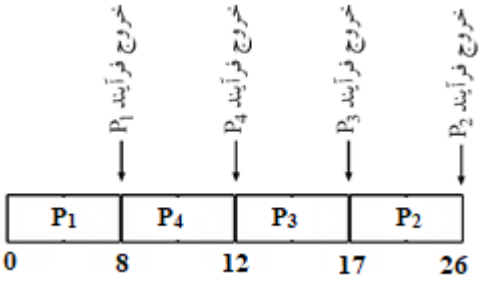
توجه: اگر لحظه ورود همه فرآیندها یکی باشد، الگوریتمSRT مشابه SJF عمل می‌کند.

توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	0	8		
P_2	1	9		
P_3	2	5		
P_4	3	4		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:



توجه: در الگوریتم‌های انحصاری HRRN و SJF لحظه خروج اول یک فرآیند با لحظه خروج کامل یک فرآیند برابر است، بنابراین در حل این سوال زمان پاسخ و زمان بازگشت یکسان در نظر گرفته شده است.

زمان ورود فرآیند – زمان خروج اول فرآیند = زمان پاسخ فرآیند

زمان ورود فرآیند – زمان خروج کامل فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 8 - 0 = 8$

P_2 زمان بازگشت $= 26 - 1 = 25$

P_3 زمان بازگشت $= 17 - 2 = 15$

P_4 زمان بازگشت $= 12 - 3 = 9$

میانگین زمان بازگشت $= \text{ATT} = \frac{8+25+15+9}{4} = \frac{57}{4} = 14.25$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 8 - 8 = 0$

P_2 زمان انتظار $= 25 - 9 = 16$

P_3 زمان انتظار $= 15 - 5 = 10$

P_4 زمان انتظار $= 9 - 4 = 5$

میانگین زمان انتظار $= \text{AWT} = \frac{0+16+10+5}{4} = \frac{31}{4} = 7.75$

میانگین زمان اجرا $= \text{AST} = \frac{8+9+5+4}{4} = \frac{26}{4} = 6.5$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

14.25= 6.5 + 7.75

توجه: مطابق رابطه فوق، تفاضل میانگین زمان بازگشت و میانگین زمان انتظار باید برابر میانگین زمان اجرا باشد.

توجه: همچنین مطابق رابطه فوق، میانگین زمان بازگشت همواره از میانگین زمان انتظار بیشتر است.

با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	8	0	8
P_2	1	9	16	25
P_3	2	5	10	15
P_4	3	4	5	9

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

14.25 7.75 6.5

در الگوریتم HRRN

برای تعیین اولویت یک فرآیند در الگوریتم HRRN از فرمول زیر استفاده می‌شود:

$$\text{اولویت} = \frac{\text{زمان انتظار + زمان اجرا}}{\text{زمان اجرا}} = \frac{\text{زمان انتظار}}{\text{زمان اجرا}} + 1$$

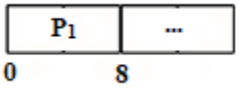
نکته: HRRN یک الگوریتم **انحصاری** است.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان پاسخ =
P_1	0	8		
P_2	1	9		
P_3	2	5		
P_4	3	4		

با توجه به مفروضات مساله، نمودار گانت زیر را داریم:

در این الگوریتم در زمان صفر فقط فرآیند P_1 قرار دارد که به شکل انحصاری اجرا می‌گردد.



در لحظه 8 فرآیندهای P_2، P_3 و P_4 در صف آماده قرار دارند. انتظار هر یک از فرآیندهای P_2، P_3 و P_4 تا لحظه 8 به صورت زیر است:

$P_2 = 8 - 1 = 7$ انتظار

$P_3 = 8 - 2 = 6$ انتظار

$P_4 = 8 - 3 = 5$ انتظار

در ادامه براساس رابطه اولویت داریم:

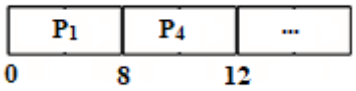
$$\text{اولویت } (P_2) = \frac{7}{9} + 1 = 1.7$$

$$\text{اولویت } (P_3) = \frac{6}{5} + 1 = 2.2$$

$$\text{اولویت } (P_4) = \frac{5}{4} + 1 = 2.25$$

$$\text{اولویت } (P_4) > \text{اولویت } (P_3) > \text{اولویت } (P_2)$$

واضح است که P_4 بالاترین اولویت را دارد، پس در زمان 8 فرآیند P_4 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.



