موسسه بابان

انتشارات بابان و انتشارات راهیان ارشد

درس و کنکور ارشد

سیستم عامل

(حل تشریحی سوالات دولتی ۱۳۹۹)

ویژه‌ی داوطلبان کنکور کارشناسی ارشد مهندسی کامپیوتر و IT

براساس کتب مرجع

آبراهام سیلبرشاتز، ویلیام استالینگز و اندور اس تننباوم

ارسطو خلیلی‌فر

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۹- الگوریتم زیر ساختار فرآیند P_i برای حل مسئله ناحیه بحرانی (Critical-Problem) در حالتی که n فرآیند وجود داشته باشد، است. در خصوص این الگوریتم کدام گزینه صحیح است؟ **(مهندسی کامپیوتر - دولتی ۹۹)**

```
do{
waiting[i] = true;
key = true;
while(waiting[i] && key)
   key = test_and_set(&lock);
waiting[i] = false;
/* Critical Section */
j = (i+1)  % n;
while (!waiting[j])
  j = (i+1) % n;
if (j == i)
   lock = false;
/* Remailder Section */
} while (true);
```

۱) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را به ازای مقدار دلخواه n برآورده می‌کند.

۲) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را تنها به ازای مقدار n=2 برآورده می‌کند.

۳) شرط پیشرفت را تنها به ازای مقدار n بزرگتر از 2 برآورده نمی‌کند.

۴) شرط پیشرفت را هرگز برآورده نمی‌کند.

عنوان کتاب: سیستم عامل

مولف: ارسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۹- گزینه (۴) 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 ناحیه‌ی 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) waiting [i]= TRUE;
(2) key = TRUE;
(3) while (waiting [i] && key)
(4) key = test_and_set (&lock);
(5)  waiting [i] = FALSE;
/*critical_section*/
(6) j = (i + 1) % n;
(7) while (!waiting [i])
(8) j = (j + 1) % n;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
 }
```

```
P1:
 {
(1) waiting [i]= TRUE;
(2) key = TRUE;
(3) while (waiting [i] && key)
(4) key = test_and_set (&lock);
(5)  waiting [i] = FALSE;
/*critical_section*/
(6) j = (i + 1) % n;
(7) while (!waiting [i])
(8) j = (j + 1) % n;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
 }
```

توجه: عملگر % که به mod معروف است، باقی مانده تقسیم را در خروجی قرار می‌دهد.

توجه: معادل عبارت j = (i + 1) % n در خط (6) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P0 به صورت زیر است:

```
j = (i + 1) % 2
j = (0 + 1) % 2
j = 1
```

توجه: معادل عبارت j = (j + 1) % n در خط (8) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P0 به صورت زیر است:

```
j = (j + 1) % 2
j = (1 + 1) % 2
```

j = 0

توجه: معادل عبارت j = (i + 1) % n در خط (6) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P1 به صورت زیر است:

j = (i + 1) % 2
j = (1 + 1) % 2
j = 0

توجه: معادل عبارت j = (j + 1) % n در خط (8) اگر n برابر 2 یا بیشتر فرآیند و i اندیس فرآیند جاری باشد، برای فرآیند P1 به صورت زیر است:

j = (j + 1) % 2
j = (0 + 1) % 2
j = 1

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

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) waiting [0]= TRUE;
(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

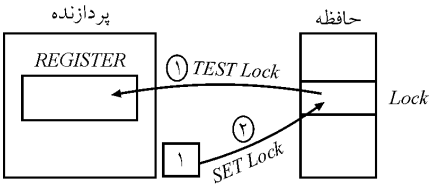
توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: بسیاری از پردازنده‌ها، دستورالعمل دو بخشی اما اتمیک خاصی دارند به نام TSL (TEST and SET Lock)، بدین نحو که این دستورالعمل به شکل تجزیه ناپذیر، عملیات زیر را انجام می‌دهد:

- ابتدا محتویات یک کلمه از حافظه به نام Lock را می‌خواند و مقدار آن را در رجیستر قرار می‌دهد.(TEST Lock)
- سپس مقدار یک را در متغیر Lock قرار می‌دهد. (SET Lock)

به شکل زیر توجه کنید:



سخت‌افزار تضمین می‌کندکه دو عامل خواندن و مقداردهی متغیر قفل به صورت اتمیک (تجزیه ناپذیر) انجام شود. بدین شکل که هیچ فرآیند و حتی پردازنده‌ی دیگر نتواند به این متغیر قفل دسترسی پیدا کند تا وقتی که اجرای دستورالعمل به پایان برسد. پردازنده‌ای که دستورالعمل TSL را اجرا می‌کند، گذرگاه حافظه را قفل می‌کند تا از دسترسی دیگر پردازنده‌ها به حافظه جلوگیری کند تا اینکه این دستورالعمل به پایان برسد.

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*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 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = TRUE است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

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 شـرط 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 تلفـن همگـانی بشـه اونوقت شرط اول انحصار متقابل نقض شده. اخلاق می‌گه اگه یه نفر داخل باجـه تلفـن همگـانی 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 سیسـتم تـک پردازنـده‌ای و یـا مـوازی در سیسـتم چنـد پردازنده‌ای حرکت بدید اگر هر دو باهم توانستند وارد ناحیه بحرانی خودشان شوند، آنگاه در ایـن حالت شرط انحصار متقابل نقض شده است.

```
P0:
 {
(1) waiting [0]= TRUE;
(2) key = TRUE;
(3) while (waiting [0] && key)
(4) key = test_and_set (&lock);
(5)  waiting [0] = FALSE;
/*critical_section*/
(6) j = 1;
(7) while (!waiting [1])
(8) j = 0;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
 }
```

```
P1:
 {
(1) waiting [1]= TRUE;
(2) key = TRUE;
(3) while (waiting [1] && key)
(4) key = test_and_set (&lock);
(5)  waiting [1] = FALSE;
/*critical_section*/
(6) j = 0;
(7) while (!waiting [j])
(8) j = 1;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
 }
```

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

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

```
P0:
(1) waiting [0]= TRUE;
(2) key = TRUE;
```

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآیند P_0 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود به ناحیه‌ی بحرانی را دارند. بنابراین آرایه فرآیندها هر دو waiting [0]= TRUE و waiting [1]= TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند. مقدار اولیه متغیر lock برابر FALSE است، هر فرآیندی که زودتر دستور key = test_and_set (&lock) را اجرا کند، داخل ناحیه بحرانی می‌شود و فرآیندی که دیرتر دستور key = test_and_set (&lock) را اجرا کند، باید شرط حلقه را چک کند.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقدام به اجرای دستور test_and_set (&lock) کنند، بنابراین فرآیند P_0 وارد ناحیه بحرانی می‌شود، اما فرآیند P_1 باید شرط حلقه را چک کند. به صورت زیر:

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

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

توجه: هم اکنون waiting [1] = TRUE و key = FALSE و lock = TRUE است.

توجه: در اجرای هم‌روند قبلا از دو خط اول فرآیند P1 عبور کردیم و key = TRUE قبلا اجرا شده است و مجدد اجرا نمی‌شود، پس در حال حاضر مقدار key برابر FALSE است که قبلا در فرآیند P0 مقداردهی شده است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآینـد P_1 قرار می‌گیرد، به صورت زیر:

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

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 هـر دو بـاهم توانسـتند وارد ناحیـه بحرانـی خودشان شوند، آنگاه در این حالت شرط انحصار متقابل نقض شده است، خب هر دو باهم موفـق شدند. فرآیند اول و دوم نتوانستند هر دو باهم وارد ناحیه بحرانی خودشان بشوند. بنـابراین شـرط دوم انحصار متقابل برقرار نیست.

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 هـر دو تونستن به طور همزمان وارد باجه تلفن همگانی بشن اونوقت شرط دوم انحصار متقابل نقـض شده. اخلاق می‌گه دو نفر نباید همزمان باهم داخل باجه تلفن همگانی باشـن، یعنـی اگـه یـه نفـر داخل باجه تلفن همگانی هست نفر دیگه‌ای نباید وارد باجه تلفن همگانی بشه و اگـه بشـه شـرط 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 نیست.

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) waiting [0]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:
(5) waiting [0] = FALSE;
/*critical_section*/

همانطور که در (گام ۱) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قـرار بـدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۲) می‌شویم. هـم اکنـون پردازنـده در ناحیـه بحرانی فرآیند P0 مشغول حرکت است.

توجه: هم اکنون waiting [1] = FALSE و key = FALSE و lock = TRUE 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*/
(6) j = 1;
(7) while (!waiting [1])

توجه: هم اکنون خروجی حلقه while (!FALSE) برابر TRUE است. پس بدنه حلقه یعنی دستور j = 0 اجرا می‌شود.

(8) j = 0;
(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (0 = = 0) برابر TRUE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا می‌شود.

(10) lock = FALSE;

حال در ادامه فرآیند P0 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانـده 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 قـرار 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 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [1] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:

(5) waiting [1] = FALSE;

/*critical_section*/

همانطور که در (گام ۳) گفتیم قرار شد که فرآیند اول را داخل ناحیه بحرانی خودش قرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۴) می‌شویم. هم اکنون پردازنده در ناحیه بحرانی فرآیند P1 مشغول حرکت است.

توجه: هم اکنون waiting [0] = FALSE و key = FALSE و lock = TRUE است.

(گام ۴): همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قرار بده، یعنی:

فرض کنید فرآیند P1 از بخش خروج از ناحیه بحرانی خودش عبور کند، به صورت زیر:

P1:

/*critical_section*/

(6) j = 0;

(7) while (!waiting [0])

توجه: هم اکنون خروجی حلقه while (!FALSE) برابر TRUE است. پس بدنه حلقه یعنی دستور j = 1 اجرا می‌شود.

(8) j = 1;

(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (1 = = 1) برابر TRUE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا می‌شود.

(10) lock = FALSE;

حال در ادامه فرآیند P1 پس از عبور از بخش خروج از ناحیه بحرانی خودش در ناحیه باقی‌مانـده خودش قرار می‌گیرد، به صورت زیر:

P1:

/*remainder_section*/

همانطور که در (گام ۴) گفتیم قرار شد که همان فرآیند دوم را داخل ناحیه باقی‌مانده خودش قـرار بدهیم، خب قرار دادیم. حال در ادامه آزمون سوم وارد (گام ۵) می‌شویم. هم اکنون پردازنده داخل ناحیه باقی‌مانده فرآیند P1 مشغول حرکت است.

(گام ۵): فرآیند دوم به ابتدای برنامه برگردد و مجددا تصمیم بگیرد وارد ناحیـه بحرانـی خـودش 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. یعنی:

فرض کنید فرآیند P1 نیز قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [1] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

همانطور که در (گام ۵) گفتیم قرار شد که 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، ، خب شد، فرآیند دوم مجددا وارد ناحیه بحرانی خودش شد. بنابراین شرط پیشرفت برقرار است.

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) waiting [0]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:

(5) waiting [0] = FALSE;

/*critical_section*/

(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 قصد دارد وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_1:

(1) waiting [1]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = TRUE است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [1] = TRUE و key = TRUE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

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*/

(6) j = 1;

(7) while (!waiting [1])

توجه: هم اکنون خروجی حلقه while (!TRUE) برابر FALSE است. پس بدنه حلقه یعنی دستور j = 0 اجرا نمی‌شود.

(8) j = 0;

(9) if (j = = i)

توجه: هم اکنون خروجی دستور شرطی if (1 = = 0) برابر FALSE است. پس بدنه دستور شرطی یعنی دستور lock = FALSE اجرا نمی‌شود.

(10) lock = 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 P_0 قصد دارد مجددا وارد ناحیه بحرانی خودش شود، به صورت زیر:

P_0:

(1) waiting [0]= TRUE;

(2) key = TRUE;

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = TRUE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

شرط حلقه TRUE است، پس کنترل برنامه از حلقه خارج نشده و داخل ناحیه بحرانی فرآیند P0 قرار نمی‌گیرد، این حلقه مدام تکرار می‌شود و فرآیند P0 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 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، دقت کنید که در سناریوی فوق فرآیند P_0 از ناحیه بحرانی خارج شد، اما مقدار lock را نتوانست FALSE کند و مقدار lock در همان مقدار TRUE باقی ماند. فرآیند P1 از قبل در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود مانده است و می‌چرخد و در ادامه فرآیند P0 هم در یک حلقه انتظار مشغول پشت ناحیه بحرانی خود مانده است و این یعنی «**بن بست**» رخ داده است.

اگر 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. 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. خب هر دو باهم پشت ناحیه بحرانی خودشان مسدود شدند. فرآیند اول نتوانست وارد ناحیه بحرانی خودش شود، همچنین فرآیند دوم هم نتوانست وارد ناحیه بحرانی خودش بشود. بنابراین «**بن بست**» رخ داده است.

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 خـواب رفته‌اند که در این حالت بن‌بست رخ داده است.

```
P0:
 {
(1) waiting [0]= TRUE;
(2) key = TRUE;
(3) while (waiting [0] && key)
(4) key = test_and_set (&lock);
(5)  waiting [0] = FALSE;
/*critical_section*/
(6) j = 1;
(7) while (!waiting [1])
(8) j = 0;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
 }
```

```
P1:
 {
(1) waiting [1]= TRUE;
(2) key = TRUE;
(3) while (waiting [1] && key)
(4) key = test_and_set (&lock);
(5)  waiting [1] = FALSE;
/*critical_section*/
(6) j = 0;
(7) while (!waiting [j])
(8) j = 1;
(9) if (j == i)
(10) lock = FALSE;
/*remainder_section*/
}
```

توجه: مقادیر اولیه به صورت زیر است:

key =FALSE , lock = FALSE, waiting [0] = FALSE, waiting [1] = FALSE

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

```
P0:
(1) waiting [0]= TRUE;
(2) key = TRUE;
```

همچنین فرض کنید فرآیند P_1 نیز به شکل همروند یا موازی با فرآیند P_0 قصـد دارد وارد ناحیـه بحرانی خودش شود، به صورت زیر:

