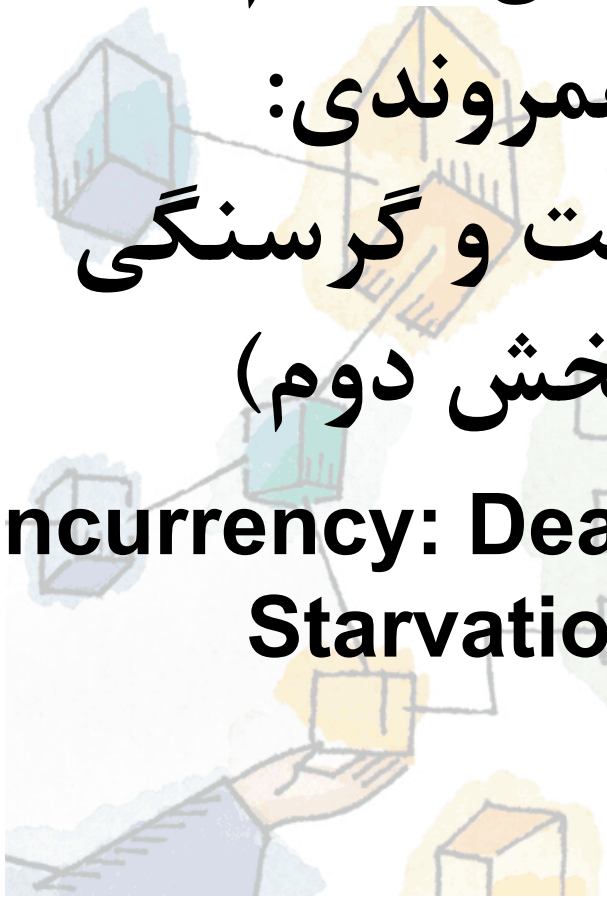
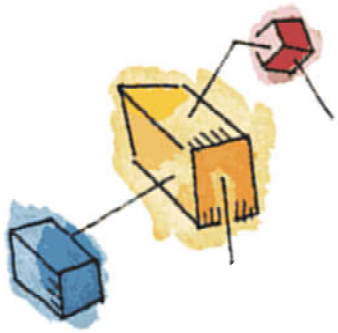
به نام خدا

فصل ششم
همروندی:
بن بست و گرسنگی
(بخش دوم)

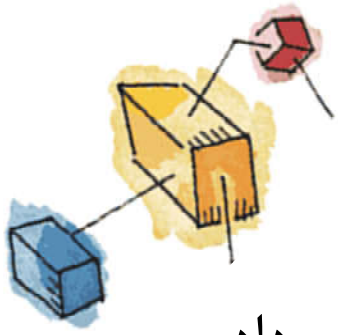
Concurrency: Deadlock and Starvation





سرفصل مطالب

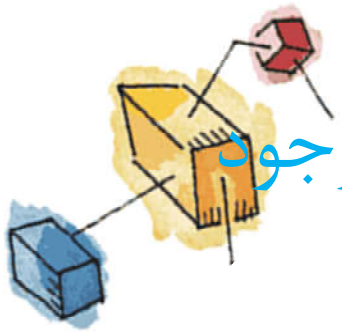
- اصول بن بست
- راه های برخورد با مساله بن بست
  - پیش گیری از بن بست (Deadlock Prevention)
  - اجتناب از بن بست (Deadlock Avoidance)
  - کشف بن بست (Deadlock Detection)



کشف بن بست (Deadlock Detection)

- روشی است که در آن اجازه برای انجام فرآیند و هر تخصیص منبع داده می شود و سیستم عامل مرتب وجود بن بست را بررسی می کند.

- این روش، یکی از متداول ترین روش های مواجهه با بن بست است که در سیستم عامل ها اتخاذ شده است.