در لحظه 12 فرآیندهای P_2 و P_3 در صف آماده قرار دارند. انتظار هر یک از فرآیندهای P_2 و P_3 تـا لحظه 12 به صورت زیر است:

انتظار $P_2 = 12 - 1 = 11$

انتظار $P_3 = 12 - 2 = 10$

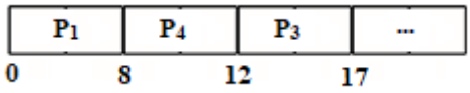
در ادامه براساس رابطه اولویت داریم:

$$\text{اولویت}(P_2) = \frac{11}{9} + 1 = 2.2$$

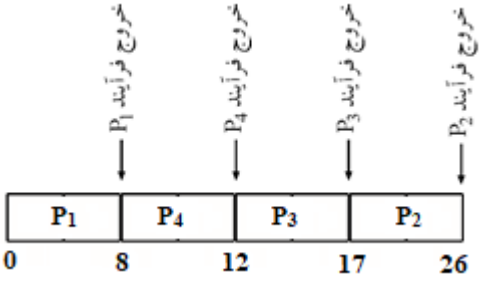
$$\text{اولویت}(P_3) = \frac{10}{5} + 1 = 3$$

$$\text{اولویت}(P_3) > \text{اولویت}(P_2)$$

واضح است که P_3 بالاترین اولویت را دارد، پس در زمان 12 فرآیند P_3 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.



در لحظه 17 فقط فرآیند P_2 در صف آماده قرار دارد. پس در زمان 17 فرآیند P_2 انتخاب می‌شود که به شکل انحصاری اجرا می‌گردد.



با توجه به اطلاعات به دست آمده، جدول قبل، به شکل زیر تکمیل می‌گردد:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت=
P_1	0	8	0	8
P_2	1	9	16	25
P_3	2	5	10	15
P_4	3	4	5	9

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

14.25 7.75 6.5

توجه: واضح است که زمان‌بندی‌های دو الگوریتم SJF و HRRN یکسان است، بنـابراین پُرواضـح است که گزینه اول پاسخ سوال است.

تست‌های فصل سوم

۱۰۰- کدام عبارت در مورد نخ‌ها درست نیست؟

(مهندسی *IT* – دولتی ۹۸)

۱) نخ‌های یک پردازه، دارای برنامه مخصوص به خود هستند.
۲) نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.
۳) نخ‌های یک پردازه، از فضای آدرس یکسان استفاده می‌کنند.
۴) نخ‌های یک پردازه، از یک پشته مشترک استفاده می‌کنند.

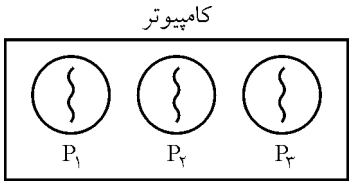
پاسخ‌های فصل سوم

۱۰۰- گزینه (۴) صحیح است.

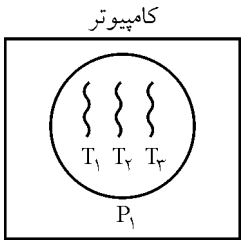
نخ (Thread)

در سیستم‌های قدیمی‌تر، به ازای هر فرآیند یک رشته نخ یا رشته اجرایی و به تبع یک شمارنده برنامه (PC) وجود داشت اما در سیستم عامل‌های امروزی به ازای هر فرآیند می‌توان چند نخ یا رشته اجرایی داشت.