```
P2:
(1) waiting [1]= TRUE;
(2) key = TRUE;
```

زمانی که هر دو فرآیند به طور تقریباً همزمان، قصد ورود بـه ناحیـه‌ی بحرانـی را دارنـد. بنـابراین آرایه فرآیندها هر دو waiting [0]= TRUE و waiting [1]= TRUE می‌شود، زیرا هر دو فرآیند به شکل تقریباً همزمان علاقه‌مند به ورود به ناحیه‌ی بحرانی هستند.

دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند. مقدار اولیه متغیـر lock برابـر FALSE است، هر فرآیندی که زودتـر دسـتور key = test_and_set (&lock) را اجـرا کنـد، داخـل ناحیـه بحرانی می‌شود و فرآیندی که دیرتر دستور key = test_and_set (&lock) را اجرا کند، باید شـرط حلقه را چک کند.

فرض کنید، فرآیند P_0 **زودتر** و فرآیند P_1 **دیرتر** اقـدام بـه اجـرای دسـتور test_and_set (&lock) کنند، بنابراین فرآیند P_0 وارد ناحیه بحرانی می‌شود، اما فرآیند P_1 باید شرط حلقه را چک کند. بـه صورت زیر:

توجه: هم اکنون waiting [0] = TRUE و key = TRUE و lock = FALSE است.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && TRUE) برابر TRUE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا می‌شود.

(4) key = test_and_set (&lock);

توجه: مطابق آنچه برای TSL گفتیم، دستور key = test_and_set (&lock) ابتدا مقدار کنونی lock را می‌خواند و داخل متغییر key قرار می‌دهد، سپس مقدار lock را به TRUE مقداردهی می‌کند.

توجه: هم اکنون waiting [0] = TRUE و key = FALSE و lock = TRUE است.

توجه: مجدد حلقه اجرا می‌شود.

(3) while (waiting [0] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

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

P0:

(5) waiting [0] = FALSE;

/*critical_section*/

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

توجه: هم اکنون waiting [1] = TRUE و key = FALSE و lock = TRUE است.

توجه: در اجرای هم‌روند قبلا از دو خط اول فرآیند P1 عبور کردیم و key = TRUE قبلا اجرا شده است و مجدد اجرا نمی‌شود پس در حال حاضر مقدار key برابر FALSE است که قبلا در فرآیند P0 مقداردهی شده است.

(3) while (waiting [1] && key)

توجه: هم اکنون خروجی حلقه while (TRUE && FALSE) برابر FALSE است. پس بدنه حلقه یعنی دستور key = test_and_set (&lock) اجرا نمی‌شود.

شرط حلقه FALSE است، پس کنترل برنامه از حلقه خارج شده و داخل ناحیه بحرانی فرآیند P_1 قرار می‌گیرد، به صورت زیر:

P1:
(5) waiting [1] = FALSE;
/*critical_section*/

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 و دوم توانستند هر دو باهم وارد ناحیه بحرانی خودشان بشوند. 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 به دلیل وجود بن بست در سناریوی گرسنگی برقرار نیست.

صورت سوال به این شکل است:

الگوریتم زیر ساختار فرآیند P_i برای حل مسئله ناحیه بحرانی (Critical-Problem) در حالتی که n فرآیند وجود داشته باشد، است. در خصوص این الگوریتم کدام گزینه صحصیح است؟

۱) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را به ازای مقدار دلخواه n برآورده می‌کند.

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

۲) سه شرط مسئله ناحیه بحرانی (انحصار متقابل، پیشرفت، انتظار محدود) را تنها به ازای مقدار n=2 برآورده می‌کند.

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

۳) شرط پیشرفت را تنها به ازای مقدار n بزرگتر از 2 برآورده نمی‌کند.

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

۴) شرط پیشرفت را هرگز برآورده نمی‌کند.

گزینه چهارم **نادرست** است، زیرا شرط پیشرفت 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: هنگامی که در راه حلی برای کنترل شرایط رقابتی شرط انحصار متقابل برقرار نیست و برآورده نمی‌کند و همچنین بن بست رخ داده است یعنی شرط انتظار محدود هم برقرار نیست و برآورده نمی‌کند، بررسی شرط پیشرفت برای داشتن یک راه حل اخلاقی برای کنترل شرایط رقابتی نیاز نیست.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه چهارم را به عنوان پاسخ اعلام کرده بود. ولی همانطور که دیدید پاسخ سوال در گزینه‌ها نبود.

تست‌های فصل هفتم: مدیریت بن بست

۱۰۰- الگوریتم بانکدار (Banker) برای حل کدام مسئله به کار می‌رود؟

(مهندسی کامپیوتر - دولتی ۹۹)

۱) دوری از بن‌بست
۲) تشخیص بن‌بست
۳) جلوگیری از بن‌بست
۴) ترمیم (Recovery) بن‌بست

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل پنجم: مدیریت حافظه مجازی

۱۰۰- گزینه (۱) صحیح است.

مقابله با بن‌بست

در حالت کلی برای مقابله با بن‌بست به چهار روش می‌توان عمل کرد:

۱- الگوریتم Ostrich (شتر مرغ)

۲- تشخیص و ترمیم بن‌بست (Deadlock Detection and Recovery)

۳- پیشگیری (جلوگیری) از بن‌بست (Deadlock Prevention)

۴- اجتناب (دوری) از بن‌بست (Deadlock Avoidance)

الگوریتم Ostrich

در این روش، سیستم عامل وقوع بن‌بست را کاملاً نادیده می‌گیرد!

در واقع در این روش هیچ راهکاری جهت مقابله با بن‌بست اندیشیده نمی‌شود و در صورت وقوع بن‌بست، سیستم دوباره شروع به کار می‌کند. این یعنی کم هزینه‌ترین روش جهت مقابله با بن‌بست!

در برخی از سیستم عامل‌های امروزی، مانند بعضی نسخه‌های UNIX، از آنجا که احتمال وقوع بن‌بست بسیار پایین است (بن‌بست به ندرت رخ می‌دهد)، از این روش استفاده می‌شود.

تشخیص و ترمیم بن‌بست

در این روش هیچ هزینه‌ای برای پیشگیری و اجتناب از بن‌بست پرداخت نمی‌شود و اجازه می‌دهیم بن‌بست رخ دهد. پس از کشف بن‌بست، باید سیستم از بن‌بست خارج شود.

در این روش سه سؤال اساسی وجود دارد. اول، چگونه متوجه بن‌بست شویم؟ دوم، الگوریتم تشخیص بن‌بست در چه زمان‌هایی اجرا شود؟ و سوم، چگونه ازبن‌بست خارج شویم؟

تشخیص بن‌بست

برای تشخیص بن‌بست می‌توان از دو رویکرد استفاده کرد:

۱- اگر از هر منبع فقط یک نمونه (Instance) در سیستم موجود باشد، می‌توان با استفاده از

گراف تخصیص منابع، بن‌بست را کشف کرد. در این حالت، اگر در گراف تخصیص منابع حلقه وجود داشته باشد بن‌بست رخ داده است.

۲- اگر نمونه‌های متعددی از یک منبع در سیستم موجود باشد، کشف بن‌بست توسط الگوریتم Coffman انجام می‌شود.

توجه: الگوریتم کافمن (Coffman)، جهت کشف بن بست مورد استفاده قرار می‌گیرد.

توجه: از آنجا که اجرای الگوریتم‌های تشخیص بن‌بست پرهزینه می‌باشد، زمان اجرای آن و تعداد اجرای آن بسیار مهم است. به عنوان مثال به صورت‌های زیر این الگوریتم را اجرا کرد:

- هنگامی که یک منبع پس از درخواست قابل اعطا نیست.
- هنگامی که بهره‌وری پردازنده از حد آستانه کمتر شود (مثلاً ۵۰٪).
- هنگامی که بار پردازشی سیستم پایین است و می‌توان الگوریتم پرهزینه‌ی تشخیص بن‌بست را اجرا کرد.
- در بازه‌های زمانی مشخص و ثابت (مثلاً هر ۳۰ دقیقه یکبار)
- در زمان‌های دلخواه یا تصادفی

ترمیم بن‌بست

پس از تشخیص بن‌بست سیستم عامل باید آن را رفع کند. در این حالت برای برطرف کردن بن‌بست، سیستم عامل معمولاً می‌تواند از دو روش استفاده کند:

۱- اتمام فرآیند (Process Termination)

سیستم عامل تعدادی از فرآیندها را خاتمه می‌دهد. در این‌صورت منابعی که در اختیار این فرآیندهاست آزاد می‌شوند و فرآیندهای دیگر می‌توانند از آن استفاده کنند. در ضمن فرآیندهایی که خاتمه یافته‌اند باید بعداً دوباره از ابتدا اجرا شوند. بنابراین خاتمه دادن فرآیندها جهت مقابله با بن‌بست هزینه زیادی دارد. نحوه‌ی انتخاب فرآیندها جهت خاتمه نیز بسیار مهم است و تأثیر زیادی بر روی کارایی این روش دارد. به عنوان مثال اگر فرآیندی که برای خاتمه انتخاب می‌شود، قسمت عمده‌ی دستورات پردازشی خود را انجام داده باشد، سربار زیادی جهت اجرای دوباره‌ی این فرآیند به سیستم تحمیل می‌شود. بنابراین مسائلی از قبیل اولویت فرآیندها، مدت زمانی که از اجرای آنها می‌گذرد، مدت زمانی که از اجرای آنها باقی مانده، منابعی که در اختیار دارند و منابعی که نیاز دارند، در انتخاب یک فرآیند جهت خاتمه تأثیر مستقیم دارند.

در حالت کلی می‌توان گفت راهکار اتمام فرآیندها به دو صورت پیاده‌سازی می‌شود:

- **اتمام کلیه فرآیندهای موجود در بن‌بست:** این روش به وضوح حلقه بن‌بست را خواهد شکست اما هزینه آن بسیار بالاست زیرا ممکن است فرآیندهای بسیاری خاتمه خاتمه یابد.
- **اتمام یک به یک فرآیندها تا شکست حلقه‌ی بن‌بست:** این روش سربار قابل توجهی به سیستم اعمال می‌کند زیرا پس از اتمام هر فرآیند، الگوریتم تشخیص بن‌بست باید دوباره فراخوانی

شود تا معین شود آیا بن‌بست هنوز وجود دارد یا خیر.

۲- پس گرفتن منابع (Resource Preemption)

می‌توان فرآیندها را خاتمه نداد، بلکه منابع را به زور از آنها پس گرفت. هنگامی که یک منبع از فرآیندی پس گرفته می‌شود، آن فرآیند باید به عقب برگردانده شود تا اگر دوباره منبع به آن اختصاص داده شد، اجرای خود را از آن نقطه آغاز کند.

به این ترتیب باز هم نحوه‌ی انتخاب فرآیند جهت گرفتن منبع از آن و حتی نحوه‌ی انتخاب منبع، تأثیر مستقیمی بر روی کارایی سیستم دارد. از طرفی دیگر، باید این اطمینان وجود داشته باشد که قطحی‌زدگی رخ نخواهد داد یعنی باید تضمین شود که منابع همواره از یک فرآیند خاص پس گرفته نمی‌شوند.

پیشگیری از بن‌بست

برای وقوع بن‌بست باید هر چهار شرط کافمن برقرار باشند. برای پیشگیری از بن‌بست باید کاری کنیم که یکی از این شروط رخ ندهد. بنابراین هر چهار شرط را بررسی می‌کنیم:

الف- انحصار متقابل

برای مقابله با این شرط می‌توان کاری کرد که فرآیندها به صورت مشترک از منابع استفاده کنند. البته امکان استفاده اشتراکی از همه‌ی منابع وجود ندارد. برای مثال می‌توان از فایل‌های خواندنی به صورت اشتراکی استفاده کرد اما چاپگر را نمی‌توان به صورت اشتراکی و همزمان مورد استفاده قرار داد. البته با استفاده از تکنیک‌هایی مانند Spooling می‌توان از منابعی مانند چاپگر نیز به صورت اشتراکی استفاده کرد. در واقع به نوعی کارهای دیگر را بافر می‌کنیم.

ب- انحصاری بودن

برای مقابله با این شرط می‌توان کاری کرد که اگر یک فرآیند منبعی را درخواست کرد که امکان تخصیص آن وجود ندارد (یعنی فرآیند باید منتظر بماند)، آنگاه منابع دیگری که در اختیار آن فرآیند هستند نیز آزاد شوند (در واقع به زور آزاد می‌شوند). البته باید دقت کرد برخی از منابع را می‌توان به زور از یک فرآیند پس گرفت و به فرآیند دیگری داد، مانند حافظه، اما برخی دیگر را نمی‌توان، مانند چاپگر.

ج- نگهداری و انتظار

برای مقابله با این شرط می‌توان کاری کرد که یک فرآیند در ابتدا و قبل از شروع به اجرا، همه‌ی منابع مورد نیاز خود را درخواست دهد و در صورتی که همه‌ی منابع مورد نیازش آزاد بودند، شروع به اجرا کند. البته این روش ممکن است باعث قحطی‌زدگی شود، مثلاً فرض کنید یک فرآیند به منابع نسبتاً زیادی نیاز داشته و هرگاه که درخواست خود را اعلام می‌کند تعدادی از منابع مورد نیازش مشغول باشند. از طرفی بهره‌وری منابع به شدت کاهش می‌یابد، زیرا مثلاً

فرآیندی که یک منبع را در انتهای کار خود نیاز دارد، از ابتدای کار منبع را در اختیار گرفته و اجازه‌ی استفاده از منبع را به دیگران نمی‌دهد.

د- انتظار چرخشی

برای مقابله با این شرط یک روش پیشنهاد شده است. در این روش کلیه‌ی منابع را شماره‌گذاری کرده و به هر منبع یک شماره یکتا نسبت می‌دهیم. سپس باید قانونی وضع کنیم که طی آن فرآیندها فقط به ترتیب صعودی یا نزولی بتوانند درخواست منبع کنند. برای مثال اگر ترتیب صعودی را وضع کردیم و یک فرآیند منبعی با شماره ۵ را در اختیار دارد، دیگر نمی‌تواند درخواست منبع ۱ تا ۴ را داشته باشد. البته این روش معمولاً در عمل قابل پیاده‌سازی نیست. زیرا یک فرآیند ممکن است ابتدا منبع ۷ و سپس منبع ۶ را نیاز داشته باشد.

نکته: در کل، مکانیزم‌های پیشگیری از بن‌بست هزینه‌ی بالایی دارند و معمولاً باعث استفاده‌ی غیربهینه از منابع و کاهش بهره‌وری سیستم می‌شوند.

اجتناب از بن‌بست

تا اینجا برای مقابله با بن‌بست یا از روش تشخیص و بازیابی استفاده می‌کردیم و یا از روش پیشگیری. روش تشخیص و بازیابی این نقص را دارد که هنگام بازیابی، تعدادی از فرآیندها باید به عقب برگردند و سربار تحمیل شده به سیستم بسیار بالاست. روش پیشگیری از منابع استفاده‌ی بهینه نمی‌کند و در کل بهره‌وری سیستم را کاهش می‌دهد.

اما ایده‌ی کلی موجود در روش اجتناب از بن‌بست این است که هنگامی که یک فرآیند منبعی را درخواست کرد، حتی اگر منبع مورد نیاز وی کاملاً آزاد است، فوراً آن منبع را در اختیار فرآیند قرار ندهیم، بلکه ابتدا بررسی کنیم با تخصیص دادن این منبع به فرآیند، احتمال وقوع بن‌بست در آینده وجود دارد یا خیر. در صورتی که احتمال وقوع بن‌بست در آینده وجود داشته باشد، فرآیند مورد نظر باید منتظر بماند و منبع مورد نیاز وی، فعلاً به وی تخصیص داده نمی‌شود.

تعریف حالت امن (Safe State)

حالتی است که در آن سیستم بتواند منابع را برای هریک از فرآیندها (تا حداکثر نیاز آنها) به ترتیبی اختصاص دهد که در ضمن آن از وقوع بن‌بست نیز جلوگیری شود. به عبارت دیگر، سیستم زمانی در یک وضعیت امن قرار دارد که یک دنباله امن وجود داشته باشد.

تعریف دنباله‌ی امن (Safe Sequence)

دنباله‌ای از فرآیندها (به صورت $< P_1, P_2, P_3, \ldots, P_n >$) را دنباله‌ی امن گویند. اگر برای هر یک از Pi ها، منابعی که Pi نیاز دارد به وسیله‌ی منابع آزاد فعلی به علاوه‌ی منابع در اختیار تمامی Pj ها (که j<i) تأمین گردد.

به عنوان مثال اگر منابعی که P_4 نیاز دارد، بلافاصله در اختیار نباشد، P_4 می‌تواند تا زمانی که

P_1، P_2 وP_3 پایان می‌یابند، منتظر بماند. هنگامی که این سه فرآیند خاتمه یافتند و منابع خود را آزاد کردند. P4 می‌تواند تمام منابع مورد نیازش را به‌دست آورده و به کارش ادامه دهد. وقتی P_4 پایان یافت P_5 می‌تواند منابع مورد درخواست خود را به دست آورده و اجرا شود و این روال ادامه می‌یابد.

تعریف حالت ناامن (Unsafe State)

اگر در یک سیستم هیچ دنباله‌ی امنی وجود نداشته باشد، سیستم در وضعیت ناامن قرار دارد.

نکته: در وضعیت امن، بن‌بست رخ نمی‌دهد اما یک وضعیت ناامن می‌تواند به بن‌بست ختم شود اما نه لزوماً. در واقع همه‌ی وضعیت‌های ناامن، بن‌بست نیستند.

مثال: یک سیستم را با سه فرآیند P_0، P_1 و P_2 در نظر بگیرید. در این سیستم ۱۱ نوارخوان به عنوان منبع وجود دارند. فرض کنید در لحظه جاری، فرآیند P_0، ۴ نوارخوان و P_1، ۲ نوارخوان و P_2 نیز ۲ نوارخوان را در اختیار گرفته‌اند، در واقع در لحظه‌ی جاری فقط فقط ۳ نوارخوان آزاد هستند. از طرفی فرض کنید اطلاع داریم فرآیندها برای تکمیل کارهای خود حداکثر به چند نوارخوان نیاز دارند، با این حساب فرآیند P_0 برای تکمیل، حداکثر به ۹ نوارخوان، فرآیند P_1 به ۴ نوارخوان و فرآیند P_2 به ۹ نوارخوان نیاز دارد.

	حداکثر نیاز	منبع در اختیار	نیاز آینده
P_0	۹	۴	۵
P_1	۴	۲	۲
P_2	۹	۲	۷

در لحظه‌ی t_0، می‌توان گفت سیستم در حالت امن است، زیرا یک توالی امن به صورت $<P_1,P_0,P_2>$ وجود دارد. زیرا در این حالت چون فقط ۳ نوارخوان آزاد داریم، فقط فرآیند P_1 می‌تواند نوارخوان‌های مورد نیازش را دریافت کند که در این صورت پس از اتمام، P_1 کل منابع در اختیارش را آزاد می‌کند و بعد از این اتفاق تعداد نوارخوان‌های آزاد به ۵ می‌رسد. سپس فرآیند P_0 می‌تواند هر ۵ نوارخوان مورد نیازش را دریافت کرده و به کار خود خاتمه دهد. پس از این مرحله P_0 تمام منابع خود را آزاد اعلام می‌کند، در این حالت تعداد نوارخوان‌های آزاد سیستم به ۹ عدد می‌رسد، سپس P_2 نیز می‌تواند نوارخوان‌های مورد نیازش را دریافت کرده و خاتمه یابد.

در نتیجه در لحظه‌ی t_0، سیستم در حالت امن قرار دارد، اما مثلاً فرض کنید سیستم در همان وضعیت t_0 قرار دارد و کمی بعد در لحظه‌ی t_1، فرآیند P_2 تقاضای یک نوارخوان را صادر می‌کند اگر با درخواست وی موافقت شود و یک نوارخوان به آن اختصاص یابد، سیستم دیگر در حالت امن قرار ندارد. زیرا در لحظه‌ی t_1 فقط ۲ نوارخوان آزاد وجود دارد و در این حالت فقط P_1 می‌تواند اجرای خود را خاتمه دهد، پس ازخاتمه P_1 و آزاد کردن منابع در اختیارش، تعداد

نوارخوان‌های آزاد سیستم به ۴ می‌رسد. که نمی‌تواند نیازهای آینده هیچ کدام از فرآیندهای P_0 یا P_2 را برآورده کند، زیرا P_0 برای تکمیل به ۵ نوارخوان و P_2 برای تکمیل به ۶ نوارخوان نیاز دارد. بنابراین عاقلانه نیست که در لحظه‌ی t_1 به درخواست P_2 عمل کنیم و یک عدد نوارخوان مورد نیاز آن را در اختیارش قرار دهیم بلکه فرآیند P_2 باید فعلاً منتظر بماند تا طبق دنباله‌ی امن لحظه‌ی t_0، یعنی $<P_2, P_0, P_1>$ پیش رویم.

بنابراین هرگاه فرآیندی، منبعی را درخواست کردکه حتی آن منبع آزاد بود، نباید بلافاصله منبع در اختیار وی قرار بگیرد، بلکه باید بررسی شود که اگر با این درخواست موافقت شود، سیستم باز هم در وضعیت امن باقی می‌ماند یا خیر.

الگوریتم بانکدار جهت اجتناب (دوری) از بن‌بست (Bankers Algorithm)

این الگوریتم جهت اجتناب از بن‌بست به کار می‌رود و همانند یک بانکدار عمل می‌کند که هیچ‌گاه همه‌ی سرمایه‌ی خود را به مشتریانش تخصیص نمی‌دهد و همواره طوری عمل می‌کند که بتواند نیازهای مشتریانش را برآورده کند.

برای پیاده‌سازی الگوریتم بانکدار به تعدادی آرایه نیاز داریم. فرض کنید که در یک سیستم تعداد n فرآیند و m نوع منبع در اختیار داشته باشیم. دقت کنید که از هر نوع منبع می‌توان هر تعدادی داشت.

در این الگوریتم آرایه‌های زیر وجود دارند:

۱- **Available**: یک آرایه یک بعدی با طول m است که تعداد منابع آزاد برای هر نوع منبع را مشخص می‌کند. به عنوان مثال اگر Available[i] = k باشد به این معناست که از منبع نوع i ام، تعداد k نمونه آزاد موجود است.

۲- **Max**: یک ماتریس دو بعدی m×n است که حداکثر نیاز هر فرآیند به منابع را مشخص می‌کند.

به عنوان مثال اگر Max[i ،j]= k باشد، به این معناست که فرآیند i ام، برای انجام کار خود حداکثر به تعداد k از منبع j ام نیاز دارد.

۳- **Allocation**: یک ماتریس دو بعدی m×n است که مشخص می‌کند از هر منبع در حال حاضر چه تعدادی به هر فرآیند اختصاص یافته است. به عنوان مثال اگر Allocation [i ،j]= k باشد، به این معناست که در حال حاضر فرآیند i ام، تعداد k نمونه از منبع j ام را در اختیار دارد.

۴- **Need**: یک ماتریس دو بعدی m×n است که مشخص می‌کند هر فرآیند ممکن است در آینده چه تعدادی از هر منبعی را درخواست کند. به عنوان مثال اگر Need[i ،j]= k باشد، به این معناست که فرآیند i ام، در آینده و برای تکمیل کار خود، ممکن است به تعدادی k نمونه از منبع j ام نیاز داشته باشد (علاوه بر منابعی که در اختیار دارد).

نکته: باید دقت کرد Need[i ،j]= Max[i j] - Allocation [i ،j]

روال کار الگوریتم: ابتدا باید بررسی کرد که چگونه می‌توان فهمید یک سیستم در حال حاضر در وضعیت امن قرار دارد یا خیر. برای این منظور به روش زیر عمل می‌کنیم:

۱- باید در ماتریس Need به دنبال سطری بگردیم که در آن همه‌ی عناصر از بردارAvailable کوچکتر باشند. در واقع با این کار به دنبال فرآیندی می‌گردیم که با منابع آزاد سیستم بتوانیم تمام نیازهای آینده‌اش را برآورده کنیم. اگر چنین سطری یافت نشد، سیستم در حالت **ناامن** قرار دارد.

۲- اگر در مرحله ۱ سطری پیدا شد، به این معناست که می‌توانیم یک فرآیند را تا انتها اجرا کرده و خاتمه دهیم. آن فرآیند پس از خاتمه، تمام منابع خود را آزاد می‌کند پس باید منابعی که قبلاً در اختیار داشته، به منابع آزاد سیستم افزوده شوند. برای این منظور، اعداد مربوط به آن فرآیند در ماتریس Allocation را، به برادر Available اضافه می‌کنیم و بعد از این کار این فرآیند به عنوان یک فرآیند خاتمه یافته به حساب می‌آید.

۳- دوباره باید مراحل ۱ و ۲، این بار با بردار Available جدید و به روز شده تکرار شوند، تا وقتی که همه‌ی فرآیندها پایان یابند. در این صورت وضعیت سیستم امن گزارش می‌شود و ترتیب خاتمه‌ی فرآیندها، به عنوان یک دنباله‌ی امن به حساب می‌آید.

نکته: اگر هنگام اجرای الگوریتم، بتوانیم در یک حالت دو فرآیند مختلف را در مرحله ۲ انتخاب کنیم، هیچ تفاوتی نمی‌کند که کدام فرآیند انتخاب شود.

حال ببینیم الگوریتم بانکدار چگونه با درخواست‌های فرآیندها برخورد می‌کند. فرض کنید فرآیند P_i درخواست تعدادی منبع از یک یا چند نوع را داشته باشد، واضح است درخواست‌های یک فرآیند از سطر مربوط به آن فرآیند در ماتریس Need بیشتر نیست. پس از اینکه فرآیند منابع خود را درخواست کرد و در صورتی که با استفاده از منابع آزاد سیستم بتوانیم درخواست وی را برآورده کنیم، به طور ضمنی فرض می‌کنیم منابع به فرآیند تخصیص یابند، سپس ماتریس‌ها و بردارها را به روز کرده و بررسی می‌کنیم وضعیت سیستم باز هم امن باقی می‌ماند یا خیر. اگر بعد از این تخصیص باز هم وضعیت سیستم امن تشخیص داده شد، منابع را به طور قطعی به فرآیند اختصاص می‌دهیم، اما اگر وضعیت سیستم ناامن تشخیص داده شد، نباید با این درخواست موافقت شود.

در هر صورت اگر منابع به فرآیند اختصاص یافت، تغییرات زیر باید اعمال شوند:

۱- منابع مورد درخواست از بردار Available کسر گردند.

۲- منابع مورد درخواست باید سطر مربوط به آن فرآیند در ماتریس Allocation اضافه شوند.

۳- منابع مورد درخواست باید از سطر مربوط به آن فرآیند در ماتریس Need کسر گردند.

(در واقع همیشه باید Need = Max– Allocation)

نکته: الگوریتم بانکدار از نظر تئوری بسیار خوب عمل می‌کند، اما در عمل به طور مؤثر استفاده نمی‌شود.

دلایلی که باعث می‌شوند عملاً از این الگوریتم استفاده نشود عبارتند از:

۱- فرآیندها معمولاً از حداکثر نیاز نهایی خود به منابع آگاهی ندارند.
۲- تعداد فرآیندها ثابت نیست و با وارد و خارج شدن کاربران تغییر می‌کند.
۳- منابعی که گمان می‌رود در اختیار سیستم هستند، ممکن است به علت خرابی یا هر علت دیگری بلا استفاده شوند.

نکته: الگوریتم بانکدار دارای مرتبه اجرایی $m \times n^2$ است.

مثال: سیستمی با پنج فرآیند P_0 تا P_4 و سه منبع A، B و C را در نظر بگیرید.

منبع A دارای ۱۰ نمونه، منبع B دارای ۵ نمونه و منبع C دارای ۷ نمونه است. فرض کنید در زمان t_0 وضعیت سیستم مطابق جدول زیر باشد:

Available

A	B	C
3	3	2

Max

منبع / فرآیند	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Allocation

منبع / فرآیند	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

با توجه به ماتریس‌های Max و Allocation، ماتریس Need به صورت زیر می‌باشد:

Need

منبع / فرآیند	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

با توجه به بردار Allocation و ماتریس Need در می‌یابیم در ابتدای کار درخواست‌های آینده‌ی فرآیند P_1 از بردارAvailable کمتر است، پس می‌توان P_1 را اجرا کرد. بنابراین بعد از اتمام P_1، تمامی منابعی که در اختیار داشته است به بردار Available اضافه می‌شوند و بردار Available به صورت زیر درمی‌آید:.

A B C

Available=(5 , 3 , 2)

در واقع سطر مربوط به P_1 در ماتریس Allocation را به Available اضافه کردیم. در این لحظه و با این بردار Available جدید، می‌توانیم P_3 (یا حتی P_4) را انتخاب کنیم. بعد از انتخاب P_3 و اتمام اجرای آن، منابعی که در اختیار P_3 بوده‌اند آزاد می‌شوند، پس بردار Available به صورت زیر در می‌آید:

A B C

Available=(7 , 4 , 3)

پس از P_3 می‌توانیم P_4 (یا حتی P_2 یا P_0) را انتخاب کنیم که بعد از اتمام P_4 بردار Available به صورت زیر در می‌آید:

A B C

Available=(7 , 4 , 5)

و بعد از آن P_2 را اجرا می‌کنیم و داریم:

A B C

Available=(10 , 4 , 7)

و در نهایت P_0 اجرا می‌شود و همه‌ی منابع آزاد می‌شوند:

A B C

Available=(10 , 5 , 7)

در واقع با اجرای مراحل الگوریتم متوجه می‌شویم دنباله‌ی <P_0، P_2، P_4، P_3، P_1 > یک دنباله‌ی امن را مشخص می‌کند، پس سیستم در وضعیت امن قرار دارد. دقت کنید این دنباله‌ی امن یک نمونه از دنباله‌های امن ممکن می‌باشد و برای یک سیستم شاید بتوان چندین دنباله‌ی امن کشف کرد.

حال فرض کنید فرآیند P_1، یک نمونه‌ی دیگر از منبع A و دو نمونه از منبع C را درخواست می‌کند، یعنی Request (P_1) = (2، 0، 1). حال بررسی می‌کنیم در صورتی که با درخواست P_1 موافقت شود و منابع مورد نیازش در اختیار وی قرار گیرند، سیستم باز هم در وضعیت امن قرار دارد یا خیر.

برای این منظور فرض می‌کنیم منابع در اختیار P_1 قرار گیرند.

در این صورت، وضعیت جدید سیستم مطابق زیر است:

Allocation

فرآیند \ منبع	A	B	C
P_0	0	1	0
P_1	3	0	2
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

فرآیند \ منبع	A	B	C
P_0	7	4	3
P_1	0	2	0
P_2	6	0	0
P_3	4	1	1
P_4	4	3	1

Available

A	B	C
2	3	0

با توجه به حالت فوق دنباله‌ی <P_2، P_0، P_4، P_3، P_1 > به عنوان یک دنباله‌ی امن وجود دارد، یعنی یک وضعیت امن موجود است، پس با درخواست P_1 موافقت می‌کنیم.

به عنوان مثالی دیگر فرض کنید در همان سیستم در لحظه‌ی t_0، فرآیند P_4 درخواست (0، 3، 4) را داشته باشد، با این درخواست P_4 موافقت نمی‌شود زیرا منابع مورد نیاز اصلاً در دسترس نیستند.

به عنوان مثالی دیگر فرض کنید در همان سیستم در لحظه‌ی t_0 فرآیند P_3 درخواست (0، 1، 1) را داشته باشد، با این درخواست نیز موافقت نمی‌شود زیرا از سطر مربوط به P_3 در ماتریس Need بیشتر است.

به عنوان مثالی دیگر فرض کنید در لحظه‌ی t_0، فرآیند P_0 درخواست (0، 3، 0) را داشته باشد، با این درخواست نیز با وجود اینکه منابع مورد نیاز در دسترس هستند، موافقت نمی‌شود. زیرا بعد از تخصیص این منابع، وضعیت حاصل ناامن است و هیچ دنباله‌ی امنی نمی‌توان در نظر گرفت.

توجه: هنگامی که رفتار فرآیندها از جهت نگهداری و درخواست و یا رهاسازی و درخواست مشخص نباشد، نیاز به ماتریس Need است که در این شرایط برای تشخیص وضعیت امن یا ناامن، الگوریتم تشخیص وضعیت امن یا ناامن بانکداران مورد استفاده قرار می‌گیرد.

توجه: هنگامی که رفتار فرآیندها نگهداری و درخواست باشد، بنابراین به جای ماتریس Need، نیاز به ماتریس Request وجود دارد، که در این شرایط برای کشف بن‌بست، الگوریتم کشف بن‌بست کافمن (Coffman) مورد استفاده قرار می‌گیرد.

توجه: الگوریتم بانکداران، جهت بررسی امن یا ناامن بودن مورد استفاده قرار می‌گیرد و در مورد بن‌بست نظر نمی‌دهد.

توجه: الگوریتم کافمن (Coffman)، جهت کشف بن‌بست مورد استفاده قرار می‌گیرد.

مثال: در سیستمی با پنج پردازه و سه نوع منبع، وضعیت تخصیص منابع به صورت زیر است. اگر در این وضعیت، درخواستی برای یک واحد دیگر از منبع A توسط پردازه P_3 صادر شود، کدام مورد درست است؟ **(مهندسی IT – دولتی ۹۶)**

(تعداد منابع اولیه)

A	B	C
8	6	10

(حداکثر نیاز)

	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

(تخصیص‌یافته)

	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	1	0	2
P_4	1	1	0

۱) بعد از انجام درخواست فوق بن‌بست قطعی است.
۲) قبل از انجام درخواست فوق وقوع بن‌بست قطعی است.
۳) قبل از درخواست فوق احتمال وقوع بن‌بست وجود دارد.
۴) بعد از انجام درخواست فوق احتمال وجود بن‌بست وجود دارد.

پاسخ: گزینه (۴) صحیح است.

ابتدا وضعیت سیستم را قبل از اجابت درخواست فرآیند P_3 بررسی می‌کنیم:

با توجه به ماتریس تخصیص منابع (Allocation) و منابع اولیه، بردار Available را بدست می‌آوریم، برای یافتن بردار Available ابتدا برای هر منبع، منابع تخصیص یافته به هر فرآیند را با هم جمع کرده و سپس از منابع اولیه کسر می‌نماییم.

A منبع موجود $= 8-(2+3+1+1) = 1$

B منبع موجود $= 6-(1+2+1) = 2$

C منبع موجود $= 10-(2+3+2) = 3$

بنابراین بردار Available به صورت زیر است:

Available	1	2	3

براساس رابطه زیر داریم:

$$Need = MAX - Allocation$$

فرآیند	Need		
	A	B	C
P_0	3	5	6
P_1	5	3	3
P_2	2	1	3
P_3	3	5	7
P_4	1	2	3

=

فرآیند	MAX		
	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

−

فرآیند	Allocation		
	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	1	0	2
P_4	1	1	0

Available	1	2	3

$$(1,2,3) \xrightarrow[+(1,1,0)]{P_4} (2,3,3) \xrightarrow[+(3,2,0)]{P_2} (5,5,3) \xrightarrow[+(2,0,3)]{P_1} (7,5,6)$$

$$\xrightarrow[+(0,1,2)]{P_0} (7,6,8) \xrightarrow[+(1,0,2)]{P_3} (8,6,10)$$

به این ترتیب و با توجه به ماتریس‌های فوق، دنباله‌ی امنِ $< P_4, P_2, P_1, P_0, P_3 >$ (از چپ به راست) برای این سیستم موجود است. بنابراین قبل از اجابتِ درخواست فرآیند P_3، سیستم در حالت امن قرار دارد و گزینه‌های دوم و سوم نادرست هستند.

الگوریتم درخواست منبع بانکداران

هنگامی که فرآیند P_i، بردار Request[i] را به عنوان اعلام نیاز به منابع مطرح می‌کند، باید این درخواست مطابق مراحل زیر بررسی گردد:

۱) اگر $Request[i] \leq Need[i]$ است به مرحله بعدی برو، در غیر اینصورت خطای تقاضای بیش از نیاز اعلام می‌شود. در واقع در این مرحله بررسی می‌شود که اگر فرآیند درخواست بیش از آن چه که در ابتدا اعلام نیاز کرده است را دارد، با این درخواست مخالفت شود.

۲) اگر $Request[i] \leq Availabe[i]$ است به مرحله 3 برو، در غیر اینصورت منبع کافی جهت تخصیص به این فرآیند وجود نداشته و این فرآیند باید منتظر بماند.

۳) با فرض این که منابع مورد نیاز به این فرآیند اختصاص می‌یابد، در این صورت با اصلاح ماتریس‌ها و بردارها یک الگوریتم تشخیص وضعیت امن اجرا می‌شود. با اجرای این الگوریتم اگر سیستم در وضعیت امن باقی بماند، این منابع به فرآیند اختصاص می‌یابند، در غیراینصورت اگر با این تخصیص سیستم به وضعیت ناامن وارد شود، ماتریس‌ها و بردارها به حالت قبل بازگردانده شده و فرآیند تا آزاد شدن منابع بیشتر منتظر می‌ماند.

اصلاح ماتریس‌ها و بردارها در صورت تخصیص به صورت زیر انجام می‌شوند:

Available (new) = Available (old) – Request[i]

Need (new) [i] = Need (old) [i] – Request[i]

Allocation (new) [i] = Allocation (old) [i] + Request[i]

مطابق فرض صورت سؤال، اگر در این وضعیت، درخواستی برای یک واحد دیگر از منبع A، توسط فرآیند P_3 صادر شود، مطابق الگوریتم درخواست منبع بانکداران، ابتدا بررسی می‌شود که آیا درخواست فرآیند، کم‌تر از نیاز اعلام شده آن است یا خیر.

$Request[P_3] \leq Need[P_3]$

$[1,0,0] \leq [3,5,7]$

چون شرط اول برقرار است، به سراغ شرط دوم می‌رویم.

در شرط دوم بررسی می‌شود که آیا درخواست داده شده، کمتر از منابع آزاد سیستم است یا خیر.

$Request[P_3] \leq Available$

$[1,0,0] \leq [1,2,3]$

چون هر دو شرط برقرار است، سراغ مرحله سوم از الگوریتم می‌رویم.
در این مرحله، با فرض اینکه درخواست داده شده، اعمال شود، ماتریس‌ها به صورت زیر بروزرسانی می‌شوند:

فرآیند	Need (new)		
	A	B	C
P_0	3	5	6
P_1	5	3	3
P_2	2	1	3
P_3	②	5	7
P_4	1	2	3

=

فرآیند	MAX		
	A	B	C
P_0	3	6	8
P_1	7	3	6
P_2	5	3	3
P_3	4	5	9
P_4	2	3	3

−

فرآیند	Allocation (new)		
	A	B	C
P_0	0	1	2
P_1	2	0	3
P_2	3	2	0
P_3	②	0	2
P_4	1	1	0

با توجه به ماتریس تخصیص منابع **جدید** ((Allocation(new) و منابع اولیه، بردار Available(new) را بدست می‌آوریم، برای یافتن بردار Available(new) ابتدا برای هر منبع، منابع تخصیص‌یافته به هر فرآیند را با هم جمع کرده و سپس از منابع اولیه کسر می‌نماییم.

منبع موجود A $= 8-(2+3+2+1) = 0$

منبع موجود B $= 6-(1+2+1) = 2$

منبع موجود C $= 10-(2+3+2) = 3$

بنابراین بردار Available(new) به صورت زیر است:

Available(new)	⓪	2	3

مطابق رابطه زیر نیز می‌توان Available (new) را محاسبه نمود:

Available (new) = Available (old) – Request [P_3]
Available (new) = [1,2,3] – [1,0,0] = [0,2,3]

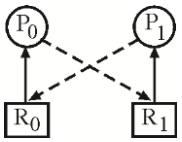
مشاهده می‌شود که با بردار Available(new) موجود، نمی‌توان نیاز هیچ فرآیندی را با توجه به ماتریس Need(new) مرتفع ساخت، بنابراین توالی امن یافت نمی‌شود و سیستم در یک وضعیت ناامن قرار دارد و احتمال وقوع بن‌بست وجود دارد. بنابراین پاسخ سؤال گزینه چهارم است.

توجه: ناامنی نه به معنی وقوع بن‌بست در گذشته و نه به معنی وقوع حتمی بن‌بست در آینده است، بلکه به معنی احتمال وقوع بن‌بست در آینده است.

توجه: در حالت ناامن نیز ممکن است، فرآیندها در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند و سیستم به بن‌بست نرسد. به بیان دیگر در حالت ناامن وقوع بن‌بست قطعی نیست. اما اگر در حالت ناامن فرآیندها علاوه بر نگهداری منابع تحت تملک خود، منابع دیگری را مورد درخواست قرار دهند، در این حالت فرآیندهایی که نیاز آن‌ها برطرف نمی‌شود یک به یک در صف انتظار قرار می‌گیرند و همچون سرنوشت‌های به هم گره خورده، تا ابد منتظر یکدیگر باقی می‌مانند، یعنی بن‌بست رخ داده است.

