انجمن جاوا کاپ تقدیم می‌کند

دوره برنامه‌نویسی جاوا

برنامه‌نویسی چندنخی
Multi-Thread Programming

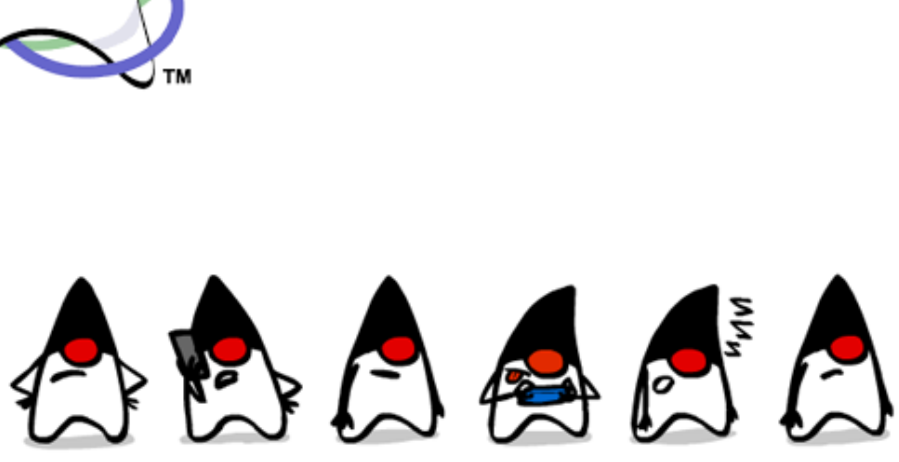
صادق علی اکبری

حقوق مؤلف

- کلیه حقوق این اثر متعلق به انجمن جاواکاپ است
- بازنشر یا تدریس آن‌چه توسط جاواکاپ و به صورت عمومی منتشر شده است، با ذکر مرجع (جاواکاپ) بلامانع است
- اگر این اثر توسط جاواکاپ به صورت عمومی منتشر نشده است و به صورت اختصاصی در اختیار شما یا شرکت شما قرار گرفته، بازنشر آن مجاز نیست
- تغییر محتوای این اثر بدون اطلاع و تأیید انجمن جاواکاپ مجاز نیست

سرفصل مطالب

- مفهوم همروندی (Concurrency) و برنامه‌های همروند
- نخ (Thread) و برنامه‌نویسی چندنخی (Multi-thread)
- همزمانی (Synchronization)
- حالت‌های یک نخ (Thread State)



مقدمه

برنامه‌نویسی ترتیبی

- برنامه‌هایی که تا این‌جا می‌نوشتیم، به صورت ترتیبی (sequential) اجرا می‌شدند
  - در این برنامه‌ها، دستورات یکی پس از دیگری اجرا می‌شدند
- اما چگونه برنامه‌ای بنویسیم که چند کار را به طور همزمان انجام می‌دهد؟
  - چنین برنامه‌هایی چه مخاطراتی دارند و چه نکاتی را باید رعایت کنیم
  - چه امکاناتی در این زمینه در زبان جاوا تعبیه شده است
- فایده: گاهی باید یک برنامه چند کار را همزمان اجرا کند
  - مثلاً یک آنتی‌ویروس، همزمان با جستجوی ویروس، امکان تعامل با کاربر را داشته باشد
- فایده: کامپیوترهای امروزی معمولاً می‌توانند چند دستور را همزمان اجرا کنند
  - اجرای موازی (parallel) و افزایش کارایی