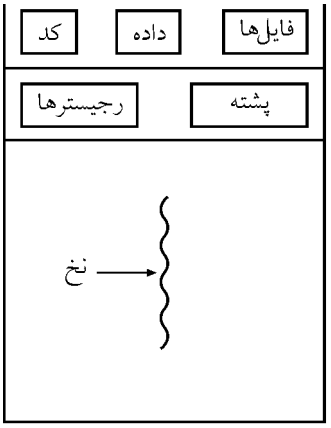
شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشته اجرایی و یک حافظه مختص به خود را دارند.



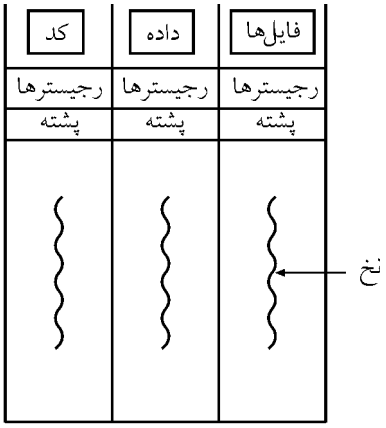
ولی در شکل زیر یک فرآیند، سه رشته اجرایی دارد که هر یک رجیستر، پشته و شمارنده برنامه (PC) مجزای خود را دارند و مانند فرآیندها می‌توانند همروند (در سیستم‌های تک‌پردازنده‌ای) و موازی (در سیستم‌های چندپردازنده‌ای) اجرا شوند.



توجه: نخ‌های همتا که در یک فرآیند قرار دارند و از کد، داده، heap و منابع مشترک استفاده می‌کنند اما هر نخ، شمارنده برنامه، مجموعه رجیستر و فضای پشته جداگانه‌ای در اختیار دارد. در واقع هر نخ، TCB مجزایی دارد.



فرآیند تک نخی



فرآیند چند نخی

توجه: از آن‌جا که نخ‌های همتا در یک فرآیند قرار داشته و اشتراکات زیادی با هم دارند، عمل تعویض متن بین آنها به راحتی و با هزینه کمتری صورت می‌گیرد، در واقع TCB مربوط به نخ‌ها، محتوی کمتری نسبت به PCB فرآیندها دارد، برای مثال لیست فایل‌های باز مربوط به فرآیندها است، بنابراین این لیست به هنگام تعویض متن فرآیندها باید داخل PCB مربوط به فرآیند ذخیره گردد، در حالی که به هنگام تعویض متن بین نخ‌ها نیازی به ذخیره‌سازی لیست فایل‌های باز مربوط به یک فرآیند در TCB یک نخ نیست. بنابراین تعویض متن بین نخ‌ها نسبت به فرآیندها ارزان‌تر است.

چند نخی در زبان #C

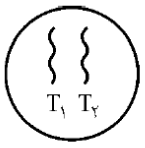
#C پیاده سازی چند نخی را پشتیبانی می‌کند. در زبان #C، هر برنامه به طور پیش فرض از یک نخ تشکیل شده است و درصورت ایجاد نخ‌های دیگر، مفهوم چندنخی پیاده‌سازی می‌گردد.

توجه: نخ اول به صورت پیش‌فرض وجود دارد و برنامه با نخ اول شروع به اجرا می‌کند.

مثال: در قطعه کد زیر نخ T_1 به طور پیش‌فرض وجود دارد و نخ T_2 ایجاد می‌گردد:

```
static void main ()
  {
Thread  T2=new  Thread (Go) ;
T2.Start( )  → نخ T۲ اجرا می‌گردد.
Go ( );  → فعالیت Go داخل نخ T۱ قرار داده شده است. می‌شود فعالیت دیگری تعریف شود و در این بخش فراخوانی شود.
  }
static void Go( )
{
for (int i = 0 ; i<5 ; i++)
      console.write("*");
 }
```

↓ در این بخش نخ T_2 ایجاد می‌گردد و فعالیت Go داخل این نخ قرار می‌گیرد.



P۱

توجه: فعالیت Go داخل نخ T_1 قرار دارد، اما فعالیت Go در نخ T_2 هم قرار داده شده است. در دو نخ T_1و T_2 که فعالیت Go داخل آن قرار دارد، متغیر محلی i در داخل پشته مربوط به هر نخ ایجاد می‌گردد.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

****

چاپ 10 عدد ستاره به دلیل اجرای همروند (سیستم تک‌پردازنده‌ای) یا موازی (سیستم چندپردازنده‌ای) دو نخ T_1 و T_2 است!