مثال: در یک سیستم تعداد منبع اولیه R_0 برابر یک و تعداد منبع اولیه R_1 برابر یک است. اگر فرآیند P_0 یک منبع R_0 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_1 نیز داشته باشد و همچنین فرآیند P_1 یک منبع R_1 را در تملک داشته باشد و برای اتمام، نیاز به یک منبع R_0 نیز داشته باشد، وضعیت سیستم را در این شرایط بررسی کنید:

حل: ماتریس و گراف منابع و فرآیندها به صورت زیر است:



فرآیند	MAX	
	R_0	R_1
P_0	1	1
P_1	1	1

=

فرآیند	Allocation	
	R_0	R_1
P_0	1	0
P_1	0	1

+

فرآیند	Need	
	R_0	R_1
P_0	0	1
P_1	1	0

با توجه به ماتریس تخصیص منابع (Allocation) و منابع اولیه، بردار Available را بدست می‌آوریم:

R_0 منبع موجود $= 1-(1) = 0$

R_1 منبع موجود $= 1-(1) = 0$

بنابراین بردار Available به صورت زیر است:

Available	0	0

مشاهده می‌شود که با بردار Available موجود، نمی‌توان نیاز هیچ فرآیندی را با توجه به ماتریس Need مرتفع ساخت، بنابراین توالی امن یافت نمی‌شود و سیستم در یک وضعیت ناامن قرار دارد و احتمال وقوع بن‌بست وجود دارد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 یا P_1 و یا هر دو، در موقعیت رهاسازی منابع در اختیار خود قرار بگیرند، یعنی رهاسازی کنند و سپس درخواست منابع باقی‌مانده خود را صادر کنند، بن‌بست رخ نمی‌دهد.

توجه: در این شرایط ناامن، اگر فرآیند P_0 در عین حال اینکه منبع R_0 را تحت تملک و نگهداری خود دارد، منبع R_1 را نیز درخواست کند، از آن‌جا که منبع R_1 در اختیار فرآیند P_1 می‌باشد، فرآیند P_0 در صف انتظار، می‌خوابد (نگهداری و انتظار) حال اگر در ادامه ماجرا، فرآیند P_1 نیز همین رویه را در پیش گیرد، یعنی فرآیند P_1 نیز در عین حال اینکه منبع R_1 را تحت تملک و نگهداری خود دارد، منبع R_0 را نیز درخواست کند، از آن‌جا که منبع R_0 در اختیار فرآیند P_0 می‌باشد، فرآیند

P_1 نیز در صف انتظار، می‌خوابد (نگهداری و انتظار). این سرنوشت‌های بـه هـم گـره خـورده، در ادامه بن‌بست را به ارمغان می‌آورد.

توجه: از دو توجه فوق این نتیجه استنباط می‌گردد که در شرایط ناامن، **احتمـال وقـوع بن‌بسـت** وجود دارد. در واقع در شرایط ناامن وقوع بن‌بست و عدم بن‌بست بـه رفتـار فرآینـدها در آینـده، بستگی دارد. یعنی فرآیندها رفتار نگهداری و درخواست را در پیش گیرند و یا رفتـار رهاسـازی و درخواست را.

تست‌های فصل هفتم: مدیریت بن بست

۱۰۱- کدام گزینه درباره ریسه‌های (Threads) سطح کاربر و سطح هسته درست است؟

(مهندسی کامپیوتر - دولتی ۹۹)

۱) زمان‌بندی ریسه‌های سطح هسته سریعتر از ریسه‌های سطح کاربر است.

۲) ریسه‌های سطح کاربر و سطح هسته از طریق فراخوانی سیستمی (System Calls) به هم سرویس می‌دهند.

۳) ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس هم دسترسی داشته و می‌توانند در فضای آدرس هم بنویسند.

۴) ریسه‌های سطح هسته به فضای آدرس ریسه‌های سطح کاربر دسترسی دارند، اما ریسه‌های سطح کاربر به فضای آدرس ریسه‌های سطح هسته دسترسی ندارند.

عنوان کتاب: سیستم عامل

مولف: ارسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

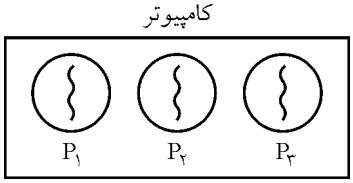
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل پنجم: مدیریت حافظه مجازی

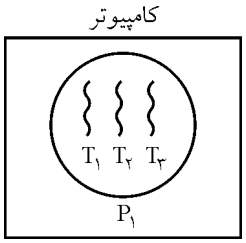
۱۰۱- گزینه (۳) صحیح است.

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.

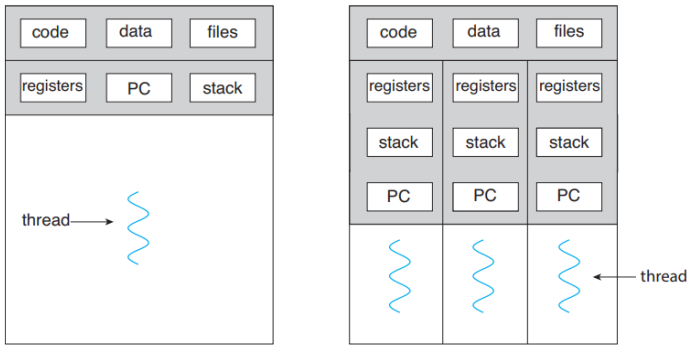
شکل زیر سه فرآیند معمولی را نشان می‌دهد که هریک برای خودشان یک رشته اجرایی و یک حافظه مختص به خود را دارند.



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.



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



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، مثلاً 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.**

توجه: گاهی از نخ به عنوان Light Weight Process (فرآیند سبک وزن) نیز یاد می‌کنند و به کل یک فرآیند، Heavy Weight Process (فرآیند سنگین وزن) نیز گویند.

توجه: نخ‌ها هم مانند فرآیندها می‌توانند حالت‌های مختلفی را تجربه کنند مانند آماده، در حال اجرا یا منتظر. در واقع پردازنده می‌تواند بین نخ‌ها به اشتراک گذاشته شود.

مزایای فرآیندهای چندنخی

۱- ساختار بسیاری از برنامه‌های کاربری ذاتاً از بخش‌های کاملاً مستقل تشکیل می‌شوند که جدا نکردن آن‌ها باعث پیچیدگی بالا و کاهش خوانایی در برنامه می‌گردد. مهندسی نرم‌افزار نیز بر ساخت برنامه‌های کاربردی توسط پیمانه‌های مستقل نیز تأکید دارد. برای مثال، خطاست اگر بیندیشید که برنامه شبیه‌سازی ۱۱ بازیکن یک تیم فوتبال در یک نخ قرار گیرد. به عنوان مثالی دیگر یک برنامه واژه‌پرداز می‌تواند از نخ‌های مستقلی مانند کنترل املا و گرامر، صفحه‌آرایی، مدیریت ورودی‌های کاربر و غیره تشکیل شده باشد.

۲- در فرآیند تک‌نخی، هرگاه فراخوان سیستمی مسدود کننده‌ای اجرا شود، کل فرآیند مسدود می‌گردد. در حالی‌که در فرآیندهای چندنخی در صورتی که سیستم عامل زمان‌بندی چند نخی را پشتیبانی کند، فقط نخی که فراخوان سیستمی مسدودکننده دارد، مسدود می‌گردد و مابقی نخ‌های یک فرآیند می‌توانند پس از در اختیار گرفتن پردازنده، اجرا گردند. فرآیند تک‌نخی مانند قانونی

می‌باشد که اگر یک نفر در خانواده خطا کند، همه خانواده محکوم می‌گردند و فرآیند چندنخی مانند قانونی می‌باشد که اگر یک نفر در خانواده خطا کند، فقط همان یک نفر محکوم می‌گردد و بقیه خانواده می‌توانند به زندگی طبیعی خود ادامه دهند.

۳- ایجاد هم‌روندی (در سیستم تک‌پردازنده‌ای) و توازی (در سیستم چندپردازنده‌ای) در نخ‌های یک فرآیند و فرآیندهای دیگر.

مثال: کاربرد چندنخی در فرآیند سمت سرویس‌دهنده.

در این مدل، فرآیند سرویس‌دهنده از چندین نخ جهت پاسخ به درخواست‌های کاربر یعنی سرویس‌گیرنده تشکیل شده است. پاسخ هر کاربر می‌تواند توسط یک نخ از سمت سرویس‌دهنده داده شود. چنانچه نخی در فرآیند سرویس‌دهنده جهت تبادل داده از روی دیسک به سمت سرویس‌گیرنده مسدود گردد، نخ‌های دیگر فرآیند سرویس‌دهنده می‌توانند به درخواست‌های دیگر، پاسخ دهند. زیرا کارکرد آن‌ها وابسته به نخ مسدود شده نیست.

توجه: شاید بگویید به جای قرار دادن کارهای مختلف یک سرویس‌دهنده در داخل نخ‌های یک فرآیند، می‌شد هریک از کارها را در داخل یک فرآیند قرار داد و چند فرآیندی را در مقابل چندنخی ابداع کرد. اما به دلایل زیر استفاده از چندنخی معقولانه‌تر به‌نظر می‌رسد:

– هزینه زمانی ایجاد (بارگذاری TCB) و پایان دادن (ذخیره‌سازی TCB) به یک نخ در یک فرآیند به مراتب کمتر از ایجاد (بارگذاری PCB) و پایان دادن (ذخیره‌سازی PCB) یک فرآیند است. نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند، در حالی که فرآیندها، منابع مختص به خود را در اختیار می‌گیرند.

– نخ‌های همتا در یک فرآیند، اشتراکات زیادی باهم دارند، بنابراین عمل تعویض متن بین آنها با هزینه کمتری انجام می‌گردد. در حالی که تعویض متن بین فرآیندها به دلیل عدم اشتراکات با هزینه بیشتری انجام می‌گردد.

توجه: اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

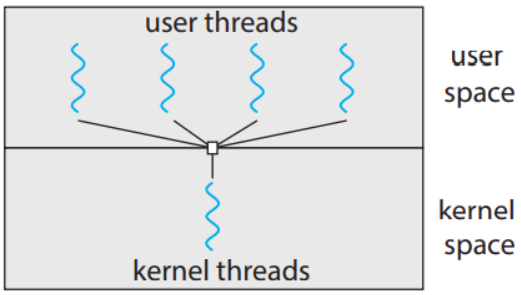
اداره نخ‌ها توسط بسته نخ (Thread Package) به سه روش زیر وجود دارد:

نخ‌های سطح کاربر: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ در مد کاربر و توسط کتابخانه نخ (Thread Library) انجام می‌شود. در این روش هسته سیستم عامل از وجود نخ اطلاع ندارد. در این روش فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح کاربر وجود دارد و زمان‌بند چند نخی در سطح هسته در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته

سیستم عامل فقط فرآیندها را می‌شناسد و هیچ اطلاعاتی از نخ‌ها ندارد. در واقع اولویت‌بندی نخ‌ها، مدیریت نخ‌ها و زمان‌بند چندنخی در سطح کاربر و توسط یک بسته نرم‌افزاری انجام می‌گردد. بدین معنی که نخ‌ها را برنامه‌نویس مشخص می‌کند و مدیریت آن‌ها را نیز بر عهده می‌گیرد. بنابراین زمان‌بند پردازنده، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از فرآیندهای آماده قرار می‌دهد، سپس زمان‌بند چند نخی در سطح کاربر، متناسب با کاربردی که در آن فرآیند به کار گرفته می‌شود، الگوریتم زمان‌بندی را انتخاب کرده و تصمیم می‌گیرد که پردازنده در اختیار کدام‌یک از نخ‌های آماده در فرآیند موردنظر قرار گیرد و تا زمانی که پردازنده در تملک فرآیند باشد و یا تا قبل از پایان برش زمانی مربوط به فرآیند، نخ‌های یک فرآیند از پردازنده بهره‌مند می‌شوند و به محض مسدود شدن یک نخ و یا پایان برش زمانی یک فرآیند، یا اتمام فرآیند، پردازنده به فرآیند بعدی تعلق می‌گیرد.

توجه: در این روش نخ ماهیت منطقی دارد و از دید کاربر فقط وجود دارد، در واقع از نظر سیستم عامل ماهیت فیزیکی ندارد، بنابراین نخ کاربر در این روش همانند یک تابع در فرآیند می‌باشد که از رجیستر و پشته مختص به خود نیز بهره‌مند می‌باشد.

توجه: در این روش چند نخ سطح کاربر به یک فرآیند تک نخی نگاشت می‌شود که به آن مدل چند به یک (Many to one model) گفته می‌شود. شکل زیر گویای مطلب است:



Many-to-one model.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، بین پدران خانواده‌ها تقسیم گردد و این پدران خانواده‌ها باشند که تصمیم بگیرند به هر عضو خانواده چه مقدار نقدینگی تعلق بگیرد. در این روش فقط پدران شماره حساب مختص به خود را دارند. اما اگر یکی از اعضای خانواده خطایی انجام دهد و محکوم گردد، آنکه تمام اعضای خانواده برای مدتی از خدمات دولت محکوم می‌گردند، زیرا در این مدل، دولت فقط پدران خانواده‌ها را می‌شناسد و از اعضای خانواده اطلاعی ندارد. بنابراین حساب پدر خانواده برای مدتی مسدود می‌گردد.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند کاربر، کاملاً در سطح کاربر و با سربار بسیار

ناچیز (در حد فراخوانی نخ بعدی) و بدون تغییر حالت پردازنده بـه مـد هسـته پردازنـده، توسـط زمان‌بند چندنخی در سطح کاربر (برنامـه‌هـای کـاربر) انجـام مـی‌گـردد. فقـط کافیسـت محتـوای ثبات‌های CPU برای نخ فعلی، ذخیره شده و سپس مقادیر قبلا ذخیره شده مربوط به نخ بعدی کـه به آن سوییچ می‌شود، بارگذاری مجدد شوند. و نیاز به دخالت هسته و تله‌های سـنگین (سـویچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعـدی و سـوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB) ندارد.

توجه: نخ‌های سطح کاربر، رابطه و تبادل داده سریع‌تر و سبک‌تری نسـبت بـه نخ‌هـای سـطح هسته دارند. چون نیاز به به دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مـد هسـته بـه مـد کـاربر شامل تغییر MMU و TLB) ندارد.

توجه: به دلیل آنکه نخ‌های یک فرآیند کاربر، تماماً در فضای کاربر مدیریت می‌شوند، اگر نـخ موجود در یک فرآیند کاربر، یک فرآیند سیستمی مسدودکننده را اجرا نماید، هسته سیستم عامل نه تنها آن نخ، بلکه کل فرآیند کاربر را که شامل تمام نخ‌های دیگر می‌باشـد، مسـدود مـی‌کنـد، زیـرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد. در واقع در این روش هسته سیسـتم عامل نخ‌ها را همانند توابع داخل یک فرآیند می‌بیند، یعنی هسته سیستم عامـل، یـک فرآینـد چنـد نخی سطح کاربر را، مانند یک فرآیند تک نخی اما دارای چند تابع مختلف می‌بیند!

توجه: یک نقص صفحه (صفحه‌بندی در حافظه مجازی) مربوط به یک نخ، باعث می‌شود کل فرآیند مسدود شود. زیرا هسته سیستم عامل خبری از نخ‌های داخل فرآیند کاربر ندارد.

توجه: زمان‌بند سطح کاربر برای زمان‌بندی نخ‌ها نمی‌تواند از Timer سیستم اسـتفاده نمایـد و باید به طور دستی زمان‌بندی شود.

توجه: این راه‌کار، منجر به عدم امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) نخ‌های داخل یک فرآیند کاربر در حالت انسداد یک نخ داخل یک فرآیند کاربر می‌گردد.

البته اگر نخی داخل یک فرآیند کاربر مسدود نگردد، امکان هـم‌رونـدی میـان نـخ‌هـای داخـل فرآیند کاربر برقرار است. مانند یک تیم فوتبال ۱۱ نفره که اگـر بـازیکنی مرتکـب خطـا گـردد، از آن‌جا که داور فقط نام تیم را می‌شناسد و نه تک‌تک بازیکنان تـیم را، آن‌گـاه کـل تـیم را جریمـه، اخراج و مسدود می‌کند. اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطایی نگردد، واضـح اسـت کـه هم‌روندی برقرار است.

توجه: فرض کنید یک کیک داریم که آن را به چهار قسمت مساوی تقسیم کرده‌ایـم، همچنـین فرض کنید خوردن هر بخش کیک یک ساعت زمان بخواهد، اگر یک نفر بخواهد تمام ایـن کیـک را بخورد، پس چهار ساعت طول می‌کشد اگر چهار نفر بخواهند تمام این کیک را بخورند و به هر نفر یک بخش کیک داده شود، آنگاه خوردن تمام کیک به‌طور موازی ۱ ساعت طول خواهد کشید. در روش سطح کاربر، یک فرآیند چند نخی کاربر نمی‌تواند از امتیازات چند پردازنده‌ای بهره ببرد،

زیرا در روش سطح کاربر، هسته سیستم عامل در هر لحظه فقط یک پردازنده را در اختیار نخ‌های یک فرآیند کاربر قرار می‌دهد. حتی اگر چند پردازنده موجود باشد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر نیست. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند کاربر وجود ندارد.

توجه: نرم‌افزارهای POSIX P-threads و Mach C-thread یه عنوان یک بسته نرم‌افزاری، می‌تواند جهت مدیریت نخ‌ها و زمان‌بند چند نخی در سطح کاربر مورد استفاده قرار گیرند.

توجه: در این روش تخصیص منابع و زمان‌بندی پردازنده، بر روی فرآیندها انجام می‌شود. همچنین زمان‌بندی نخ‌های سطح کاربر، بر عهده زمان‌بند چندنخی سطح کاربر خواهد بود.

توجه: سیستم عامل سولاریس، مدل چند به یک را پیاده سازی می‌کند.

نخ‌های سطح هسته: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ در مد هسته و توسط هسته سیستم عامل انجام می‌شود. در این روش هسته از وجود نخ اطلاع دارد. در این روش فقط زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته وجود دارد و زمان‌بند چند نخی در سطح کاربر در این روش مورد استفاده قرار نمی‌گیرد. در واقع هسته سیستم عامل نه تنها فرآیندها را می‌شناسد بلکه از وجود نخ‌های داخل یک فرآیند نیز در صورت وجود نخ آگاه است. در واقع مدیریت نخ‌ها و زمان‌بند چندنخی (نخ‌های فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل) در سطح هسته، توسط هسته سیستم عامل انجام می‌گردد و کاربر هیچ دیدی از این کار ندارد. انگار که اجتماع تمام نخ‌های فرآیندهای سیستم عامل و کاربر را در نظر بگیرید، حال بر روی تک نخ‌ها بر اساس یک الگوریتم خاص حرکت کنید. بنابراین زمان‌بند چند نخی در سطح هسته، براساس الگوریتم مشخصی مثلاً نوبت چرخشی پردازنده را در اختیار یکی از نخ‌های آماده قرار می‌دهد. توجه کنید در این روش پردازنده دیگر در تملک فرآیند نیست بلکه در تملک نخ‌ها است. در واقع تا زمانی که پردازنده در تملک یک نخ باشد و تا قبل از مسدود شدن نخ، و یا تمام شدن نخ و یا پایان برش زمانی مربوط به نخ، نخ می‌تواند از پردازنده بهره ببرد و به محض مسدود شدن یک نخ، یا پایان برشی زمان مربوط به نخ یا اتمام نخ، پردازنده می‌تواند به نخ بعدی که ممکن است، نخ بعدی، نخی هم‌خانواده با نخ قبلی در یک فرآیند باشد، یا نخی در یک فرآیند دیگر باشد، تعلق بگیرد. در واقع زمان‌بند چند نخی در سطح هسته سیستم عامل تعیین می‌کندکه نخ بعدی که باید شروع به کار کند متعلق به همان فرآیند باشد و یا از یک فرآیند دیگر انتخاب شود.

توجه: برای انجام زمان‌بندی چند نخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند. مجدداً تأکید می‌کنیم که در این حالت هر نخ TCB خاص خود را دارد. به عبارت دیگر هر نخ رجیستر و پشته مختص به خود را دارد.

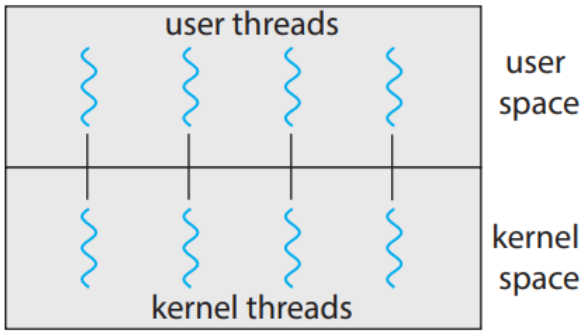
توجه: در این روش نخ ماهیت فیزیکی دارد، و از دید کاربر و سیستم عامل وجود دارد،

بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، بـه عبـارت دیگـر هـر نـخ TCB مختص به خود را دارد.

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند ایجاد (بارگذاری TCB مخـتص بـه نـخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بنـد چند نخی در سطح هسته سیستم عامل انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولـت حاصـل از منابع کشور، بدون اعمال هیچ‌گونه اولویت‌بندی بین تک تک اعضاء یـک کشـور تقسـیم گـردد (حال این اعضاء شهروند عام باشد یا خـاص) در ایـن روش هـر یـک از اعضـای کشـور شـماره حساب مختص به خود را دارند. اما اگر یکی از افراد کشور خطایی انجام دهـد و محکـوم گـردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خـانواده آن فرد همچنان خدمات ارائه می‌دهد. زیرا دولت از مشخصات تک تک اعضاء جامعه آگاه است.

توجه: در این روش هر نخ سطح کاربر به یک نخ سطح هسته نگاشت می‌شود که بـه آن مـدل یک به یک (one to one model) گفته می‌شود. شکل زیر گویای مطلب است:



One-to-one model.

توجه: عمل تعویض متن، مابین نخ‌های یک فرآیند (فرآیندهای کـاربر یـا فرآینـدهای سیسـتم عامل) در سطح هسته سیستم عامل، در مد هسته سیستم پردازنـده توسـط زمـان‌بنـد چنـدنخی در سطح هسته سیستم عامل انجام می‌گیرد. بنابراین زمان‌بندی نخ‌های سطح هسـته بـه دلیـل دخالـت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض مـتن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییـر MMU و TLB)، کنـدتر از ریسه‌های سطح کاربر است. در نتیجه نخ‌های هسته به طور کلی فرایند ایجـاد، حـذف و مـدیریت کُندتری نسبت به نخ‌های کاربر دارند.

توجه: نخ‌های سطح هسته رابطه و تبادل داده کندتر و سنگین‌تری نسبت به نخ‌های سطح کاربر دارند. چون نیاز به به دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییـر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته بـه مـد کـاربر شـامل

تغییر MMU و TLB) دارد.

توجه: به دلیل آنکه نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل)، تماماً در فضای هسته سیستم عامل مدیریت می‌شوند و هسته سیستم عامل از وجود نخ‌های یک فرآیند آگاه است، اگر نخ موجود در یک فرآیند، یک فراخوان سیستمی مسدود کننده را اجرا نماید، هسته سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌توانند از پردازنده بهره ببرند.

توجه: یک نقص صفحه (صفحه‌بندی در حافظه مجازی) مربوط به یک نخ، باعث نمی‌شود کل فرآیند مسدود شود. زیرا هسته سیستم عامل از نخ‌های داخل فرآیند کاربر باخبر است.

توجه: زمان‌بند سطح هسته برای زمان‌بندی نخ‌ها می‌تواند از Timer سیستم استفاده نماید.

توجه: این راه کار، منجر به امکان هم‌روندی (در سیستم‌های تک پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف و یا میان نخ‌های داخل یک فرآیند می‌شود. البته اگر نخی مسدود نگردد، درجه همروندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند. مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، همروندی یا توازی همچنان برقرار است، چون داور تک‌تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ یک از بازیکنان تیم مرتکب خطا نگردند، واضح است که همروندی و توازی بالاتر هم خواهد بود.

توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش سطح هسته، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در روش سطح هسته، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. بنابراین در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.

توجه: در این روش، منابع به فرآیندها اختصاص می‌یابد ولی زمان‌بندی پردازنده، بر روی نخ‌ها انجام می‌گیرد و زمان‌بندی نخ‌های سطح کاربر و نخ‌های سطح هسته، برعهده زمان‌بند چندنخی سطح هسته می‌باشد.

توجه: سیستم عامل لینوکس، خانواده سیستم عامل ویندوز و سولاریس ۹ مدل یک به یک را پیاده‌سازی می‌کنند.

نخ‌های ترکیبی: در این روش اداره نخ شامل ایجاد، حذف و زمان‌بندی نخ به طور ترکیبی از سطح کاربر و سطح هسته انجام می‌شود.

این روش از اجتماع دو روش سطح کاربر و هسته ابداع گردیده است. در این روش علاوه بر

زمان‌بند پردازنده و زمان‌بند چندنخی در سطح هسته سیستم عامل، زمان‌بند چند نخی نیز در سطح کاربر برای زمان‌بندی و اولویت‌دهی نخ‌های فرآیند کاربر وجود دارد. در واقع زمان‌بندی نخ‌های فرآیندهای سیستم عامل، توسط هسته سیستم عامل و زمان‌بندی نخ‌های فرآیندهای کاربر، توسط برنامه کاربر انجام می‌گردد. که این امر منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. در این روش، زمان‌بند چند نخی در سطح کاربر، نخ‌های کاندید خود را از میان نخ‌های متعدد در فرآیندهای مختلف کاربر براساس یک الگوریتم خاص انتخاب و تحویل یک زمان‌بند چند نخی در سطح هسته می‌دهد. در واقع انتخاب نخ‌های کاندید (زمان‌بندی) در فرآیندهای کاربر به خود کاربر واگذار شده است که همانطور که گفتیم منجر به اولویت‌بندی نخ‌های فرآیندهای کاربر می‌گردد. حال اجتماع حاصل از نخ‌های کاندید فرآیندهای کاربر و نخ‌های فرآیندهای سیستم عامل توسط زمان‌بند چندنخی هسته سیستم عامل براساس یک الگوریتم خاص زمانبندی می‌شود.

از توضیحات فوق این مهم برداشت می‌شود که هسته سیستم عامل باید این قابلیت را داشته باشد که کاربر بتواند نخ‌های فرآیندهای سطح خود را به هسته سیستم عامل معرفی کند. بنابراین در این روش هسته سیستم عامل نه تنها فرآیندهای کاربر را می‌شناسد، بلکه از وجود نخ‌های کاربر داخل فرآیندهای کاربر نیز در صورت وجود نخ آگاه است.

توجه: برای معرفی نخ‌های کاندید فرآیندهای سطح کاربر به زمانبند چندنخی در سطح هسته سیستم عامل، علاوه بر جایگاه‌های مخصوص نخ‌های فرآیندهای سیستم عامل در زمان‌بند چندنخی در سطح هسته سیستم عامل، تعدادی جایگاه، ویژه نخ‌های سطح کاربر نیز، در زمانبند چندنخی در سطح هسته سیستم عامل در نظر گرفته شده است، به این جایگاه ویژه که **محیط اجرای نخ** نیز نامیده می‌شود، LWP گفته می‌شود. در واقع هر نخ کاندید انتخاب شده توسط زمانبند چندنخی در سطح کاربر، پس از معرفی به زمانبند چندنخی در سطح هسته، در یکی از جایگاه‌های ویژه که همان LWP است، جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد.

توجه: LWP سرواژه‌ی عبارت Light Weight Process و به معنی فرآیند سبک وزن است.

توجه: هنگامی که یک نخ به زمان‌بند چندنخی در سطح هسته معرفی می‌گردد و در ادامه در یک LWP جهت زمان‌بندی توسط زمان‌بند چندنخی در سطح هسته، قرار می‌گیرد، در طول حیات خود ممکن است، در LWP های متفاوتی بخش‌هایی از اجرای خود را طی کند، مثلاً یک نخ در صورت رسیدن به عملیات ورودی و خروجی، جایگاه خود یعنی LWP را واگذار می‌کند و در اجرای بعدی پس از پایان عملیات ورودی و خروجی ممکن است به یک LWP دیگر منتسب شود. توجه کنید که LWP محیط اجرای نخ می‌باشد.

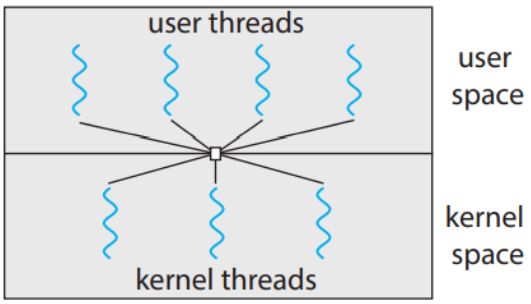
توجه: برای انجام زمان‌بندی چندنخی، هسته سیستم عامل باید علاوه بر جدول فرآیندها، یک جدول نخ (شبیه جدول فرآیند) داشته باشد که اطلاعات تمامی نخ‌های موجود در سیستم را نگهداری کند.

توجه: در این روش نخ ماهیت فیزیکی دارد و از دید کاربر و سیستم عامل وجود دارد، بنابراین در این روش، هر نخ، رجیستر و پشته مختص به خود را دارد، یه عبارت دیگر هر نخ TCB مختص به خود را دارد.

توجه: عملیات مرتبط با نخ‌های سطح کاربر، مانند (بارگذاری TCB مختص به نخ) و پایان دادن (ذخیره‌سازی TCB مختص به نخ) بر عهده هسته سیستم عامل است و توسط زمان‌بند چندنخی سطح هسته انجام می‌گردد.

توجه: مدل غیرکامپیوتری این روش نیز وجود دارد، مانند حالتی که درآمدهای دولت حاصل از منابع کشور، با اعمال نوعی اولویت‌بندی بین برخی از اعضاء یک کشور تقسیم گردد (حال این اعضاء شهروند عام باشند یا خاص) در این روش هریک از اعضاء اولویت‌دار و منتخب کشور شماره حساب مختص به خود را دارند. اما اگر یکی از اعضاء خانواده خطایی انجام دهد و محکوم گردد، آنگاه فقط همان فرد برای مدتی از خدمات دولت محروم می‌گردد و دولت به بقیه اعضاء خانواده آن فرد، همچنان خدمات ارائه می‌دهد، زیرا دولت از مشخصات تک تک اعضاء آن خانواده آگاه است.

توجه: در این روش چند نخ سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته (LWP ها) نگاشت می‌شود که به آن مدل چند به چند (Many to Many model) گفته می‌شود.



Many-to-many model.

توجه: عمل تعویض متن مابین نخ‌های یک فرآیند (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل، در مد هسته پردازنده توسط زمانبند چندنخی در سطح هسته سیستم عامل انجام می‌گردد و عمل تعویض متن مابین فرآیندها (فرآیندهای کاربر یا فرآیندهای سیستم عامل) در سطح هسته سیستم عامل و در مد هسته پردازنده توسط زمانبند پردازنده انجام می‌گردد.

توجه: به دلیل آنکه نخ‌های کاندید فرآیندهای سطح کاربر، به هسته سیستم عامل معرفی می‌گردد و هسته سیستم عامل از وجود نخ‌های کاندید فرآیندهای سطح کاربر آگاه است، اگر یک نخ کاندید موجود در یک فرآیند کاربر، یک فراخوان سیستمی مسدودکننده را اجرا نماید، هسته

سیستم عامل فقط آن نخ مربوطه را مسدود می‌کند و نخ‌های دیگر هم‌خانواده با آن نخ مسدود شده، همچنان می‌تواند از پردازنده بهره ببرند. این راه‌کار، منجر به امکان هم‌روندی (در سیستم‌های تک‌پردازنده‌ای) و امکان توازی (در سیستم‌های چندپردازنده‌ای) میان فرآیندهای مختلف کاربر و یا میان نخ‌های داخل یک فرآیندکاربر می‌شود. البته اگر نخی مسدود نگردد، درجه هم‌روندی و توازی بالاتر هم خواهد رفت، زیرا در اینصورت همه نخ‌ها به طور هم‌روند یا موازی در حال حرکت هستند.

مانند یک تیم فوتبال ۱۱ نفره که اگر بازیکنی مرتکب خطا گردد، هم‌روندی یا توازن همچنان برقرار است، چون داور تک تک بازیکنان را می‌شناسد و فقط بازیکن خاطی را مسدود، جریمه و اخراج می‌کند و بقیه بازیکنان تیم به بازی خود ادامه می‌دهند، اما اگر هیچ‌یک از بازیکنان تیم مرتکب خطا نگردد، واضح است هم‌روندی و توازی بالاتر هم خواهد رفت.

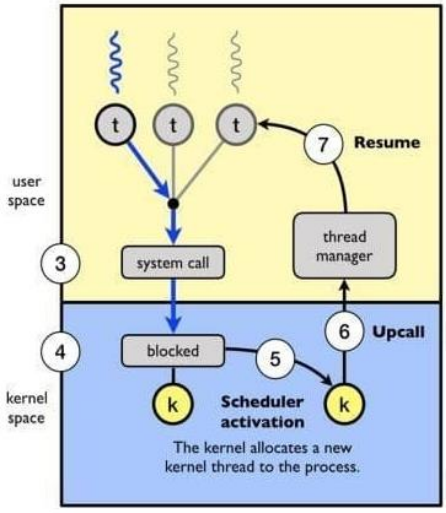
توجه: همانند نخ‌های سطح کاربر، همروندی و توازی برای نخ‌های سطح هسته سیستم عامل نیز، در حالت انسداد یک نخ یا عدم انسداد یک نخ، برقرار است.

توجه: مثال کیک مطرح شده را مجدداً به یاد آورید، در روش ترکیبی، یک فرآیند چندنخی (فرآیندهای کاربر یا فرآیندهای سیستم عامل) می‌تواند از امتیازات چندپردازنده‌ای بهره ببرد. در روش ترکیبی، هسته سیستم عامل می‌تواند در هر لحظه چندین پردازنده را در اختیار نخ‌های یک فرآیند قرار دهد. یعنی در این روش خوردن کیک بخش‌بندی شده به صورت چند نفری امکان‌پذیر است. بنابراین در این روش امکان پردازش موازی نخ‌های داخل یک فرآیند وجود دارد.

توجه: نخ‌ها به دو صورت ممکن است مسدود شوند، یکی با فراخوان سیستمی و توسط هسته سیستم عامل و دیگری با فراخوان تابعی برای خوابیدن از کتابخانه سطح کاربر، بنابراین در روش ترکیبی علاوه بر اداره نخ‌ها در سطح هسته، کتابخانه سطح کاربر نیز می‌تواند بدون دخالت هسته اولویت اجرای نخ‌ها را در زمان اجرا جا به جا کند. برای مثال اگر نخ درحال اجرای فعلی بر روی یک LWP بخواهد بخوابد تا نخ دیگری عملیاتی را انجام داده و سپس او را بیدار کند، می‌تواند با فراخوانی تابع کتابخانه در سطح کاربر بخوابد. کتابخانه سطح کاربر ثبات‌ها و داده‌های نخی که کاندید خوابیدن است را در جدول نخ ذخیره می‌کند و سپس، نخ دیگری را برای اجرا انتخاب می‌کند و ثبات‌ها و داده‌هایش از جدول نخ به پردازنده بارگذاری می‌شود. بنابراین نخ جدید بر روی همان LWP نخ قبلی اجرا می‌شود. دقت کنید که زمان‌بند سطح هسته متوجه این تعویض نخ سطح کاربر نمی‌شود.

توجه: هسته به یک فرآیند کاربر تعدادی LWP اختصاص می‌دهد تا نخ‌های خود را بر روی آنها زمان‌بندی کند. همچنین سطح هسته موظف است وقوع رویدادها را به سطح کاربر اطلاع دهد. شروع این ارتباط بین کتابخانه نخ سطح کاربر و سطح هسته از فعال‌سازی زمان‌بند (Scheduler Activation) است و ادامه آن فراخوان رو به بالا (Upcall) است. در کتابخانه نخ سطح کاربر برای اداره Upcall ها، Upcall Handler وجود دارد که داخل LWP اجرا می‌شود. هرگاه نخ درحال اجرا

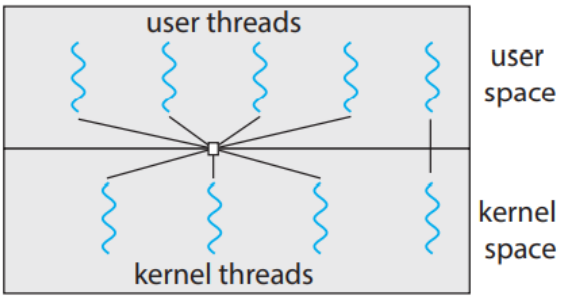
بر روی یک LWP از طریق هسته مسدود شود، هسته یک Upcall صادر می‌کند و مشخص می‌کند که کدام نخ را مسدود کرده است. در این لحظه هسته یک LWP دیگر را در اختیار سطح کاربر می‌گذارد تا Upcall Handler بر روی آن اجرا شده و پس از ذخیره وضعیت نخ مسدود شده در جداول فضای کاربر، یک نخ جدید را برای اجرا بر روی LWP جدید زمان‌بندی کرده و وضعیت آنرا از جداول مربوطه به CPU بارگذاری کند. همچنین هرگاه رویدادی که نخ مسدود منتظر آن بوده است رخ دهد برای مثال تکمیل ورودی و خروجی، هسته یک Upcall دیگر به کتابخانه نخ کاربر می‌دهد تا آنرا از آماده اجرا بودن نخ مورد نظر آگاه سازد.



در **سیستم‌های بی‌درنگ** به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسیع و گسترده اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها داده شود. در روش چند به چند، سطح هسته وجود دارد و به تبع همروندی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمام مهلت زمانی تولید شود. اما یک ریسک دارد اینکه در روش چند به چند برنامه‌نویس به تعداد دلخواه نخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتر شود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یا مساوی از نخ‌های سطح هسته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشته باشد، اگر چهار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمان پاسخ طولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. هرچند که در روش چند به چند، سطح کاربر وجود

دارد، که این امر منجر به این می‌شود که اجـازه تعیـین اولویت‌هـا، مهلت‌هـا و کنتـرل زمان‌بنـدی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود.

توجه: روش دو سطحی به نوعی روش **چند به چند پلاس** اسـت، یعنـی روش دوسـطحی بهبـود یافته روش چند به چند است، در واقع روش دوسطحی علاوه بر اینکه روش چنـد بـه چنـد را در خود دارد، جهت بهبود کارایی روش یک به یک را نیز در خـود قـرار داده اسـت. در سیسـتم‌های بی‌درنگ به خصوص بی‌درنگ سخت لازم است به کاربر به طور وسـیع و گسـترده اجـازه تعیـین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآینـدها و نخ‌هـا داده شـود. در روش دوسـطحی (Two-Level Model) یا همان چند به چند پـلاس، سـطح هسـته وجـود دارد و بـه تبـع همرونـدی (در سیستم‌های تک‌پردازنده‌ای) پس از مسدود شدن یک نخ و توازی (در سیستم‌های چندپردازنده‌ای) وجود دارد که این امر منجر به این می‌شود که زمان پاسخ کوتاه باشد و پاسخ قبل از اتمـام مهلـت زمانی تولید شود. اما یک ریسک دارد اینکه در روش چند به چند برنامه‌نویس به تعداد دلخواه نـخ ایجاد می‌کند، اما ممکن است تعداد LWP کمتری به آنها منتسب شود تا سربار هسته کمتـر شـود. زیرا در روش چند به چند نخ‌های سطح کاربر به تعداد کم‌تر یـا مسـاوی از نخ‌هـای سـطح هسـته نگاشت می‌شود. برای مثال اگر یک فرآیند پنج درخواست (نخ) همزمان داشـته باشـد، اگـر چهـار LWP داشته باشیم، یکی از درخواست‌ها (نخ‌ها) تا بازگشت نتیجه یکی از چهار درخواست (نـخ) دیگر از سطح هسته به جریان نخواهد افتاد و این یعنی تاخیر و ممکن است زمـان پاسـخ طـولانی شود و به تبع این احتمال وجود دارد که پاسخ بعد از اتمام مهلت زمانی تولید شود. اما از آنجا کـه روش دوسطحی علاوه بر داشتن روش چند به چند، روش یک بـه یـک را نیـز دارد، می‌توانـد در مواقع لزوم برای پاسخ سریع از روش یک به یک نیز استفاده نماید، زیرا در روش یک به یک هـر رشته نخ کاربر به یک رشته نخ هسته نگاشت می‌شود. شکل زیر گویای مطلب است:



Two-level model.

همچنین در روش دوسطحی (Two-Level Model) یا همان چند بـه چنـد پـلاس، سـطح کـاربر وجود دارد، که این امر منجر به این می‌شود که اجازه تعیین اولویت‌ها، مهلت‌ها و کنترل زمان‌بندی فرآیندها و نخ‌ها به طور وسیع و گسترده به کاربر داده شود. روش دوسطحی در سیستم عامل‌هایی

نظیر IRIX، HP-UX و True64nIx پشتیبانی می‌شود. نسخه‌های قبل از Solaris9 نیز مدل دوسطحی را پشتیبانی می‌کنند، اما Solaris9 از روش یک به یک استفاده می‌کند.

صورت سوال به این شکل است:

کدام گزینه درباره ریسه‌های (Threads) سطح کاربر و سطح هسته درست است؟

۱) زمان‌بندی ریسه‌های سطح هسته سریعتر از ریسه‌های سطح کاربر است.

گزینه اول نادرست است، زیرا زمان‌بندی ریسه‌های سطح هسته کندتر از ریسه‌های سطح کاربر است. زمان‌بندی نخ‌های سطح هسته به دلیل دخالت هسته و تله‌های سنگین (سوییچ از مد کاربر به مد هسته شامل تغییر MMU و TLB، تعویض متن از نخ فعلی به نخ بعدی و سوییچ از مد هسته به مد کاربر شامل تغییر MMU و TLB)، کندتر از ریسه‌های سطح کاربر است. در نتیجه نخ‌های هسته به طور کلی فرایند ایجاد، حذف و مدیریت کُندتری نسبت به نخ‌های کاربر دارند.