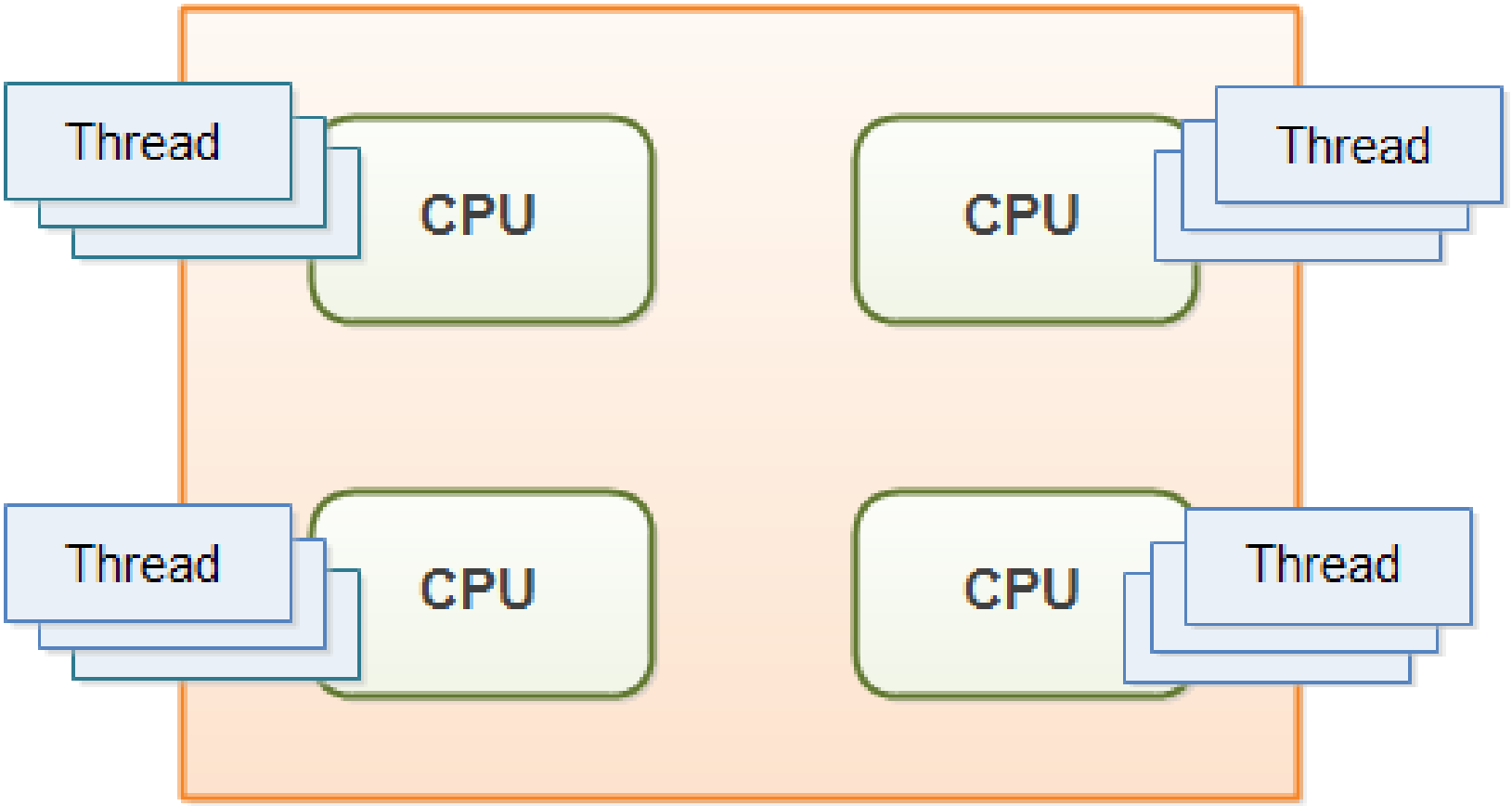
چندپردازشی، چندنخی

- مفهوم چند پردازشی (multi-tasking یا multi-processing):
  - یعنی سیستم‌عامل بتواند چند «برنامه» را همزمان اجرا کند
  - سیستم‌عامل‌های مهم و معمولی این امکان را دارند (ویندوز، لینوکس و ...)
  - مثلاً در ویندوز همزمان با Eclipse می‌توانیم Chrome را هم اجرا کنیم
- مفهوم چندنخی (multi-thread)
  - یعنی یک برنامه بتواند چند بخش را به صورت همزمان اجرا کند
  - هر جریان اجرایی در یک برنامه: یک **نخ** اجرایی (thread of execution)
  - مثلاً همزمان با یک متد، متدی دیگری را در اجرا داشته باشد
  - به همزمانی در اجرای چند بخش، همروندی (concurrency) می‌گویند