توجه: چاپ 10 عدد ستاره نشانه این است که هر نخ پشته مختص به خود را دارد.

توجه: نخ از دو بخش ساختاری (ظرف) و محتوایی تشکیل شده است، ساختار نخ در زبان C# توسط دستور new ایجاد می‌شود، و محتوای نخ توسط قرار دادن یک قطعه کد به درون آن پُر می‌شود. در مثال بالا دو ساختار و ظرف نخ T_1و T_2 توسط دستور new ایجاد شده است و قطعه کد Go به عنوان محتوای نخ T_1و T_2 در نظر گرفته شده است. بنابراین نخ‌های یک پردازه(فرآیند)، دارای برنامه(قطعه کد) مخصوص به خود هستند.

مثال: در قطعه کد زیر نخ T_1 به طور پیش‌فرض وجود دارد و نخ T_2 ایجاد می‌گردد:

```
static void main()
  {
    Thread  T2=new  Thread (func) ;
    T2.start( ) ;
    console.write("X");
  }
static void func( )
{
console.write("Y");
 }
```

در این بخش نخ T_2 ایجاد می‌گردد و فعالیت func داخل این نخ قرار می‌گیرد.

نخ T_2 اجرا می‌گردد.

فعالیت چاپ "X" داخل نخ T_1 قرار داده شده است.

بنابراین خروجی این برنامه به صورت زیر خواهد بود:

XY

توجه: نخ از دو بخش ساختاری (ظرف) و محتوایی تشکیل شده است، ساختار نخ در زبان C# توسط دستور new ایجاد می‌شود، و محتوای نخ توسط قرار دادن یک قطعه کد به درون آن پُر می‌شود. در مثال بالا دو ساختار و ظرف نخ T_1و T_2 توسط دستور new ایجاد شده است و قطعه کد write("X") به عنوان محتوای نخ T_1 و قطعه کد write("Y") به عنوان محتوای نخ T_2 در نظر گرفته شده است. بنابراین نخ‌های یک پردازه(فرآیند)، دارای برنامه(قطعه کد) مخصوص به خود هستند.

توجه: اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، heap، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد.

صورت سوال به این شکل است:

کدام عبارت در مورد نخ‌ها درست نیست؟

۱) نخ‌های یک پردازه، دارای برنامه مخصوص به خود هستند.

گزینه اول گزاره درستی است، زیرا نخ‌های یک پردازه(فرآیند)، دارای برنامه(قطعه کد) مخصوص به خود هستند.

۲) نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.

گزینه دوم گزاره درستی است، زیرا نخ‌های یک پردازه، از فضای heap مشترک استفاده می‌کنند.

۳) نخ‌های یک پردازه، از فضای آدرس یکسان استفاده می‌کنند.

گزینه سوم گزاره درستی است، زیرا نخ‌های یک پردازه، از فضای آدرس یکسان استفاده می‌کنند. کل ساختار و محتوای یک فرآیند داخل یک فضای آدرس قرار می‌گیرد، نخ یک مفهوم برنامه‌نویسی است که بخشی از یک فرآیند محسوب می‌شود.

۴) نخ‌های یک پردازه، از یک پشته مشترک استفاده می‌کنند.

گزینه چهارم گزاره درستی نیست، زیرا نخ‌های یک پردازه، از یک پشته مشترک استفاده نمی‌کنند. هر نخ پشته مختص به خود را دارد.

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

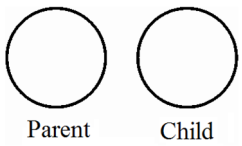
پدر در PCB فرآیند فرزند کپی می‌شـــود و تنها چیزی که در محتوای PCB فرآیند پدر و PCB فرآیند فرزند تفاوت دارد، مقدار pid است چون هر فرآیند pid مختص به خودش را دارد، در واقع pid فرآیند پدر با pid فرآیند فرزند متفاوت اســـت. همچنین دقت کنید که فضـــای آدرس PCB فرآیند پدر از فضای آدرس PCB فرآیند فرزند مستقل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

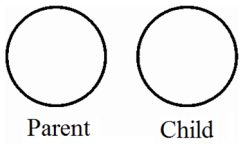
۱- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد اســتفاده قرار نگیرد، یعنی محتوای فرآیند پدر و فرآیند فرزند بســته به شــرایط، **فقط در حد تغییر مقدار متغیرها و نه تغییر 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.