۲) ریسه‌های سطح کاربر و سطح هسته از طریق فراخوانی سیستمی (System Calls) به هم سرویس می‌دهند.

گزینه دوم نادرست است، زیرا زمان‌بندی نخ‌های سطح هسته نیاز به فراخوانی سیستمی و دخالت هسته و تله‌های سنگین دارد. اما سرویس ریسه‌های سطح کاربر و سطح هسته به هم ارتباطی به فراخوان سیستمی ندارد. فراخوان سیستمی برای صدا زدن یک تابع از سیستم عامل مورد استفاده قرار می‌گیرد و نه صدا زدن یک نخ از یک فرآیند. اصولا نخ توسط زمان‌بند سطح کاربر یا هسته زمان‌بندی و اجرا می‌شود و صدا زده نمی‌شود.

۳) ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس هم دسترسی داشته و می‌توانند در فضای آدرس هم بنویسند.

گزینه سوم درست است، زیرا اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

۴) ریسه‌های سطح هسته به فضای آدرس ریسه‌های سطح کاربر دسترسی دارند، اما ریسه‌های سطح کاربر به فضای آدرس ریسه‌های سطح هسته دسترسی ندارند.

گزینه چهارم نادرست است، زیرا اشتراکات نخ‌های داخل یک فرآیند شامل سگمنت داده (داده سراسری)، فضای آدرس، فایل‌های باز و اختلاف نخ‌های داخل یک فرآیند شامل شمارنده برنامه (PC)، رجیسترها و پشته می‌باشد. از آنجاکه نخ‌های داخل یک فرآیند، از برخی منابع به صورت مشترک استفاده می‌کنند. بنابراین ریسه‌های سطح کاربر و سطح هسته می‌توانند به فضای آدرس مشترک هم دسترسی داشته و می‌توانند در فضای آدرس مشترک هم بنویسند.

تست‌های فصل پنجم: مدیریت حافظه اصلی

۱۰۲- در یک سیستم عامل از صفحه‌بندی وارون (Inverted Paging) استفاده می‌شود. کدام گزینه در مورد جدول صفحه درست است؟

(مهندسی کامپیوتر - دولتی ۹۹)

۱) یک جدول صفحه عمومی که بر اساس شماره قاب مرتب شده است.
۲) یک جدول صفحه عمومی که بر اساس شماره پردازه مرتب شده است.
۳) یک جدول صفحه عمومی که بر اساس شماره آدرس مجازی مرتب شده است.
۴) هر پردازه دارای یک جدول صفحه اختصاصی است که بر اساس شماره قاب مرتب شده است.

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل پنجم: مدیریت حافظه مجازی

۱۰۲- گزینه (۱) صحیح است.

در تکنیک صفحه‌بندی، برای هر فرآیند یک جدول صفحه معمولی تشکیل می‌شود که در آن به ازای همه صفحه‌های یک فرآیند، درایه وجود دارد و هر درایه مشخص می‌کند که کدام قاب فیزیکی به این صفحه اختصاص یافته است. در این روش وقتی فرآیندها بزرگ باشند، هزینه نگهداری جداول صفحه بسیار زیاد می‌شود، در ضمن به ازای هر فرآیند نیز باید یک جدول صفحه داشته باشیم. به عبارتی وقتی تعداد و اندازه فرآیندها بزرگ شود، این روش برای ترجمه آدرس مجازی به فیزیکی مقرون به صرفه نیست.

برای حل این مشکل **جدول صفحه چندسطحی** و **جدول صفحه معکوس** ابداع شده است. جدول صفحه معکوس از نظر سربار حافظه بهتر از روش جدول صفحه چندسطحی است و حالت حداقلی است. اغلب سیستم‌های کامپیوتری، فضای آدرس منطقی(مجازی) بزرگی را پشتیبانی می‌کنند، مانند کامپیوترهایی که آدرس‌های 32 یا 64 بیتی را پشتیبانی می‌کنند، در چنین محیط‌هایی جدول صفحه معمولی بسیار بزرگ خواهد شد. برای مثال یک سیستم را با 32 بیت فضای آدرس منطقی(مجازی) در نظر بگیرید. اگر اندازه صفحه در این سیستم برابر با 4KB (2^{12}) باشد، در اینصورت جدول صفحه شامل بیش از یک میلیون درایه خواهد بود. $\left(2^{32} / 2^{12} = 2^{20}\right)$. فرض کنید هر درایه جدول صفحه معمولی، شامل 4 بایت باشد، در اینصورت هر فرآیند به $\left(4 \times 2^{20}\right)$ 4MB فضای آدرس فیزیکی فقط برای نگهداری جدول صفحه معمولی نیاز دارد. همچنین مشخص است که با توجه به محدودیت اندازه قاب (4KB) نمی‌توان این جدول صفحه معمولی را بصورت پیوسته درون یک قاب جا داد. 4MB اندازه جدول صفحه معمولی در فضای 4KB مربوط به یک قاب جا نمی‌شود. یک راه حل تقسیم جدول صفحه معمولی به جداول صفحه جزئی و ایجاد جدول صفحه چندسطحی است، در این حالت جدول صفحه معمولی نیز همانند فرآیند صفحه‌بندی می‌شود. راه حل دیگر استفاده از جدول صفحه معکوس است. در راه حل جدول صفحه معکوس به جای اینکه در جداول صفحه مربوط به هر فرآیند، به ازای هر صفحه مجازی یک درایه داشته باشیم، به ازای هر قاب در حافظه فیزیکی یک درایه در جدول صفحه معکوس

نگهداری می‌کنیم. در واقع به ازای هر قاب حافظه اصلی اینکه در حال حاضر کدام صفحه مربوط به کدام فرآیند در این قاب ذخیره شده است نگهداری می‌شود. بنابراین تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی(اصلی) است.

مثال: در یک سیستم صفحه‌بندی که دارای 34 بیت آدرس است 23 بیت اول برای شماره صفحه و 11 بیت بعدی برای آدرس‌دهی درون صفحه است. در یک سیستم با صفحه‌بندی معکوس (Inverted Page Table) با 128 مگابایت حافظه، جدول صفحه دارای چند خانه (درایه یا مدخل) است؟

پاسخ: تعداد درایه‌های جدول صفحه معکوس، مطابق رابطه زیر محاسبه می‌گردد:

تعداد قاب‌های حافظه فیزیکی = تعداد درایه‌های جدول صفحه معکوس

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

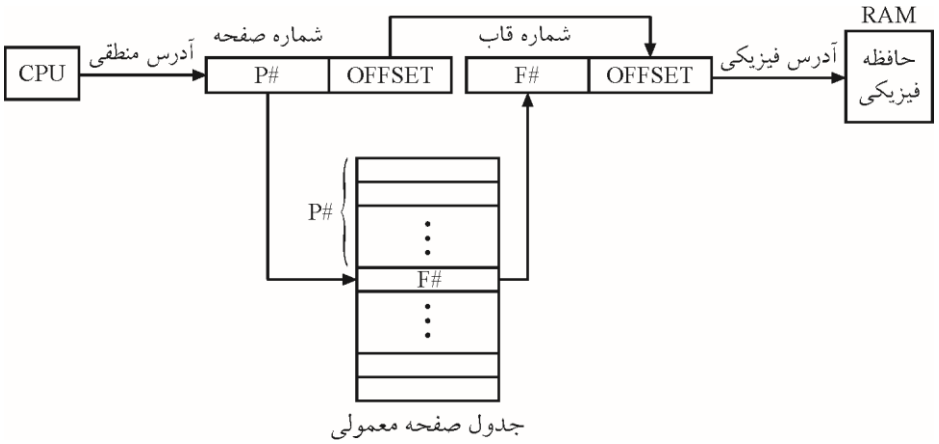
همانطور که گفتیم تعداد درایه‌های جدول صفحه معکوس برابر تعداد قاب‌های حافظه فیزیکی است، بنابراین تعداد درایه‌های جدول صفحه معکوس به صورت زیر خواهد بود:

$$\text{تعداد درایه‌های جدول صفحه معکوس} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = \frac{2^7 \times 2^{20}}{2^{11}} = 2^{16}$$

در یک عبارت ساده، هدف از استفاده از جدول صفحه معکوس، **کاهش** میزان حافظه فیزیکی مورد نیاز برای ترجمه آدرس مجازی به فیزیکی است. با استفاده از تکنیک جدول صفحه معکوس، مقدار زیادی در حافظه فیزیکی صرفه‌جویی می‌شود اما تبدیل آدرس مجازی به فیزیکی کندتر و وقت‌گیرتر می‌شود. در واقع **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم. زیرا جدول صفحه معکوس بر اساس #F اندیس شده است و نه مانند جدول صفحه معمولی که بر اساس #P اندیس شده است. در واقع وقتی یک فرآیند با PID مختص به خود به صفحه مجازی #P مراجعه می‌کند، سخت‌افزار دیگر نمی‌تواند از #P به عنوان اندیس جدول صفحه استفاده کند و صفحه فیزیکی را بیابد و از این جهت باید سرتاسر جدول صفحه معکوس (وارونه) را برای یافتن درایه (PID,P#) از آدرس مجازی (PID,P#,OFFSET) جستجو کند. اگر یک تطبیق پیدا شود، در اینصورت آدرس فیزیکی (F#,OFFSET) تولید خواهد شد و اگر هیچ تطبیقی یافته نشود، در اینصورت یک دسترسی آدرس غیر مجاز می‌باشد. ضمن اینکه این جستجو باید به ازای هربار دسترسی به حافظه انجام شود. همانطور که گفتیم در جدول صفحه معکوس **صرفه‌جویی** در مصرف حافظه فیزیکی را بدست می‌آوریم، اما **سرعت** تبدیل آدرس مجازی به فیزیکی را از دست می‌دهیم، یعنی سرعت نگاشت آدرس منطقی به فیزیکی **کاهش** می‌یابد. درواقع جدول صفحه معکوس **زمان** نگاشت آدرس

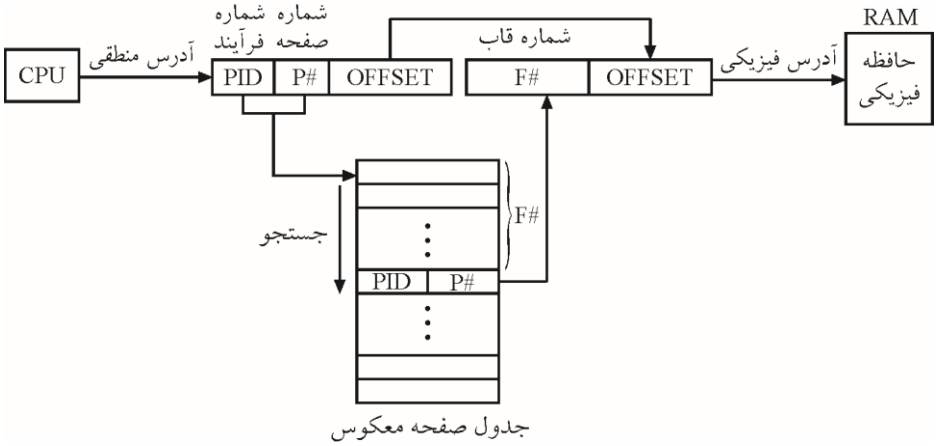
منطقی به آدرس فیزیکی را **افزایش** می‌دهد. هرچند موجب صرفه‌جویی در مصرف حافظه فیزیکی می‌شود. **بنابراین گزینه اول درست و گزینه‌ای دوم، سوم و چهارم نادرست است.**

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معمولی** به صورت زیر است:



جدول صفحه معمولی

نحوه ترجمه آدرس منطقی به فیزیکی، در سیستم جدول **صفحه معکوس** به صورت زیر است:



جدول صفحه معکوس

توجه: برای افزایش سرعت تبدیل آدرس مجازی به فیزیکی در جدول صفحه معکوس می‌توان از تکنیک TLB استفاده نمود.

توجه: جدول صفحه معکوس، اطلاعات کاملی در مورد فضای آدرس منطقی یک فرآیند را ندارد. در حالی که سیستم صفحه‌بندی مبتنی بر تقاضا به این اطلاعات کامل جهت پردازش نقص‌های صفحه نیازمند است. برای دسترسی به این اطلاعات کامل، یک **جدول صفحه خارجی**، یکی به ازای هر فرآیند، باید نگهداری شود. جدول صفحه خارجی شامل فیلدهای Valid، Present و آدرس صفحه مجازی دچار نقص صفحه شده بر روی رسانه ذخیره‌سازی است.

توجه: هنگامی که یک نقص صفحه رخ می‌دهد، ممکن است یک **نقص صفحه دیگر** برای بارگذاری **جدول صفحه خارجی** از رسانه ذخیره‌سازی به حافظه فیزیکی رخ دهد. بنابراین در جدول صفحه معکوس، زمان سرویس نقص صفحه(page fault) به دلیل ایجاد یک نقص صفحه دیگر، **افزایش** می‌یابد.

توجه: در سیستم‌هایی که از جدول صفحه معکوس‌شده استفاده می‌کنند، استفاده اشتراکی از صفحات میان فرآیندها در حالت عادی امکان‌پذیر نیست، در حالت کلی اشتراک به شکل چند به یک است، یعنی صفحه مشترک میان فرآیندها در یک قاب مشخص قرار می‌گیرد. که پیاده‌سازی حالت چند به یک در جدول صفحه معکوس درحالت ابتدایی امکان‌پذیر نیست. زیرا در جدول صفحه معکوس برای هر درایه آن فقط می‌توان شماره یک صفحه از یک فرآیند را قرار داد. البته توسط مکانیزم‌هایی می‌توان اشتراک‌گذاری صفحات میان فرآیندها را در جدول صفحه معکوس نیز پیاده‌سازی نمود. برای مثال به جدول صفحه معکوس اجازه دهیم برای هر درایه، بیش از یک شماره صفحه را آدرس‌دهی کند.

تست‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۳- در یک سیستم که تخصیص حافظه در آن بر اساس صفحه‌بندی (Paging) انجام می‌شود، اندازه هر فریم (Frame) برابر 2048 بایت است. شکل زیر، حافظه اصلی سیستم است، که قسمت‌های خاکستری فریم‌های تخصیص داده شده به یک پردازه هستند. اگر Internal Fragmentation برابر 900 بایت باشد، اندازه پردازه و External Fragmentation چند بایت است؟

(مهندسی کامپیوتر - دولتی ۹۹)

۱) اندازه پردازه برابر 12288 بایت و اندازه External Fragmentation برابر 14336 بایت است.
۲) اندازه پردازه برابر 12288 بایت و اندازه External Fragmentation برابرصفراست.
۳) اندازه پردازه برابر 11388 بایت و اندازه External Fragmentation برابر صفر است.
۴) اندازه پردازه برابر 6 بایت و اندازه External Fragmentation برابر7 بایت است.

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۳- گزینه (۳) صحیح است.

تکه تکه شدن داخلی (Internal Fragmentation) زمانی رخ می‌دهد که حافظه از قبل مرزبندی شده باشد و در هر قسمت بتوان فقط یک قطعه را قرار داد. در این حالت چون احتمالاً قطعه‌ای که برای یک جایگاه انتخاب شده است قدری از آن کوچکتر است، در انتهای جایگاه مقداری فضای بلااستفاده به‌وجود خواهد آمد. به این مشکل تکه تکه شدن داخلی گویند.

اما وقتی از قبل مرزبندی مشخصی برای حافظه در نظر نگیریم احتمال وقوع تکه تکه شدن

خارجی (External Fragmentation) وجود دارد. در این حالت احتمالاً فضاهای خالی و قابل استفاده حافظه به صورت جایگاه‌های کوچک، لابه لای فرآیندها پخش شده‌اند که مجموع این جایگاه‌های کوچک پراکنده احتمالاً نیاز ما را برآورده می‌کنند، اما چون همجوار نیستند بلااستفاده باقی می‌مانند.

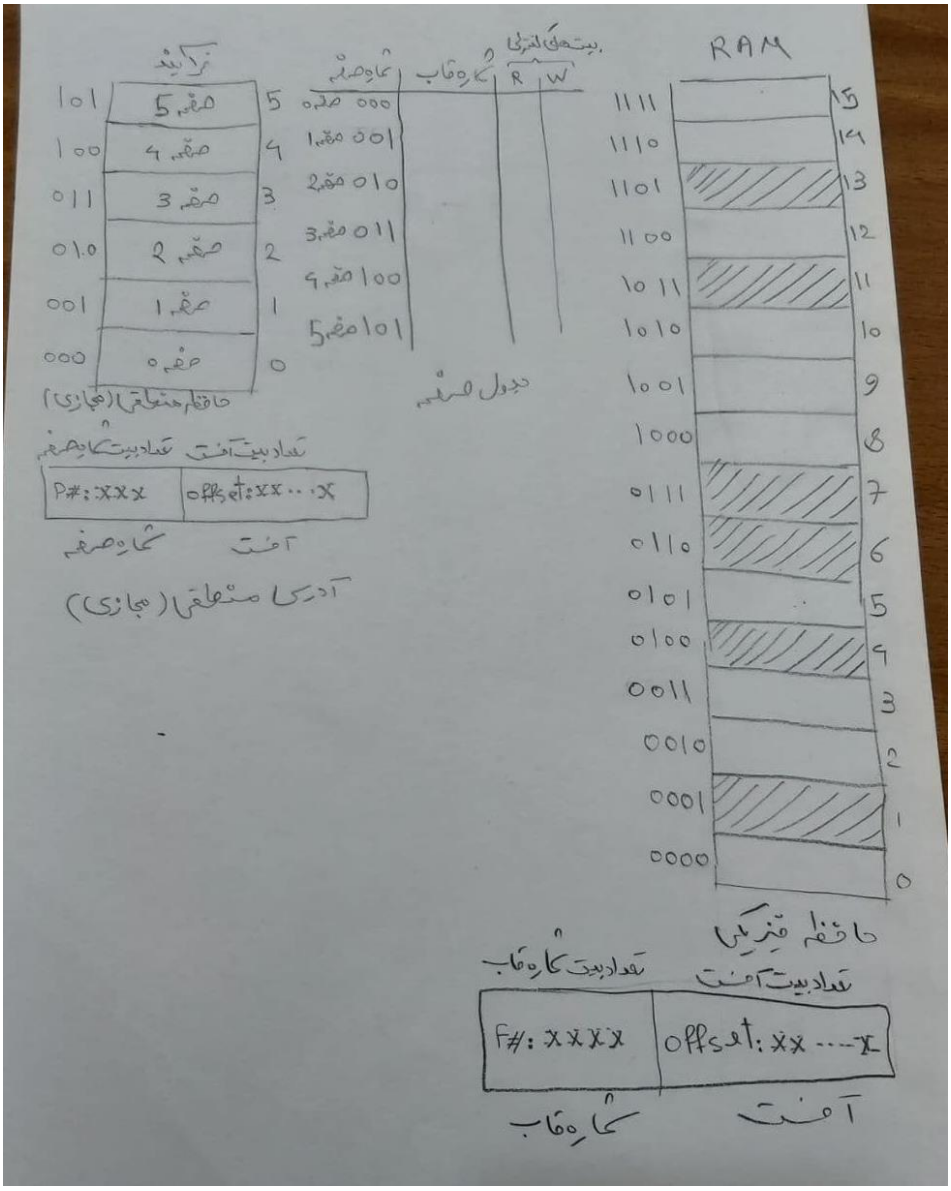
در «**تکنیک صفحه‌بندی**» حافظه به بخش‌هایی با اندازه‌ی یکسان به نام قاب (Frame) تقسیم می‌شود. از طرفی برنامه‌ها نیز به قسمت‌های مساوی و هم‌اندازه با قاب‌ها تقسیم می‌شوند که به آن‌ها صفحه (Page) می‌گویند. حال هنگامی که برنامه‌ای به حافظه منتقل می‌شود باید تمام صفحاتش به داخل قاب‌های خالی آورده شوند. در این حالت اصلاً نیازی نیست صفحات مربوط به یک فرآیند در قاب‌های همجوار قرار گیرند.

مزیت عمده این روش «**از بین بردن و صفر شدن تکه تکه شدن خارجی**» و به حداقل رساندن تکه تکه شدن داخلی می‌باشد، اما در عوض عملیات محاسبه آدرس‌ها و مدیریت این صفحات قدری هزینه‌بر و زمان‌گیر است.

توجه: برای پیاده‌سازی این روش و مدیریت صفحه‌ها و از همه مهم‌تر تبدیل و نگاشت آدرس‌ها باید یک جدول صفحه به ازای هر فرآیند در نظر گرفت. در واقع جدول صفحه‌ی هر فرآیند دارای یک درایه به ازای هر صفحه می‌باشد که مشخص می‌کند هر صفحه از یک فرآیند در کدام قاب حافظه نگهداری می‌شود.

توجه: نقطه ضعف اصلی مکانیزم صفحه‌بندی این است که اگر فقط احتیاج به ناحیه بسیار کوچکی از حافظه باشد، در این صورت مقداری از فضای حافظه تلف می‌شود، زیرا کوچکترین واحدی از حافظه که می‌توان آن را به استفاده‌کننده اختصاص داد، یک صفحه است.

توجه: مطابق اطلاعات مساله شکل زیر گویای مطلب است:



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

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

$$b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^6 = 3\text{bit} = \text{تعداد بیت شماره صفحه}$$

تعداد بیت شماره قاب = b = $\log_2$ تعداد قاب‌های حافظه فیزیکی = $\log_2^{16}$ = 4bit

تعداد بیت آفست = b = $\log_2$ اندازه صفحه یا اندازه قاب = $\log_2^{2048}$ = 11bit

اندازه جدول صفحه = تعداد درایه‌های جدول صفحه) × تعدادصفحات فرآیند 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 سطح جدول صفحه می‌باشد.

توجه: در صورت سوال مطرح شده است که اگر Internal Fragmentation برابر 900 بایت باشد، اندازه پردازه و External Fragmentation چند بایت است؟

توجه: مطابق فرض سوال قسمت‌های خاکستری فریم‌های تخصیص داده شده به یک پردازه هستند، بنابراین تعداد صفحات فرآیند برابر 6 عدد است.

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

اندازه فرآیند = تعداد صفحات فرآیند (اندازه قاب) × اندازه صفحه =6 ×2048 =12288 بایت

توجه: در ایده‌ی صفحه‌بندی تکه تکه شدن خارجی نداریم اما ممکن است قدری تکه تکه شدن داخلی داشته باشیم و آن هم در هر فرآیند و به ازای «**آخرین صفحه**» رخ می‌دهد. در واقع چون فرآیند می‌تواند هر طولی داشته باشد، ممکن است مضرب صحیحی از اندازه صفحه‌ها نباشد و آخرین صفحه قدری خالی بماند. به همین دلیل به طور متوسط نیم صفحه به ازای هر فرآیند تکه تکه شدن داخلی خواهیم داشت. بنابراین اگر مقدار 900 بایت تکه تکه شدن داخلی که آن هم در هر فرآیند و به ازای «**آخرین صفحه**» رخ می‌دهد را از مقدار حداکثر اندازه فرآیند یعنی 12288 بایت کسر کنیم به مقدار 11388 بایت که اندازه واقعی پردازه است می‌رسیم. به صورت زیر:

$$12288 - 900 = 11388$$

بنابراین پُرواضح است که گزینه سوم پاسخ سوال است.

تست‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۴- در یک سیستم صفحه‌بندی (Paging)، طول آدرس منطقی 19 بیت است. اگر تعداد صفحات موجود در فضای آدرس منطقی 129 صفحه باشد و قرار باشد به یک فضای آدرس فیزیکی 1 مگابایتی نگاشت صورت گیرد، هر مدخل (entry) از جدول صفحه (Page Table) باید چند بیت باشد؟ (بدون در نظر گرفتن valid یا invalid در page table)

(مهندسی کامپیوتر – دولتی ۹۹)

۱) 8 ۲) 9 ۳) 17 ۴) 20

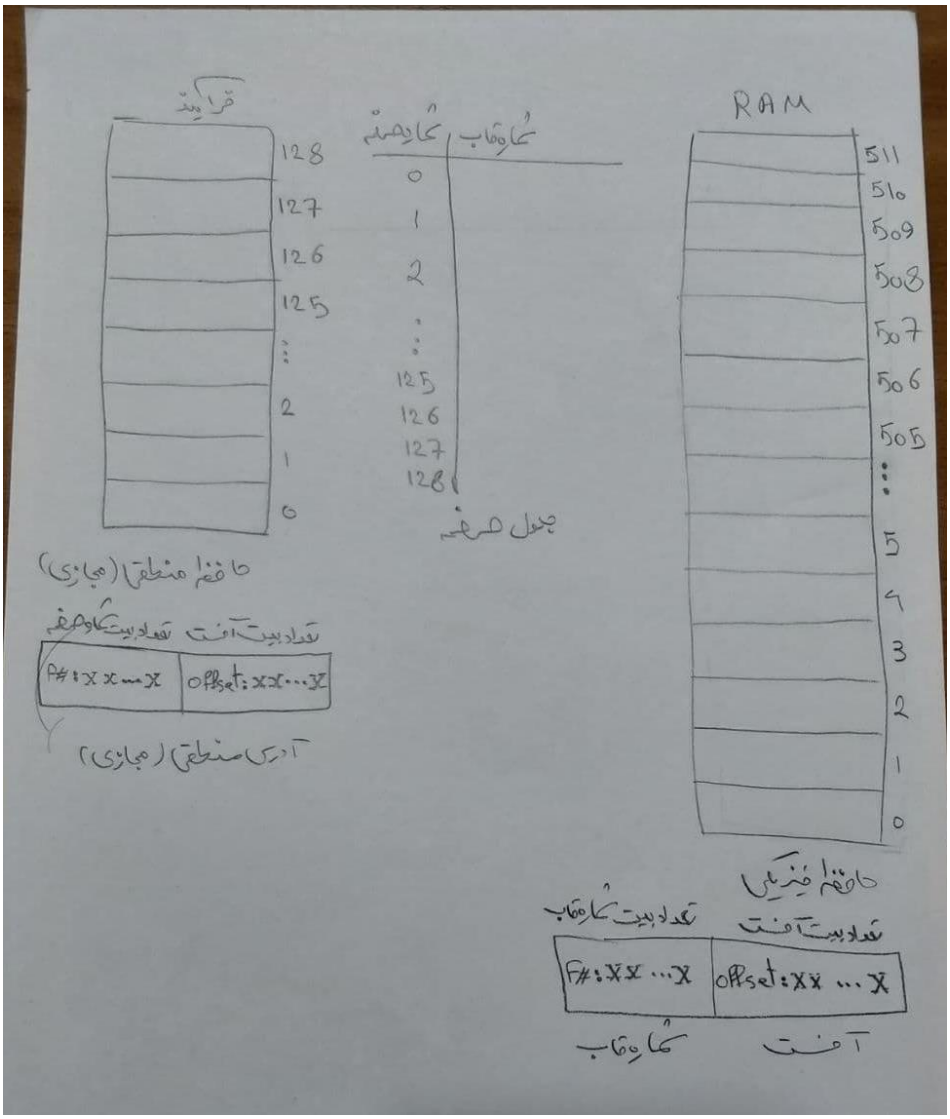
عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل چهارم: مدیریت حافظه اصلی

۱۰۴- گزینه (۲) صحیح است.

در صورت سوال مطرح شده است که طول آدرس منطقی 19 بیت، تعداد صفحات فرآیند در فضای آدرس منطقی 129 صفحه و اندازه فضای آدرس فیزیکی (حافظه فیزیکی) 1MB است. همچنین مطابق فرض سوال طول بیت‌های کنترلی valid یا invalid در page table در جدول صفحه نادیده گرفته شده است و خواسته شده است طول هر مدخل (entry) از جدول صفحه (Page Table) باید چند بیت باشد؟

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بندی به صورت زیر است:



توجه: مطابق فرض سوال تعداد صفحات فرآیند در فضای آدرس منطقی (فرآیند) 129 صفحه است.

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

$$b = \log_2^{\text{تعداد صفحات فرآیند}} = \lceil \log_2^{129} \rceil = 8 \text{ bit} = \text{تعداد بیت شماره صفحه}$$

توجه: مطابق فرض سوال طول آدرس منطقی 19 بیت است.

همچنین، اندازه آدرس منطقی (مجازی) به صورت زیر است:

$19\,\text{bit} - 8\,\text{bit} = 11\,\text{bit}$ = تعداد بیت شماره صفحه - طول آدرس منطقی = تعداد بیت آفست

$8\text{bit} + 11\text{bit} = 19\,\text{bit}$ = تعداد بیت آفست + تعداد بیت شماره صفحه = طول آدرس منطقی

$2^{\text{تعداد بیت آفست}} = 2^{11} = 2048\ \text{B}$ = اندازه صفحه یا اندازه قاب

همچنین داریم:

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

$2^{19} = 2^8 \times 2^{11} = 2^{19} = 512\,\text{KB}$ = اندازه حافظه منطقی (فرآیند)

توجه: مطابق فرض سوال اندازه فضای آدرس فیزیکی (حافظه فیزیکی) $1\text{MB} = 2^{20}\ \text{B}$ است.

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

$$\text{تعداد قاب‌های حافظه‌ی فیزیکی} = \frac{2^{20}}{2^{11}} = 2^9 = 512$$

$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{512} = \log_2^{2^9} = 9\,\text{bit}$

$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{2^{11}} = 11\,\text{bit}$

همچنین داریم:

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

$2^{20} = 2^9 \times 2^{11} = 2^{20} = 1\ \text{MB}$ = اندازه حافظه فیزیکی (RAM)

همچنین، اندازه آدرس فیزیکی به صورت زیر است:

$20\,\text{bit} - 11\,\text{bit} = 9\,\text{bit}$ = تعداد بیت آفست - طول آدرس فیزیکی = تعداد بیت شماره قاب

$9\text{bit} + 11\text{bit} = 20\,\text{bit}$ = تعداد بیت آفست + تعداد بیت شماره قاب = طول آدرس فیزیکی

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

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

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:

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

در سوال مطرح شده 0 بیت مربوط به بیت‌های کنترلی (مطابق فرض سوال طول بیت‌های کنترلی valid یا invalid در page table در جدول صفحه نادیده گرفته شده است) و 9 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق (طول هر مدخل (entry) از جدول صفحه (Page Table)) برابر $9bit + 0bit = 9bit$ می‌باشد.

همچنین 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:

$$\text{اندازه جدول صفحه} = 129 \times 9bit = 1161bit$$

بنابراین پُرواضح است که گزینه دوم پاسخ سوال است.

تست‌های فصل دوم

۱۰۵- در یک سیستم زمان‌بندی MLFQ دو صف RR با برش زمانی 3 و ∞ وجود دارد. میانگین زمان بازگشت برای پردازش‌های زیر کدام است؟ **(مهندسی کامپیوتر - دولتی ۹۹)**

	Arrival time	CPU Burst1	I/O Burst1	CPU Burst2
P1	0	3	6	2
P2	3	7	5	1
P3	4	2	4	4
P4	11	5	3	1

۱) 21.5 ۲) 17 ۳) 16.5 ۴) 14

عنوان کتاب: سیستم عامل

مولف: ارسطو خلیلی‌فر

ناشر: انتشارات راهیان ارشد

آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم

۱۰۵- گزینه (۲) صحیح است.

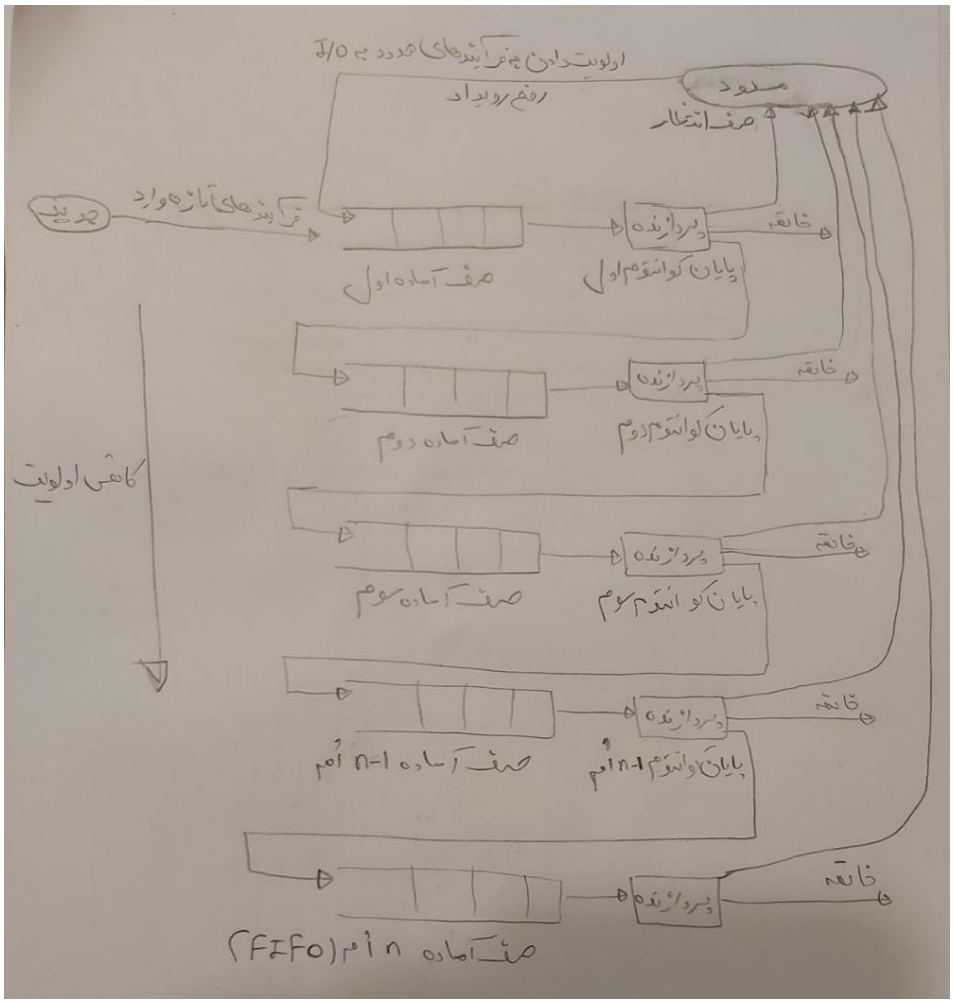
میانگین سربار تعویض متن در الگوریتم نوبت چرخشی (RR) به علت ثابت بودن برش زمانی زیاد است. پس مسئله این است که چگونه در کنار کاهش سربار سوئیچ، زمان بازگشت کارهای کوچک را نیز پایین نگه داریم؟ راه حل این است که با ایجاد صف‌های چندگانه (Multiple Queues)، کلاس‌های اولویت ایجاد شود. همه فرآیندهای تازه وارد در انتهای بالاترین صف (بالاترین کلاس اولویت) قرار می‌گیرند. فرآیندها در بالاترین کلاس، به مدت ۱ کوانتوم اجرا می‌شوند. فرآیندهای کلاس بعدی ۲ کوانتوم، فرآیندهای کلاس بعدی ۴ کوانتوم و به همین ترتیب، فرآیند درون صف Q_n برابر 2^n کوانتوم می‌گیرد. اگر یک فرآیند از تمام کوانتوم مختص خودش در یک کلاس مورد نظر به صورت کامل استفاده کند و تمام نشود، به کلاس پایین‌تر منتقل می‌شود. پس از اولین اجرا (اولین کوانتوم) در صف اول برای فرآیند ممکن است سه حالت پیش بیاید:

حالت اول: ممکن است فرآیند مورد نظر با همان کوانتوم اول در صف اول تمام شود.

حالت دوم: ممکن است اجرای فرآیند مورد نظر ادامه داشته باشد که در این صورت، به صف دوم منتقل می‌شود. به همین ترتیب، فرآیندها پس از پایان کوانتوم زمانی در هر صف و تمام نشدن، به صف‌های پایین‌تر و کم اولویت‌تر بعدی منتقل می‌شوند که کوانتوم‌های متناظر با صف‌ها، با پایین آمدن سطح صف، افزایش می‌یابند. به این ترتیب هرچه فرآیندها در صف‌های اولویت، عمیق‌تر و عمیق‌تر فرو می‌روند، به همان اندازه فاصله زمانی مابین دوبار اجرای پشت سرهم آنها بیشتر و بیشتر می‌شود. در این الگوریتم فرآیندهای کوتاه به سرعت تکمیل می‌شوند بدون اینکه به صف‌های پایین‌تر منتقل شوند و از طرفی فرآیندهای طولانی کم کم به صف‌های پایین و کم اولویت‌تر هدایت می‌شوند. بنابراین فرآیندهای جدیدتر و کوتاه‌تر به فرآیندهای قدیمی و بلندتر ارجحیت می‌یابند.

حالت سوم: ممکن است فرآیند مورد نظر در یکی از صف‌ها برای وقوع رخدادی مثل ورودی و خروجی، منتظر بماند که در این صورت مسدود و به صف انتظار منتقل می‌شود. و پس از اتمام انتظار در انتهای صف اول به عنوان با اولویت‌ترین و بالاترین کلاس قرار می‌گیرد.

شکل زیر گویای مطلب است:



ایده موجود در روش MLFQ این است که فرآیندها با توجه به رفتارشان بین صف‌ها حرکت کنند. به عنوان مثال فرآیندی که در یک صف با اولویت بالاتر قرار دارد و زمان پردازنده را زیاد مصرف می‌کند، به صف پایین و پایین‌تر منتقل می‌شود و یا فرآیندی که مسدود می‌شود پس از رفع انسداد به صف اول با بالاترین اولویت منتقل می‌شود. زمانی سراغ پردازش فرآیندهای داخل صف‌های پایین‌تر می‌رویم که تمام صف‌های بالاتر از آن خالی باشد. اگر پردازش فرآیندی در یکی از صف‌های پایین‌تر در جریان باشد، و فرآیند جدیدی وارد صف اول شود، سیستم، پردازش ادامه فرآیندهای صف جاری را رها می‌کند و به صف اول باز می‌گردد، البته در مورد فرآیند جاری، این اجازه داده می‌شود که کوانتوم مربوطه را تمام کند و سپس بازگشت به صف اول انجام

می‌شود. زیرا این الگوریتم «**غیرانحصاری مبتنی بر کوانتوم**» است.

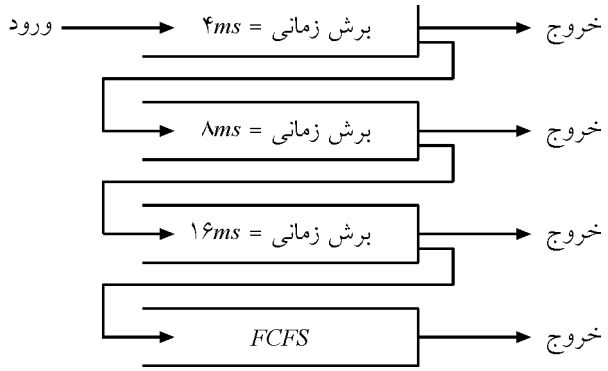
توجه: مطابق سیاست‌های الگوریتم MLFQ منطقی است که هر فرآیند پس از عمل I/O به صف اول بازگردد، زیرا ممکن است فرآیندی در ابتدای روند پردازشش به مدت طولانی CPU مصرف کند که این امر منجر به این می‌شود که در گذر زمان فرآیند مورد نظر ته نشین شده و به صف‌های پایین‌تر و با اولویت کمتر منتقل شود، درحالیکه بعد از مدتی فرآیند مورد نظر به واسطه داشتن عملیات I/O به یک فرآیند تعاملی تبدیل شود، طبیعی است که جهت انجام عملیات تعاملی به CPU و به تبع اولویت بالاتر و سریع‌تر نیاز است، پس فرآیند مورد نظر پس از عمل I/O نباید در صف با اولویت پایین‌تر بماند و باید به صف اول جهت دریافت سریعتر CPU منتقل شود. در نتیجه این الگوریتم به نفع فرآیندهای با زمان اجرای کوتاه مدت یا محدود به I/O (I/O Bound) است. به طور مثال، یک فرآیند با وقفه I/O وارد صف انتظار شده و پس از انجام عمل I/O و به تبع رفع انتظار، برای اجرای مجدد به صف اول با بالاترین اولویت باز می‌گردد، بنابراین این الگوریتم به نفع فرآیندهای محدود به I/O هستند.

توجه: این الگوریتم به کارهای کوتاه اولویت می‌دهد، بدون آنکه لازم باشد قبل از اجرای یک کار، مدت اجرای آنرا بداند بر خلاف الگوریتم‌های SJF، SRT و HRRN که باید زمان اجرا را از قبل دانست. چون این الگوریتم بر اساس مکانیزم فیدبک (FB : FeedBack) رفتار می‌کند، ابتدا فرآیند وارد صف اول می‌شود و یک کوانتوم جهت اجرا به آن اختصاص می‌یابد، اگر در صف اول و کوانتوم اول فرآیند تمام شد که شد، اگر تمام نشد و بازهم CPU می‌خواست به صف دوم می‌رود و یک کوانتوم دیگر جهت اجرا به آن اختصاص می‌یابد. و این رویه ته نشینی به سمت صف‌های پایین‌تر آنقدر ادامه پیدا می‌کند تا فرآیند تمام شود. طبیعی است که اگر یک فرآیند خیلی کوتاه باشد به طور خودکار با بالاترین اولویت اجرا شده و از سیستم خارج می‌شود. هرچند که مشکل قحطی زدگی کارهای طولانی وجود دارد.

توجه: در این الگوریتم نرخ عملیات تعویض متن تا حد قابل توجهی کاهش می‌یابد، چون اگر فرآیندی زمان پردازش زیادی را لازم داشته باشد، به جای تخصیص مکرر کوانتوم‌های کوچک، به صف‌های پایین‌تر که کوانتوم زمانی‌شان بیش‌تر است هدایت می‌شود.

توجه: این الگوریتم با نام‌های صف‌های چندگانه یا صف‌های چندسطحی بازخوردی (**MLFQ**: Multi Level FeedBack Queus) یا MFQ یا FB شناخته می‌شود.

مثال: فرض کنید سیستمی با ۴ صف در اختیار داریم، در صف اول از روشRR با برش زمانی ۴ میلی‌ثانیه، در صف دوم از روش RR با برش زمانی ۸ میلی‌ثانیه، در صف سوم از روش RR با برش زمانی ۱۶ میلی‌ثانیه و در صف آخر از روشFCFS استفاده می‌کنیم(شکل زیر).



مثالی از الگوریتم MLFQ

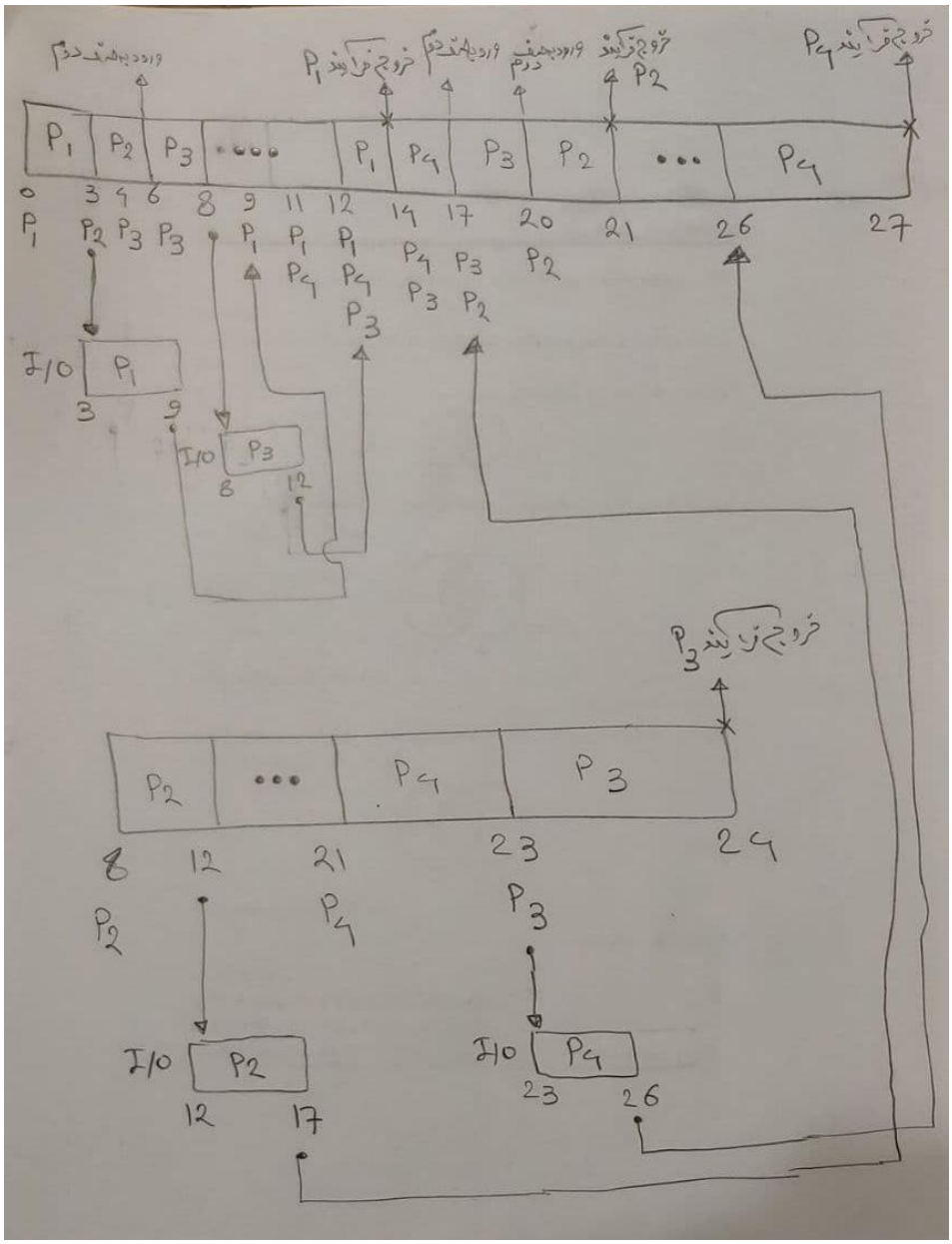
در این حالت فرآیندها ابتدا به صف اول وارد می‌شوند، اگر در این صف به پایان نرسیدند به انتهای صف دوم، اگر در این صف نیز کامل نشدند به انتهای صف سوم و اگر باز هم به پایان نرسیدند به انتهای صف چهارم وارد می‌شوند که از روش FCFS استفاده می‌کنند.

با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	CPU Burst1	I/O Burst1	CPU Burst2	زمان انتظار+	زمان بازگشت =
P_1	0	3	6	2		
P_2	3	7	5	1		
P_3	4	2	4	4		
P_4	11	5	3	1		

ابتدا فرآیندها در صف اول با تکه زمانی معادل 3 واحد زمانی قرار می‌گیرند. و اگر در مدت 3 واحد زمانی تمام نشوند به صف دوم با تکه زمانی معادل ∞ واحد زمانی یعنی الگوریتم FCFS منتقل می‌گردند، تا بالاخره تمام شوند.

نمودار گانت کلی به صورت زیر است:



زمان ورود فرآیند – زمان خروج فرآیند = زمان بازگشت فرآیند

P_1 زمان بازگشت $= 14 - 0 = 14$

P_2 زمان بازگشت $= 21 - 3 = 18$

P_3 زمان بازگشت $= 24 - 4 = 20$

P_4 زمان بازگشت $= 27 - 11 = 16$

میانگین زمان بازگشت $= ATT = \frac{14+18+20+16}{4} = \frac{68}{4} = 17$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار $= 14 - 5 = 9$

P_2 زمان انتظار $= 18 - 8 = 10$

P_3 زمان انتظار $= 20 - 6 = 14$

P_4 زمان انتظار $= 16 - 6 = 10$

میانگین زمان انتظار $= AWT = \frac{9+10+14+10}{4} = \frac{43}{4} = 10.75$

میانگین زمان اجرا $= AST = \frac{5+8+6+6}{4} = \frac{25}{4} = 6.25$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان انتظار + میانگین زمان اجرا = میانگین زمان بازگشت

17 = 6.25 + 10.75

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	5	9	14
P_2	3	8	10	18
P_3	4	6	14	20
P_4	11	6	10	16

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

17 10.75 6.25

تست‌های فصل پنجم

۱۰۶- یک سامانه مدیریت حافظه را در نظر بگیرید، که تنها **چهار قاب به پردازه اختصاص داده شده است که در ابتدا خالی هستند. حال پردازه صفحات را به ترتیب از چپ به راست 1, 2, 3, 4, 3, 2, 2, 1, 6, 7 ارجاع می‌کند. اگر این سیستم مدیریت حافظه از روش LRU برای جایگزینی صفحه استفاده کند، چه تعداد نقص صفحه خواهیم داشت؟** **(مهندسی کامپیوتر - دولتی ۹۹)**

۱) 7 ۲) 8 ۳) 9 ۴) 10

۷۶- گزینه () صحیح است.

الگوریتم LRU (Least Recently Used)

ایده اصلی این الگوریتم این است که اگر صفحه‌ای در چند دستور اخیر مراجعات زیادی داشته است، به احتمال قوی در دستورات بعدی هم ارجاعات زیادی خواهد داشت، همچنین اگر یک صفحه، اخیرا هیچ مراجعه‌ای نداشته، احتمالاً در آینده نزدیک هم ارجاعی نخواهد داشت. در واقع این الگوریتم بیان می‌کند هنگام وقوع خطای نقص صفحه، صفحه‌ای را حذف کنید که طولانی‌ترین زمان عدم استفاده را دارد.

می‌توان گفت LRU تقریبی از الگوریتم بهینه می‌باشد که در آن به جای توجه به آینده، به گذشته توجه می‌شود.

توجه: الگوریتم LRU ناهنجاری بی‌لیدی ندارد، بنابراین با افزایش تعداد قاب، همواره، نقص صفحه کاهش می‌یابد.

اگر 4 قاب در نظر گرفته شود:

نگاه به گذشته ←

رشته مراجعات	1	2	3	4	3	2	2	1	6	7										
قاب 1	1	1	1	1	1	1	1	1	1	1										
قاب 2		2	2	2	2	2	2	2	2	2										
قاب 3			3	3	3	3	3	3	3	7										
قاب 4				4	4	4	4	4	6	6										
نقص صفحه	*	*	*	*					*	*										

در این حالت 6 نقص صفحه به وقوع پیوست. پاسخ سوال در گزینه‌ها نیست.

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

تست‌های فصل هشتم: مدیریت دیسک

۱۰۷- در یک دستگاه دیسک‌خوان، بازو روی سیلندر 35 قرار دارد و جهت حرکت آن به سمت شیارهای بزرگتر با شماره بزرگتر است. اگر زمان انتقال از یک سیلندر به بعدی 5ms باشد و از الگوریتم آسانسور برای دستیابی به سیلندرها استفاده شود، کل زمان جستجو برای دستیابی به سیلندرهای زیر (به ترتیب از چپ به راست) چند میلی ثانیه است؟

34, 45, 39, 80, 12, 25, 44

(مهندسی کامپیوتر – دولتی ۹۹)

۱) 565 ۲) 575 ۳) 665 ۴) 675

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل هشتم: مدیریت دیسک

۱۰۷- گزینه (۱) صحیح است.

در ویراست دهم سال 2018 کتاب مرجع آبراهام سیلبرشاتز صفحه 458 الگوریتم‌های Look و C-Look از کتاب حذف شده‌اند. و از SCAN algorithm به صورت elevator algorithm به معنی «**الگوریتم آسانسور**» یاد شده است. به متن صریح کتاب در ادامه توجه نمایید:

SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

در الگوریتم SCAN ، بازوی دیسک از یک انتهای دیسک شروع کرده و در جهت انتهای دیگر حرکت می کند و به درخواست هایی که در هر سیلندر به آنها می‌رسد، سرویس دهی می کند، تا زمانی که به انتهای دیگر برسد. در انتهای دیگر، جهت حرکت هد، برعکس می شود و سرویس دهی ادامه می یابد. هد به صورت پیوسته به عقب و جلو در عرض دیسک، حرکت می کند. الگوریتم SCAN گاهی اوقات **الگوریتم آسانسور (elevator algorithm)** نامیده می‌شود.

به مثال زیر از متن کتاب سیستم عامل آبراهام سیلبرشاتز دقت نمایید:

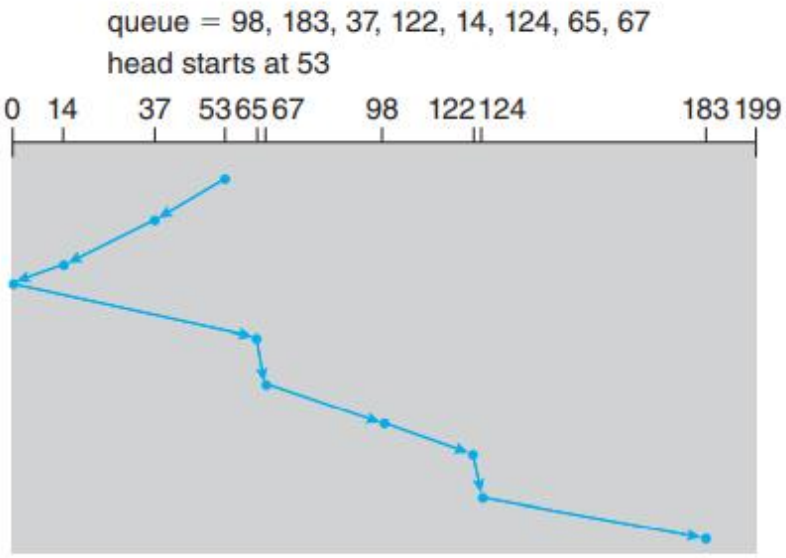


Figure 11.7 SCAN disk scheduling.

توجه: در این مورد کتاب اندرو تنن‌بام با کتاب آبراهام سیلبرشاتز اختلاف نظر دارد و جناب تنن‌بام به الگوریتم Look می‌گوید آسانسور، در چند مورد اختلافی حتی تست کنکور حذف شد.

توجه: در صورت سوال طراح محترم شماره ابتدا و انتهای دیسک را مطرح نکرده است، که به ناچار می‌بایست در حل این سوال الگوریتم آسانسور را همان الگوریتم Look در کتاب تنن‌بام در نظر گرفت.

توجه: در کتاب اندرو تنن‌بام الگوریتم آسانسور (Look) به صورت زیر تعریف شده است:

در الگوریتم آسانسور (Look)، هد تا جایی حرکت می‌کند که درخواستی وجود داشته باشد و سپس جهت آن بالعکس می‌شود. به عبارت دیگر در الگوریتم آسانسور (Look) حرکت هد لزوما تا انتهای دیسک و تا آخرین شیار ادامه نمی‌یابد. بلکه فقط تا آخرین درخواست جلو می‌رود و بعد از آخرین درخواست تغییر جهت داده و به سمت درخواست‌های دیگر حرکت می‌کند. در واقع در این الگوریتم کار از اولین درخواست شروع شده و به آخرین درخواست ختم می‌گردد.

توجه: مطابق فرض سوال بازو روی سیلندر 35 قرار دارد و جهت حرکت آن به سمت شیارهای بزرگتر با شماره بزرگتر است. و مطرح شده است که اگر زمان انتقال از یک سیلندر به بعدی 5ms باشد و از الگوریتم آسانسور برای دستیابی به سیلندرها استفاده شود، کل زمان جستجو برای دستیابی به سیلندرهای زیر (به ترتیب از چپ به راست) چند میلی ثانیه است؟

34, 45, 39, 80, 12, 25, 44

مطابق فرض سؤال دنباله درخواست‌ها به صورت زیر است:

34, 45, 39, 80, 12, 25, 44

و هد در ابتدای کار بر روی سیلندر 35 قرار دارد.

در این روش ابتدا درخواست 39 سرویس داده می‌شود. زیرا نزدیکترین درخواست به مکان فعلی هد در جهت صعودی است، سپس درخواست‌های 44، 45، 80 در ادامه پیمایش تغییر جهت هد صورت می‌گیرد و سپس درخواست‌های 34، 25 و در نهایت 12 سرویس می‌گیرند.

یعنی داریم:

$$35 \xrightarrow[4]{} 39 \xrightarrow[5]{} 44 \xrightarrow[1]{} 45 \xrightarrow[35]{} 80 \xrightarrow[46]{} 34 \xrightarrow[9]{} 25 \xrightarrow[13]{} 12$$

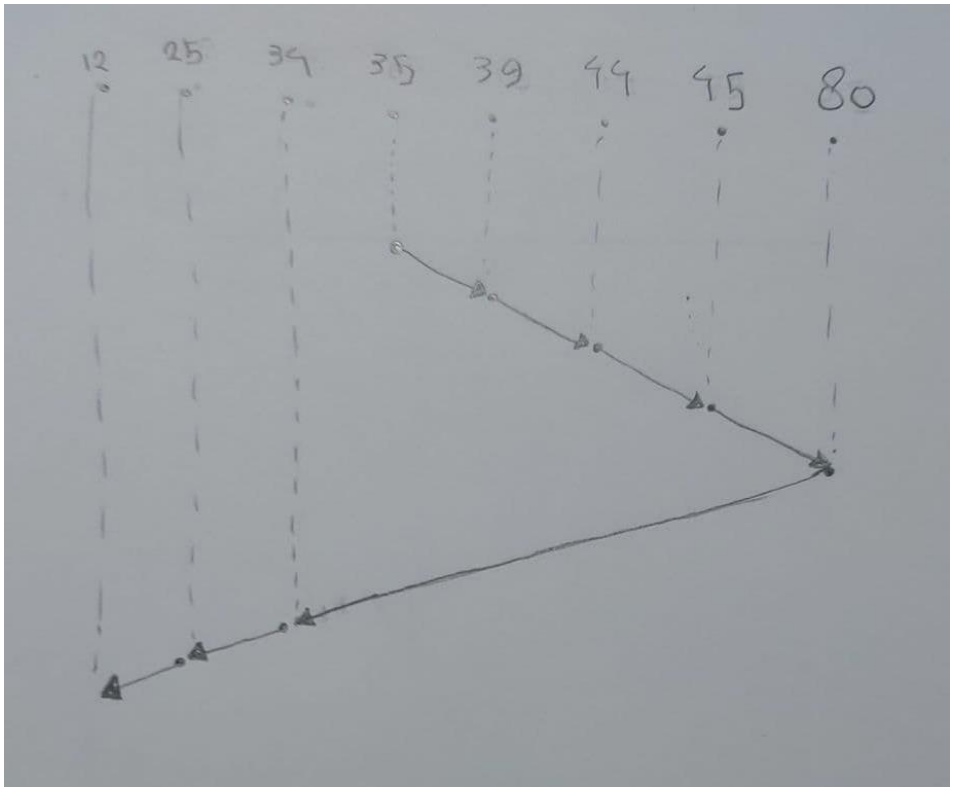
در نتیجه، تعداد کل حرکات هد به صورت زیر خواهد بود:

$4+5+1+35+46+9+13 = 113$

مطابق فرض سؤال، اگر حرکت از هر سیلندر به سیلندر دیگر (سیلندرهای متوالی) یعنی مجاور 5 میلی‌ثانیه طول بکشد، مجموع زمان جستجو در این درخواست‌ها، به صورت زیر خواهد بود:

$$T_{Total} = 113 \times 5 = 565 \text{ ms}$$

شکل زیر گویای مطلب است:



$(80-35)+(80-12)=45+68=113$

$T_{Total}=113\times 5=565\,ms$

بنابراین پُرواضح است که گزینه اول پاسخ سوال است.

تست‌های فصل ششم: مدیریت فرآیندها و نخ‌ها هم‌روند

۱۰۸- کدام عبارت ذیل نمی‌تواند خروجی اجرای هم‌رونـد فرآینـدهای P_1 و P_2 باشـد؟ (S_1 متغیر سراسری با مقدار اولیه صفر می‌باشد.)

(مهندسی کامپیوتر – دولتی ۹۹)

P1	P2
Print "A"	Print "C"
wait (S1)	signal (S1)
Print "B"	Print "D"

۱) ABCD ۲) ACDB ۳) CABD ۴) CADB

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۱۰۸- گزینه (۱) صحیح است.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
Add  this process to s.queue;
block ();
  }
}
```