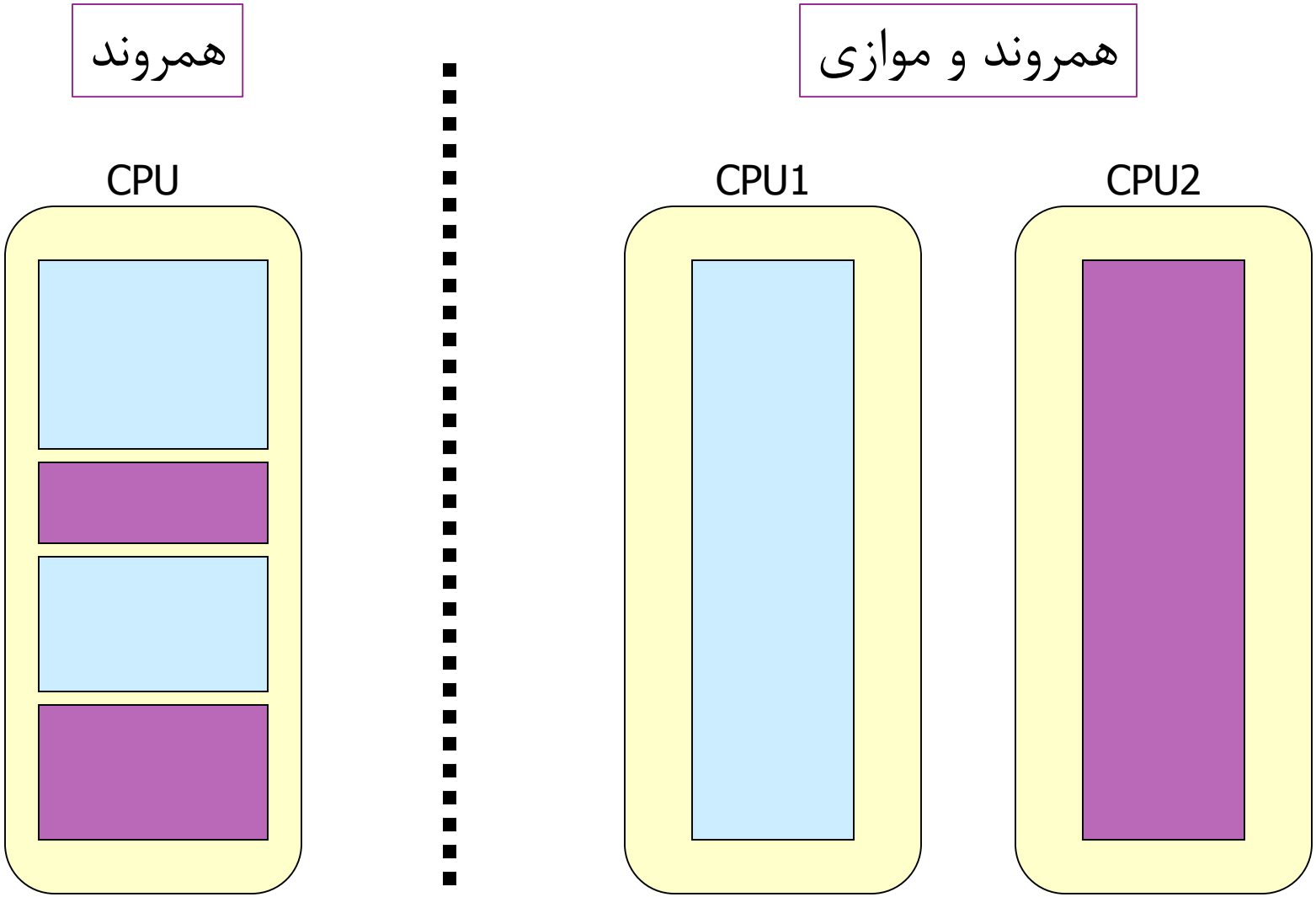
اجرای موازی و اجرای همروند

- مفهوم اجرای موازی (Parallel)
  - یعنی دو دستور واقعاً همزمان با هم در حال اجرا باشند
  - مثلاً همزمان که یک پردازنده (CPU) یک متد را اجرا می‌کند، یک پردازنده دیگر متدی دیگر را اجرا کند
- اجرای همروند (concurrency)
  - یعنی ظاهراً چند بخش همزمان با هم در حال اجرا باشند
  - چند بخش همزمان در حال پیشرفت هستند
  - ولی لزوماً به صورت موازی اجرا نمی‌شوند
  - شاید در هر لحظه، یکی از این کارها در حال اجرا باشد