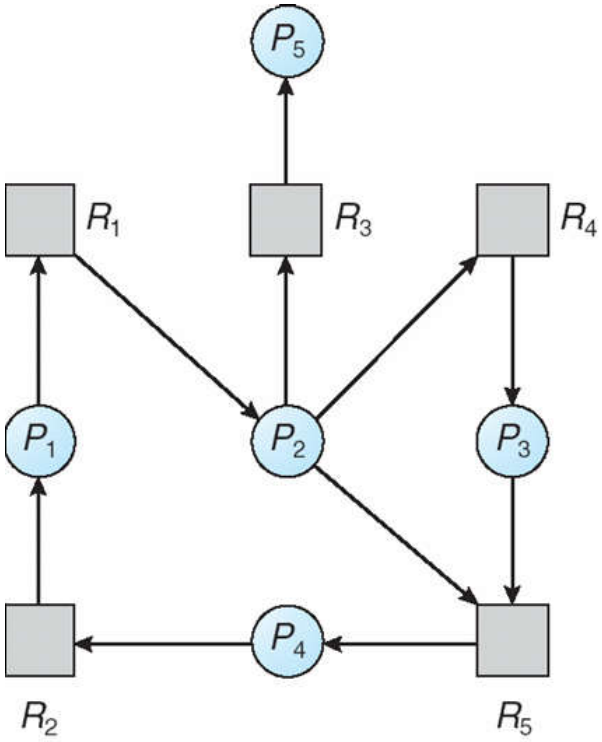
کشف بن بست در حالتی که یک نمونه از هر منبع وجود دارد

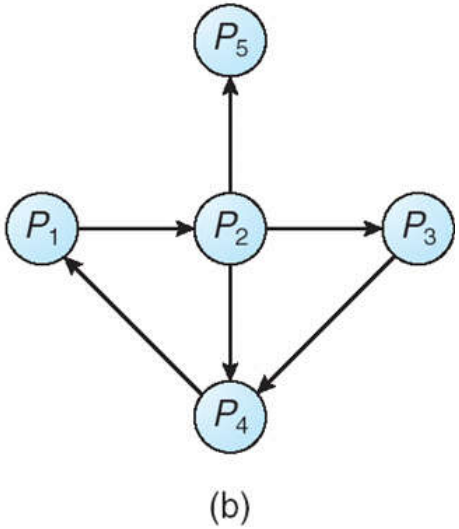
- ایجاد گراف انتظار
  - نودهای گراف، فرآیندها هستند
  - $P_i \rightarrow P_j$ اگر P_i منتظر P_j است.

- به طور متناوب الگوریتمی صدا زده می شود تا وجود دور در گراف را جستجو کند.
- اگر دوری در گراف وجود داشته باشد، سیستم در حالت بن بست است.

- الگوریتمی که وجود دور در گراف را تشخیص می دهد، از مرتبه n^2 است که n تعداد نودها در گراف است.

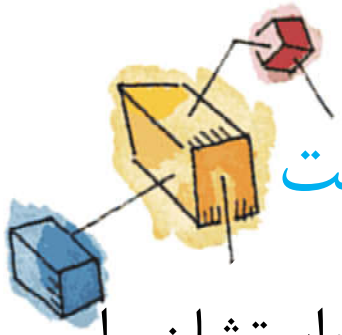
گراف تخصیص منابع و گراف انتظار





گراف تخصیص منابع

گراف انتظار مربوطه



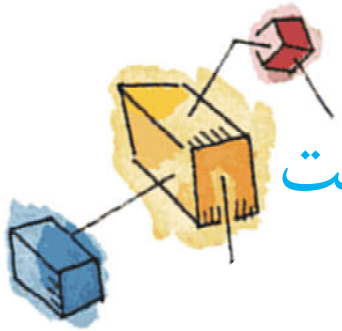
کشف بن بست وقتی چند نمونه از منابع موجود است

- به روشی مانند الگوریتم بانکدار، فرآیندهایی که می توان درخواستشان را پاسخ داد شناسایی می شوند و از لیست خارج می شوند.

- الگوریتم با علامت زدن فرآیندهایی که در بن بست نیستند پیش می رود.

- اگر فرآیندهایی باقی بمانند که نتوان درخواست آنها را پاسخ داد، سیستم در حالت بن بست است.

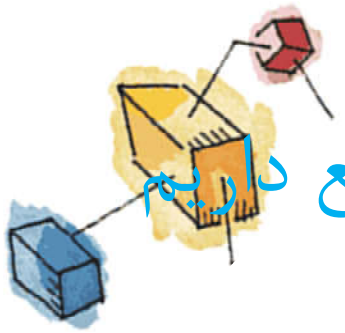
- ماتریس Q ماتریس درخواست های فعلی است (به جای ماتریس نیازها یا need که در الگوریتم بانکدار استفاده می شد و همه نیازهای آتی فرآیند را نشان می داد).