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if(s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در غیر اینصورت، فرآیند، وارد خط بعدی می‌شود.

تابع signal(s): عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    Remove a process from queue;
    Wake up ();
     }
}
```

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به خط بعدی است که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake () up بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

ترتیب خروجی گزینه اول امکان پذیر نیست:

فرآیند	دستور	S1	خروجی
P1	Print "A"	0	A
P1	wait (S1)	-1	A
فرآیند P1 در سطر دوم توسط دستور wait (S1) در صف سمافور S1 می‌خوابد.			
P2	Print "C"	-1	AC
P2	signal (S1)	0	AC
فرآیند P1 در سطر چهارم توسط دستور signal (S1) بیدار می‌شود، و دقیقا دستور بعد از wait (S1) یعنی Print "B" در صف آماده قرار می‌گیرد و می‌تواند اجرا شود.			
P1	Print "B"	0	ACB
P2	Print "D"	0	**ACBD**

نتیجه: در این سوال اجرای دستور Print "B" در فرآیند P1 بدون اجرای دستور signal (S1) در P2 و به تبع اجرای دستور Print "C" امکان پذیر نیست. به عبارت دیگر Print "C" همواره

باید قبل از اجرای "Print "B باشد.

ترتیب خروجی گزینه دوم امکان پذیر است:

فرآیند	دستور	S1	خروجی
P1	Print "A"	0	A
P2	Print "C"	0	AC
P2	signal (S1)	1	AC
P2	Print "D"	1	ACD
P1	wait (S1)	0	ACD
P1	Print "B"	0	**ACDB**

ترتیب خروجی گزینه سوم امکان پذیر است:

فرآیند	دستور	S1	خروجی
P2	Print "C"	0	C
P2	signal (S1)	1	C
P1	Print "A"	1	CA
P1	wait (S1)	0	CA
P1	Print "B"	0	CAB
P2	Print "D"	0	**CABD**

ترتیب خروجی گزینه چهارم امکان پذیر است:

فرآیند	دستور	S1	خروجی
P2	Print "C"	0	C
P2	signal (S1)	1	C
P1	Print "A"	1	CA
P2	Print "D"	0	CAD
P1	wait (S1)	0	CAD
P1	Print "B"	0	**CADB**

بنابراین پُرواضح است که گزینه اول پاسخ سوال است.

تست‌های فصل اول: مفاهیم اولیه

۹۴- کدام گزینه از مزایای ساختار سیستم عامل لایه‌ای (Layered) نسبت به ساختار سیستم عامل یکپارچه (Monolithic) نیست؟ **(مهندسی *IT* – دولتی ۹۹)**

۱) قابلیت گسترش بیشتر
۲) خطایابی ساده‌تر
۳) مدیریت ساده‌تر
۴) سرعت بیشتر

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل اول: مفاهیم اولیه

۹۴- گزینه (۴) صحیح است.

ساختارهای درونی طراحی سیستم عامل‌ها به صورت زیر است:

۱- ساختار یکپارچه (Monolithic)

ساختار سیستم عامل‌های یکپارچه به این صورت است که هیچ ساختاری ندارد! سیستم‌های یکپارچه به صورت مجموعه‌ای از رویه‌ها نوشته شده‌اند که هر یک می‌توانند دیگری را به هنگام نیاز فراخوانی کرده و برای انجام محاسبات خود از آن‌ها کمک بگیرند. هر یک از رویه‌ها دارای یک واسط شامل پارامترهای ورودی و خروجی است که به سادگی تعریف شده‌اند. همانطور که مشخص است این سیستم هیچ‌گونه نظم، ترتیب، دسته‌بندی و سلسله مراتب خاصی ندارد.

برای ساختن یک سیستم یکپارچه (ساختن object واقعی سیستم عامل) ابتدا باید تمام رویه‌ها در قالب تعدادی فایل و یا به صورت جداگانه کامپایل شوند و سپس با استفاده از یک پیونددهنده (Linker) به هم متصل شده و در فایل object نهایی قرار گیرند. در این ساختار تمام رویه‌ها همدیگر را می‌بینند. این به آن معنی است که بین رویه‌ها هیچ تعیین سطحی وجود ندارد و اصطلاحاً سیستم تخت است. در واقع برای مخفی کردن اطلاعات و محصورسازی (Encapsulation)، محدویت‌هایی وضع نشده است و از آنجاکه دسترسی به هر رویه‌ای امکان‌پذیر است لذا حفاظت وجود ندارد. پنهان‌سازی اطلاعات (information hiding) نتیجه پیمانه‌ای کردن (Modularity) است. به بیان دیگر شرط لازم برای برقراری پنهان‌سازی اطلاعات، پیمانه‌ای کردن است و شرط کافی برای برقراری پنهان‌سازی اطلاعات تعریف متغیرهای محلی و دستورالعمل‌های مرتبط با تابع است. در یک بیان ساده پنهان‌سازی اطلاعات می‌گوید بخشی از نرم‌افزار در داخل یک پیمانه (تابع) محصور شود. هدف از پنهان‌سازی اطلاعات، پنهان کردن روال انجام دستورات تابع و متغیرهای محلی در پس واسط یا بلاک تابع است. استفاده‌کنندگان پیمانه‌ها (توابع) نیازی به دانستن جزئیات داخلی پیمانه‌ها (توابع) ندارند.

توجه: تعریف متغیر سراسری در تابع ناقض اصل پنهان‌سازی اطلاعات است.

اما به هر حال حتی در سیستم‌های یکپارچه حداقل یک تابع اصلی برای فراخوانی توابع و رویه‌های دیگر وجود دارد

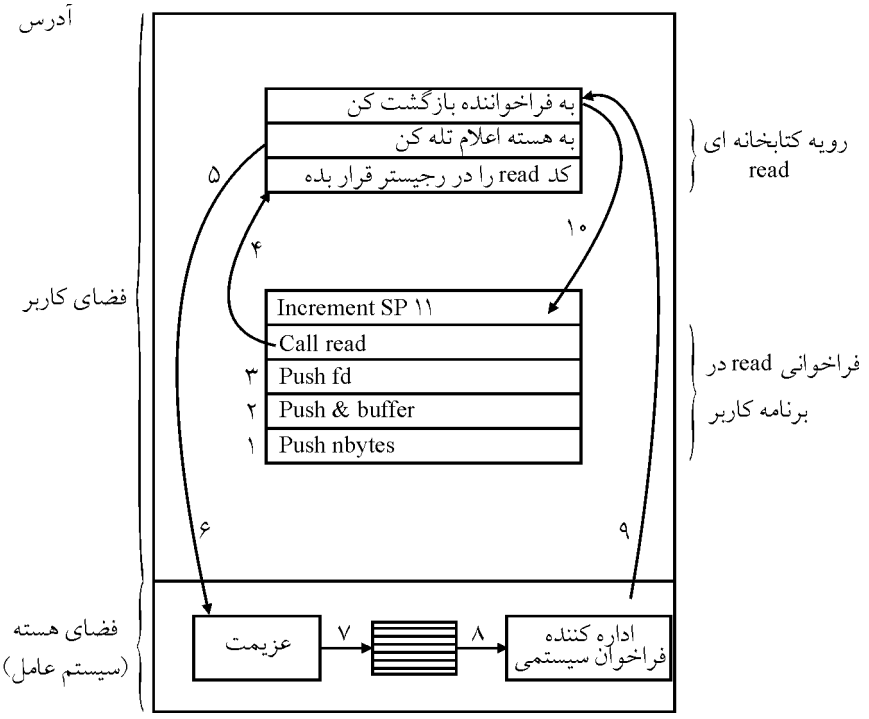
در این سیستم برای تقاضای یک سرویس از سرویس‌های سیستم عامل، ابتدا پارامترهای فراخوانی را در محل‌های از پیش تعیین شده مانند رجیسترها یا پشته قرار داده و سپس یک دستورالعمل تله مربوط به سرویس موردنظر اجرا می‌شود. این عمل به فراخوانی هسته یا فراخوانی راهبری معروف است. در واقع با این روش ماشین از مُد کاربر به مُد هسته می‌رود تا جهت انجام سرویس مورد نظر، کنترل در اختیار سیستم عامل قرار گیرد. اکنون با یک مثال چگونگی عملکرد فراخوان‌های سیستمی شرح داده می‌شود. فرض کنید باید فراخوانی زیر توسط برنامه اصلی (برنامه کاربر) انجام شود:

Count = Read (fd, buffer, nbyted);

تابع Read دارای سه پارامتر است، پارامتر اول مشخص‌کننده فایل است، پارامتر دوم یک اشاره‌گر به بافر است، و سومین پارامتر بیانگر تعدادی بایتی است که باید خوانده شود. این تابع تعداد بایت‌هایی که واقعاً از فایل خوانده و در بافر قرار داده است را برمی‌گرداند. در حالت عادی مقدار Count برابر با nbytes است ولی امکان دارد که کمتر از آن هم باشد (مثلاً به علت اتمام فایل).

در ابتدا برای صدا زدن رویه کتابخانه‌ای Read از کامپایلر، پارامترهای آن در پشته Push می‌شود (در شکل زیر این عمل با توجه به زبان برنامه‌نویسی C و C++ از پارامتر آخر به اول انجام می‌شود). با توجه به اینکه پارامتر دوم یعنی buffer یک اشاره‌گر است در هنگام push در ابتدای آن یک علامت & استفاده می‌شود، یعنی محتوای آن به عنوان یک آدرس در نظر گرفته شده است. مراحل push کردن پارامترها در شکل زیر با شماره‌های 1 تا 3 نمایش داده شده‌اند. اکنون باید رویه کتابخانه‌ای را فراخوانی کرد. برای این کار از یک دستورالعمل ساده فراخوانی رویه که برای صدا زدن همه رویه‌ها به کار می‌رود، استفاده می‌شود (مرحله 4). شماره فراخوانی سیستمی باید در محلی مانند یک رجیستر (که سیستم عامل انتظار دارد و در آنجا باشد) قرار گیرد (مرحله 5). اکنون باید دستور TRAP یا همان وقفه نرم‌افزاری (تله) رخ دهد تا کنترل اجرا از مُد کاربر به مُد هسته رفته و اجرای کُد مربوط به آن فراخوان از یک آدرس ثابت درون هسته آغاز شود (مرحله 6). اکنون هسته، شماره فراخوانی که در یک رجیستر گرفته بود را بررسی می‌کند و سپس به سراغ یک جدول که حاوی اشاره‌گرهایی به اداره‌کننده‌های فراخوانی‌های سیستمی (Systemcall Handler) هستند، می‌رود و با استفاده از شماره فراخوانی از بین درایه‌های جدول، محل قرار گرفتن اداره‌کننده فراخوانی مذکور (Read) را می‌یابد که عموماً محلی از حافظه است (مرحله 7) و آن را به اجرا درمی‌آورد (مرحله 8). در ادامه، اجرای اداره‌کننده فراخوانی سیستمی به طور کامل تمام می‌شود یا ممکن است تمام نشود و کنترل به فرآیند جدید دیگری منتقل شود و به تبع مراحل 1 تا 6 رویه کتابخانه‌ای در فضای کاربر برای فرآیند جدید هم اجرا شود. اینکه گفته می‌شود ممکن است به دلیل آن است که اگر فراخوان سیستمی در فرآیند جاری مجبور به انتظار باشد، مانند اینکه فراخوان سیستمی تلاش کند از صفحه کلید بخواند ولی هنوز چیزی از طرف

کاربر تایپ نشده باشد، از آنجا فراخوان سیستمی منتظر و مسدود شده داخل فرآیند جاری است بنابراین فرآیند جاری از طرف سیستم عامل به صف مسدود و منتظر منتقل می‌شود. در این شرایط سیستم عامل به صف فرآیندهای آماده اجرا نگاه می‌کند تا ببیند آیا فرآیند دیگری می‌تواند پس از آن اجرا شود یا خیر. در صورت وجود فرآیند آماده جدید سیستم عامل تعویض متن انجام داده و پردازنده را در اختیار فرآیند جدید قرار می‌دهد. در ادامه هرگاه ورودی مورد نظر فراخوان سیستمی فرآیند قبل آماده شود، برای مثال کلیدی فشرده شود، فرآیند قبلی به صف آماده سیستم عامل منتقل می‌شود و ممکن است در دفعه بعد توسط زمان‌بند کوتاه مدت سیستم عامل جهت اجرا انتخاب شود و مراحل 9 تا 11 انجام می‌شود. (مرحله 9) سپس این رویه به روش بازگشت به برنامه کاربر بازمی‌گردد. (مرحله 10) برای خاتمه کار، برنامه کاربر پشته را پاک‌سازی می‌کند، همان‌گونه که پس از بازگشت از همه رویه‌ها این عمل صورت می‌گیرد. (مرحله 11)



ساختار کلی این سیستم عامل‌ها از سه بخش زیر تشکیل شده است:

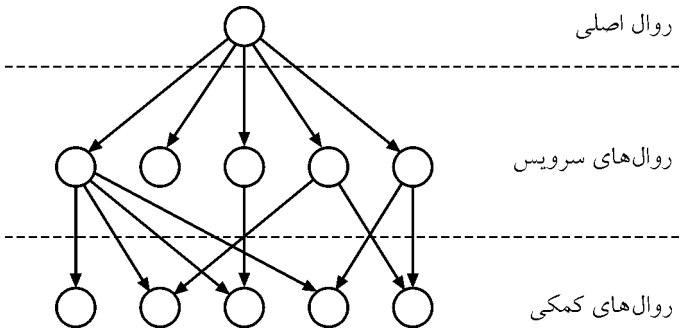
۱- یک برنامه اصلی که می‌تواند رویه سرویس‌های خواسته شده را فراخوانی کند. این برنامه به ازای هر درخواست، یک روال سرویس را صدا می‌زند.

۲- مجموعه‌ای از رویه‌های سرویس که می‌توانند تعدادی فراخوانی سیستمی انجام دهند و یا تعدادی روال کمکی را فراخوانی کنند.

۳- تعدادی رویه کمکی و سودمند (Utility Procedures) که می‌توانند به رویه‌های سرویس

کمک کنند و توسط رویه‌های سرویس فراخوانی شوند.

این مفاهیم در شکل زیر نشان داده شده است:



ساختار سیستم عامل‌های یکپارچه

. پیمانه‌ای کردن یعنی تقسیم نرم‌افزار (در اینجا خود برنامه سیستمی سیستم عامل) به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک و فهم نرم‌افزار یکپارچه (monolithic) که تنها از یک پیمانه (ماژول) تشکیل شده است، بسیار مشکل است، زیرا تعداد متغیرها، حجم ارجاعات، تعداد مسیرهای کنترلی و پیچیدگی سراسری آن بسیار زیاد است. بنابراین اگر بتوان نرم‌افزار را به پیمانه‌هایی مناسب تقسیم نمود، پیچیدگی و هزینه کلی آن کاهش خواهد یافت. اما تعداد این پیمانه‌ها نباید بیش از حد باشد زیرا هزینه مجتمع‌سازی یا یکپارچه‌سازی و اتصال آن‌ها افزایش می‌یابد.

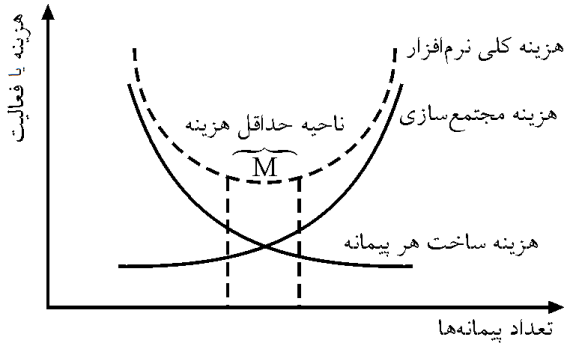
در یک نرم‌افزار با کاهش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها کاهش می‌یابد. اما هزینه ساخت هر پیمانه افزایش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با افزایش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها افزایش می‌یابد. اما هزینه ساخت هر پیمانه کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با داشتن تعداد مناسب پیمانه (ماژول)، همه هزینه‌ها کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد.

شکل زیر رابطه تعداد پیمانه‌های نرم‌افزار با هزینه توسعه نرم‌افزار را نشان می‌دهد. در این شکل، یکی از منحنی‌ها هزینه توسعه یک پیمانه را نشان می‌دهد و منحنی دیگر، هزینه

یکپارچه‌سازی پیمانه‌ها را نشان می‌دهد. به تبع هزینه کلی نرم‌افزار برابر حاصل جمع این دو منحنی در هر نقطه است، که با منحنی خط‌چین نشان داده شده است.



نمودار فوق نشان می‌دهد که هزینه یا فعالیت لازم برای ساخت پیمانه‌های نرم‌افزاری با افزایش تعداد پیمانه‌ها الزاما کاهش می‌یابد ولی به موازات رشد بیش از حد پیمانه‌ها، هزینه مربوط به اجتماع و یکپارچه‌سازی پیمانه‌ها نیز رشد می‌کند. در واقع همواره با تعداد مناسبی از پیمانه‌ها که با M نشان داده شده است، می‌توان هزینه و کار را کمینه کرد.

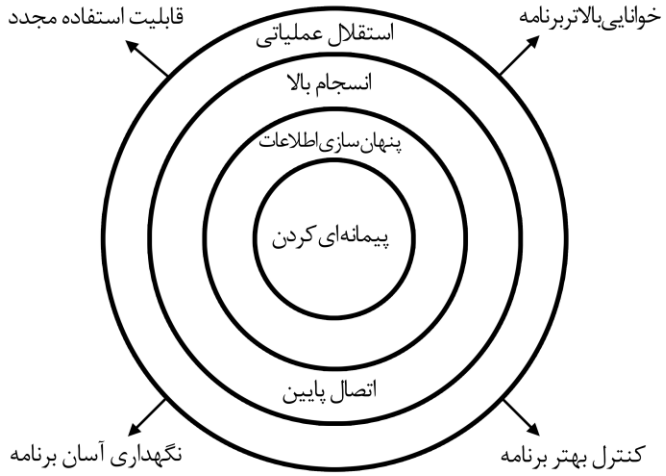
استقلال عملیاتی، نتیجه پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی انسجام بالا و اتصال پایین است. در صورتی که پیمانه‌هایی را با عملکرد تک‌منظوره(انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد.

توجه: استقلال عملیاتی خوب، کلید طراحی خوب و طراحی خوب، کلید یک نرم‌افزار با کیفیت می‌باشد. استقلال عملیاتی خوب با دو مفهوم انسجام (Cohesion) و اتصال (Coupling) مورد ارزیابی قرار می‌گیرد. انسجام، معیاری است که توان نسبی کارکردی یک پیمانه را نشان می‌دهد و اتصال، معیاری است که میزان نسبی وابستگی پیمانه‌ها به یکدیگر را نشان می‌دهد.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد** پیمانه‌های (توابع) برنامه جاری در برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.

شکل زیر گویای مطلب است:



توجه: مهمترین مزیت سیستم‌های یکپارچه، سرعت نسبتاً بالای آنهاست، زیرا معمولاً هر تابع و روالی می‌تواند بدون محدودیت توابع دیگر را فراخوانی کند. اما در مقابل معایب اساسی زیادی نسبت به ساختارهای سیستم عامل‌های بعدی دارد که می‌توان از آن‌ها (نقیض آن‌ها) به عنوان معیار سیستم عامل خوب نیز یاد کرد. این معایب عبارتند از:

۱- عدم وجود (یا ناچیز بودن) طراحی پیمانه‌ای (Modularity).

۲- عدم انعطاف‌پذیری (Flexibility) یعنی برای اینکه بتوان آن را با عملکردهای جدید تطبیق داد باید تغییرات اساسی میان رویه‌های آن ایجاد کرد.

۳- پیچیدگی نگهداری (Maintainability)

۴- پیچیدگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).

۵- پیچیدگی پیاده‌سازی.

۶- قابلیت اطمینان (Reliability) کم (به علت پیچیدگی و درهم ریختگی ساختار آن).

۲- ساختار لایه‌ای (Layered)

در این ساختار، سیستم عامل به صورت چند لایه طراحی می‌شود که هر لایه بر روی دیگری قرار گرفته است و می‌توان هر لایه را مستقل از لایه دیگر طراحی کرد و گسترش داد. هر لایه در این ساختار به لایه بالاتر از خود سرویس داده و جزئیات کار را از دید آن مخفی می‌سازد. به این ترتیب هر لایه می‌تواند از توابع و سرویس‌های لایه پایین‌تر استفاده کند بدون اینکه بداند این سرویس چگونه پیاده‌سازی شده است. تنها باید بداند هر سرویس چه می‌کند و چگونه و از طریق چه واسطهایی می‌توان به آن سرویس دسترسی پیدا کرد.

اولین سیستمی که به این روش طراحی شد، سیستم THE بود که توسط دِکسترا و

دانشجویانش در سال 1968 ساخته شد. این سیستم دارای شش لایه بود که در شکل زیـر نمـایش داده شده است:

لایه	وظیفه
5	اپراتور
4	برنامه‌های کاربردی
3	مدیریت ورودی / خروجی
2	ارتباط فرآیند – اپراتور
1	مدیریت حافظه اصلی و جانبی (drum)
0	تخصیص پردازنده و چندبرنامگی

ساختار سیستم عامل THE

در ادامه شرح مختصری برای این لایه‌ها بیان می‌شود:

لایه صفر (تخصیص پردازنده و چندبرنامگی): وظیفه این لایه تعویض مـتن در زمـان وقـوع وقفه یا پایان برش زمانی می‌باشد.

لایه یک (مدیریت حافظه اصلی و جانبی): ایـن لایـه فضـایی از حافظـه اصـلی و همچنـین بخشی به اندازه 512 هزار کلمه بر روی drum را به فرآینـدها تخصـیص می‌دهـد. از drum بـرای نگهداری بخش‌هایی (صفحه‌هایی) از اطلاعات مورد نیاز فرآیندها که در حافظـه اصـلی بـه علـت نبودن فضای کافی جا نگرفته‌اند استفاده می‌شود تا در صورت لزوم با داده‌های داخل حافظه اصلی در فضای آدرس خود فرآیند، جابه‌جا شوند.

لایه دو (ارتباط فرآیند و اپراتور): وظیفه این لایـه برقـراری ارتبـاط بـین فرآینـد و کنسـول اپراتور است، یعنی در بالای این لایه، هر فرآیند به کنسول اپراتور مخصـوص بـه خـودش متصـل می‌شود.

لایه سه (مدیریت ورودی و خروجی): وظیفه این لایه مدیریت دستگاه‌های I/O و بافر کردن جریان اطلاعات مربوطه می‌باشد.

لایه چهار (برنامه‌های کاربردی): در این لایه برنامه‌های کاربردی قـرار می‌گیرنـد کـه از نظـر نیازهای تخصیص پردازنده، مدیریت حافظه، ارتباط با کنسول و I/O توسط لایه‌های زیرین تـأمین شده‌اند.

لایه پنج (اپراتور): فرآیند اپراتور سیستم در این لایه قرار دارد.

مزیت‌های سیستم‌های لایه‌ای:

۱- طراحی پیمانه‌ای (Modularity) مناسب.

توجه: مزیت اصلی روش لایه‌ای، پیمانه‌ای بودن (Modularity) است، یعنی لایه‌ها به نحـوی

تقسیم‌بندی می‌شوند که هر لایه فقط توابع و سرویس‌های لایه‌های پایین‌تر را استفاده می‌کند. بدین ترتیب هر لایه را می‌توان مستقل از لایه‌های دیگر و به راحتی طراحی کرد، خطایابی و نگهداری کرد و توسعه داد.

۲- انعطاف‌پذیری (Flexibility) مناسب در برابر تغییرات.

۳- قابلیت نگهداری (Maintainability) مناسب.

۴- سادگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).

۵- سادگی در پیاده‌سازی.

۶- قابلیت اطمینان (Reliability) مناسب. (توانایی در محافظت لایه‌ها از یکدیگر)

معایب سیستم‌های لایه‌ای:

۱- کاهش کارایی

توجه: یک نقض سیستم‌های لایه‌ای این است که ممکن است قدری نسبت به سیستم‌های دیگر کند به نظر برسند و به تبع منجر به «**کاهش کارایی**» گردد، زیرا دستورات و فراخوانی‌ها از لایه بالا به سمت لایه پایین حرکت می‌کنند و زمان زیادی برای این روال صرف می‌شود. در واقع هر لایه قدری سربار (Overhead) به دستورات و فراخوانی‌ها می‌افزاید، در نتیجه یک فراخوانی سیستمی در این ساختار، زمان بیشتری نسبت به ساختار غیر لایه‌ای صرف می‌کند. برای رفع این نقص سعی می‌شود تعداد لایه‌های کمتری با قابلیت عمل بیشتری طراحی شود. به عنوان مثال محصول اولیه Windows NT با لایه‌های زیاد، کارایی کمتری نسبت به ویندوز 95 داشت. در NT 4.0 سعی شد لایه‌ها به همدیگر نزدیکتر و مجتمع‌تر شوند تا کارایی بیشتر گردد.

۲- نیاز به دقت بالا در پیمانه‌ای کردن و تقسیم‌بندی لایه‌ها دارد، مسأله اصلی در سیستم‌های لایه‌ای، تعریف مناسب هر لایه و سرویس‌های درون آن می‌باشد. از آنجا که هر لایه فقط می‌تواند از سرویس‌های لایه پایین‌تر استفاده کند، در طراحی لایه‌ها باید دقت بسیار به خرج داد.

۳- ساختار ماشین مجازی (Virtual Machine)

در این ساختار، یک سیستم عامل بر روی سخت‌افزار اجرا شده و برای لایه بالاتر چندین ماشین مجازی با تمام جزئیات سخت‌افزاری فراهم می‌کند که بر روی هر یک از این ماشین‌های مجازی می‌توان یک سیستم عامل جداگانه نصب و اجرا کرد.

این ماشین مجازی (که پایین‌ترین لایه را برای لایه‌های بالاتر فراهم می‌کند) فقط یک ماشین توسعه یافته (مثلاً با سیستم فایل و دیگر امکانات پیشرفته) نیست، بلکه کپی دقیقی از سخت‌افزار (شامل همه قسمت‌هایی که یک سخت‌افزار واقعی دارد) است.

توجه: جهت درک چگونگی پیاده‌سازی سیستم عامل‌های ماشین مجازی به شکل زیر دقت کنید:

فرآیندها	فرآیندها	فرآیندها
هسته سیستم عامل ۱	هسته سیستم عامل ۲	هسته سیستم عامل ۳
ماشین مجازی		
سخت‌افزار		

مدل ماشین مجازی

فرآیندها
هسته سیستم عامل
سخت افزار

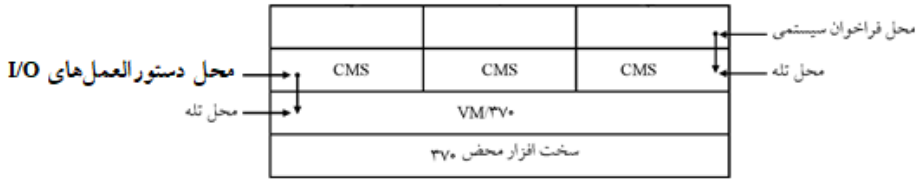
مدل ماشین غیر مجازی

یک سیستم اشتراک زمانی دو وظیفه عمده و مجزا دارد:

۱- مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع)

۲- مدیریت واسط کاربر (یک ماشین توسعه‌یافته با استفاده از یک واسط که بر روی سخت‌افزار خام قرار گیرد.)

اما ایده ساختار ماشین مجازی این است که دو وظیفه فوق از یکدیگر جدا شوند، سیستم عامل VM/370 بر روی سیستم‌های IBM بهترین مثال از مفهوم ماشین مجازی است. قلب سیستم که به نام مانیتور ماشین مجازی (VMM یا Virtual machin manitor) یا Hypervisor معروف است بر روی سخت‌افزار عریان اجرا شده و فقط وظیفه اول یعنی مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع) را انجام می‌دهد و اکنون می‌توان یک یا چند ماشین مجازی را بر روی لایه بالاتر از آن قرار داد، انگار که یک ماشین فیزیکی کامل، به چندین نسخه ماشین مجازی کامل تکثیر شده است. البته باید توجه داشته باشید که ماشین‌های مجازی برخلاف سیستم‌های عامل یک ماشین توسعه‌یافته نیستند بلکه هر کدام یک نسخه دقیق از سخت‌افزار خام هستند که شامل تمام خصوصیات سخت‌افزار مانند مد کاربر، مد هسته، I/O و وقفه می‌باشند. این ماشین مجازی انگار که همان وظیفه دوم یعنی مدیریت واسط کاربر است، اما فقط در حد یک سخت افزار خشک و خالی است. اکنون می‌توان بر روی هر کدام از این ماشین‌های مجازی یک سیستم عامل معمولی نصب نمود، این عمل همان عمل نصب سیستم عامل بر روی یک سخت‌افزار خام حقیقی است. شکل زیر ساختار VM/370 را نشان می‌دهد.



توجه: هر کاربر یک برنامه (CMS: Conversational sational monitor system) مخصوص به خود را دارد که یک سیستم عامل تک کاربره محاوره‌ای است.

توجه: سیستم عامل‌هایی که بر روی ماشین‌های مجازی اجرا می‌شوند، هیچ‌گونه ارتباطی با یکدیگر ندارند، در واقع هر کدام خود را در یک سیستم مجزا تصور می‌کنند بنابراین می‌توان بر روی هر ماشین مجازی یک نوع سیستم عامل مجزا نصب نمود.

همانطور که مشخص است هر ماشین مجازی باید از دیگران کاملاً مستقل باشد یا حداقل اینطور تصور کند. حال سوال این است که این استقلال چگونه بر روی یک سخت‌افزار مشترک پیاده‌سازی می‌شود؟ جواب تسهیم (Multiplexing) و Slooping می‌باشد.

در واقع کافی است منابع سیستم با توجه به خصوصیاتش به دو صورت زمانی یا فضایی بین ماشین‌های مجازی تسهیم و یا Spool شوند. به عنوان مثال:

- ایجاد پردازنده مجازی با تسهیم زمانی پردازنده حقیقی که این عمل به کمک PCBهای مجازی انجام می‌شود.
- ایجاد حافظه‌های مجازی با تسهیم فضای حافظه اصلی بین ماشین‌های مجازی.
- Spool کردن دستگاه‌های ورودی و خروجی.
- تشکیل دیسک مجازی با تسهیم فضای دیسک حقیقی.

همانطور که مشخص است دستیابی به سخت‌افزار و فراخوان‌های سیستمی نمی‌تواند بر روی CMS اجرا شود و در واقع CMS قدرت این کار را ندارد، پس برنامه اجرا شده در CMS چگونه فراخوانی سیستمی انجام می‌دهد؟ فرض کنید برنامه‌ای که روی CMS اجرا می‌شود قصد انجام عمل I/O دارد و یک فراخوانی سیستمی صادر می‌کند. این فراخوانی سیستمی باعث تغییر مد کاربر به مد هسته و می‌شود و سیستم عامل روی CMS (از آنجا که نمی‌داند CMS سخت‌افزار حقیقی نیست) سعی می‌کند که این عمل را روی CMS اجرا کند و در واقع یک تله مجازی در CMS رخ می‌دهد، اکنون CMS که در حقیقت یک نرم‌افزار است، این درخواست را به VM/370 ارجاع می‌دهد و اکنون یک تله واقعی در VM/370 رخ می‌دهد و درخواست (با توجه به تسهیم یا Spool) به اجرا رسیده و پاسخ به لایه‌های بالاتر ارجاع می‌شود. از آنجا که وظیفه چندبرنامگی و ارائه ماشین توسعه یافته کاملاً مجزا از یکدیگر هستند، هر یک از این تکه برنامه‌ها بسیار ساده‌تر شده و از انعطاف‌پذیری (Flexibility) بیشتر و قابلیت نگهداری (Maintainability) آسان‌تری برخوردار هستند.

توجه: در ساختار ماشین مجازی دو وظیفه اصلی چندبرنامگی و ایجاد واسط راحت (مستقل از سخت افزار) از یکدیگر مجزا شده‌اند. مانیتور ماشین مجازی وظیفه چندبرنامگی را بر عهده دارد و لایه بالای آن وظیفه ایجاد واسط کاربر با سخت افزار را بر عهده دارد. لذا هر یک از این بخش‌ها ساده‌تر شده و از قابلیت انعطاف بیشتری برخوردارند.

توجه: از آنجا که هر ماشین مجازی کاملا مشابه سخت افزار واقعی است، هر یک از آنها می‌توانند هر سیستم عاملی را مستقلا اجرا کنند لذا می‌توان همزمان سیستم‌عامل‌های مختلفی را روی این

ماشین اجرا کرد.

یکی دیگر از کاربردهای ماشین‌های مجازی به مجازی‌سازی محیط برنامه‌نویسی (Programming-environment Virtualization) موسوم است، در این روش VMMها سخت افزار واقعی را جهت اجرای سیستم‌عامل‌های متعدد نسخه‌برداری و مجازی‌سازی نمی‌کنند، بلکه هدف ایجاد قابلیت حمل (Portability) برنامه‌ها بر روی سخت‌افزار و سیستم‌عامل‌های مختلف است. کامپایلر زبان java که توسط شرکت Sun Microsystem طراحی شده است، یک خروجی بایت کد (byte code) تولید می‌کند. این بایت کدها دستوراتی هستند که بر روی ماشین مجازی جاوا (JVM) اجرا می‌شوند. جهت اجرای برنامه‌های java در یک ماشین، آن کامپیوتر می‌بایست دارای یک JVM باشد. بدین ترتیب برنامه‌هایی که به زبان java نوشته شده‌اند به راحتی بر روی انواع کامپیوترها اجرا می‌شوند. فقط کافی است بایت کدها را روی آن ماشین کامپایل کرد. بدیهی است به علت نیاز به کامپایل شدن بایت کدها، برنامه‌های جاوا سرعت کمتری نسبت به برنامه‌هایی نظیر C دارد. برنامه‌های C توسط کامپایلر بومی یک کامپیوتر، برای یکبار تبدیل به زبان ماشین آن کامپیوتر می‌گردد. پس خروجی زبان ماشین کامپایلر C از یک نوع کامپیوتر به کامپیوتر دیگر متفاوت است ولی بایت کدهای خروجی java برای همه ماشین‌ها یکسان است.

مزیت‌های سیستم‌های ماشین‌های مجازی:

۱- امکان اجرای چند سیستم عامل بر روی یک ماشین حقیقی (این سیستم عامل‌ها از یک زیرساخت مشترک پیروی می‌کنند).

۲- استقلال کامل ماشین‌های مجازی قرار گرفته روی یک ماشین حقیقی و امنیت بالای آن‌ها.

۳- با در اختیار داشتن چند ماشین مجازی بدون استفاده از سخت‌افزار اضافی و با هزینه پایین می‌توان نصب و تست سیستم عامل‌های مختلف و حتی شبکه‌های کامپیوتری انجام داد.

۳- در ماشین مجازی java قابلیت حمل (Protability) بالایی وجود دارد.

معایب سیستم‌های ماشین‌های مجازی:

۱- به دلیل ساخت نسخه‌های متعدد از سخت افزار پیچیدگی پیاده‌سازی آن بسیار زیاد است.

۲- به دلیل سربار حاصل از دوبار وقفه (وقفه مجازی و وقفه واقعی) کارایی آن پایین است.

۴- ساختار ریز هسته (Microkernel)

ایده‌ی اصلی در این ساختار، هر چه کوچک‌تر و ساده‌تر نمودن قسمت هسته (Kernel) سیستم عامل است. بنابراین به دلیل اینکه یک هسته بسیار کوچک خواهیم داشت به ساختار **ریز هسته** (Microkernel) **ساختار مشتری – سرویس‌دهنده** (Client - Server) نیز گفته می‌شود.

در سیستم عامل‌های مشتری – سرویس‌دهنده، قسمت اعظم کد سیستم عامل در حالت کاربر اجرا می‌شود و وظیفه هسته برقراری ارتباط میان فرایندهای سرویس‌دهنده و مشتری است. در این ساختار درخواست فرایند کاربر (که به آن فرایند مشتری گویند) به یک فرایند سرویس‌دهنده فرستاده شده و پاسخ آن به مشتری برگشت داده می‌شود.

توجه: امروزه سیستم عامل‌ها به جای داشتن یک هسته بزرگ به سمت داشتن هسته هرچه کوچکتر حرکت می‌کنند که به آن اصطلاحاً **ریز هسته** می‌گویند. در واقع فقط چند عمل اصلی مانند ارتباط بین فرآیندها و زمانبندی پایه‌ای را به هسته واگذار می‌کنند و دیگر خدمات سیستم عامل توسط تعدادی فرآیند سرویس‌دهنده صورت می‌گیرد. این همان ایده‌ی مدل مشتری – سرویس‌دهنده است که به آن **معماری ریز هسته** نیز می‌گویند.

توجه: روند طراحی سیستم عامل‌های جدید همواره با این این ایده همراه بوده است که تا جایی که ممکن است کدها به لایه‌های بالاتر منتقل شوند تا نهایتا یک هسته کمینه پدید آید. برای این منظور اکثر وظایف سیستم عامل را در سطح کاربر و مشابه فرآیندهای کاربر پیاده‌سازی می‌کنند. برای درخواست یک سرویس، مانند خواندن یک بلوک از فایل، فرآیند کاربر (که اکنون به عنوان فرآیند مشتری شناخته می‌شود) یک درخواست به فرآیند سرویس‌دهنده ارسال می‌نماید و از آن می‌خواهد که کارش را انجام دهد و پاسخ را برگرداند. کار هسته در این مدل برقراری ارتباط بین مشتری‌ها و سرویس‌دهنده از طریق تبادل پیام است. البته هسته، وظایف دیگری نیز دارد. به طور کلی وظایف هسته در این مدل به صورت زیر است:

۱-تبادل پیام مابین مشتری‌ها و سرویس‌دهنده‌ها

۲-زمان‌بندی پردازنده، فرآیندها و ایجاد چندبرنامگی

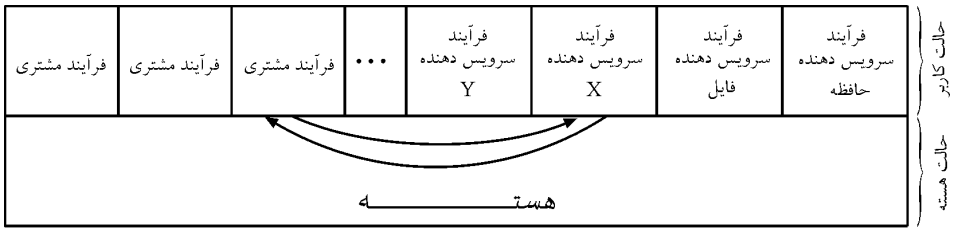
۳-بخش سطح پایین مدیریت حافظه مانند برنامه‌ریزی ثبات‌های سخت‌افزار مدیریت حافظه

۴- بخش سطح پایین مدیریت I/O که برنامه‌ریزی ثبات‌های ویژه کنترل کننده‌ها و کارهایی را بر عهده دارند که اگر در سطح کاربر انجام شود، آنگاه امنیت کل سیستم به مخاطره خواهد افتاد.

توجه: در این روش، سیستم عامل به چند بخش (خدمات فایل، خدمات فرآیند، خدمات ترمینال و خدمات حافظه) تقسیم شده است، که هریک به عنوان یک سرویس‌دهنده فقط یکی از وظایف سیستم عامل را انجام می‌دهد، بنابراین سیستم عامل را می‌توان ساده‌تر تحلیل، طراحی، پیاده‌سازی و نگهداری کرد.

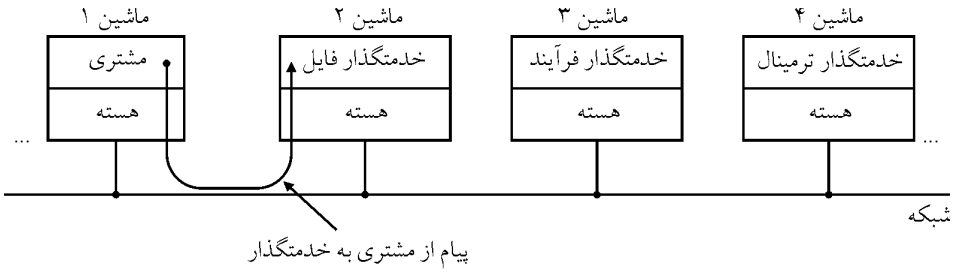
توجه: سرویس‌دهنده‌ها در مُد کاربر اجرا شده و ریز هسته با آنها همانند فرآیندهای کاربر رفتار می‌کند. بنابراین هیچ یک از سرویس‌دهنده‌ها دسترسی مستقیم به سخت‌افزار را ندارد. همچنین اگر کارکرد یکی از سرویس‌دهنده‌ها مختل شود، فقط همان بخش مختل می‌شود و این اختلال به دلیل استقلال عملیاتی هر سرویس‌دهنده به سایر سرویس‌دهنده‌ها و به تبع کل سیستم منتقل نمی‌شود.

توجه: نحوه‌ی پیاده‌سازی و عملکرد مدل مشتری – سرویس‌دهنده در شکل زیر نشان داده شده است:



ساختار مشتری – سرویس‌دهنده

در حالت دیگر می‌توان از مدل مشتری و سرویس‌دهنده در سیستم‌های توزیع شده استفاده نمود. به شکل زیر توجه کنید.



با توجه به شکل فوق فرض کنید، مشتری بر روی یک ماشین مجزا از سرویس‌دهنده اجرا شده است، در این حالت همانطور که در مدل مشتری و سرویس‌دهنده ماشین محلی، یک پیام درخواست از مشتری به سرویس‌دهنده از طریق هسته ارسال می‌شد و سرویس‌دهنده پاسخ می‌داد، در سیستم توزیع شده هم این پیام از مشتری به هسته ماشین مشتری و از آن به یک شبکه ارتباطی و از آن به هسته ماشین سرویس‌دهنده و سپس به سرویس‌دهنده انتقال می‌یابد و پاسخ هم همین مسیر را باز می‌گردد. نکته‌ای که در این روش وجود دارد این است که برای مشتری و یا سرویس‌دهنده بین پیامی که از ماشین محلی آن‌ها می‌آید با پیامی که از ماشین راه دور دیگر می‌آید، تفاوتی وجود ندارد (و در واقع برای آن‌ها قابل تشخیص نیست). به عبارت دیگر از آنجا که یک مشتری به وسیله ارسال پیام‌هایش با یک سرویس‌دهنده ارتباط برقرار می‌کند، مشتری نیاز ندارد که بداند آیا به پیغام وی به صورت محلی در ماشین خودش رسیدگی می‌شود و یا اینکه پیغام از طریق شبکه به یک ماشین دور ارسال می‌شود.

مزیت‌های سیستم‌های مشتری و سرویس‌دهنده:

مهمترین مزیت سیستم‌های مشتری و سرویس‌دهنده طراحی پیمانه‌ای عالی آن‌ها می‌باشد. از این رو که وظایف سیستم عامل به طور مفصل و منسجم از هم جدا شده و هر کدام در یک پیمانه

قرار گرفته‌اند، مزیت پیمانه‌ای بودن این روش باعث بروز خصوصیات زیر می‌شود:

۱- انعطاف‌پذیری (Flexibility) بالا

۲- قابلیت نگهداری (Maintainability) ، اشکال‌زدایی (Debug) و ترمیم (Recovery) بالا

۳- قابلیت اطمینان (Reliability) بالا

۴- مقیاس‌پذیری (قابلیت توسعه) بالا

۵- سهولت پیاده‌سازی

۶- مناسب بودن برای سیستم‌های شبکه‌ای و توزیعی

معایب سیستم‌های مشتری و سرویس‌دهنده:

در مقابل همین خاصیت پیمانه‌ای بودن و جدا کردن سرویس‌دهنده‌ها از مشتری‌ها باعث پایین آمدن کارایی (سرعت) این سیستم نسبت به سیستم‌های دیگر شده است. فرض کنید که مشتری اقدام به ارسال یک پیام درخواست می‌کند این پیام در حین ارسال توسط مشتری و دریافت توسط سرویس‌دهنده باعث وقوع 2 تله می‌شود و پاسخ آن نیز به همین صورت، بنابراین برای هر درخواست و پاسخ 4 تله رخ می‌دهد، 2 فراخوان سیستمی برای ارسال و دریافت «**پیام درخواست**» و همچنین 2 فراخوان سیستمی برای ارسال و دریافت «**پیام پاسخ**». همچنین در موازات آن فراخوانی‌های سیستمی برای مدیریت حافظه و I/O نیز رخ می‌دهد که کارایی را تا حد زیادی کاهش می‌دهد.

تست‌های فصل اول: مفاهیم اولیه

۹۵- کدام مورد از مزایای ساختار ریز هسته (Micro Kernel) در طراحی سیستم عامل نیست؟
(مهندسی *IT* – دولتی ۹۹)

۱) کارایی سیستم را افزایش می‌دهد.
۲) برای سیستم‌های توزیع شده مناسب است.
۳) اضافه کردن سرویس جدید نیازی به اصلاح هسته سیستم عامل ندارد.
۴) در صورت بروز خرابی در سرویس خارج از هسته، کل سیستم عامل از کار نمی‌افتد.

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل اول: مفاهیم اولیه

۹۵- گزینه (۱) صحیح است.

ساختارهای درونی طراحی سیستم عامل‌ها به صورت زیر است:

۱- ساختار یکپارچه (Monolithic)

ساختار سیستم عامل‌های یکپارچه به این صورت است که هیچ ساختاری ندارد! سیستم‌های یکپارچه به صورت مجموعه‌ای از رویه‌ها نوشته شده‌اند که هر یک می‌توانند دیگری را به هنگام نیاز فراخوانی کرده و برای انجام محاسبات خود از آن‌ها کمک بگیرند. هر یک از رویه‌ها دارای یک واسط شامل پارامترهای ورودی و خروجی است که به سادگی تعریف شده‌اند. همانطور که مشخص است این سیستم هیچ‌گونه نظم، ترتیب، دسته‌بندی و سلسله مراتب خاصی ندارد.