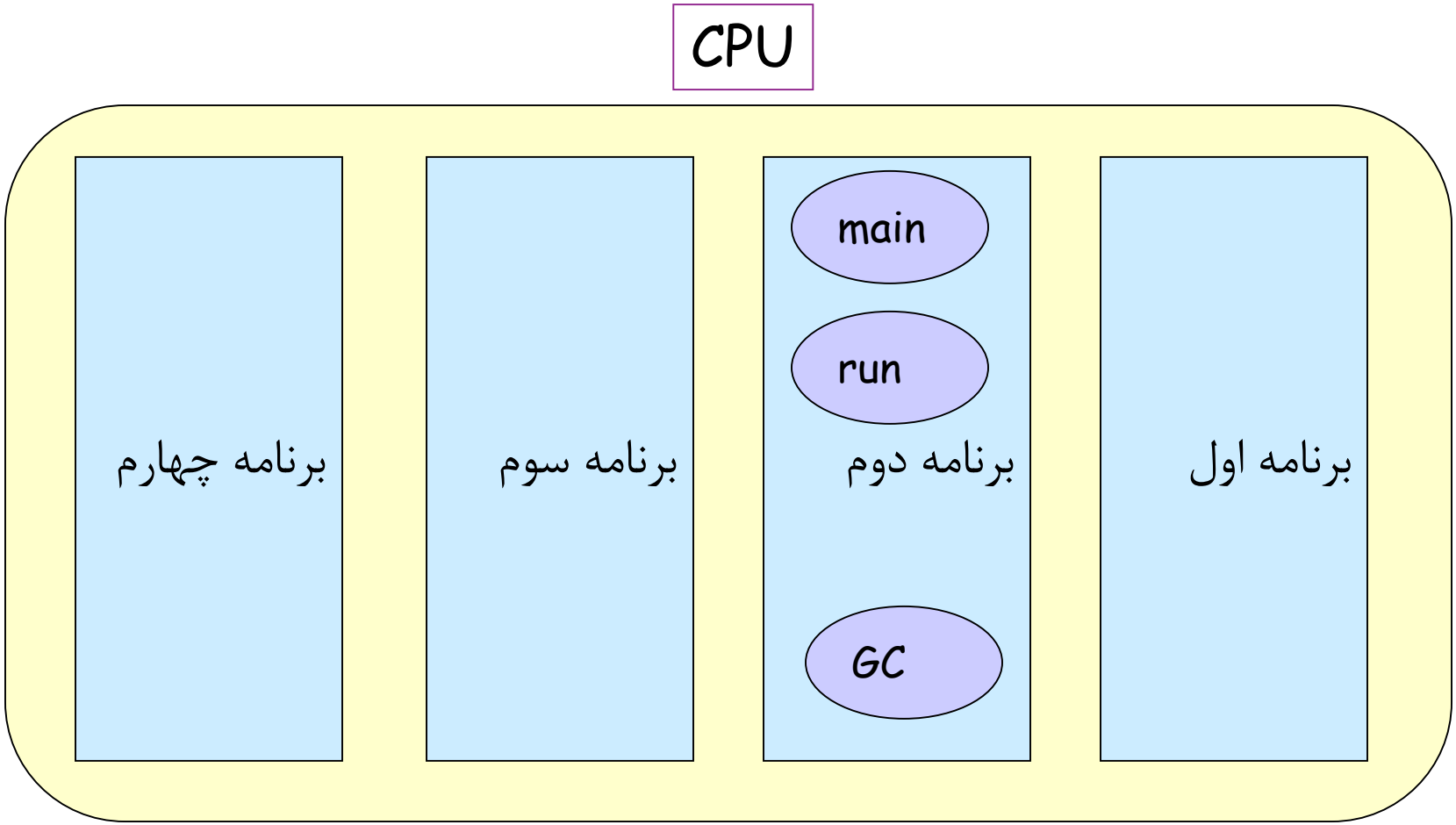
اجرای موازی، اجرای همروند



همروندی و موازی بودن



پردازش‌ها و نخ‌ها



مزایای همروندی

- همروندی در یک برنامه چه فوایدی دارد؟
  - افزایش کارایی
    - مثلاً اگر چند پردازنده و یا چند هسته پردازشی داشته باشیم
    - کامپیوترهای چندپردازنده‌ای و پردازنده‌های چندهسته‌ای (مثل core i7)
  - پیشرفت همزمان چند کار (مثلاً ذخیره و پردازش اطلاعات)
  - برنامه‌های پاسخگو (همزمان با پردازش، تعامل با کاربر ممکن است)
- حتی بدون امکان اجرای موازی، امکان همروندی برنامه مفید است

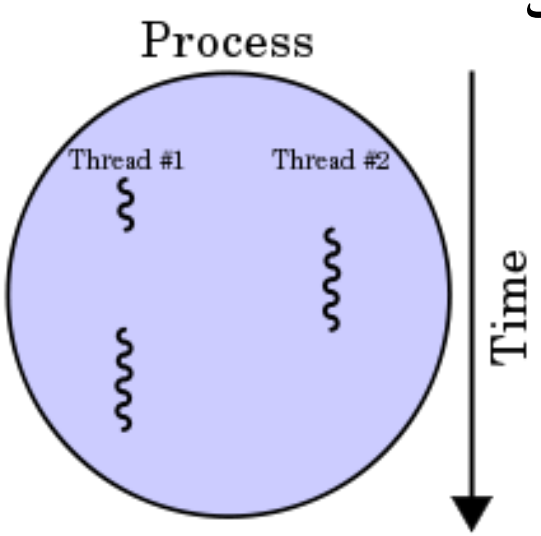
مفهوم نخ (Thread)

مفهوم نخ (Thread) در برنامه‌نویسی

- وقتی یک برنامه جاوا را اجرا می کنیم:

یک نخ (thread) ایجاد می‌شود که متد main() را اجرا می‌کند

- برنامه می‌تواند نخ‌های جدیدی ایجاد کند و سپس آن‌ها را اجرا کند
- نخ‌های مختلف به صورت همروند اجرا می‌شوند
  - شاید به صورت موازی