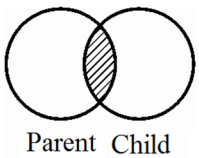
۲- اگر فضای آدرس فرآیند پدر و فرآیند فرزند **مستقل** باشد، یعنی تکنیک Copy-On-Write مورد استفاده قرار نگیرد و همچنین دستور exec به معنی ساخت محتوای کاملا جدید در فرآیند فرزند مورد استفاده قرار بگیرد و یک برنامه و قطعه کد جدید به درون خود بار کند، یعنی محتوای فرآیند فرزند بسته به شرایط، **در حد تغییر 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:



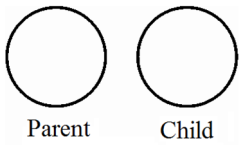
توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه مشترک، برای **فضای آدرس مستقل** برقرار است.

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:

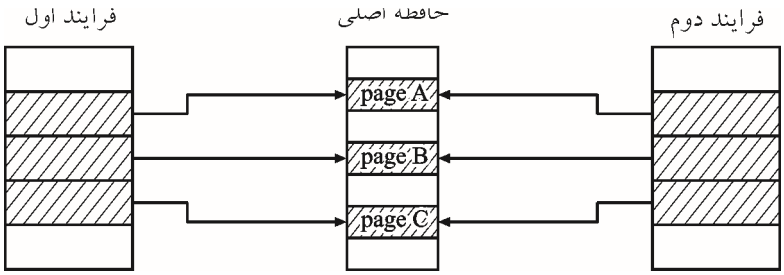


توجه: بنابراین رابطه $card(Parent \cap Child) = 0$ به معنی 0 صفحه م شترک، برای **ف ضای آدرس مستقل** برقرار است.

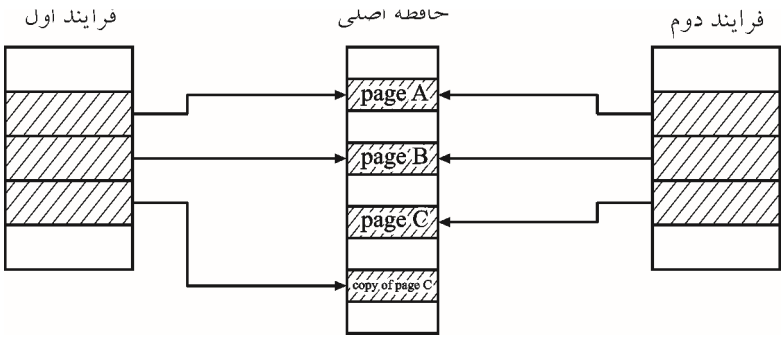
توجه: در ســیســـتم عامل Unix یک فرآیند فرزند پس از ایجاد توســط فراخوانی fork، می‌تواند بلافاصله از فراخوانی سیستمی ()exec استفاده کند و کل فضای آدرس حافظه‌ی خود را جایگزین کند و از ادامه شراکت با پدر صرفه نظر کند. بنابراین، با این کار، حافظه‌ی جداگانه‌ای برای فرآیند فرزند ایجاد می‌شـــود. با توجه به 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.

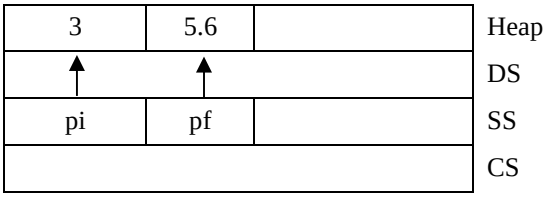
توجه: ppid نشان دهنده pid فرآیند پدر است. بنابراین شماره ppid فرآیند فرزند، برابر pid فرآیند پدر است.