برای ساختن یک سیستم یکپارچه (ساختن object واقعی سیستم عامل) ابتدا باید تمام رویه‌ها در قالب تعدادی فایل و یا به صورت جداگانه کامپایل شوند و سپس با استفاده از یک پیونددهنده (Linker) به هم متصل شده و در فایل object نهایی قرار گیرند. در این ساختار تمام رویه‌ها همدیگر را می‌بینند. این به آن معنی است که بین رویه‌ها هیچ تعیین سطحی وجود ندارد و اصطلاحاً سیستم تخت است. در واقع برای مخفی کردن اطلاعات و محصورسازی (Encapsulation)، محدویت‌هایی وضع نشده است و از آنجاکه دسترسی به هر رویه‌ای امکان‌پذیر است لذا حفاظت وجود ندارد. پنهان‌سازی اطلاعات (information hiding) نتیجه پیمانه‌ای کردن (Modularity) است. به بیان دیگر شرط لازم برای برقراری پنهان‌سازی اطلاعات، پیمانه‌ای کردن است و شرط کافی برای برقراری پنهان‌سازی اطلاعات تعریف متغیرهای محلی و دستورالعمل‌های مرتبط با تابع است. در یک بیان ساده پنهان‌سازی اطلاعات می‌گوید بخشی از نرم‌افزار در داخل یک پیمانه (تابع) محصور شود. هدف از پنهان‌سازی اطلاعات، پنهان کردن روال انجام دستورات تابع و متغیرهای محلی در پس واسط یا بلاک تابع است. استفاده‌کنندگان پیمانه‌ها (توابع) نیازی به دانستن جزئیات داخلی پیمانه‌ها (توابع) ندارند.

توجه: تعریف متغیر سراسری در تابع ناقض اصل پنهان‌سازی اطلاعات است.

اما به هر حال حتی در سیستم‌های یکپارچه حداقل یک تابع اصلی برای فراخوانی توابع و رویه‌های دیگر وجود دارد

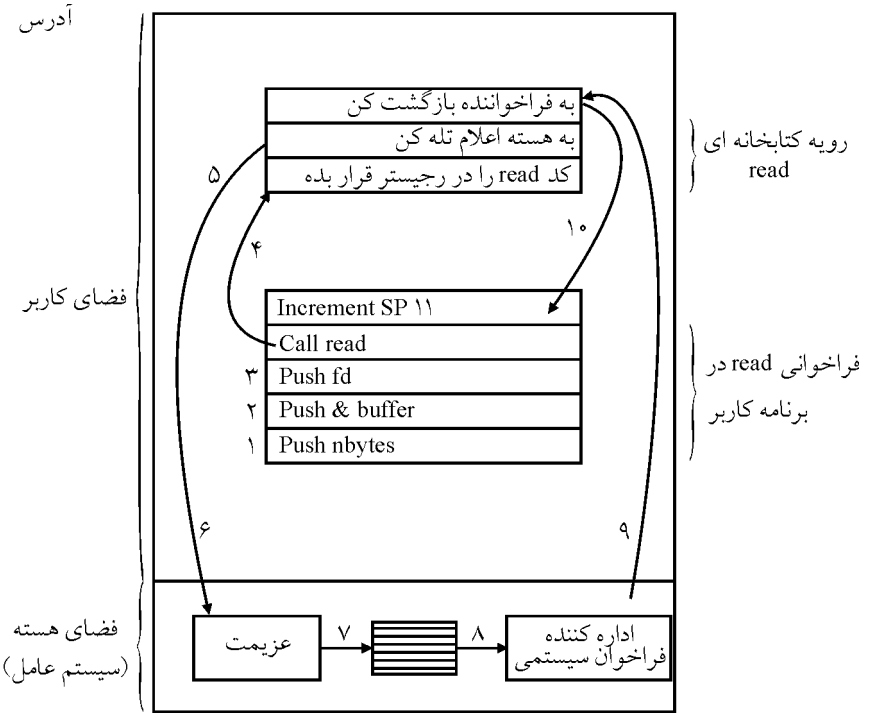
در این سیستم برای تقاضای یک سرویس از سرویس‌های سیستم عامل، ابتدا پارامترهای فراخوانی را در محل‌های از پیش تعیین شده مانند رجیسترها یا پشته قرار داده و سپس یک دستورالعمل تله مربوط به سرویس موردنظر اجرا می‌شود. این عمل به فراخوانی هسته یا فراخوانی راهبری معروف است. در واقع با این روش ماشین از مُد کاربر به مُد هسته می‌رود تا جهت انجام سرویس مورد نظر، کنترل در اختیار سیستم عامل قرار گیرد. اکنون با یک مثال چگونگی عملکرد فراخوان‌های سیستمی شرح داده می‌شود. فرض کنید باید فراخوانی زیر توسط برنامه اصلی (برنامه کاربر) انجام شود:

Count = Read (fd, buffer, nbyted);

تابع Read دارای سه پارامتر است، پارامتر اول مشخص‌کننده فایل است، پارامتر دوم یک اشاره‌گر به بافر است، و سومین پارامتر بیانگر تعدادی بایتی است که باید خوانده شود. این تابع تعداد بایت‌هایی که واقعاً از فایل خوانده و در بافر قرار داده است را برمی‌گرداند. در حالت عادی مقدار Count برابر با nbytes است ولی امکان دارد که کمتر از آن هم باشد (مثلاً به علت اتمام فایل).

در ابتدا برای صدا زدن رویه کتابخانه‌ای Read از کامپایلر، پارامترهای آن در پشته Push می‌شود (در شکل زیر این عمل با توجه به زبان برنامه‌نویسی C و C++ از پارامتر آخر به اول انجام می‌شود). با توجه به اینکه پارامتر دوم یعنی buffer یک اشاره‌گر است در هنگام push در ابتدای آن یک علامت & استفاده می‌شود، یعنی محتوای آن به عنوان یک آدرس در نظر گرفته شده است. مراحل push کردن پارامترها در شکل زیر با شماره‌های 1 تا 3 نمایش داده شده‌اند. اکنون باید رویه کتابخانه‌ای را فراخوانی کرد. برای این کار از یک دستورالعمل ساده فراخوانی رویه که برای صدا زدن همه رویه‌ها به کار می‌رود، استفاده می‌شود (مرحله 4). شماره فراخوانی سیستمی باید در محلی مانند یک رجیستر (که سیستم عامل انتظار دارد و در آنجا باشد) قرار گیرد (مرحله 5). اکنون باید دستور TRAP یا همان وقفه نرم‌افزاری (تله) رخ دهد تا کنترل اجرا از مُد کاربر به مُد هسته رفته و اجرای کُد مربوط به آن فراخوان از یک آدرس ثابت درون هسته آغاز شود (مرحله 6). اکنون هسته، شماره فراخوانی که در یک رجیستر گرفته بود را بررسی می‌کند و سپس به سراغ یک جدول که حاوی اشاره‌گرهایی به اداره‌کننده‌های فراخوانی‌های سیستمی (Systemcall Handler) هستند، می‌رود و با استفاده از شماره فراخوانی از بین درایه‌های جدول، محل قرار گرفتن اداره‌کننده فراخوانی مذکور (Read) را می‌یابد که عموماً محلی از حافظه است (مرحله 7) و آن را به اجرا درمی‌آورد (مرحله 8). در ادامه، اجرای اداره‌کننده فراخوانی سیستمی به طور کامل تمام می‌شود یا ممکن است تمام نشود و کنترل به فرآیند جدید دیگری منتقل شود و به تبع مراحل 1 تا 6 رویه کتابخانه‌ای در فضای کاربر برای فرآیند جدید هم اجرا شود. اینکه گفته می‌شود ممکن است به دلیل آن است که اگر فراخوان سیستمی در فرآیند جاری مجبور به انتظار باشد، مانند اینکه فراخوان سیستمی تلاش کند از صفحه کلید بخواند ولی هنوز چیزی از طرف

کاربر تایپ نشده باشد، از آنجا فراخوان سیستمی منتظر و مسدود شده داخل فرآیند جاری است بنابراین فرآیند جاری از طرف سیستم عامل به صف مسدود و منتظر منتقل می‌شود. در این شرایط سیستم عامل به صف فرآیندهای آماده اجرا نگاه می‌کند تا ببیند آیا فرآیند دیگری می‌تواند پس از آن اجرا شود یا خیر. در صورت وجود فرآیند آماده جدید سیستم عامل تعویض متن انجام داده و پردازنده را در اختیار فرآیند جدید قرار می‌دهد. در ادامه هرگاه ورودی مورد نظر فراخوان سیستمی فرآیند قبل آماده شود، برای مثال کلیدی فشرده شود، فرآیند قبلی به صف آماده سیستم عامل منتقل می‌شود و ممکن است در دفعه بعد توسط زمان‌بند کوتاه مدت سیستم عامل جهت اجرا انتخاب شود و مراحل 9 تا 11 انجام می‌شود. (مرحله 9) سپس این رویه به روش بازگشت به برنامه کاربر بازمی‌گردد. (مرحله 10) برای خاتمه کار، برنامه کاربر پشته را پاک‌سازی می‌کند، همان‌گونه که پس از بازگشت از همه رویه‌ها این عمل صورت می‌گیرد. (مرحله 11)



ساختار کلی این سیستم عامل‌ها از سه بخش زیر تشکیل شده است:

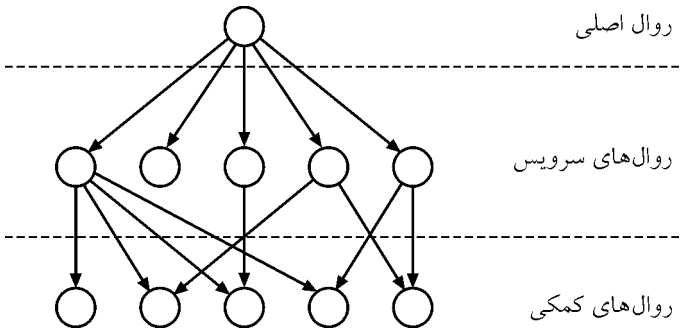
۱- یک برنامه اصلی که می‌تواند رویه سرویس‌های خواسته شده را فراخوانی کند. این برنامه به ازای هر درخواست، یک روال سرویس را صدا می‌زند.

۲- مجموعه‌ای از رویه‌های سرویس که می‌توانند تعدادی فراخوانی سیستمی انجام دهند و یا تعدادی روال کمکی را فراخوانی کنند.

۳- تعدادی رویه کمکی و سودمند (Utility Procedures) که می‌توانند به رویه‌های سرویس

کمک کنند و توسط رویه‌های سرویس فراخوانی شوند.

این مفاهیم در شکل زیر نشان داده شده است:



ساختار سیستم عامل‌های یکپارچه

. پیمانه‌ای کردن یعنی تقسیم نرم‌افزار (در اینجا خود برنامه سیستمی سیستم عامل) به چند مولفه جداگانه و متمایز که این مولفه‌ها به عنوان پیمانه‌هایی هستند که از اجتماع آن‌ها، خواسته‌های مساله برآورده می‌شود. با این عمل مدیریت مفهومی یک برنامه حاصل می‌شود و قابلیت خوانایی نرم‌افزار چندین برابر می‌گردد. درک و فهم نرم‌افزار یکپارچه (monolithic) که تنها از یک پیمانه (ماژول) تشکیل شده است، بسیار مشکل است، زیرا تعداد متغیرها، حجم ارجاعات، تعداد مسیرهای کنترلی و پیچیدگی سراسری آن بسیار زیاد است. بنابراین اگر بتوان نرم‌افزار را به پیمانه‌هایی مناسب تقسیم نمود، پیچیدگی و هزینه کلی آن کاهش خواهد یافت. اما تعداد این پیمانه‌ها نباید بیش از حد باشد زیرا هزینه مجتمع‌سازی یا یکپارچه‌سازی و اتصال آن‌ها افزایش می‌یابد.

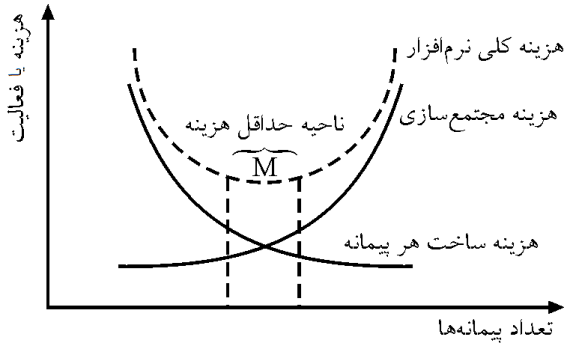
در یک نرم‌افزار با کاهش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها کاهش می‌یابد. اما هزینه ساخت هر پیمانه افزایش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با افزایش تعداد پیمانه‌ها (ماژول‌ها)، هزینه یکپارچه‌سازی آن‌ها افزایش می‌یابد. اما هزینه ساخت هر پیمانه کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، افزایش می‌یابد.

همچنین در یک نرم‌افزار با داشتن تعداد مناسب پیمانه (ماژول)، همه هزینه‌ها کاهش می‌یابد. زیرا هم هزینه یکپارچه‌سازی و هم هزینه ساخت هر پیمانه، هر دو کاهش می‌یابد، بنابراین هزینه کلی آن نیز که حاصل جمع دو مقدار هزینه یکپارچه‌سازی و هزینه ساخت هر پیمانه است، کاهش می‌یابد.

شکل زیر رابطه تعداد پیمانه‌های نرم‌افزار با هزینه توسعه نرم‌افزار را نشان می‌دهد. در این شکل، یکی از منحنی‌ها هزینه توسعه یک پیمانه را نشان می‌دهد و منحنی دیگر، هزینه

یکپارچه‌سازی پیمانه‌ها را نشان می‌دهد. به تبع هزینه کلی نرم‌افزار برابر حاصل جمع این دو منحنی در هر نقطه است، که با منحنی خط‌چین نشان داده شده است.



نمودار فوق نشان می‌دهد که هزینه یا فعالیت لازم برای ساخت پیمانه‌های نرم‌افزاری با افزایش تعداد پیمانه‌ها الزاما کاهش می‌یابد ولی به موازات رشد بیش از حد پیمانه‌ها، هزینه مربوط به اجتماع و یکپارچه‌سازی پیمانه‌ها نیز رشد می‌کند. در واقع همواره با تعداد مناسبی از پیمانه‌ها که با M نشان داده شده است، می‌توان هزینه و کار را کمینه کرد.

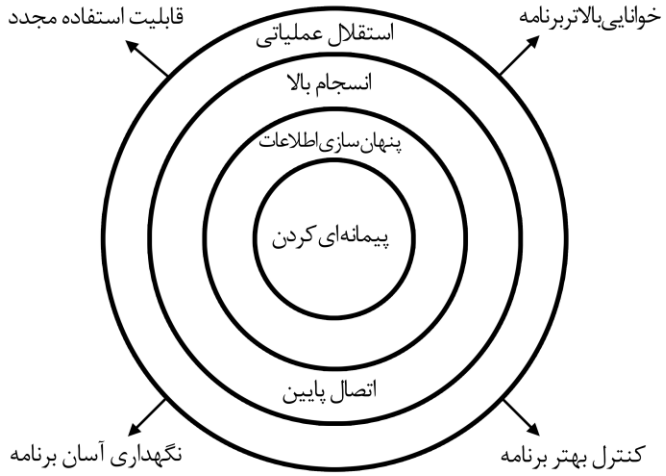
استقلال عملیاتی، نتیجه پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است. به بیان دیگر شرط لازم برای برقراری استقلال عملیاتی، پیمانه‌ای‌کردن و پنهان‌سازی اطلاعات است و شرط کافی برای برقراری استقلال عملیاتی انسجام بالا و اتصال پایین است. در صورتی که پیمانه‌هایی را با عملکرد تک‌منظوره(انسجام بالا) و عدم ارتباط بیش از حد با پیمانه‌های دیگر (اتصال پایین) ایجاد کنیم، استقلال عملیاتی تحقق می‌یابد.

توجه: استقلال عملیاتی خوب، کلید طراحی خوب و طراحی خوب، کلید یک نرم‌افزار با کیفیت می‌باشد. استقلال عملیاتی خوب با دو مفهوم انسجام (Cohesion) و اتصال (Coupling) مورد ارزیابی قرار می‌گیرد. انسجام، معیاری است که توان نسبی کارکردی یک پیمانه را نشان می‌دهد و اتصال، معیاری است که میزان نسبی وابستگی پیمانه‌ها به یکدیگر را نشان می‌دهد.

توجه: در یک طراحی ایده‌آل، هدف، محقق کردن بالاترین سطح انسجام (Cohesion بالا) داخل پیمانه‌های برنامه و کمترین سطح اتصال (Coupling پایین) مابین پیمانه‌های برنامه است.

توجه: پیمانه‌ای‌کردن، برقراری پنهان‌سازی اطلاعات، انسجام بالا، اتصال پایین و برقراری استقلال عملیاتی، به عنوان اصول و معیارهای طراحی معماری مطلوب، منجر به **خوانایی بالاتر برنامه، کنترل بهتر برنامه، قابلیت استفاده مجدد** پیمانه‌های (توابع) برنامه جاری در برنامه‌های آتی و **نگهداری آسان برنامه** جاری می‌گردد.

شکل زیر گویای مطلب است:



توجه: مهمترین مزیت سیستم‌های یکپارچه، سرعت نسبتاً بالای آنهاست، زیرا معمولاً هر تابع و روالی می‌تواند بدون محدودیت توابع دیگر را فراخوانی کند. اما در مقابل معایب اساسی زیادی نسبت به ساختارهای سیستم عامل‌های بعدی دارد که می‌توان از آن‌ها (نقیض آن‌ها) به عنوان معیار سیستم عامل خوب نیز یاد کرد. این معایب عبارتند از:

۱- عدم وجود (یا ناچیز بودن) طراحی پیمانه‌ای (Modularity).

۲- عدم انعطاف‌پذیری (Flexibility) یعنی برای اینکه بتوان آن را با عملکردهای جدید تطبیق داد باید تغییرات اساسی میان رویه‌های آن ایجاد کرد.

۳- پیچیدگی نگهداری (Maintainability)

۴- پیچیدگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).

۵- پیچیدگی پیاده‌سازی.

۶- قابلیت اطمینان (Reliability) کم (به علت پیچیدگی و درهم ریختگی ساختار آن).

۲- ساختار لایه‌ای (Layered)

در این ساختار، سیستم عامل به صورت چند لایه طراحی می‌شود که هر لایه بر روی دیگری قرار گرفته است و می‌توان هر لایه را مستقل از لایه دیگر طراحی کرد و گسترش داد. هر لایه در این ساختار به لایه بالاتر از خود سرویس داده و جزئیات کار را از دید آن مخفی می‌سازد. به این ترتیب هر لایه می‌تواند از توابع و سرویس‌های لایه پایین‌تر استفاده کند بدون اینکه بداند این سرویس چگونه پیاده‌سازی شده است. تنها باید بداند هر سرویس چه می‌کند و چگونه و از طریق چه واسطهایی می‌توان به آن سرویس دسترسی پیدا کرد.

اولین سیستمی که به این روش طراحی شد، سیستم THE بود که توسط دِکسترا و

دانشجویانش در سال 1968 ساخته شد. این سیستم دارای شش لایه بود که در شکل زیـر نمـایش داده شده است:

لایه	وظیفه
5	اپراتور
4	برنامه‌های کاربردی
3	مدیریت ورودی / خروجی
2	ارتباط فرآیند – اپراتور
1	مدیریت حافظه اصلی و جانبی (drum)
0	تخصیص پردازنده و چندبرنامگی

ساختار سیستم عامل THE

در ادامه شرح مختصری برای این لایه‌ها بیان می‌شود:

لایه صفر (تخصیص پردازنده و چندبرنامگی): وظیفه این لایه تعویض مـتن در زمـان وقـوع وقفه یا پایان برش زمانی می‌باشد.

لایه یک (مدیریت حافظه اصلی و جانبی): ایـن لایـه فضـایی از حافظـه اصـلی و همچنـین بخشی به اندازه 512 هزار کلمه بر روی drum را به فرآینـدها تخصـیص می‌دهـد. از drum بـرای نگهداری بخش‌هایی (صفحه‌هایی) از اطلاعات مورد نیاز فرآیندها که در حافظـه اصـلی بـه علـت نبودن فضای کافی جا نگرفته‌اند استفاده می‌شود تا در صورت لزوم با داده‌های داخل حافظه اصلی در فضای آدرس خود فرآیند، جابه‌جا شوند.

لایه دو (ارتباط فرآیند و اپراتور): وظیفه این لایـه برقـراری ارتبـاط بـین فرآینـد و کنسـول اپراتور است، یعنی در بالای این لایه، هر فرآیند به کنسول اپراتور مخصـوص بـه خـودش متصـل می‌شود.

لایه سه (مدیریت ورودی و خروجی): وظیفه این لایه مدیریت دستگاه‌های I/O و بافر کردن جریان اطلاعات مربوطه می‌باشد.

لایه چهار (برنامه‌های کاربردی): در این لایه برنامه‌های کاربردی قـرار می‌گیرنـد کـه از نظـر نیازهای تخصیص پردازنده، مدیریت حافظه، ارتباط با کنسول و I/O توسط لایه‌های زیرین تـأمین شده‌اند.

لایه پنج (اپراتور): فرآیند اپراتور سیستم در این لایه قرار دارد.

مزیت‌های سیستم‌های لایه‌ای:

۱- طراحی پیمانه‌ای (Modularity) مناسب.

توجه: مزیت اصلی روش لایه‌ای، پیمانه‌ای بودن (Modularity) است، یعنی لایه‌ها به نحـوی

تقسیم‌بندی می‌شوند که هر لایه فقط توابع و سرویس‌های لایه‌های پایین‌تر را استفاده می‌کند. بدین ترتیب هر لایه را می‌توان مستقل از لایه‌های دیگر و به راحتی طراحی کرد، خطایابی و نگهداری کرد و توسعه داد.

۲- انعطاف‌پذیری (Flexibility) مناسب در برابر تغییرات.

۳- قابلیت نگهداری (Maintainability) مناسب.

۴- سادگی در اشکال‌زدایی (Debug) و ترمیم (Recovery).

۵- سادگی در پیاده‌سازی.

۶- قابلیت اطمینان (Reliability) مناسب. (توانایی در محافظت لایه‌ها از یکدیگر)

معایب سیستم‌های لایه‌ای:

۱- کاهش کارایی

توجه: یک نقض سیستم‌های لایه‌ای این است که ممکن است قدری نسبت به سیستم‌های دیگر کند به نظر برسند و به تبع منجر به «**کاهش کارایی**» گردد، زیرا دستورات و فراخوانی‌ها از لایه بالا به سمت لایه پایین حرکت می‌کنند و زمان زیادی برای این روال صرف می‌شود. در واقع هر لایه قدری سربار (Overhead) به دستورات و فراخوانی‌ها می‌افزاید، در نتیجه یک فراخوانی سیستمی در این ساختار، زمان بیشتری نسبت به ساختار غیر لایه‌ای صرف می‌کند. برای رفع این نقص سعی می‌شود تعداد لایه‌های کمتری با قابلیت عمل بیشتری طراحی شود. به عنوان مثال محصول اولیه Windows NT با لایه‌های زیاد، کارایی کمتری نسبت به ویندوز 95 داشت. در NT 4.0 سعی شد لایه‌ها به همدیگر نزدیکتر و مجتمع‌تر شوند تا کارایی بیشتر گردد.

۲- نیاز به دقت بالا در پیمانه‌ای کردن و تقسیم‌بندی لایه‌ها دارد، مسأله اصلی در سیستم‌های لایه‌ای، تعریف مناسب هر لایه و سرویس‌های درون آن می‌باشد. از آنجا که هر لایه فقط می‌تواند از سرویس‌های لایه پایین‌تر استفاده کند، در طراحی لایه‌ها باید دقت بسیار به خرج داد.

۳- ساختار ماشین مجازی (Virtual Machine)

در این ساختار، یک سیستم عامل بر روی سخت‌افزار اجرا شده و برای لایه بالاتر چندین ماشین مجازی با تمام جزئیات سخت‌افزاری فراهم می‌کند که بر روی هر یک از این ماشین‌های مجازی می‌توان یک سیستم عامل جداگانه نصب و اجرا کرد.

این ماشین مجازی (که پایین‌ترین لایه را برای لایه‌های بالاتر فراهم می‌کند) فقط یک ماشین توسعه یافته (مثلاً با سیستم فایل و دیگر امکانات پیشرفته) نیست، بلکه کپی دقیقی از سخت‌افزار (شامل همه قسمت‌هایی که یک سخت‌افزار واقعی دارد) است.

توجه: جهت درک چگونگی پیاده‌سازی سیستم عامل‌های ماشین مجازی به شکل زیر دقت کنید:

فرآیندها	فرآیندها	فرآیندها
هسته سیستم عامل ۱	هسته سیستم عامل ۲	هسته سیستم عامل ۳
ماشین مجازی		
سخت‌افزار		

مدل ماشین مجازی

فرآیندها
هسته سیستم عامل
سخت افزار

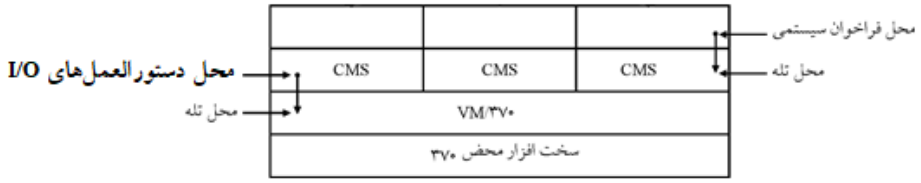
مدل ماشین غیر مجازی

یک سیستم اشتراک زمانی دو وظیفه عمده و مجزا دارد:

۱- مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع)

۲- مدیریت واسط کاربر (یک ماشین توسعه‌یافته با استفاده از یک واسط که بر روی سخت‌افزار خام قرار گیرد.)

اما ایده ساختار ماشین مجازی این است که دو وظیفه فوق از یکدیگر جدا شوند، سیستم عامل VM/370 بر روی سیستم‌های IBM بهترین مثال از مفهوم ماشین مجازی است. قلب سیستم که به نام مانیتور ماشین مجازی (VMM یا Virtual machin manitor) یا Hypervisor معروف است بر روی سخت‌افزار عریان اجرا شده و فقط وظیفه اول یعنی مدیریت منابع (چندبرنامگی و مالتی‌پلکس کردن پردازنده و سایر منابع) را انجام می‌دهد و اکنون می‌توان یک یا چند ماشین مجازی را بر روی لایه بالاتر از آن قرار داد، انگار که یک ماشین فیزیکی کامل، به چندین نسخه ماشین مجازی کامل تکثیر شده است. البته باید توجه داشته باشید که ماشین‌های مجازی برخلاف سیستم‌های عامل یک ماشین توسعه‌یافته نیستند بلکه هر کدام یک نسخه دقیق از سخت‌افزار خام هستند که شامل تمام خصوصیات سخت‌افزار مانند مد کاربر، مد هسته، I/O و وقفه می‌باشند. این ماشین مجازی انگار که همان وظیفه دوم یعنی مدیریت واسط کاربر است، اما فقط در حد یک سخت افزار خشک و خالی است. اکنون می‌توان بر روی هر کدام از این ماشین‌های مجازی یک سیستم عامل معمولی نصب نمود، این عمل همان عمل نصب سیستم عامل بر روی یک سخت‌افزار خام حقیقی است. شکل زیر ساختار VM/370 را نشان می‌دهد.



توجه: هر کاربر یک برنامه (CMS: Conversational sational monitor system) مخصوص به خود را دارد که یک سیستم عامل تک کاربره محاوره‌ای است.

توجه: سیستم عامل‌هایی که بر روی ماشین‌های مجازی اجرا می‌شوند، هیچ‌گونه ارتباطی با یکدیگر ندارند، در واقع هر کدام خود را در یک سیستم مجزا تصور می‌کنند بنابراین می‌توان بر روی هر ماشین مجازی یک نوع سیستم عامل مجزا نصب نمود.

همانطور که مشخص است هر ماشین مجازی باید از دیگران کاملاً مستقل باشد یا حداقل اینطور تصور کند. حال سوال این است که این استقلال چگونه بر روی یک سخت‌افزار مشترک پیاده‌سازی می‌شود؟ جواب تسهیم (Multiplexing) و Slooping می‌باشد.

در واقع کافی است منابع سیستم با توجه به خصوصیاتش به دو صورت زمانی یا فضایی بین ماشین‌های مجازی تسهیم و یا Spool شوند. به عنوان مثال:

- ایجاد پردازنده مجازی با تسهیم زمانی پردازنده حقیقی که این عمل به کمک PCBهای مجازی انجام می‌شود.
- ایجاد حافظه‌های مجازی با تسهیم فضای حافظه اصلی بین ماشین‌های مجازی.
- Spool کردن دستگاه‌های ورودی و خروجی.
- تشکیل دیسک مجازی با تسهیم فضای دیسک حقیقی.

همانطور که مشخص است دستیابی به سخت‌افزار و فراخوان‌های سیستمی نمی‌تواند بر روی CMS اجرا شود و در واقع CMS قدرت این کار را ندارد، پس برنامه اجرا شده در CMS چگونه فراخوانی سیستمی انجام می‌دهد؟ فرض کنید برنامه‌ای که روی CMS اجرا می‌شود قصد انجام عمل I/O دارد و یک فراخوانی سیستمی صادر می‌کند. این فراخوانی سیستمی باعث تغییر مد کاربر به مد هسته و می‌شود و سیستم عامل روی CMS (از آنجا که نمی‌داند CMS سخت‌افزار حقیقی نیست) سعی می‌کند که این عمل را روی CMS اجرا کند و در واقع یک تله مجازی در CMS رخ می‌دهد، اکنون CMS که در حقیقت یک نرم‌افزار است، این درخواست را به VM/370 ارجاع می‌دهد و اکنون یک تله واقعی در VM/370 رخ می‌دهد و درخواست (با توجه به تسهیم یا Spool) به اجرا رسیده و پاسخ به لایه‌های بالاتر ارجاع می‌شود. از آنجا که وظیفه چندبرنامگی و ارائه ماشین توسعه یافته کاملاً مجزا از یکدیگر هستند، هر یک از این تکه برنامه‌ها بسیار ساده‌تر شده و از انعطاف‌پذیری (Flexibility) بیشتر و قابلیت نگهداری (Maintainability) آسان‌تری برخوردار هستند.

توجه: در ساختار ماشین مجازی دو وظیفه اصلی چندبرنامگی و ایجاد واسط راحت (مستقل از سخت افزار) از یکدیگر مجزا شده‌اند. مانیتور ماشین مجازی وظیفه چندبرنامگی را بر عهده دارد و لایه بالای آن وظیفه ایجاد واسط کاربر با سخت افزار را بر عهده دارد. لذا هر یک از این بخش‌ها ساده‌تر شده و از قابلیت انعطاف بیشتری برخوردارند.

توجه: از آنجا که هر ماشین مجازی کاملا مشابه سخت افزار واقعی است، هر یک از آنها می‌توانند هر سیستم عاملی را مستقلا اجرا کنند لذا می‌توان همزمان سیستم‌عامل‌های مختلفی را روی این

ماشین اجرا کرد.

یکی دیگر از کاربردهای ماشین‌های مجازی به مجازی‌سازی محیط برنامه‌نویسی (Programming-environment Virtualization) موسوم است، در این روش VMMها سخت افزار واقعی را جهت اجرای سیستم‌عامل‌های متعدد نسخه‌برداری و مجازی‌سازی نمی‌کنند، بلکه هدف ایجاد قابلیت حمل (Portability) برنامه‌ها بر روی سخت‌افزار و سیستم‌عامل‌های مختلف است. کامپایلر زبان java که توسط شرکت Sun Microsystem طراحی شده است، یک خروجی بایت کد (byte code) تولید می‌کند. این بایت کدها دستوراتی هستند که بر روی ماشین مجازی جاوا (JVM) اجرا می‌شوند. جهت اجرای برنامه‌های java در یک ماشین، آن کامپیوتر می‌بایست دارای یک JVM باشد. بدین ترتیب برنامه‌هایی که به زبان java نوشته شده‌اند به راحتی بر روی انواع کامپیوترها اجرا می‌شوند. فقط کافی است بایت کدها را روی آن ماشین کامپایل کرد. بدیهی است به علت نیاز به کامپایل شدن بایت کدها، برنامه‌های جاوا سرعت کمتری نسبت به برنامه‌هایی نظیر C دارد. برنامه‌های C توسط کامپایلر بومی یک کامپیوتر، برای یکبار تبدیل به زبان ماشین آن کامپیوتر می‌گردد. پس خروجی زبان ماشین کامپایلر C از یک نوع کامپیوتر به کامپیوتر دیگر متفاوت است ولی بایت کدهای خروجی java برای همه ماشین‌ها یکسان است.

مزیت‌های سیستم‌های ماشین‌های مجازی:

۱- امکان اجرای چند سیستم عامل بر روی یک ماشین حقیقی (این سیستم عامل‌ها از یک زیرساخت مشترک پیروی می‌کنند).

۲- استقلال کامل ماشین‌های مجازی قرار گرفته روی یک ماشین حقیقی و امنیت بالای آن‌ها.

۳- با در اختیار داشتن چند ماشین مجازی بدون استفاده از سخت‌افزار اضافی و با هزینه پایین می‌توان نصب و تست سیستم عامل‌های مختلف و حتی شبکه‌های کامپیوتری انجام داد.

۳- در ماشین مجازی java قابلیت حمل (Protability) بالایی وجود دارد.

معایب سیستم‌های ماشین‌های مجازی:

۱- به دلیل ساخت نسخه‌های متعدد از سخت افزار پیچیدگی پیاده‌سازی آن بسیار زیاد است.

۲- به دلیل سربار حاصل از دوبار وقفه (وقفه مجازی و وقفه واقعی) کارایی آن پایین است.

۴- ساختار ریز هسته (Microkernel)

ایده‌ی اصلی در این ساختار، هر چه کوچک‌تر و ساده‌تر نمودن قسمت هسته (Kernel) سیستم عامل است. بنابراین به دلیل اینکه یک هسته بسیار کوچک خواهیم داشت به ساختار **ریز هسته** (Microkernel) **ساختار مشتری – سرویس‌دهنده** (Client - Server) نیز گفته می‌شود.

در سیستم عامل‌های مشتری – سرویس‌دهنده، قسمت اعظم کد سیستم عامل در حالت کاربر اجرا می‌شود و وظیفه هسته برقراری ارتباط میان فرایندهای سرویس‌دهنده و مشتری است. در این ساختار درخواست فرایند کاربر (که به آن فرایند مشتری گویند) به یک فرایند سرویس‌دهنده فرستاده شده و پاسخ آن به مشتری برگشت داده می‌شود.

توجه: امروزه سیستم عامل‌ها به جای داشتن یک هسته بزرگ به سمت داشتن هسته هرچه کوچکتر حرکت می‌کنند که به آن اصطلاحاً **ریز هسته** می‌گویند. در واقع فقط چند عمل اصلی مانند ارتباط بین فرآیندها و زمانبندی پایه‌ای را به هسته واگذار می‌کنند و دیگر خدمات سیستم عامل توسط تعدادی فرآیند سرویس‌دهنده صورت می‌گیرد. این همان ایده‌ی مدل مشتری – سرویس‌دهنده است که به آن **معماری ریز هسته** نیز می‌گویند.

توجه: روند طراحی سیستم عامل‌های جدید همواره با این این ایده همراه بوده است که تا جایی که ممکن است کدها به لایه‌های بالاتر منتقل شوند تا نهایتا یک هسته کمینه پدید آید. برای این منظور اکثر وظایف سیستم عامل را در سطح کاربر و مشابه فرآیندهای کاربر پیاده‌سازی می‌کنند. برای درخواست یک سرویس، مانند خواندن یک بلوک از فایل، فرآیند کاربر (که اکنون به عنوان فرآیند مشتری شناخته می‌شود) یک درخواست به فرآیند سرویس‌دهنده ارسال می‌نماید و از آن می‌خواهد که کارش را انجام دهد و پاسخ را برگرداند. کار هسته در این مدل برقراری ارتباط بین مشتری‌ها و سرویس‌دهنده از طریق تبادل پیام است. البته هسته، وظایف دیگری نیز دارد. به طور کلی وظایف هسته در این مدل به صورت زیر است:

۱-تبادل پیام مابین مشتری‌ها و سرویس‌دهنده‌ها

۲-زمان‌بندی پردازنده، فرآیندها و ایجاد چندبرنامگی

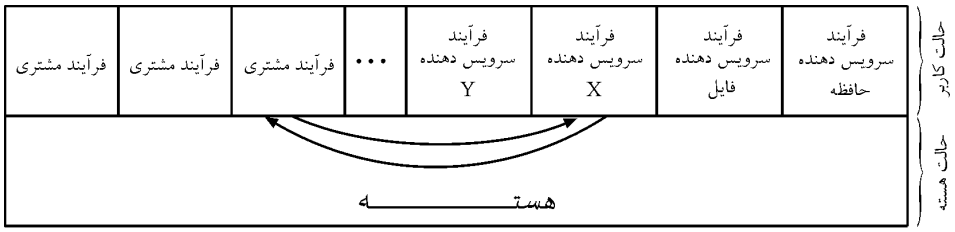
۳-بخش سطح پایین مدیریت حافظه مانند برنامه‌ریزی ثبات‌های سخت‌افزار مدیریت حافظه

۴- بخش سطح پایین مدیریت I/O که برنامه‌ریزی ثبات‌های ویژه کنترل کننده‌ها و کارهایی را بر عهده دارند که اگر در سطح کاربر انجام شود، آنگاه امنیت کل سیستم به مخاطره خواهد افتاد.

توجه: در این روش، سیستم عامل به چند بخش (خدمات فایل، خدمات فرآیند، خدمات ترمینال و خدمات حافظه) تقسیم شده است، که هریک به عنوان یک سرویس‌دهنده فقط یکی از وظایف سیستم عامل را انجام می‌دهد، بنابراین سیستم عامل را می‌توان ساده‌تر تحلیل، طراحی، پیاده‌سازی و نگهداری کرد.

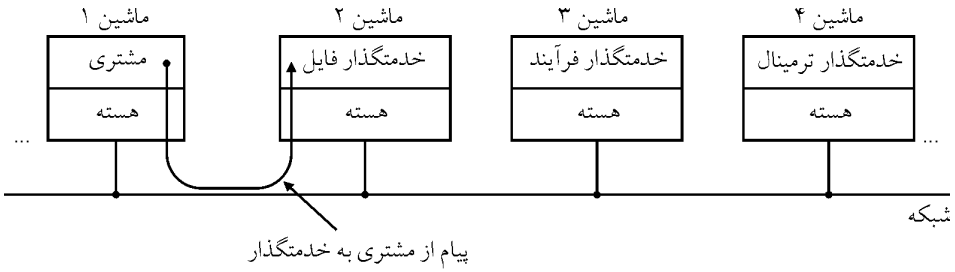
توجه: سرویس‌دهنده‌ها در مُد کاربر اجرا شده و ریز هسته با آنها همانند فرآیندهای کاربر رفتار می‌کند. بنابراین هیچ یک از سرویس‌دهنده‌ها دسترسی مستقیم به سخت‌افزار را ندارد. همچنین اگر کارکرد یکی از سرویس‌دهنده‌ها مختل شود، فقط همان بخش مختل می‌شود و این اختلال به دلیل استقلال عملیاتی هر سرویس‌دهنده به سایر سرویس‌دهنده‌ها و به تبع کل سیستم منتقل نمی‌شود.

توجه: نحوه‌ی پیاده‌سازی و عملکرد مدل مشتری – سرویس‌دهنده در شکل زیر نشان داده شده است:



ساختار مشتری – سرویس‌دهنده

در حالت دیگر می‌توان از مدل مشتری و سرویس‌دهنده در سیستم‌های توزیع شده استفاده نمود. به شکل زیر توجه کنید.



با توجه به شکل فوق فرض کنید، مشتری بر روی یک ماشین مجزا از سرویس‌دهنده اجرا شده است، در این حالت همانطور که در مدل مشتری و سرویس‌دهنده ماشین محلی، یک پیام درخواست از مشتری به سرویس‌دهنده از طریق هسته ارسال می‌شد و سرویس‌دهنده پاسخ می‌داد، در سیستم توزیع شده هم این پیام از مشتری به هسته ماشین مشتری و از آن به یک شبکه ارتباطی و از آن به هسته ماشین سرویس‌دهنده و سپس به سرویس‌دهنده انتقال می‌یابد و پاسخ هم همین مسیر را باز می‌گردد. نکته‌ای که در این روش وجود دارد این است که برای مشتری و یا سرویس‌دهنده بین پیامی که از ماشین محلی آن‌ها می‌آید با پیامی که از ماشین راه دور دیگر می‌آید، تفاوتی وجود ندارد (و در واقع برای آن‌ها قابل تشخیص نیست). به عبارت دیگر از آنجا که یک مشتری به وسیله ارسال پیام‌هایش با یک سرویس‌دهنده ارتباط برقرار می‌کند، مشتری نیاز ندارد که بداند آیا به پیغام وی به صورت محلی در ماشین خودش رسیدگی می‌شود و یا اینکه پیغام از طریق شبکه به یک ماشین دور ارسال می‌شود.

مزیت‌های سیستم‌های مشتری و سرویس‌دهنده:

مهمترین مزیت سیستم‌های مشتری و سرویس‌دهنده طراحی پیمانه‌ای عالی آن‌ها می‌باشد. از این رو که وظایف سیستم عامل به طور مفصل و منسجم از هم جدا شده و هر کدام در یک پیمانه

قرار گرفته‌اند، مزیت پیمانه‌ای بودن این روش باعث بروز خصوصیات زیر می‌شود:

۱- انعطاف‌پذیری (Flexibility) بالا

۲- قابلیت نگهداری (Maintainability) ، اشکال‌زدایی (Debug) و ترمیم (Recovery) بالا

۳- قابلیت اطمینان (Reliability) بالا

۴- مقیاس‌پذیری (قابلیت توسعه) بالا

۵- سهولت پیاده‌سازی

۶- مناسب بودن برای سیستم‌های شبکه‌ای و توزیعی

معایب سیستم‌های مشتری و سرویس‌دهنده:

در مقابل همین خاصیت پیمانه‌ای بودن و جدا کردن سرویس‌دهنده‌ها از مشتری‌ها باعث پایین آمدن کارایی (سرعت) این سیستم نسبت به سیستم‌های دیگر شده است. فرض کنید که مشتری اقدام به ارسال یک پیام درخواست می‌کند این پیام در حین ارسال توسط مشتری و دریافت توسط سرویس‌دهنده باعث وقوع 2 تله می‌شود و پاسخ آن نیز به همین صورت، بنابراین برای هر درخواست و پاسخ 4 تله رخ می‌دهد، 2 فراخوان سیستمی برای ارسال و دریافت «**پیام درخواست**» و همچنین 2 فراخوان سیستمی برای ارسال و دریافت «**پیام پاسخ**». همچنین در موازات آن فراخوانی‌های سیستمی برای مدیریت حافظه و I/O نیز رخ می‌دهد که کارایی را تا حد زیادی کاهش می‌دهد.

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۹۶- با توجه به جدول زیر، متوسط زمان پاسخ‌دهی (Response Time) و متوسط زمان انتظار (Waiting Time) پردازه‌ها برای الگوریتم Preemptive Shortest Remaining Job First چند واحد زمانی است؟ **(مهندسی *IT* – دولتی ۹۹)**

پردازه	زمان محاسبات	زمان ورود پردازه
P1	5	3
P2	8	1
P3	6	2

۱) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر صفر است.
۲) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر 1.6 است.
۳) متوسط زمان پاسخ‌دهی برابر 6 و متوسط زمان انتظار برابر 6 است.
۴) متوسط زمان پاسخ‌دهی برابر 6.6 و متوسط زمان انتظار برابر 6 است.

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۹۶- گزینه () 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 ممکن 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 اگر 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 پاسخ و میـانگین زمان گردش کار (بازگشت) فرآیندهاست.

توجه: در عمل نمی‌توان الگوریتم SJF را پیاده‌سازی کرد، زیرا سیستم عامل زمان اجرای فرآیندها را از قبل نمی‌داند و تنها کاری که می‌تواند انجام دهد این است که زمان اجرای فرآیندها را فقط 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.

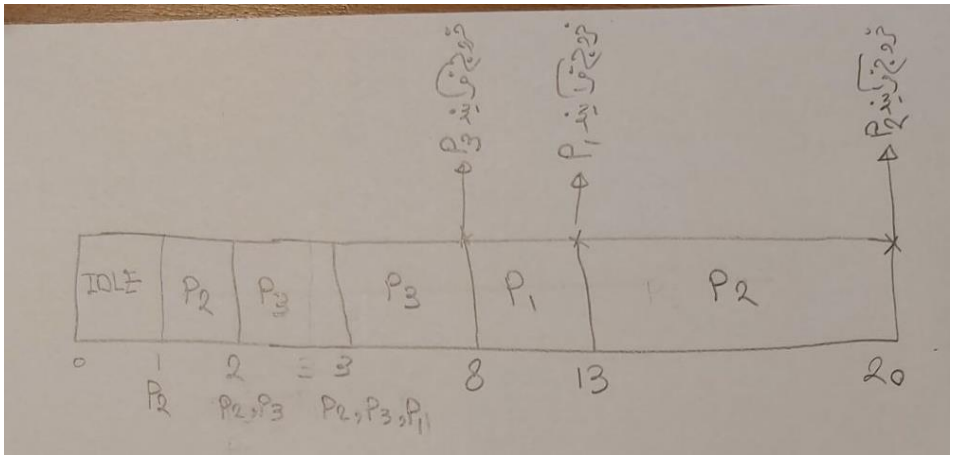
توجه: در الگوریتم **SRT** نیز همانند الگوریتم SJF، احتمال وقوع **قحطی‌زدگی برای کارهای بزرگ** وجود دارد.

توجه: مطابق فرض سوال الگوریتم «**Preemptive Shortest Remaining Job First**» در نظر گرفته شده است که همان SRT **غیرانحصاری** است.

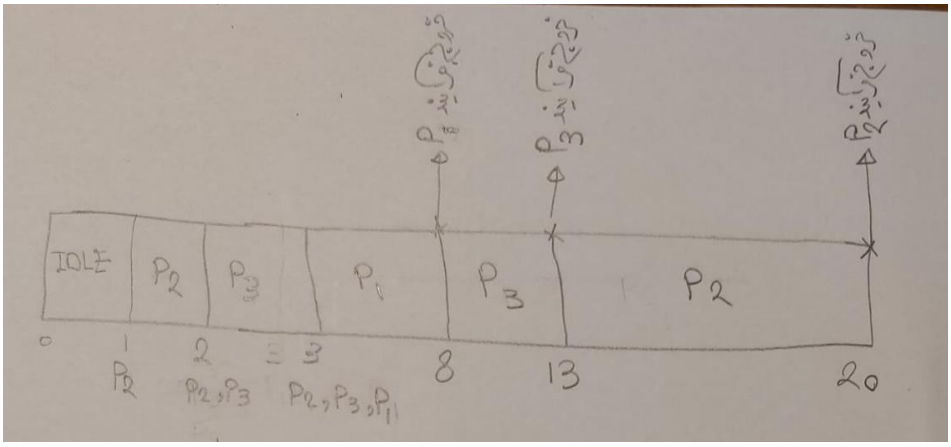
با توجه به مفروضات مطرح شده در صورت سؤال داریم:

فرآیند	زمان ورود	زمان اجرا	زمان انتظار+	زمان بازگشت =
P_1	3	5		
P_2	1	8		
P_3	2	6		

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



توجه: در لحظه 3 و لحظه ورود فرآیند P_1 ، زمان باقی‌مانده فرآیند P_1 و P_3 هر دو برابر مقدار 5 واحد زمانی است که به دلیل جلوگیری از تعویض متن و به تبع رعایت اصل بهینگی و بالابردن بهره‌وری CPU در لحظه 3 پردازنده به همان فرآیند P_3 اختصاص می‌یابد. که البته اگر اول فرآیند P_1 و سپس فرآیند P_3 هم اجرا شود بازهم پاسخ در گزینه‌ها وجود ندارد.



توجه: در ادامه، برای حل سوال نمودار گانت اول مورد استفاده قرار گرفته است.

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

P_1 زمان پاسخ $= 13 - 3 = 10$

P_2 زمان پاسخ $= 2 - 1 = 1$

P_3 زمان پاسخ $= 8 - 2 = 6$

میانگین زمان پاسخ = ART = $\frac{10+1+6}{3} = \frac{17}{3} = 5.66$

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

P_1 زمان بازگشت = $13-3=10$

P_2 زمان بازگشت = $20-1=19$

P_3 زمان بازگشت = $8-2=6$

میانگین زمان بازگشت = ATT = $\frac{10+19+6}{3} = \frac{35}{3} = 11.67$

زمان اجرای فرآیند – زمان بازگشت فرآیند = زمان انتظار فرآیند

P_1 زمان انتظار = $10-5=5$

P_2 زمان انتظار = $19-8=11$

P_3 زمان انتظار = $6-6=0$

میانگین زمان انتظار = AWT = $\frac{5+11+0}{3} = \frac{16}{3} = 5.34$

میانگین زمان اجرا = AST = $\frac{5+8+6}{3} = \frac{19}{3} = 6.33$

AVG Turnaround Time=AVG Service Time + AVG Waiting Time

میانگین زمان بازگشت = میانگین زمان اجرا + میانگین زمان انتظار

11.67= 6.33 + 5.34

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	3	5	5	10
P_2	1	8	11	19
P_3	2	6	0	6

میانگین زمان بازگشت = میانگین زمان انتظار + میانگین زمان اجرا

11.67 5.34 6.33

توجه: همانطور که پُر واضح است، پاسخ سوال در گزینه‌ها موجود نیست.
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 ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۷- در یک سیستم، 50 پردازه (Proccess) با کد زیر به صورت هم‌روند (Concurrent) در حال اجرا هستند. اگر مقدار اولیه سمافورها x=20 ، y=15 و z=1 باشند، حداکثر چند پردازه ممکن است پشت سمافور z در حالت انتظار بلوکه شوند؟ **(مهندسی *IT* – دولتی ۹۹)**