ایجاد نخ

- دو راه اولیه برای تعریف رفتار یک نخ جدید در برنامه وجود دارد
- در هر دو راه، کلاس جدیدی می‌سازیم
  ۱- کلاس جدید زیرکلاس java.lang.Thread باشد
  ۲- کلاس جدید واسط java.lang.Runnable را پیاده‌سازی کند
- متد run را در کلاس جدید پیاده‌سازی می‌کنیم
  - این متد، دستورات نخ (thread) جدید را توصیف می‌کند

راه اول

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Hello");
        System.out.println("Bye");
    }
}
public class ThreadExample{
    public static void main(String[] args) {
        System.out.println("Salam");
        MyThread t = new MyThread();
        t.start();
        System.out.println("Khodahafez");
    }
}
```

خروجی محتمل:

```
Salam
Khodahafez
Hello
Bye
```

```
Salam
Hello
Khodahafez
Bye
```

- راه اول: ایجاد زیرکلاس Thread
- برای ایجاد نخ جدید: یک شیء از این کلاس بسازیم و متد start آن را فراخوانی کنیم
- برنامه فوق دو نخ (جریان اجرایی همروند) دارد
- یکی Salam و Khodahafez را چاپ می‌کند و دیگری Hello و Bye

راه دوم: پیاده‌سازی واسط Runnable

```
class MyRunnable implements Runnable{
    @Override
    public void run() {
        System.out.println("Hello");
        System.out.println("Bye");
    }
}
```

```
Thread t = new Thread(new MyRunnable());
t.start();
```

- برای ایجاد نخ جدید: یک شیء (مثلاً با نام r) از این کلاس جدید بسازیم
- یک شیء از کلاس Thread بسازیم (مثلاً با نام t) و در سازنده آن r را پاس کنیم
- متد start را روی t فراخوانی کنیم

دو سؤال

- راه اول (زیرکلاس Thread) بهتر است یا راه دوم (پیاده‌سازی واسط Runnable) ؟
  - هرچند راه اول پیاده‌سازی ساده‌تری دارد
  - در راه دوم دست طراح بازتر است تا کلاس موردنظر از کلاسی دلخواه ارث‌بری کند
    - اگر کلاس ما زیرکلاس Thread باشد نمی‌تواند از کلاس دیگری ارث‌بری کند
    - معمولاً واسط Runnable پیاده‌سازی می‌شود
- چرا متد run را پیاده‌سازی می‌کنیم ولی متد start را فراخوانی می‌کنیم؟
  - متد start یک متد خاص در کلاس Thread است
    که یک فرایند سطح پایین و سیستمی (ایجاد نخ جدید) را اجرا می‌کند
    و در نخ جدید، متد run را صدا می‌زند
- فراخوانی متد run فراخوانی تابعی معمولی است که به ایجاد نخ جدید منجر نمی‌شود

امکانات کلاس Thread

متدهای Thread

- برای هر نخی که اجرا می‌شود، یک شیء از کلاس Thread ساخته شده است
- متدهای شیء Thread امکاناتی برای نخ متناظر ارائه می‌کنند
- متدهای کلاس Thread
  - run, start, getId, setPriority, setDaemon, …
  - **متد استاتیک currentThread** : نخ جاری را برمی‌گرداند

```
Thread t = Thread.currentThread();
```

  - **متد استاتیک sleep** : نخ جاری مدتی به خواب می‌رود

(اجرای آن به اندازه مشخصی متوقف می‌شود و سپس ادامه می‌یابد)

    - نحوه فراخوانی: sleep(m) یا sleep(m, n)
    - اجرای این نخ به مدت m میلی ثانیه و n نانوثانیه متوقف می‌شود

متد join

- گاهی لازم است کار یک نخ تمام شود، تا اجرای یک بخش از کد ادامه یابد
  - مثلاً نخ «ارسال پیام» متوقف شود تا کار نخ «جستجوی ویروس» تمام شود
- یک نخ می‌تواند تا اتمام یک نخ دیگر منتظر بماند (موقتاً متوقف شود)
- این کار با کمک متد join انجام می‌شود

```
Thread virusScan = new VirusScanThread();
virusScan.start();
prepareEmail();
virusScan.join();
sendEmail();
```

- نکته:

متدهای sleep و join ممکن است خطای InterruptedException پرتاب کنند

- توضیح بیشتر درباره این خطا در ادامه خواهد آمد

اولویت نخ

- اولویت (priority) یک نخ قابل تنظیم است
- اولویت نخ، با کمک متد setPriority تغییر می‌کند
- اولویت یک عدد بین ۱ تا ۱۰ است که میزان اهمیت نخ را نشان می‌دهد
- سیستم‌عامل سعی می‌کند نخ‌های بااولویت بالا را بیشتر اجرا کند
  - زمان بیشتری از CPU به نخ‌های بااولویت تخصیص می‌یابد