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.

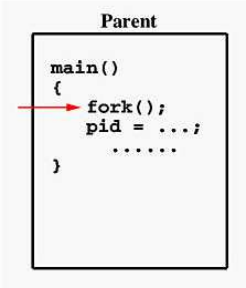
مثال: برنامه‌ی زیر یک قطعه کد به زبان C ا ست که مقدار pid را تو سط د ستور printf داخل یک حلقه‌ی for پس از اجرای دسـتور fork در خروجی نمایش می‌دهد: (فرض کنید pid واقعی فرآیند پدر برابر مقدار 2600 و pid واقعی فرآیند فرزند برابر مقدار 2603 باشد.)

```
//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;
}
```

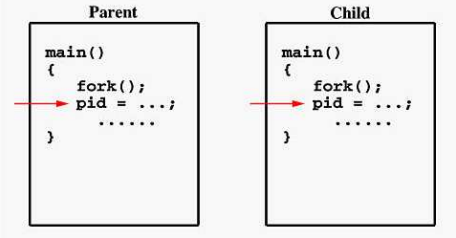
توجه: برنامه اجرا می‌شـــود و در اولین خط مقدار متغیر محلی i برابر با 0 می‌شـــود. دقت کنید که متغیر i داخل تابع main تعریف شده ا ست و یک متغیر محلی محسوب می شود که داخل Stack Segment تعریف و مقداردهی می‌شـود. در خط بعدی، دسـتور fork قرار دارد، وقتی که دسـتور fork اجرا می‌شـــود یک فرآیند فرزند (جدید)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

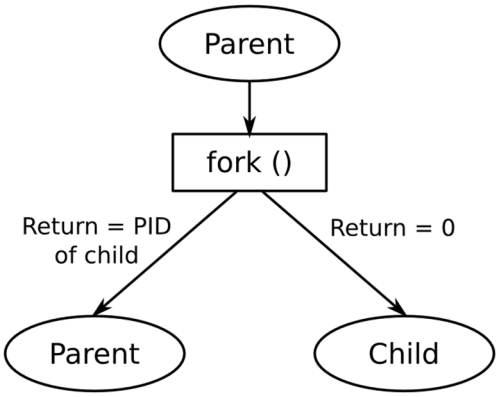
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع () getpid استفاده نماید.

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

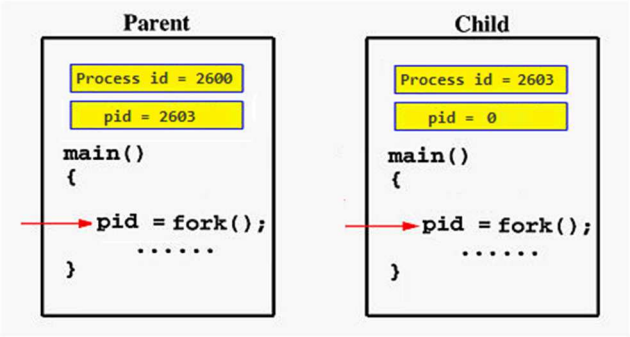
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