```
wait (x);
 wait (y);
   wait (z);
      a= a+1;
   signal (z);
 signal (y);
signal (x);
```

۱) 14
۲) 30
۳) 49
۴) 50

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل ششم: مدیریت فرآیندها و نخ‌های هم‌روند

۹۷- گزینه (۱) صحیح است.

تابع wait(s): عملیات آن به ترتیب شامل، کاهش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً خواباندن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Wait (semaphore)
   {
s.count = s.count – 1;
if (s.count< 0)
{
Add  this process to s.queue;
block ();
  }
}
```

شرح تابع: پس از فراخوانی تابع wait(s) توسط یک فرآیند، ابتدا یک واحد از شمارنده‌ی سمافور کاسته می‌شود (s.count = s.count - 1)، سپس اگر شرط مربوط به دستور if(s.count< 0) برقرار بود (مقدار شمارنده سمافور منفی بود) این فرآیند داخل صف سمافور قرار گرفته و توسط تابع block مسدود و به خواب می‌رود، یعنی از وضعیت اجرا به وضعیت منتظر منتقل می‌گردد، در

غیر اینصورت، فرآیند، وارد خط بعدی می‌شود.

تابع signal(s): عملیات آن به ترتیب شامل، افزایش مقدار شمارنده، تست کردن مقدار شمارنده و احیاناً بیدار کردن یک فرآیند است.

ساختار این تابع به صورت زیر است:

```
Signal (semaphore s)
{
s.count = s.count + 1
   if (s.count< = 0)
   {
    Remove a process from queue;
    Wake up ();
     }
}
```

شرح تابع: پس از فراخوانی تابع signal(s) توسط یک فرآیند، ابتدا یک واحد به مقدار شمارنده سمافور اضافه می‌شود (s.count = s.count + 1)، سپس اگر شرط مربوط به دستور if (s.count< = 0) برقرار بود (مقدار شمارنده سمافور مثبت نبود) به معنی وجود فرآیندهای علاقه‌مند ورود به خط بعدی است که در حال حاضر در صف سمافور قرار دارند، به شکل خروج به ترتیب ورود (FIFO) فقط یک فرآیند به ازای هر بار فراخوانی تابع signal(s) توسط تابع wake () up بیدار شده، یعنی تغییر وضعیت داده و از وضعیت منتظر به صف آماده منتقل می‌گردد. بنابراین این فرآیند پس از حضور در صف آماده‌ی پردازنده، این شانس را دارد تا توسط زمانبند کوتاه‌مدت، انتخاب شود و پردازنده را در اختیار بگیرد و در وضعیت اجرا قرار بگیرد.

به بیان دیگر هر فرآیند با اجرای تابع signal(s)، فرآیند سر صف سمافور را بیدار می‌کند و اگر صف سمافور خالی باشد و هیچ فرآیند خوابیده‌ای در آن سمافور وجود نداشته باشد در تابع signal فقط یک واحد به مقدار شمارنده سمافور اضافه می‌شود و تابع خاتمه می‌یابد.

توجه:

در صورت سوال مطرح شده است که سه سمافور با مقدار اولیه x=20، y=15 و z=1 در نظر گرفته شود. همچنین گفته شده است که قطعه کد زیر توسط 50 پردازه (process) اجرا می‌شود.

```
wait (x);
 wait (y);
   wait (z);
      a= a+1;
```

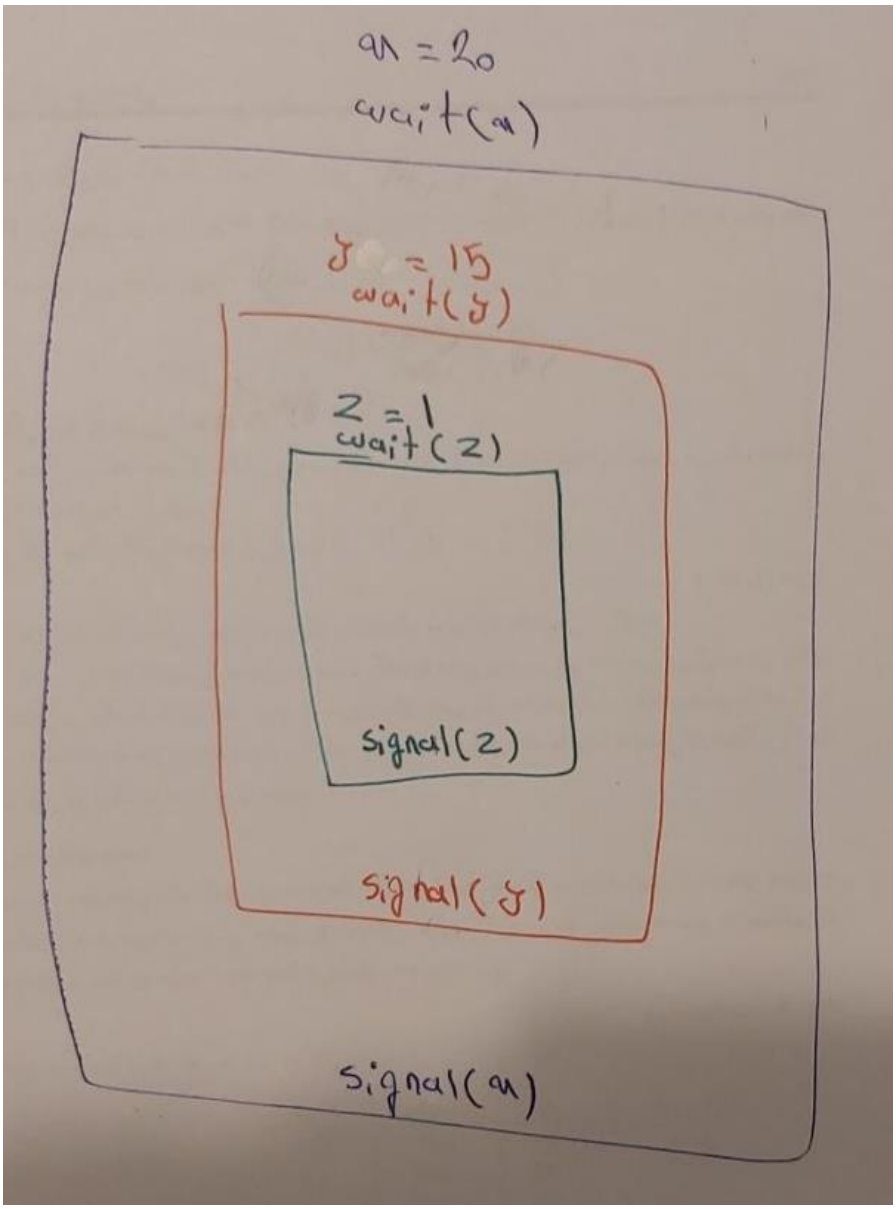
signal (z);

signal (y);

signal (x);

در ادامه خواسته سوال این است که، حداکثر طول صفی که برای سمافور z تشکیل می‌شود، چقدر است؟

شکل زیر را در نظر بگیرید:



دقت کنید که در صورت سوال مطرح شده است که قطعه کـد فـوق توسـط 50 پـردازه (process) اجرا می‌شود. پس می‌بایست قطعه کد فوق توسط 50 فرآیند اجرا گردد و از آن عبور کنند.
50 فرآیند P_1، P_2، P_3 ، ...، P_{47}، P_{48}، P_{49}، P_{50} را در نظـر بگیریـد. از ایـن تعـداد بـا توجـه بـه شمارنده سمافور x که برابر مقدار 20 است، ابتدا فرآیندهای P_1، P_2، P_3، ...، P_{19} ، P_{20} به ترتیـب اجرا می‌شوند و از سـمافور x عبـور می‌کننـد. و فرآینـدهای P_{21}، P_{22}، ... ، P_{47}، P_{48}، P_{49} ، P_{50} پشت سمافور x در صف سمافور x آرام می‌گیرند و می‌خوابند. در ادامه از بـین فرآینـدهای عبـور کرده از سمافور x یعنی فرآیندهای P_1 ، P_2 ، P_3 ، ...، P_{19}، P_{20} با توجه به شمارنده سمافور y که برابر مقدار 15 است، فقط فرآیندهای P_1، P_2، P_3 ،...، P_{14} ، P_{15} این شانس را پیدا می‌کننـد کـه از سمافورy عبور کنند و عبور می‌کنند، و فرآیندهای P_{16}، P_{17}، P_{18}، P_{19} ، P_{20} پشـت سـمافور y در صف سمافور y آرام می‌گیرند و می‌خوابند. پس درحال حاضر 5 فرآینـد در صـف سـمافور y بـه خواب رفته‌اند، بنابراین طول صفی که برای سمافور y تشکیل شده است، هم اکنون برابـر مقـدار5 است. در ادامه از بین فرآیندهای عبور کرده از سمافور y یعنـی فرآینـدهای P_1، P_2، ...، P_{14} ، P_{15} با توجه به شمارنده سمافور z که برابر مقدار 1 است، فقط فرآیند P_1 این شانس را پیدا می‌کند کـه از سمافورz عبور کند و عبور می‌کنـد، و فرآینـدهای P_2، P_3، ...، P_{14} ، P_{15} پشـت سـمافور z در صف سمافور z آرام می‌گیرند و می‌خوابند. پس درحال حاضر 14 فرآیند در صـف سـمافور z بـه خواب رفته‌اند، بنابراین طول صفی که برای سمافور z تشکیل شده است، هم اکنون برابر مقـدار14 است.

در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور z تابع signal(z) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_2 می‌گردد، در این لحظه فرآیند P_2 در خط بعد از تابع wait(z) و قبل از تـابع signal(z) قرار می‌گیرد، همچنین در ادامه فرآیند P_1 پس از عبور از ناحیه سمافور y تابع signal(y) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{16} می‌گردد، در این لحظه فرآینـد P_{16} در خـط بعد از تابع wait(y) و قبل از تابع wait(z) قرار می‌گیرد، همچنین در ادامه فرآیند P_1 پـس از عبـور از ناحیه سمافور x تابع signal(x) را اجرا می‌کند، این امر منجر به بیدار شدن فرآیند P_{21} می‌گردد، در این لحظه فرآیند P_{21} در خط بعد از تابع wait(x) و قبل از تابع wait(y) قرار می‌گیرد، این روند تا اتمام ملاقات هر سه ناحیه x، y و z برای همه فرآیندها از P_1 تا P_{50} ادامه پیدا می‌کند.

تست‌های فصل چهارم: مدیریت حافظه اصلی

۹۸- یک سامانه داری 64 صفحه مجازی (Virtual Pages) است که به 16 قاب فیزیکی (Physical Frames) بر اساس رابطه زیر نگاشت داده می‌شود. طول هر صفحه یک کیلو کلمه (1 K Words) است. اگر آدرس مجازی برابر 1010101000111101 باشد، کدام گزینه آدرس فیزیکی را نشان می‌دهد؟ **(مهندسی *IT* – دولتی ۹۹)**

۱) 10101010001111
۲) 10111000111101
۳) 10101000111101
۴) 11001000111101

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل چهارم: مدیریت فرآیندها و زمان‌بندی پردازنده

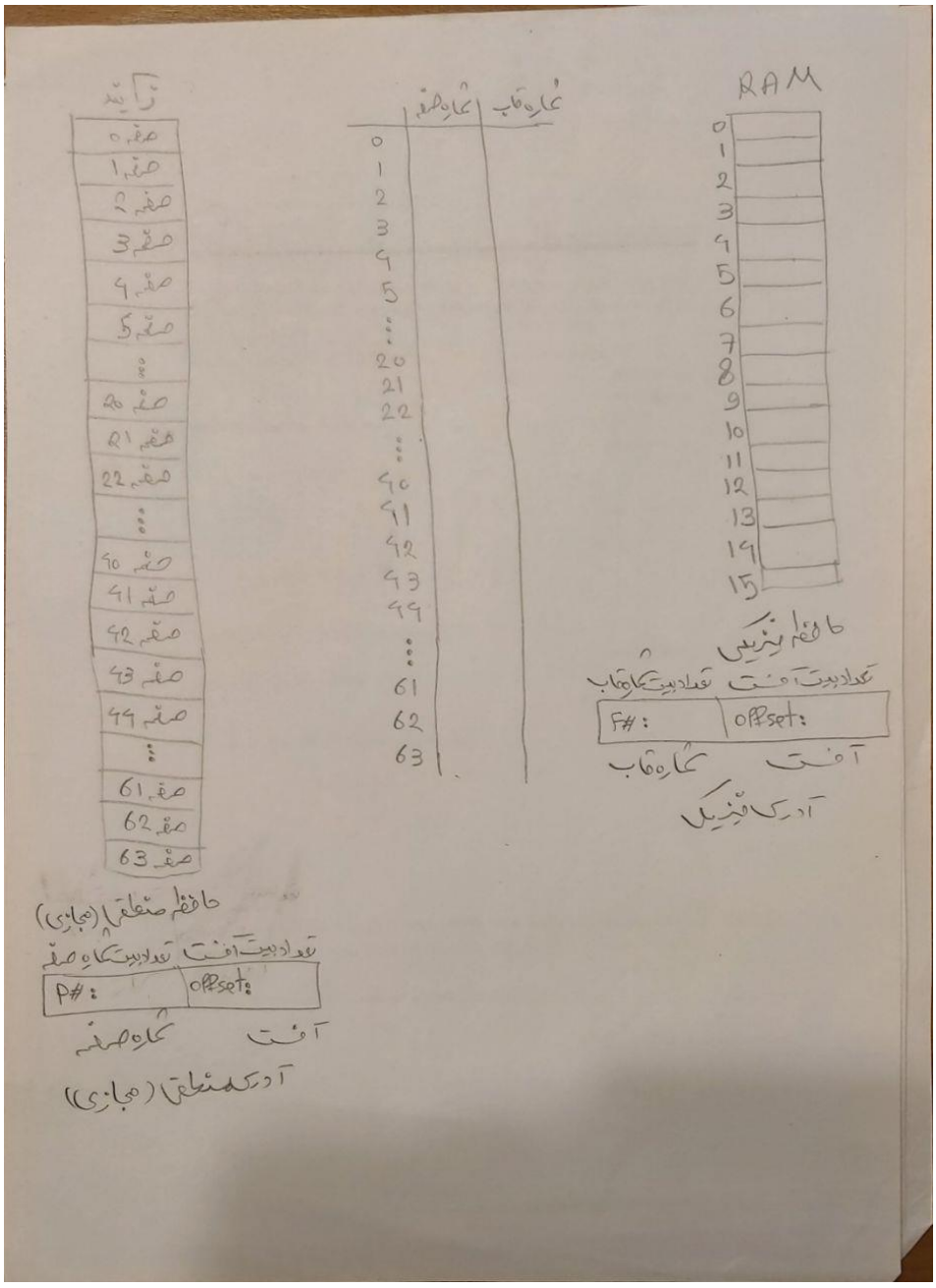
۹۸- گزینه () صحیح است.

در صورت سوال مطرح شده است که سامانه داری 64 صفحه مجازی (Virtual Pages) است که به 16 قاب فیزیکی (Physical Frames) بر اساس رابطه زیر نگاشت داده می‌شود. طول هر صفحه یک کیلو کلمه (1 K Words) است. اگر آدرس مجازی برابر 1010101000111101 باشد، کدام گزینه آدرس فیزیکی را نشان می‌دهد؟

توجه: همانطور که پُر واضح است، به دلیل بیان نشدن رابطه نگاشت، سوال غیرقابل حل است.

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

به طور کلی روابط میان آدرس منطقی و آدرس فیزیکی در راه حل صفحه‌بندی به صورت زیر است:


RAM
حافظه فیزیکی
تعداد بیت آفست
تعداد بیت شماره قاب
F#:
offset:
آفست
شماره قاب
آدرس فیزیکی
شماره قاب
شماره صفحه
حافظه منطقی (مجازی)
تعداد بیت آفست
تعداد بیت شماره صفحه
P#:
offset:
آفست
شماره صفحه
آدرس منطقی (مجازی)

$$\text{تعداد صفحات فرآیند (تعداد درایه‌های جدول صفحه)} = \frac{\text{اندازه فرآیند}}{\text{اندازه صفحه یا اندازه قاب}} = 64$$

$$\text{تعداد قاب‌های حافظه فیزیکی} = \frac{\text{اندازه حافظه فیزیکی}}{\text{اندازه صفحه یا اندازه قاب}} = 16$$

$$\text{تعداد بیت شماره صفحه} = b = \log_2^{\text{تعداد صفحات فرآیند}} = \log_2^{64} = 6bit$$

$$\text{تعداد بیت شماره قاب} = b = \log_2^{\text{تعداد قاب‌های حافظه فیزیکی}} = \log_2^{16} = 4bit$$

توجه: در صورت سوال ذکر نشده است که یک کلمه یک بایت در نظر گرفته شود یا چهار بایت، اما بر اساس طول 16 بیتی آدرس مجازی و طول 6 بیتی تعداد بیت شماره قاب، یک کلمه معادل یک بایت در نظر گرفته می‌شود تا طول تعداد بیت آفست همان 10 بیت باشد.

$$\text{تعداد بیت آفست} = b = \log_2^{\text{اندازه صفحه یا اندازه قاب}} = \log_2^{1KB} = \log_2^{2^{10}} = \log_2^{2^{10}} = 10bit$$

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

طول آدرس منطقی = تعداد بیت شماره صفحه + تعداد بیت آفست $= 6bit + 10bit = 16bit$

طول آدرس فیزیکی = تعداد بیت شماره قاب + تعداد بیت آفست $= 4bit + 10bit = 14bit$

همچنین داریم:

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

$$\text{اندازه حافظه منطقی (فرآیند)} = 2^{16} = 2^6 \times 2^{10} = 2^{16} = 64KB$$

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

$$\text{اندازه حافظه فیزیکی (RAM)} = 2^{14} = 2^4 \times 2^{10} = 2^{14} = 16KB$$

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

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 4 بیت مربوط به تعداد بیت‌های شماره قاب می‌باشد.

پس: عرض جدول صفحه فوق برابر $4\text{bit} + 0\text{bit} = 4\,\text{bit}$ 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:

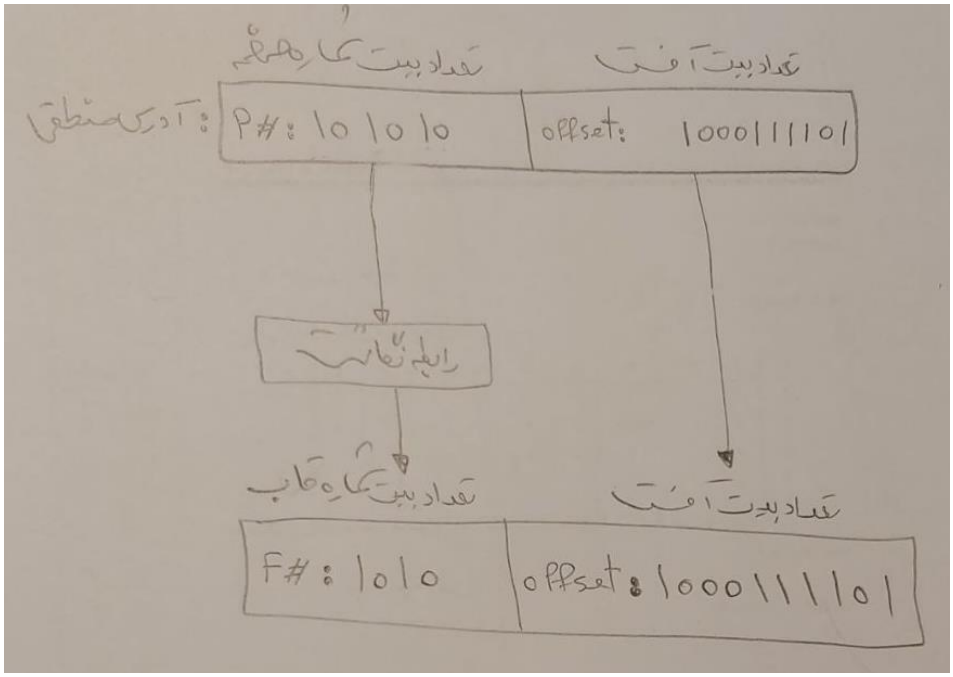
$$\text{اندازه جدول صفحه} = 64 \times 4\,\text{bit} = 256\,\text{bit} = 32\,\text{Byte}$$

توجه: در صورت سوال مطرح شده، برای تبدیل شماره صفحه به شماره قاب یک الگوی ثابت و از قبل تعیین شده بر اساس یک رابطه نگاشت، مشخص گردیده است که خود این فرض نادرست می‌باشد. تعیین شماره قاب در حافظه فیزیکی برای یک صفحه از یک فرآیند براساس قاب‌های آزاد در حافظه فیزیکی و توسط سیستم عامل انجام می‌گردد و نه براساس یک الگوی از قبل مشخص شده و نگاشت ثابت، زیرا یک صفحه از فرآیند همواره به یک قاب خاص از حافظه فیزیکی نگاشت نمی‌شود.

از آنجا که فرآیند موجود در این سیستم شامل 64 صفحه است، بنابراین 6 بیت سمت چپ $(b = \log_2^{64} = 6\,\text{bit})$ از آدرس منطقی 16 بیتی مربوط به شماره صفحه و 10 بیت سمت راست باقیمانده مربوط به آفست است. همچنین از آنجا که حافظه فیزیکی موجود در این سیستم نیز شامل 16 قاب است، بنابراین 4 بیت سمت چپ $(b = \log_2^{16} = 4\,\text{bit})$ از آدرس فیزیکی نیز مربوط به تعداد بیت‌های شماره قاب می‌باشد. با توجه به اینکه در ترجمه آدرس منطقی به فیزیکی در سیستم‌های صفحه‌بندی شده، تعداد بیت‌های آفست ثابت مانده و فقط شماره صفحه با شماره قاب جای‌گزین می‌گردد، بنابراین آدرس فیزیکی نیز دارای طولی برابر 14 بیت خواهد بود.

توجه: باتوجه به فرض مطرح شده در صورت سؤال، باید رابطه نگاشت مطرح می‌شد که نشده‌است، بنابراین همانطور که گفتیم به دلیل بیان نشدن رابطه نگاشت، سوال غیرقابل حل است.

توجه: اما با توجه به کلید اولیه سازمان سنجش یعنی گزینه سوم می‌توان حدس زد که رابطه نگاشت دوبار شیفت به راست بوده است که در صورت سوال فراموش شده است مطرح شود، پس برای این کار، کافی است مقدار دودویی شماره صفحه را دوبار به سمت راست شیفت دهیم تا مقدار شماره قاب بدست بیاید و سپس 10 بیت سمت راست آدرس منطقی را به عنوان آفست به آن بیافزاییم:



که مقدار حاصل مطابق کلید اولیه سازمان سنجش آموزش کشور یعنی گزینه سوم است. هرچند که گفتیم این نوع نگاشت ثابت نادرست است. چرا که همیشه مجموع صفحاتی که در 2 بیت سمت راست شماره صفحه و به شکل 00، 01، 10، 11 متفاوت ولی در 4 بیت باقی‌مانده یکسان هستند به دلیل انتقال 2 بیت به سمت راست، همگی یک شماره قاب واحد خواهند داشت. یعنی به هر قاب موجود در حافظه فیزیکی، 4 صفحه از آدرس منطقی نسبت داده می‌شود. مثلاً صفحات شماره 0، 1، 2 و3 همگی به شماره قاب 0 نگاشت می‌شوند و نمی‌توانند همزمان با هم درون حافظه فیزیکی قرار داده شوند.

000000 : شماره صفحه 0
000001 : شماره صفحه 1
000010 : شماره صفحه 2
000011 : شماره صفحه 3

نگاشت (2 بیت شیفت به راست) ⟶ شماره قاب صفر: 0000

تست‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۹۹- کدام گزینه معیار (Criterion) یک زمان‌بند پردازنده نیست؟ **(مهندسی *IT* - دولتی ۹۹)**

۱) زمان پاسخ
۲) بهره‌وری پردازنده
۳) گذردهی (Throughput)
۴) زمان Burst (Burst time)

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل دوم: مدیریت فرآیندها و زمان‌بندی پردازنده

۹۹- گزینه (۴) صحیح است.

وظیفه زمانبند فرآیند، انتخاب یک فرآیند جهت در اختیار گرفتن پردازنده است (در واقع مشخص می‌کند که در لحظه حاضر، پردازنده در اختیار کدام فرآیند می‌باشد).

الگوریتم‌های زمانبندی اهداف زیر را دنبال می‌کند (در واقع موارد زیر معیارهایی جهت مقایسه عملکرد الگوریتم‌های مختلف می‌باشند):

عدالت (Fairness)

منظور از عدالت این است که فرآیندهای هم‌ارز از دید زمانبند، یکسان تلقی شوند. به عبارت دیگر یک فرآیند نباید از فرآیندهای هم‌سطح خود سهم بیشتری ازCPU دریافت کند.

البته این امر درباره فرآیندهایی که اولویت یکسان ندارند صادق نیست و منطقی است که فرآیندی با اولویت بالاتر، سهم بیشتری از CPU دریافت کند.

زمان پاسخ (Response)

یک الگوریتم زمانبندی باید زمان پاسخ فرآیندها را به حداقل برساند. منظور از زمان پاسخ، مدت زمان بین لحظه ورود کار و لحظه دریافت نتیجه و خروجی اول می‌باشد. این معیار در سیستم‌های محاوره‌ای و تعاملی اهمیت ویژه‌ای دارد.

زمان گردش (برگشت یا بازگشت) (Turnaround Time)

یک الگوریتم زمانبندی باید زمان گردش کار فرآیندها را به حداقل برساند. منظور از زمان گردش کار، مدت زمان بین لحظه ورود کار و لحظه خروج کامل از سیستم می‌باشد.

توجه: به تفاوت بین زمان پاسخ و زمان بازگشت دقت کنید:

زمان پاسخ، مدت زمان بین صدور فرمان و تولید اولین نتیجه و خروجی آن می‌باشد اما زمان بازگشت، مدت زمان بین ورود کار و تکمیل نهایی و نتیجه و خروج کامل آن است.

زمان انتظار (Waiting Time)

یک الگوریتم زمانبندی باید زمان انتظار فرآیندها را به حداقل برساند. منظور از زمان انتظار، مجموع زمان‌هایی است که یک فرآیند در صف فرآیندهای آماده و انتظار، منتظر دریافت CPU

می‌باشد.

واضح است که یک الگوریتم زمانبندی، بر روی مدت زمان اجرای یک فرآیند و مدت زمان ورودی و خروجی تأثیری ندارد، بلکه فقط بر روی مدت زمان انتظار یک فرآیند در صف آماده مؤثر است.

بهره‌وری پردازنده (CPU Utilization or CPU Efficiency)

یک الگوریتم زمانبندی باید بهره‌وری پردازنده را به حداکثر برساند، به این معنی که حتی الامکان در تمام زمان‌ها پردازنده مشغول باشد تا زمان‌های هدر رفته پردازنده به حداقل برسد. که به آن درصد استفاده مفید از CPU نیز گفته می‌شود. زمان‌های بیکاری و زمان‌های تعویض متن، زمان‌های غیرمفید محسوب می‌شوند. برای محاسبه آن، زمان مفید را بر زمان کل یعی حاصل جمع زمان مفید و زمان غیرمفید، تقسیم می‌کنیم و در 100 ضرب می‌کنیم.

توان گذردهی (توان عملیاتی) (Throughput)

یک الگوریتم زمانبندی باید توان گذردهی را به حداکثر برساند. منظور از گذردهی، تعداد فرآیندهایی است که در واحد زمان تکمیل می‌شوند. به عبارت دیگر توان گذردهی تعداد کارهای انجام شده، تکمیل شده و خارج شده در واحد زمان است. که برای محاسبه آن تعداد کل کارها بر زمان اجرای کل کارها تقسیم می‌شود.

مهلت (Deadline)

مدت زمانی که از لحظه ورود یک فرآیند بی‌درنگ فرصت وجود دارد تا پردازش آن تمام شود. ملاک حفظ و گارانتی کردن مهلت زمانی است.

مطلوبیت (Proportionality)

مطلوبیت به معنی برآورده‌سازی نیازهای غیروظیفه‌مندی کاربران نهایی است، یعنی اگر به کمک سیستم عامل نیازهای فرآیندها مثل پردازنده و حافظه برآورده می‌شود و به تبع آن نیازهای کاربران نهایی برآورده می‌شود، کاربران نهایی دوست دارند که کار فرآیندها به کمک سیستم عامل در زمان کوتاه هم انجام شوند و این نیاز غیروظیفه مندی کاربران نهایی است. کاربران نهایی از سیستم عامل توقع دارند هم وظیفه فرآیندها به درستی انجام شود و هم غیروظیفه‌مندی (سرعت انجام) فرآیندها مطلوب باشد.

توجه: علاوه بر معیارهای ذکر شده، معیارهای زیر نیز کم و بیش برای مقایسه الگوریتم‌های زمانبندی به‌کار می‌روند:

- **تعادل یا توازن (Balance) در استفاده از منابع:** در واقع الگوریتم‌های زمانبندی باید باعث شود از تمام منابع سیستم به خوبی استفاده و بهره‌برداری شود. به عبارت دیگر منابع را مشغول نگه

دارد. به بیان دیگر مشغول نگه داشتن همه بخش‌های سیستم و چندبرنامگی کارهای محدود به CPU و محدود به I/O در کنار یکدیگر و ایجاد توازن بار در استفاده از همه منابع.

– **پیش بینی پذیری (Predicatability)**: الگوریتم زمانبندی باید به گونه‌ای عمل کند که در اجراهای مختلف و چندباره یک کار، به‌خصوص زمان انتظار و زمان پاسخ تغییرات شدیدی نداشته باشند. این معیار برای کاربران اهمیت دارد. مانند پرهیز از تنزل کیفیت در سیستم‌های چند رسانه‌ای.