کشف بن بست وقتی چند نمونه از منابع موجود است

مراحل الگوریتم:

1. هر فرآیندی که یک سطر تماما صفر در Allocation دارد را علامت بزن.
2. بردار موقتی W را برابر با بردار Available قرار بده.
3. اندیس i را به گونه ای پیدا کن که فرآیند i ام تا کنون علامت نخورده باشد و $Q_{ik} \leq W_k \quad 1 \leq k \leq m$
   - اگر چنین فرآیندی پیدا نشد، به الگوریتم پایان بده.
   - اگر پیدا شد، فرآیند i ام را علامت بزن و سطر متناظر از ماتریس Allocation را با W جمع بزن. به گام ۳ برو.



مثالی برای کشف بن بست وقتی چند نمونه از منابع داریم

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

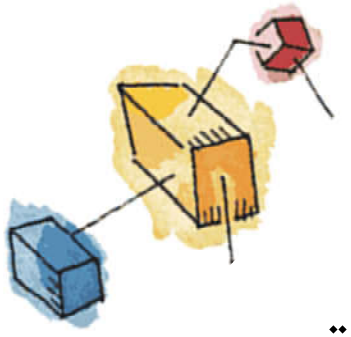
R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

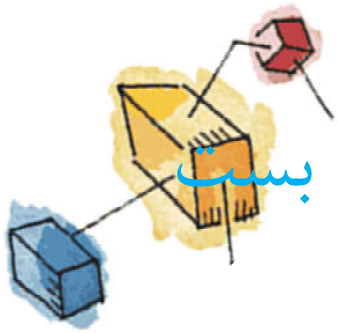
Available vector

- در این مثال، P4 و P3 علامت می خورند ولی P1 و P2 علامت نمی خورند پس در بن بست هستند .



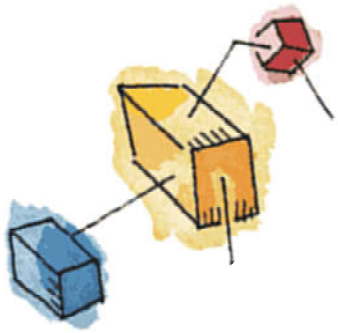
فراخوانی الگوریتم کشف بن بست

- بسته به احتمال بروز بن بست در سیستم، تعداد دفعات بررسی وجود بن بست می تواند مساوی تعداد دفعات درخواست منابع یا کمتر باشد.
  - اگر با هر درخواست منبع، الگوریتم کشف بن بست اجرا شود، بن بست سریع تر کشف می شود و الگوریتم ساده تری خواهد داشت.
  - ولی از طرف دیگر، این کار زمان بر خواهد بود و بخش زیادی از وقت پردازنده را می گیرد.



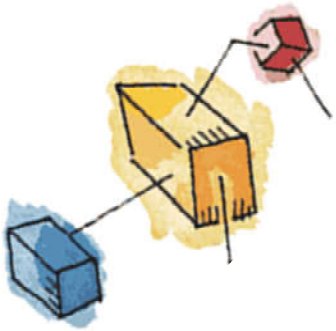
استراتژی های ترمیم قابل استفاده پس از کشف بن بست

1. قطع تمام فرآیندهای در بن بست
2. برگشت هر یک از فرآیندهای قرار گرفته در بن بست به یک checkpoint قبلی و سپس شروع مجدد فرآیندها.
   - ممکن است بن بست اولیه دوباره رخ دهد.
3. متوقف کردن یک به یک فرآیندهای در بن بست تا جایی که دیگر بن بست وجود نداشته باشد.
4. قبضه کردن پی در پی منابع تا جایی که دیگر بن بست وجود نداشته باشد.