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 ***
```

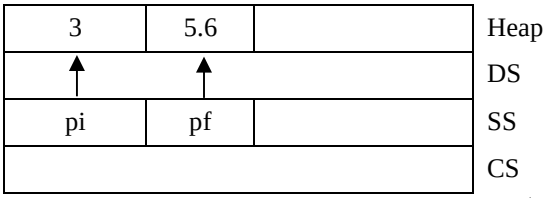
توجه: همانطور که گفتیم، مقادیر برگشتی فراخوانی fork بهتر است با یک شرط if کنترل شود تا مسیر اجرای فرآیند پدر و فرآیند فرزند مشخص و متمایز شود. در مثال فوق مسیر فرآیند پدر و فرزند به دلیل وجود شرط if از هم متمایز شده بود.

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**

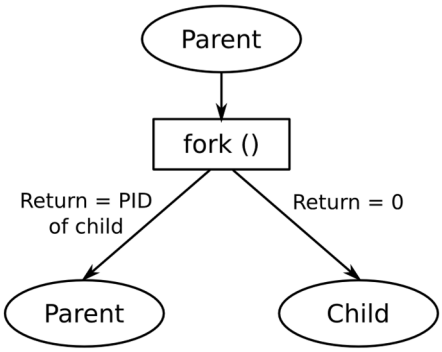
توجه: یک فرآیند فرزند می‌تواند جهت بازیابی مقدار process id یا همان pid منتسب شده به خودش، از تابع getpid () استفاده نماید.

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

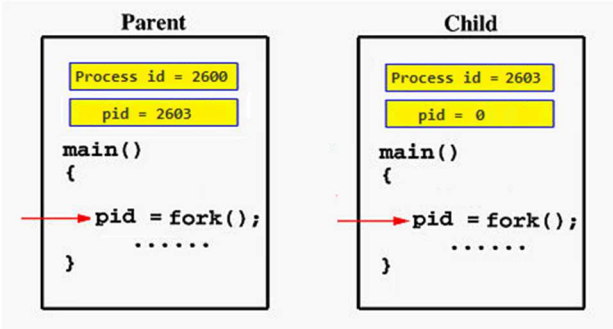
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند برابر مقدار صفر است که به آن پاس داده می‌شود. عدد صفر پاس داده شده به فرآیند فرزند به این معنی است که فرآیند فرزند، فعلا هیچ فرزندی ندارد.

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 ());
```

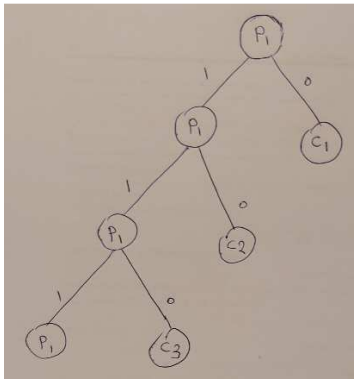
توجه: مقدار بازگشتی fork در حالت اجرای موفق برای فرآیند فرزند سوم برابر مقدار صفر ا ست که به آن پاس داده می‌شود.

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:

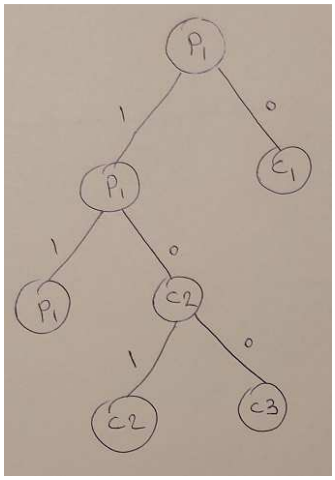
```
0
printf ("Process id = %d \n", getpid ());
```

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:

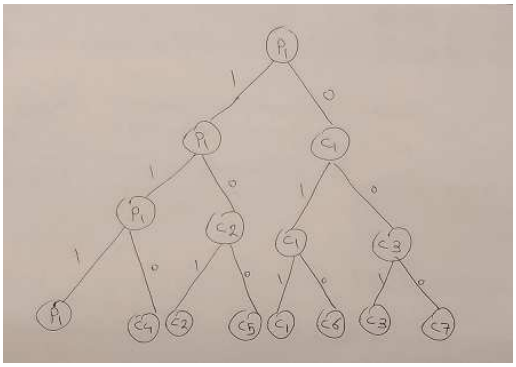
0
printf ("Process id = %d \n", getpid ());

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:

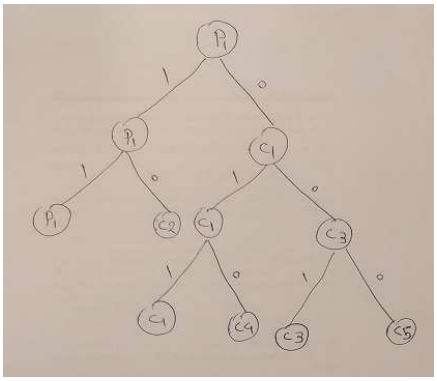
0
printf ("Process id = %d \n", getpid ())

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 ***
```