– رعایت اصول و اولویت‌ها

– رعایت ضرب العجل‌ها

نکته: برخی از معیارها و اهداف فوق با هم محقق نمی‌شوند و ممکن است با یکدیگر در تضاد باشند. به عنوان مثال ممکن است جهت برقراری عدالت، سیاستی اعمال شود که بهره‌وری CPU را کاهش دهد.

توجه: الگوریتم‌های زمانبندی به دو دسته تقسیم می‌شوند:

۱- Non Preemptive (غیر قابل پس‌گیری، انحصاری)

۲- Preemptive (قابل پس‌گیری، غیر انحصاری)

در زمانبندی انحصاری، هنگامی که پردازنده در اختیار فرآیندی قرار گیرد نمی‌توان پردازنده را از وی پس گرفت، مگر این‌که خود داوطلبانه آن را آزاد کند و یا فرآیند خاتمه یابد اما در زمانبندی غیر انحصاری می‌توان پردازنده را از فرآیندی پس گرفت و به دیگری تحویل داد.

توجه: در سیستم‌های تعاملی، الگوریتم‌های غیر انحصاری (قابل پس‌گیری) تنها گزینه قابل قبول هستند، زیرا در غیر این‌صورت یک فرآیند CPU را به انحصار خود درآورده و به دیگران اجازه کار نمی‌دهد. شاید به‌نظر برسد برای سیستم‌های بی‌درنگ تنها باید از الگوریتم‌های غیر انحصاری (قابل پس‌گیری) استفاده کرد، اما در این سیستم‌ها از هر دو نوع الگوریتم می‌توان بهره برد. از آن جا که فرآیندها در سیستم‌های بی‌درنگ اغلب بسیار کوچک هستند و به سرعت اجرا شده و خاتمه می‌یابند (یا مسدود می‌شوند)، الگوریتم‌های انحصاری (غیر قابل پس‌گیری) نیز در این سیستم‌ها کاربرد دارند.

توجه: در متون فارسی به الگوریتم **انحصاری** (Non Preemptive)، الگوریتم «**غیرقبضه‌ای**» یا «**غیرقابل پس گرفتن**» یا «**غیرقابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: در متون فارسی به الگوریتم **غیرانحصاری** (Preemptive)، الگوریتم «**قبضه‌ای**» یا «**قابل پس گرفتن**» یا «**قابل تخلیه پیش هنگام**» نیز گفته می‌شود.

توجه: زمان Burst (Burst time) مربوط به زمان اجرای یک فرآیند است و به معیارهای الگوریتم‌های زمان‌بندی فرآیند مرتبط نیست، بنابراین پُرواضح است که گزینه چهارم پاسخ سوال است.

تست‌های فصل هشتم: مدیریت دیسک

۱۰۰- برای خواندن از دیسک، در کدام لایه نرم‌افزاری محاسبات مربوط به شیار(Track)، قطاع (Sector) و هد دیسک صورت می‌پذیرد؟ **(مهندسی IT – دولتی ۹۹)**

۱) لایه Device Driver
۲) لایه روتین سرویس‌دهی به وقفه
۳) لایه مدیریت دستگاه‌های سیستم عامل
۴) لایه نخ سطح هسته که برنامه سطح کاربر را اجرا می‌کند.

عنوان کتاب: سیستم عامل
مولف: ارسطو خلیلی‌فر
ناشر: انتشارات راهیان ارشد
آدرس سایت گروه بابان: khalilifar.ir

پاسخ‌های فصل هشتم: مدیریت دیسک

۱۰۰- گزینه (۲) صحیح است.

توجه: سازمان سنجش آموزش کشور، در کلید اولیه و نهایی خود، گزینه دوم را بـه عنـوان پاسـخ اعلام کرده بود. اما بهتر بود در کلید نهایی گزینه اول به عنوان پاسخ نهایی اعلام می‌شد.

توجه: اصولا کار لایه‌های نرم افزار ورودی و خروجی سطح کاربر(درخواست ورودی و خروجی)، نرم افزار ورودی و خروجی مستقل از دستگاه (جستجوی ورودی و خروجی)، گرداننده دستگاه (Device Driver) (جستجوی ورودی و خروجی) برای جستجوی ورودی و خروجی است و کار لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) برای تحویل ورودی و خروجی مورد نظر به فرآیند مرتبط است. بنابراین برای خواندن از دیسک، محاسبات مربوط به شیار(Track)، قطاع (Sector) و هد دیسک در لایه‌های قبل از لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) یعنی لایه گرداننده دستگاه (Device Driver) و گزینه اول انجام می‌شود. مگر اینکه نظر طراح این باشد که لایه روتین سرویس دهنده وقفه (اداره‌کننده وقفه) دو بخشی است و بخشی از آن که مربوط به محاسبات است قبل از وقفه تحویل ورودی و خروجی و در لایه گرداننده دستگاه (Device Driver) انجام می‌شود که البته فرض نادرستی است.

توجه: یکی از مهم‌ترین وظایف سیستم عامل کنترل و مدیریت دستگاه‌های I/O در کامپیوتر است. سیستم عامل باید فرامینی را برای دستگاه‌ها صادر نماید، وقفه‌های ایجاد شده به وسیله دسـتگاه‌ها را دریافت نماید، خطاها را پردازش نمایـد و بـرای سـهولت اسـتفاده از سیسـتم رابط‌هـایی بـین دستگاه‌ها و بقیه سیستم ایجاد کند.

توجه: هر دستگاه I/O از دو بخش مولفه **الکترونیکی** و مولفه **مکانیکی** تشکیل شده اسـت. مولفـه الکترونیکی همـان **کنترل‌کننده دستگاه** (Device Controller) و مولفـه مکـانیکی همـان **دسـتگاه** است. و علت ذکر تفاوت بین کنترل‌کننده و دستگاه این است که سیستم عامل با کنترل‌کننـده کـار می‌کند و نه خود دستگاه. هر کنترل‌کننده دارای تعداد محدودی رجیستر (جهت دریافـت فرمـان و یه به منظور تعیین وضعیت دستگاه) است که برای ارتباط برقرار کردن با پردازنده به کار می‌رونـد. سیستم عامل با نوشتن در این رجیسترها قادر است، دستوراتی مبتنی بر تحویل و یا دریافـت داده، خاموش و روشن کردن دستگاه را صادر نماید و همچنین با خوانـدن از آن می‌توانـد، از وضـعیت

دستگاه باخبر گردد. علاوه بر رجیسترهای کنترلی، بسیاری از دستگاه‌ها دارای بافر داده‌ای نیز هستند که سیستم عامل می‌تواند از آنها بخواند یا در آنها بنویسد. به عنوان مثال می‌توان به بافر داده Video RAM اشاره نمود که سیستم عامل و یا برنامه‌های کاربردی جهت نمایش پیکسل روی صفحه مانیتور، در آن می‌نویسد. برای مثالی دیگر هد دیسک تحت کنترل نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک قرار دارد، و به خودی خود هد دیسک فهم و شعور ندارد، یعنی بدون نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک، هد دیسک نمی‌داند کجا برود و کجا نرود، چه کند و چه نکند. بلکه نرم‌افزار زمان‌بندی دیسک به واسطه استفاده از یک الگوریتم خاص، توسط کنترل‌کننده هد دیسک، هد دیسک را کنترل و جابه‌جا می‌کند، در واقع این نرم‌افزار زمان‌بندی دیسک و کنترل‌کننده هد دیسک است که به هد دیسک فهم و شعور می‌دهد و به هد می‌گوید که کجا برود و کجا نرود، چه کند و چه نکند. بنابراین در زمان حرکت مکانیکی هد به سمت یک سیلندر، هد شخصاً تصمیم جدیدی نمی‌گیرد، بلکه تابع تصمیمات نرم‌افزار زمان‌بندی و کنترل‌کننده خود است

توجه: سیستم عامل با نوشتن فرامین در ثبات‌های خاص کنترل‌کننده، عملیات ورودی و خروجی مورد نظر را انجام می‌دهد. هنگامی که فرمانی پذیرفته شد، پردازنده، کنترل‌کننده دستگاه را رها می‌کند و به سایر کارها می‌پردازد. بعد از کامل شدن فرمان، کنترل‌کننده، وقفه‌ای ایجاد می‌کند که به سیستم عامل امکان می‌دهد، کنترل پردازنده را به دست آورد و نتایج عمل را بررسی نماید، پردازنده نتایج دلخواه و وضعیت دستگاه را، با خواندن چند بایت از ثبات‌های کنترل‌کننده به دست می‌آورد.

توجه: نرم‌افزار I/O ، به صورت مجموعه‌ای از لایه‌هاست که رابط مناسبی را برای کاربر فراهم نماید. نرم‌افزار I/O باید به گونه‌ای طراحی شود که «**استقلال از دستگاه**» را به همراه داشته باشد. این امکان باید وجود داشته باشد که برنامه‌ها به گونه‌ای نوشته شوند که بتوانند با انواع دستگاه‌ها کار کنند، بدون آنکه با تعویض نوع دستگاه نیازی به اصلاح آنها باشد.

توجه: نرم‌افزار I/O کنترل و پردازش خطا یعنی کشف و تصحیح خطا را نیز انجام می‌دهد. البته خطاها باید تا حد امکان نزدیک به سخت افزار پردازش شود. اگر کنترلر یک خطای خواندن را کشف کند، باید در صورتی که می‌تواند، خود به تصحیح آن اقدام کند، در غیر این صورت نرم‌افزار گرداننده آن خطا را پردازش می‌نماید و احتمالا این کار را با خواندن مجدد آن بلوک انجام می‌دهد. بسیاری از خطاها گذرا هستند و در صورت تکرار عمل برطرف خواهند شد. تنها در صورتی که لایه‌های زیرین نرم‌افزاری قادر به رفع مشکل نباشند، لایه‌های بالایی از وجود مشکل باخبر خواهند شد و خطا را تصحیح خواهند کرد.

توجه: نرم‌افزار I/O یک ساختار چهار لایه‌ای دارد به طوری که هر لایه وظیفه خاصی را بر عهده دارد، به صورت زیر:

فرآیندهای کاربران
نرم افزار ورودی و خروجی سطح کاربر (درخواست ورودی و خروجی)
نرم افزار ورودی و خروجی مستقل از دستگاه (جستجوی ورودی و خروجی)
گرداننده دستگاه (Device Driver) (جستجوی ورودی و خروجی)
روتین سرویس دهنده وقفه (اداره‌کننده وقفه) (تحویل ورودی و خروجی)
سخت افزار

توجه: هنگامی که یک برنامه کاربردی سعی می‌کند که یک فایل را بخواند، به ترتیب، عملیات ذیل طبق لایه‌های سیستم I/O در شکل فوق انجام می‌شود:

۱- فراخوان سیستمی (System Call) در سطح کاربر به واسطه دستور ورودی و خروجی انجام می‌شود.

۲-بخشی از سیستم عامل جهت اجرای فراخوان سیستمی صدا زده می‌شود. (اجرای ISR ورود به I/O)

۳-نرم‌افزار مستقل از دستگاه، بلوک مورد درخواست را در حافظه پنهان جستجو می‌کند. در صورتی که بلوک مورد نظر در آن موجود نباشد، مرحله بعدی اجرا می‌شود. در غیر اینصورت رکورد یافت شده به برنامه کاربر تحویل داده می‌شود.

۴-گرداننده دستگاه فراخوانی می‌شود تا تقاضای کاربر به سخت‌افزار و دیسک صادر گردد.

۵- گرداننده دستگاه و برنامه کاربر تا اجرای کامل I/O مسدود می‌گردد.

۶-زمانی که عملیات I/O توسط کنترل‌کننده دستگاه به پایان می‌رسد، سخت‌افزار مربوطه وقفه‌ای را تولید می‌کند.

۷-روتین سرویس‌دهنده وقفه (اداره کننده وقفه یا اجرای ISR خروج از I/O) اجرا می‌شود تا دریابد چه اتفاقی رخ داده است؟

۸-وضعیت دستگاه وقفه دهنده را بررسی کرده، گرداننده دستگاه، نرم افزار مستقل از دستگاه و فرآیند مسدود شده را بیدار می‌کند. سپس رکورد خوانده شده، به برنامه کاربردی تحویل داده می‌شود. واضح است که اجرای روتین سرویس دهی به وقفه بعد از اتمام عملیات ورودی خروجی است. بدین ترتیب فرآیند ورودی و خروجی به پایان می‌رسد و برنامه کاربردی کار خود را ادامه می‌دهد.

توجه: در ادامه جزئیات بیشتری در مورد لایه‌های نرم افزار I/O **با رویکـرد بـالا بـه پـایین** بیـان می‌کنیم:

نرم افزار ورودی و خروجی سطح کاربر

در این لایه جهت «**درخواست ورودی و خروجی**» مورد نظر فرآیند، توسط فرآینـد جـاری، لایـه نرم افزار ورودی و خروجی سطح کاربر فراخوانی می‌شود.

یک برنامه کامپیوتری (فرآیند کاربر) که در سطح کاربر اجرا می‌شود، علاوه بر دستورات پردازشی و محاسباتی، شامل دستورات ورودی و خروجی نیز می‌باشد، مانند دستورات خوانـدن (Read) و نوشتن (Write) در پاسکال و یا دستورات خوانـدن (scanf) و نوشـتن (printf) در زبـان C و یـا دستورات خواندن (cin) و نوشتن (cout) در زبان C++ که بدنه این توابـع داخـل کتابخانـه زبـان برنامه‌نوسی تعریف شده است. که هنگام کامپایل، بدنه تعریف ایـن توابـع ورودی و خروجـی بـه فرآیند کاربر متصل می‌گردد و در نهایت بدنه دستورات پردازشی و بدنه تعریـف توابـع ورودی و خروجی در یک برنامه کاربردی یکپارچه می‌گردد تا آماده اجرا گردد. هنگـام اجـرای بدنـه توابـع ورودی و خروجی برای مثال دستور خوانـدن (Read) جهـت انجـام کـار از دل ایـن تـابع عمـل فراخوان سیستمی (System Call) صورت می‌گیرد. هنگامی‌که رویه فراخوان سیسـتمی در سـطح کاربر فراخوانی می‌شود، به واسطه این فراخوانی و وقفه ورود به I/O بخشی از سیستم عامل جهت اجرای فراخوان سیستمی صدا زده می‌شود. (اجرای ISR ورود به I/O)

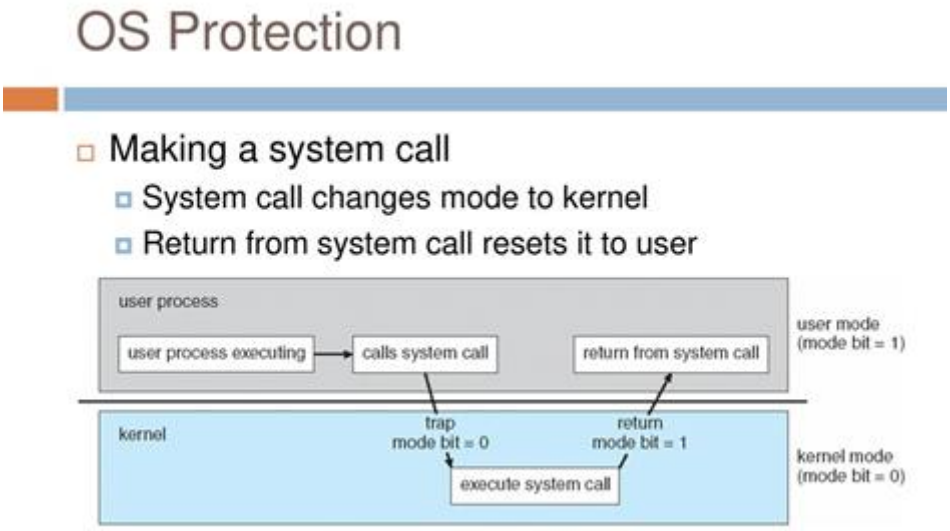
فراخوان سیستمی، یک روش برنامه‌نویسی است که در آن یک برنامه کاربردی از هسته سیستم‌عاملی که روی آن اجرا می‌گردد، یک خدمت (سرویس) درخواست می‌کند. این موضوع می‌تواند شامل سرویس‌های مرتبط با سخت‌افزار (برای مثال، دسترسی به دیسک سخت) باشد. فراخوان‌های سیستمی یک واسط اساسی بین یک فرآیند کاربر و سیستم‌عامل فراهم می‌کند.

هرگاه یک برنامه سطح کاربر نیاز به دسترسی به منابع سیستم و سخت افزار را داشته باشد، یکی از توابع درون سیستم عامل را فراخوانی می‌کند. که به این عمل فراخوان سیستمی (System Call یا Syscall) می‌گویند فراخوان سیستمی را گاه فراخوان هسته‌ای (kernel call) نیز می‌نامند چراکه برای انجام فراخوان سیستمی پردازنده باید در حالت مد هسته (kernel mode) باشد. به زبانی دیگر هیچ برنامه‌ای حق دسترسی مستقیم به سخت‌افزار را ندارد و باید توسط واسطه‌ای که سیستم‌عامل در اختیارش قرار می‌دهد و نامش فراخوان سیستمی است به سخت‌افزار سیستم دسترسی پیدا کند. در حقیقت فراخوان سیستمی پس از مدیریت منابع (Resource Management)، دومین هدف اصلی یک سیستم عامل می‌باشد.

سیستم عامل استفاده از کامپیوتر را ساده می‌سازد. این بدان معناست که برای مثال کاربر یا برنامه نویس بدون درگیر شدن با مسائل سخت افزاری دیسک ها، به راحتی فایلی را بر روی دیسک ذخیره و حذف کند. این کار در واقع با به کار بردن دستورهای ساده‌ای که فراخوان‌های

سیستمی را صدا می‌زنند انجام می‌پذیرد. در صورت عدم وجود سیستم عامل، کاربر یا برنامه‌نویس می‌بایست آشنایی کاملی با سخت افزارهای مختلف کامپیوتر داشته باشد و روتین‌هایی برای خواندن یا نوشتن آن‌ها به زبان‌های سطح پائین بنویسد. از این جنبه، گاه از سیستم عامل با عنوان ماشین توسعه یافته (Extended machine) یا ماشین مجازی (Virtual machine) یاد می‌شود که واقعیت سخت افزار را از دید کاربران مخفی می‌سازد.

توجه: اگرچه بخش عظیمی از لایه‌های نرم افزار ورودی و خروجی در داخل سیستم عامل است، اما همانطور که گفتیم بخش اندکی از آن شامل کد برنامه و کتابخانه‌های ورودی و خروجی که توسط کامپایلر به برنامه‌های کاربران پیوند می‌خورد در ناحیه خارج از هسته اجرا می‌گردد. و همانطور که گفتیم فراخوان‌های سیستمی ورودی و خروجی از بدنه تعریف توابع کتابخانه‌ای فراخوانی می‌شوند. که مجموعه تمام بدنه تعریف توابع کتابخانه‌ای متناظر با عملیات ورودی و خروجی در لایه نرم افزار ورودی و خروجی سطح کاربر قرار دارد. شکل زیر گویای مطلب است:



بخش مهم دیگری که در لایه نرم افزار ورودی و خروجی سطح کاربر موجود می‌باشد «سیستم Spooling» را تشکیل می‌دهد. Spooling روشی است که به وسیله آن دستگاه‌های ورودی و خروجی انحصاری در یک سیستم چندبرنامگی کاربرد پیدا می‌کند. به عنوان مثال یک دستگاه چاپگر را در نظر بگیرید. در صورتی که فرآیندی یک فایل متنی را برای چاپ باز کند و سپس ساعت‌ها کاری انجام ندهد، هیچ فرآیند دیگری قادر نخواهد بود چیزی را چاپ نماید. در عوض می‌توان از یک فرآیند خاصی به نام سرویس‌دهنده (Deamon) چاپگر و یک فهرست خاص به نام

فهرست Spooling استفاده نمود. برای چاپ کردن یک فایل، فرآیند ابتدا تمام فـایلی را کـه بایـد چاپ شود، ایجاد می‌نماید و آنرا در فهرست Sppling قرار می‌دهد. این برعهـده برنامـه Deamon است که فایل‌های این فهرست را چاپ نماید و این فرآیند تنها فرآیندی است که اجازه استفاده از فایل خاص چاپگر را دارد.

مفهوم Spooling تنها در چاپگر کاربرد ندارد، بلکه در شرایط دیگری نیز به کار گرفته می‌شود. بـه عنوان مثال جهت انتقال فایل بـر روی شـبکه، اغلـب از یـک برنامـه سـرویس‌دهنده (Deamon) شبکه‌ای استفاده می‌شود. کاربر برای ارسال یک فایل به مکانی، آن فایل را در فهرسـت Spooling شبکه قرار می‌دهد و مدتی بعد برنامه Deamon شبکه آن فایل را دریافـت می‌کنـد و مبـادرت بـه ارسال آن می‌نماید.

توجه: در ادامه جهت «**جستجوی ورودی و خروجی**» مورد نظر فرآیند، توسط system call ، لایه نرم افزار ورودی و خروجی مستقل از دستگاه فراخوانی می‌شود.

نرم افزار ورودی و خروجی مستقل از دستگاه

اگرچه قسمتی از نرم افزار ورودی و خروجی وابسته به دستگاه است، ولی بخش قابل تـوجهی از آن به صورت مستقل از دستگاه می‌باشد. مرز بین نرم افزار گرداننده دستگاه و نرم افزار مسـتقل از دستگاه به سیستم و دستگاه وابستگی دارد. زیرا کارهای چندی که می‌توانند به صـورت مسـتقل از دستگاه انجام شود، برای کارایی بیشتر و یا به دلایل دیگر در گرداننده دستگاه انجام می‌شود.

به طور کلی اعمال زیر در لایه نرم افزار مستقل از دستگاه قابل اجرا می‌باشند:

۱-رابط یک شکل و یکنواخت برای گردانندههای دستگاه

وظیفه اصلی نـرم افـزار ورودی و خروجـی مسـتقل از دسـتگاه مهیـا کـردن توابـع ورودی و خروجی مشترک برای تمام دستگاه‌ها و ایجاد یک رابط یک شکل برای استفاده در نرم افزار سطح کاربردی است. در واقع نویسندگان گردانندهها خود را با پارامترهـای واسـط یکسـان و اسـتاندارد تطابق می‌دهند. برای مثال یک برنامه که یک فایل را بـه عنـوان ورودی می‌خوانـد، بایـد قـادر بـه خواندن فایل از روی یک دیسک نرم یا سخت باشد بدون اینکه نیازی به تغییر برنامه برای هر نوع دستگاه متفاوت وجود داشته باشد. پلی‌مورفیسم (که در زبان یونانی به معنی «many forms» است) کیفیتی است که این امکان را به وجود می‌آورد تا یک نام، برای یـک، دو، یـا چنـدین تکنیـک بـا منظورهای متفاوت ولی در یک راستا و در ارتباط با یک مفهوم، مـورد اسـتفاده قـرار گیـرد. پلـی مورفیسم (سر بارگذاری تابع) آن گونه که در برنامه‌نویسی شیءگرا به کار گرفته شده اسـت، ایـن امکان را فراهم می‌کند تا بتوان برای مشخص کردن یک تابع عمومی حاوی عملیات مختلف، تنها از یک نام استفاده کرد. در این تابع عمومی نحوه انتخاب هر یک از عملیات مورد نظـر صـرفاً بـه وسیله نوع داده‌ای که مورد استفاده قرار می‌گیرد، تعیین می‌شود.

برای مثال در زبان C که در آن پلی‌مورفیسم مورد تاکید و اهمیـت قـرار نگرفتـه اسـت، بـرای

محاسبه قدرمطلق یک مقدار، سه تابع abs()، labs() و fabs() وجـود دارد. ایـن توابـع بـه ترتیـب مقدار قدر مطلق مقادیر عدد صحیح، عدد صحیح از نوع long و ممیز شناور را بـر می‌گردانـد. در حالی که در C++ به لحاظ تاکید روی پلی‌مورفیسم هر یک از توابع فـوق می‌تواننـد بـا نـام abs() مورد استفاده قرارگیرند، نوع داده‌ای که به عنوان آرگومان تابع abs() مورد اسـتفاده قـرار می‌گیـرد، گونه خاصی از تابع را انتخاب می‌کند. به بیـان کلی‌تـر مفهـوم پلـی مورفیسـم همـان ایـده « **one interface, multiple methods**» است. بـدین ترتیـب امکـان طراحـی یـک واسـط کـاربر کلـی (generic interface) برای گروهی از فعالیت‌های مرتبط فراهم می‌شود، در حالی که انتخاب نحوه اجرای هریک از فعالیت‌ها صرفاً بستگی به نوع داده مورد استفاده خواهد داشت.

یکی از مزیت‌های پلی‌مورفیسم، کمک در کاهش پیچیدگی برنامه است، ایـن عمـل بـا اسـتفاده از بکارگیری یک واسط کاربر یکسان برای مجموعه‌ای از عملیات تحت یک نام عمومی فراهم شـده است.

۲-نام‌گذاری یکسان

چگونگی نام‌گذاری دستگاه‌های ورودی و خروجی، جنبه دیگری از داشتن رابط تقریبا یکسـان را در دستگاه‌ها نشان می‌دهد. نگاشت اسامی سمبولیک دستگاه به گرداننده مناسب آن، در نـرم افـزار مستقل از دستگاه صورت می‌گیرد. در واقع نام‌های سیستمی دستگاه‌ها مورد استفاده قرار می‌گیرد و نه نامی که توسط کاربران مشخص شده است. برای مثال در سیستم عامل unix نام دستگاهی مانند «/dev/disk0» به طور منحصر به فرد شامل شماره دستگاه اصلی (Major device number) اسـت که جهت یافتن گرداننده مناسب آن دستگاه بکار می‌رود و همچنین حاوی شـماره دسـتگاه فرعـی (Minor device number) می‌باشد که به عنوان پارامتری به گرداننده داده می‌شود تا دستگاه فرعی مورد نظر جهت اجرای عملیات خواندن و نوشتن بر روی آن تعیین گردد. تمـام دسـتگاه‌ها دارای چنین شماره اصلی و فرعی می‌باشند و تمام گردانندەهای دستگاه‌ها از طریق شماره اصـلی مـورد دستیابی قرار می‌گیرند.

۳-حفاظت

آنچه در ارتباط با نام‌گذاری مطرح است، «**مسئله حفاظت**» می‌باشد. چگونه سیستم، کاربرانی را که حق دستیابی به دستگاه را ندارند منع نمایـد؟ در سیسـتم عامل‌هـای windows و unix شـیوه‌های امنیتی متفاوتی حاکم است. فایل‌های مخصوص (Special files) متناظر بـا دسـتگاه‌های ورودی و خروجی در unix با استفاده از بیت‌های حفاظتی (برای مثال بیت‌های rwx در unix) مورد حفاظت قرار می‌گیرند و تمام قوانین حفاظتی فایل‌ها برای دستگاه‌های ورودی و خروجی نیز به کار گرفته شده و قابل اعمال می‌باشد.

۴-بافر کردن

بافربندی موضوع دیگری است که به دلایل متفاوت هم در دستگاه‌های بلوکی و هم در دستگاه‌های کاراکتری مورد بحث قرار می‌گیرد. فرض کنیـد فرآینـدی داده‌ای را از مـودم می‌خواهـد بخوانـد،

فرآیند، فراخوان سیستمی Read را درخواست نماید و در انتظار یک کاراکتر ورودی مسدود شود و سپس با ورود هر کاراکتر یک وقفه رخ دهد و روتین سرویس‌دهنده وقفه کاراکتر را تحویل فرآیند کاربر داده و فرآیند را از حالت مسدود خارج نماید. بعد از اینکه کاراکتر در جایی ذخیره شد، فرآیند کاراکتر بعدی را می‌خواند و مجددا تا دریافت کاراکتر بعدی مسدود می‌شود. اشکالی که در این روش دیده می‌شود این است که فرآیند برای هر کاراکتر ورودی راه اندازی می‌گردد. در نتیجه تعداد دفعات اجرا بسیار زیاد اما مدت اجرا کوتاه مدت و ناکارآمد می‌باشد. اصلاح این روش می‌تواند به این صورت باشد که فرآیند، بافری n کاراکتری در فضای کاربر تهیه نماید و سپس n کاراکتر را بخواند. روتین سرویس‌دهنده وقفه کاراکترهای ورودی را یک به یک در بافر قرار می‌دهد تا کاملا پُر شود و سپس فرآیند کاربر را بیدار می‌کند.

۵-گزارش خطا

بسیاری از خطاها وابسته به دستگاه هستند و باید توسط گرداننده‌های دستگاه کنترل، کشف، تصحیح و پردازش گردند، اما گزارش خطا در بخش مستقل از دستگاه قرار دارد. یک خطای متعارف زمانی رخ می‌دهد که بلوک دیسک خراب شده باشد و نتواند خوانده شود. پس از اینکه گرداننده دستگاه به تعداد مشخصی، سعی در خواندن آن بلوک کند و در این کار موفق نباشد، نرم افزار مستقل از دستگاه را از وجود این خطا آگاه می‌سازد. چگونگی اداره خطا از این مرحله به بعد مستقل از دستگاه خواهد بود. به عبارت دیگر، تنها اگر لایه‌های پایینی قادر به حل مشکل به صورت شفاف نباشند، لایه‌های بالایی باید درگیر اداره خطا شوند.

۶-تخصیص و آزادسازی دستگاه‌های اختصاصی

دستگاه‌های انحصاری (اختصاصی یا غیراشتراکی) مانند نوارمغناطیسی در هر لحظه، فقط می‌توانند مورد استفاده یک فرآیند قرار گیرند و تا انتهای کار هر فرآیند، فقط به همان فرآیند اختصاص می‌یابند. و اگر منبع انحصاری به یک فرآیند تعلق گیرد دیگر نمی‌توان به زور منبع را از فرآیند پس گرفت و باید خودش داوطلبانه منبع را رها سازد. اگر فرآیندی منبعی انحصاری را درخواست کند که از قبل در اختیار فرآیند دیگری قرار گرفته است، آنگاه سیستم عامل فرآیند درخواست‌کننده منبع را در انتهای صف انتظار منبع قرار می‌دهد. پس از رها شدن منبع مورد درخواست توسط فرآیند جاری، فرآیندهای خوابیده در صف انتظار منبع، به ترتیب و یک به یک از ابتدای صف، منبع مورد درخواست را در اختیار می‌گیرند و کار خود را ادامه می‌دهند. تخصیص دستگاه‌های اختصاصی بر عهده نرم افزار مستقل از دستگاه در سیستم عامل است.

۷-اداره صف‌ها و زمان‌بندی

اگر در هنگام درخواست یک منبع (دستگاه) اختصاصی، آن منبع در اختیار فرآیند دیگری باشد، درخواست جدید باید در صف انتظار منبع مورد درخواست قرار داده شود. مدیریت صف‌های منابع اختصاصی بر عهده نرم افزار مستقل از دستگاه در سیستم عامل است.

۸-تعیین اندازه بلوک‌های داده مستقل از دستگاه

اندازه بلوک‌ها در دیسک‌های متفاوت ممکن است مختلف باشد و این امر بر عهده نرم افزار مستقل از دستگاه است که با ایجاد «**یک اندازه بلوک یکسان**» برای لایه‌های بالاتر این مشکل را حل نماید. به عنوان مثال ممکن است، چندین بلوک فیزیکی به عنوان یک بلوک منطقی تعیین گردد. بدین ترتیب لایه‌های بالاتر فقط از دستگاه‌های انتزاعی که همگی دارای اندازه بلوک منطقی یکسانی هستند، استفاده می‌نمایند. اندازه بلوک منطقی مستقل از اندازه بلوک فیزیکی می‌باشد. به عبارت دیگر اندازه بلوک‌ها در دیسک‌های مختلف یکسان نیست. پنهان کردن این موضوع و ایجاد بلوک‌های هم اندازه برای لایه‌های بالاتر، بر عهده نرم افزار مستقل از دستگاه است.

توجه: در ادامه جهت «**جستجوی بیشتر ورودی و خروجی**» مورد نظر فرآیند، توسط لایه نرم افزار ورودی و خروجی مستقل از دستگاه ، لایه گرداننده دستگاه (Device Driver) فراخوانی می‌شود.

گرداننده دستگاه (Device Driver)

توجه: در حالت کلی، کار گرداننده دستگاه، پذیرش درخواست‌های لایه نرم افزار ورودی و خروجی مستقل از دستگاه و مراقبت از اجرا شدن درخواست‌ها است. برای مثال یکی از درخواست‌های متداول برای یک گرداننده دستگاه دیسک، خواندن بلوک‌های دیسک است.

توجه: همانطور که گفتیم هر کنترل‌کننده دستگاه از چندین رجیستر (جهت دریافت فرمان و یا به منظور تعیین وضعیت دستگاه) برخوردار است. تعداد این رجیسترها و طبیعتا فرمان‌ها از دستگاهی به دستگاه دیگر متفاوت است. برای مثال **گرداننده** ماوس اطلاعاتی را که با حرکت دادن ماوس از چپ به راست و یا بالعکس و یا همچنین با فشار دادن دکمه‌ها در ماوس صورت می‌گیرد، دریافت می‌کند. در حالی که گرداننده دیسک باید از قطاع‌ها، شیارها، سیلندرها و حرکت بازوی دیسک اطلاع داشته باشد.

توجه: همه دستگاه‌های I/O متصل به یک کامپیوتر نیازمند یک کد خاص یعنی **گرداننده دستگاه** (Device Driver) برای کنترل آن دستگاه هستند. این کد توسط سازندگان دستگاه و برای انواع سیستم عامل‌ها نوشته می‌شود. به تفاوت گرداننده دستگاه و کنترل کننده دستگاه دقت نمایید. گرداننده دستگاه به کنترل کننده دستگاه فرمان می‌دهد.

توجه: گردانندەهای دستگاه‌ها به دلیل دسترسی مستقیم به سخت افزار دستگاه (رجیسترهای کنترل کننده دستگاه) مجبورند که بخشی از هسته سیستم عامل را تشکیل دهند. البته امکان این وجود دارد که گردانندەها در سطح کاربر به کمک یک سری فراخوان‌های سیستمی جهت نوشتن و خواندن از رجیسترها اجرا شوند. در حقیقت این طراحی (طراحی میکروکرنل) می‌تواند ایده خوبی باشد، زیرا هسته از گردانندەها جدا می‌گردد و چنانچه خطایی در گرداننده ایجاد گردد هسته را متاثر نمی‌نماید و باعث فروپاشی (Crash) سیستم نخواهد شد.

توجه: به طور کلی سیستم عامل‌ها گرداننده‌های دستگاه‌های I/O را به دو گروه دستگاه‌های **بلوکی** و دستگاه‌های **کاراکتری** تقسیم می‌کنند. دستگاه‌های بلوکی آدرس‌پذیر هستند مانند انواع دیسک‌های ذخیره‌سازی و دستگاه‌های کاراکتری آدرس‌پذیر نیستند و قابلیت هیچگونه عمل جستجویی را فراهم نمی‌کنند مانند کیبورد، ماوس، چاپگرهای خطی و سایر دستگاه‌هایی که عملکردشان شبیه دیسک‌ها نمی‌باشد.

توجه: گرداننده دستگاه دارای وظایف متعددی است اما مهمترین وظیفه آن، پذیرش تقاضای خواندن و نوشتن انتزاعی از کد نرم افزار مستقل از دستگاه در لایه بالایی خود و نیز نظارت و کنترل اجرای آنها می‌باشد. علاوه بر این، وظایف دیگری را مثل راه اندازی دستگاه‌ها و مدیریت و ثبت حوادث (Log event) را نیز به عهده می‌گیرد. در صورتی که گرداننده دستگاه در لحظه رسیدن تقاضا بیکار باشد، بلافاصله به تقاضا پاسخ می‌دهد در غیر اینصورت تقاضای جدید را به صف تقاضاهای منتظر اضافه می‌نماید تا هرچه زودتر به آنها رسیدگی شود. روش کار جهت پاسخ‌دهی به یک تقاضا بدین صورت است که ابتدا صحت پارامترهای ورودی توسط گرداننده بررسی می‌شود. اگر پارامترها صحیح نباشند، خطا برگردانده می‌شود. چنانچه پارامترها صحیح باشند، اولین گام جهت اجرای تقاضای ورودی و خروجی «**تبدیل تقاضا از فرم انتزاعی به فرم واقعی**» آن می‌باشد، به عبارت دیگر محل بلوک مورد تقاضا (شماره سیلندر و قطاع) بر روی دیسک تعیین می‌شود.

سپس گرداننده، وضعیت دستگاه را بررسی می‌کند. اگر مشغول باشد، تقاضا در صف قرار می‌گیرد، در غیر اینصورت، چک می‌شود که آیا وضعیت سخت افزاری به گونه‌ای است که بتوان تقاضا را هم اکنون پردازش کرد؟ ممکن است قبل از شروع انتقال داده، لازم باشد دستگاه را روشن کند و موتور را به راه اندازد. زمانی که دستگاه روشن می‌شود و آماده کار می‌گردد، کنترل واقعی می‌تواند شروع شود. منظور از کنترل، صدور دنباله‌ای از فرمان‌ها به دستگاه است. برنامه گرداننده دستگاه، باید معین کند چه عملیاتی توسط کنترل کننده و به چه ترتیبی باید انجام پذیرد. بعد از مشخص شدن این که چه فرامینی باید به کنترل کننده صادر گردد، برنامه گرداننده دستگاه با نوشتن فرامین در رجیسترهای دستگاه کنترل کننده، شروع به ارسال فرامین خواهد کرد.

توجه: بعضی از کنترل کننده‌ها، در هر لحظه یک فرمان را از گرداننده دستگاه دریافت کرده و اجرا می‌نمایند و زمانی که آماده اجرای فرمان بعدی باشند، گرداننده فرمان بعدی را به آنها تحویل می‌دهد و این کار تا آخرین فرمان یک به یک صورت می‌گیرد. برخی دیگر از کنترل کننده‌ها قادرند فهرستی از فرامین را بپذیرند و پس از آن بدون نیاز به گرداننده به ترتیب دستورات را اجرا نمایند. بعد از اینکه فرمان‌ها صادر شدند یکی از دو حالت ذیل پیش می‌آید:

۱-در بسیاری از مواقع، گرداننده دستگاه باید در انتظار بماند تا کنترل کننده، کارهای مورد درخواست را انجام دهد، بنابراین خود را مسدود می‌نماید و زمانی که وقفه‌ای فرا رسید، از حالت

مسدود خارج می‌شود.
۲-در سایر حالات بدون تاخیر سریعا کار انجام می‌شود و نیازی به مسدود شدن گرداننده نیست. مانند پیمایش صفحه در برخی از پایانه‌ها، که فقط با نوشتن تعدادی بایت در رجیسترهای کنترل کننده انجام پذیر است و حرکت مکانیکی در این راستا صورت نمی‌پذیرد. بنابراین، کل کار در چند میکروثانیه انجام می‌شود.
توجه: در هر دو حالت پس از پایان عملیات، باید خطاهای احتمالی بررسی شوند. اگر همه چیز صحیح باشد، گرداننده می‌تواند داده را به نرم افزار مستقل از دستگاه ارسال نماید. اگر تقاضاهای دیگر در صف انتظار دستگاه وجود داشته باشد در آن لحظه، یکی از آنها انتخاب می‌شود و به تقاضای آن رسیدگی می‌گردد. در صورتی که صف انتظار خالی باشد، گرداننده دستگاه به حالت مسدود در می‌آید و منتظر تقاضای بعدی می‌ماند.
توجه: در ادامه جهت «**تحویل ورودی و خروجی**» مورد نظر فرآیند، توسط لایه گرداننده دستگاه (Device Driver) ، لایه روتین سرویس‌دهنده وقفه فراخوانی می‌شود.

روتین سرویس‌دهنده وقفه (اداره‌کننده وقفه)

پس از تکمیل شدن عملیات ورودی و خروجی، از سوی کنترل‌کننده دستگاه مورد نظر وقفه خروج از I/O رخ می‌دهد، که منجر به فعال شدن روتین سرویس‌دهنده وقفه (اجرای ISR خروج از I/O) می‌شود، این روتین کارش بیدارکردن لایه‌های I/O قبل به صورت یک به یک و رو به بالا و در نهایت «**تحویل**» ورودی و خروجی مورد نظر به فرآیند کاربر است. روند و جزئیات خوابیدن و بیدارشدن یک به یک کل لایه‌های I/O از بالا به پایین و از پایین به بالا و انجام وقفه از سوی برنامه‌های کاربردی پنهان است و فقط نتایج به برنامه کاربردی «**تحویل**» داده می‌شود.