```
MyThread th = new MyThread();
th.setPriority(Thread.MAX_PRIORITY);
th.start();
```

```
MIN_PRIORITY = 1;
NORM_PRIORITY = 5;
MAX_PRIORITY = 10;
```

نخ‌های شبح (Daemon Threads)

- نوع خاصی از نخ‌ها هستند که در پس‌زمینه اجرا می‌شوند
- معمولاً خدماتی به سایر نخ‌ها ارائه می کنند و مستقلاً و به تنهایی معنا ندارند
- مثلاً زباله‌روب (garbage collector) یک daemon thread است
- از آن‌جا که اجرای مستقل و تنهای آن‌ها بی‌معنی است:
  - اگر فقط نخ‌های شبح در یک برنامه زنده باشند و نخ‌های معمولی پایان یافته باشند، JVM نخ‌های شبح را هم خاتمه می‌دهد و برنامه پایان می‌پذیرد
- با استفاده از متد setDaemon() : نخ به صورت شبح یا معمولی تغییر می‌کند
- مثال:

```
MyThread th = new MyThread();
th.setDaemon(true);
th.start();
```

کوییز

کوییز

```
class T extends Thread {
    public void run() {
        for (int i = 1; i <= 100; i++)
            System.out.println(i);
    }
}
class R implements Runnable{
    public void run() {
        for (char c = 'A'; c < 'Z'; c++)
            System.out.println(c);
    }
}
public class Threading{
    public static void main(String[] args) {
        new Thread(new R()).start();
        new T().start();
        new Thread(new R()).start();
        new T().start();
        for (char c = 'a'; c < 'z'; c++)
            System.out.println(c);
    }
}
```

- این برنامه چند نخ دارد؟
  - پنج نخ
(main و ۴ نخ همروند جدید)
- خروجی؟ چاپ موارد زیر:
  - دو بار از ۱ تا ۱۰۰
  - دو بار از A تا Z
  - یک بار از a تا z
- اما ترتیب چاپ قابل پیش‌بینی نیست
  - مثلاً شاید بعد از A عدد ۱ و سپس a چاپ شود

تمرین عملی

تمرین عملی

- ایجاد چند نخ
- مرور همروندی نخ‌ها
  - همزمان اجرا می‌شوند
  - ممکن است در هر اجرا، ترتیب متفاوتی از اجرا داشته باشیم
- متد sleep و currentThread
  - متد getId
- مرور این که هر نخ، پشته مخصوص خودش را دارد
  - مشاهده پشته با کمک دیباگ کردن
  - تأکید بر پیچیدگی دیباگ در برنامه‌های چندنخی

بخش بحرانی (Critical Section)

نکته