اقدام پس از کشف بن بست

- برای بند های ۳ و ۴ معیار انتخاب می تواند یکی از موارد زیر باشد:
  - کمترین وقت مصرفی از پردازنده تا کنون
  - کمترین خروجی تولید شده تا کنون
  - بیشترین زمان باقیمانده تخمینی
  - کمترین منابع تخصیص داده شده تا کنون
  - کمترین اولویت



مساله غذا خوردن فیلسوفان

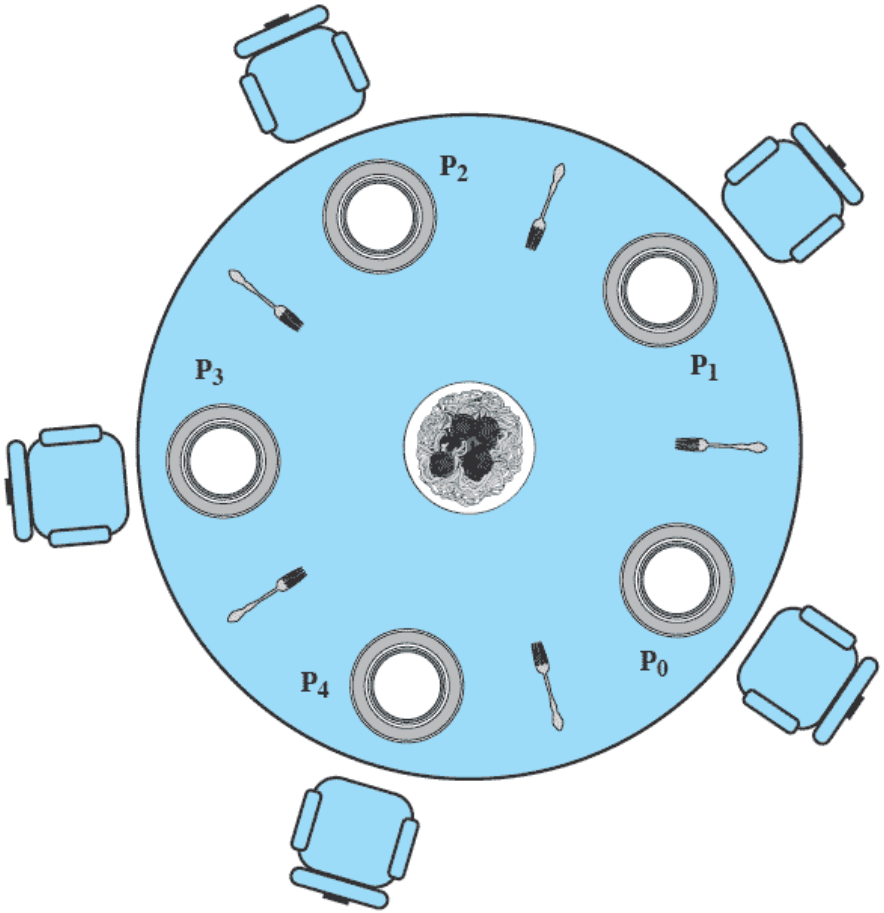
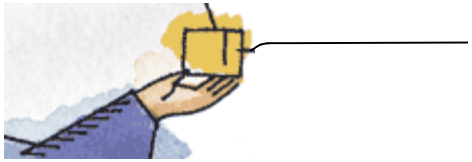


Figure 6.11 Dining Arrangement for Philosophers



حل مساله غذا خوردن فیلسوفان با استفاده از سمافور (راه اول)

این راه حل ممکن است منجر به بن بست شود.

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
      while (true) {
            think();
            wait (fork[i]);
            wait (fork [(i+1) mod 5]);
            eat();
            signal(fork [(i+1) mod 5]);
            signal(fork[i]);
      }
}
void main()
{
      parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
      }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

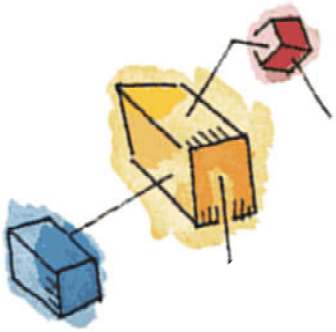
حل مساله غذا خوردن فیلسوفان با استفاده از سمافور (راه دوم)

در هر زمان تنها چهار فیلسوف را به اتاق غذاخوری راه دهیم. بنابراین حداقل یک فیلسوف به دو چنگال دسترسی دارد. این راه بدون بن بست و گرسنگی است.

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem



- پایان فصل ششم