- قبلاً دیده بودیم که:
  - در حافظه هر برنامه، بخش‌هایی مثل پشته (stack) و Heap وجود دارد
  - متغیرهای محلی در پشته و اشیاء در Heap نگهداری می‌شوند
- در واقع هر نخ، یک پشته مخصوص خودش دارد
- مثلاً اگر دو نخ مختلف، یک متد یکسان را فراخوانی کنند،

هر نخ، حافظه مجزایی برای متغیرهای محلی آن متد، در پشته خودشان خواهند داشت

- ولی همه نخ‌ها از حافظه Heap به‌طور مشترک استفاده می‌کنند
- دو نخ مختلف، می‌توانند از یک شیء مشترک استفاده کنند

بخش‌های بحرانی (Critical Section)

- دو نخ مختلف، می‌توانند همزمان از یک شیء مشترک استفاده کنند
- این وضعیت ممکن است مشکلاتی را ایجاد کند. مثال:
  - همزمان که یک نخ در حال تغییر یک شیء است، یک نخ دیگر همان شیء را تغییر دهد
  - در زمانی که یک نخ مشغول کار با یک فایل است، یک نخ دیگر آن فایل را ببندد
- بخش‌های بحرانی (critical section) :

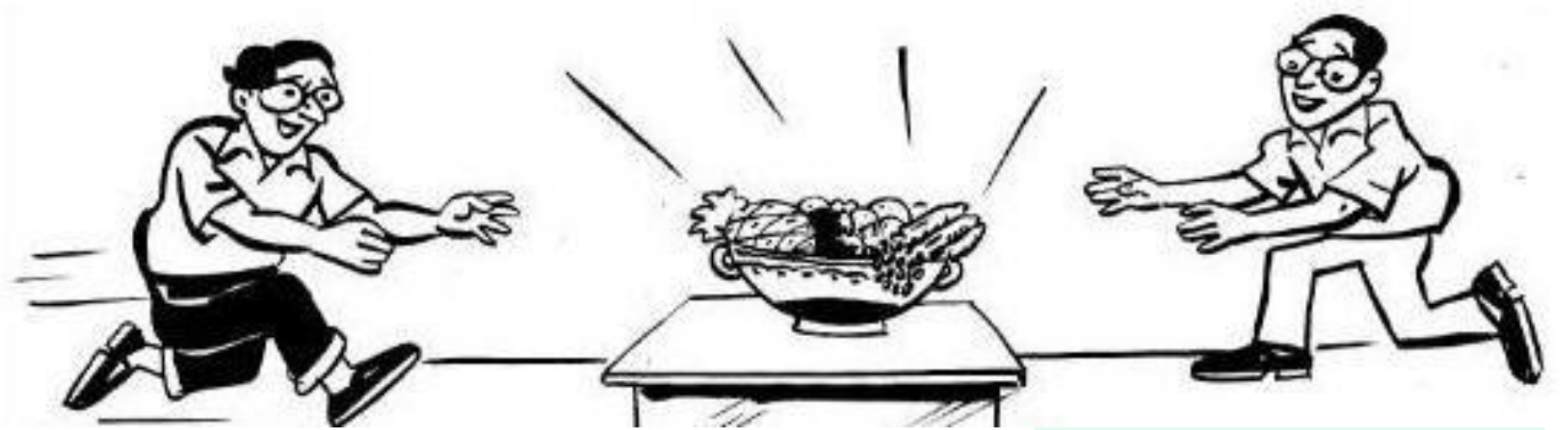
  بخش‌هایی از برنامه که نمی‌خواهیم همزمان توسط چند نخ اجرا شوند
- اگر یک نخ وارد بخش بحرانی شد، نباید نخ دیگری وارد آن شود
  - اجرای نخ دوم باید متوقف شود، تا زمانی که اجرای بخش بحرانی در نخ اول خاتمه یابد

چند اصطلاح

- منبع مشترک (shared resource)
  - یک موجود (متغیر، شیء، فایل، دستگاه، ...)که همزمان در چند نخ، مورداستفاده است
- شرایط مسابقه (race condition)
  - شرایطی که در آن چند نخ، همزمان به یک منبع مشترک دسترسی می‌یابند
  - و حداقل یکی از نخ‌ها سعی در تغییر منبع مشترک دارد
- بخش بحرانی (critical section)
  - بخشی از برنامه‌ی هر نخ، که در آن وارد شرایط مسابقه می‌شود
- انحصار متقابل (Mutual Exclusion یا Mutex)
  - چند نخ نباید همزمان بخش بحرانی را اجرا کنند
  - با ورود یک نخ به بخش بحرانی، باید از ورود نخ‌های دیگر به بخش بحرانی جلوگیری شود

شرایط مسابقه

- شرایط مسابقه، منبع مشترک، بخش بحرانی و انحصار متقابل

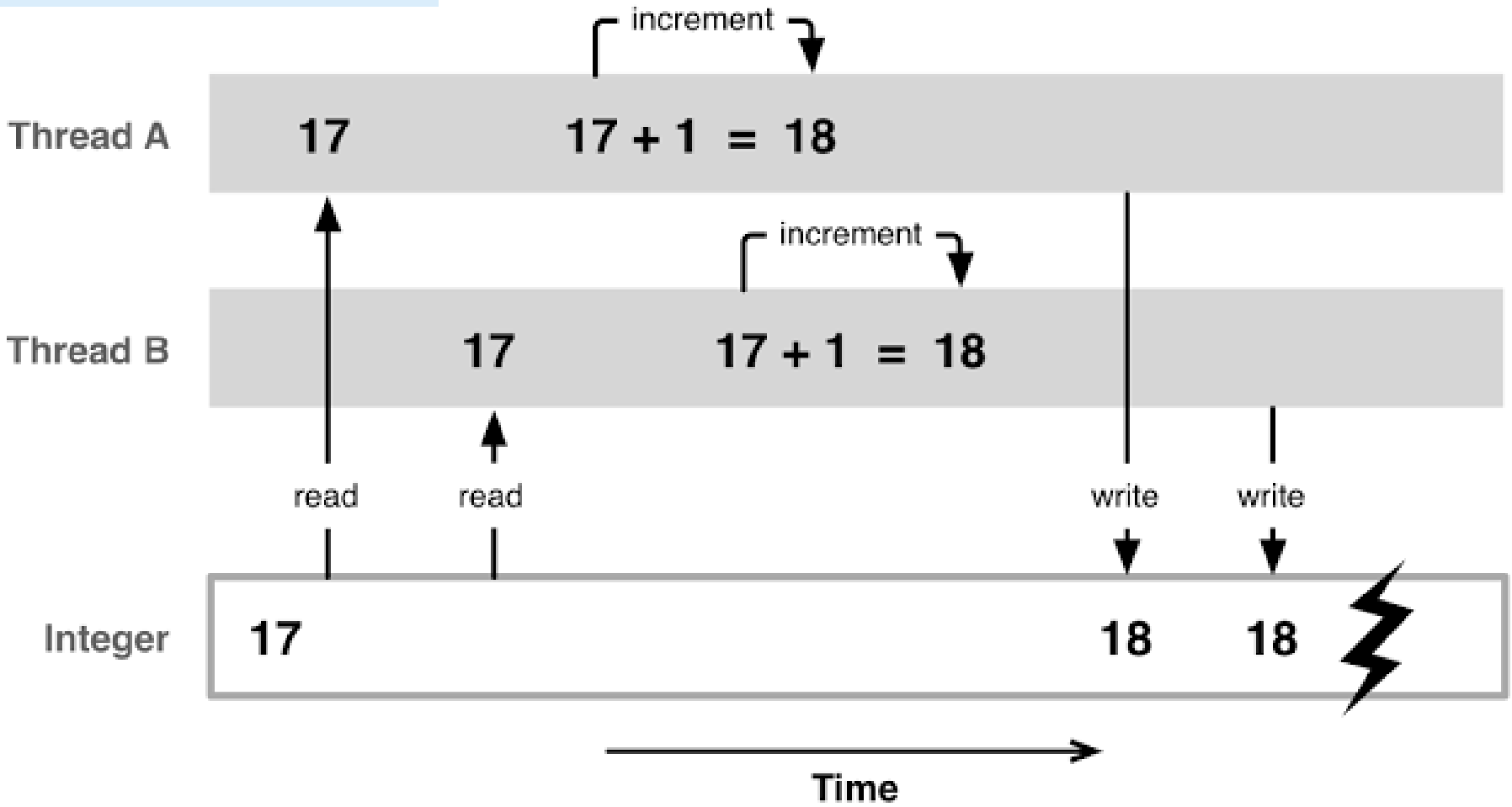


- تا ظرف میوه بدو
- آناناس را بردار
- بخشی از آن را بخور
- بقیه را به ظرف برگردان
- برگرد

- تا ظرف میوه بدو
- آناناس را بردار
- آن را خوب ببین
- آن را به ظرف برگردان
- برگرد

مثال از شرایط مسابقه

```
X=17;
Thread1: X++;
Thread2: X++;
```



بلوک‌های Synchronized

جلوگیری از ورود به بخش بحرانی

- توقف نخ‌ها در زمان لازم به صورت خودکار توسط جاوا انجام می‌شود
  - هنگام ورود یک نخ به بخش بحرانی، وقتی نخی دیگر مشغول اجرای بخش بحرانی است
  - زبان جاوا امکانی برای تعیین بخش‌های بحرانی فراهم کرده است
  - برنامه‌نویس باید بخش‌های بحرانی برنامه و شرایط ورود به آن‌ها را مشخص کند
- هر نخ، هنگام ورود به یک بخش بحرانی یک قفل (lock) را در اختیار می‌گیرد
  - اگر همین قفل را قبلاً یک نخ دیگر گرفته باشد، نمی‌تواند وارد بخش بحرانی شود
  - و تا زمان آزاد شدن قفل منتظر می‌ماند
- هنگام خروج از بخش بحرانی، قفلی که گرفته را آزاد می‌کند
- برنامه‌نویس مشخص می‌کند که برای ورود به هر بخش، چه قفلی لازم است

مثال

```
public class BankAccount {
    private float balance;
    public synchronized void deposit(float amount) {
        balance += amount;
    }
    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
}
```

- در این کلاس:

متدهای deposit و withdraw (واریز و برداشت) بخش‌های بحرانی هستند

- اگر یک نخ مشغول تغییر موجودی یک حساب (balance) است،

نباید یک نخ دیگر سعی در تغییر موجودی بدهد

- باید صبر کند تا کار نخ قبلی تمام شود

- بخش بحرانی با کلیدواژه **synchronized** مشخص می‌شود

معنای synchronized

- هر شیئی در جاوا می‌تواند به عنوان مجوز ورود به بخش بحرانی استفاده شود
- هرگاه یک متد synchronized روی یک شیء فراخوانی شود،
  قبل از ورود به این متد، سعی می‌کند قفل همان شیء را بگیرد
  - یعنی قفل this را بگیرد
  - به ازای هر شیء، یک قفل وجود دارد
- وقتی یک متد synchronized در حال اجراست:
  - همزمان هیچ متد synchronized دیگری روی همان شیء آغاز نمی‌شود
  - چون تا پایان این متد، متد دیگری نمی‌تواند قفل this را بگیرد

کوییز

سؤال: کدام گزینه‌ها صحیح هستند؟

```
public class BankAccount {
    private float balance;
    public synchronized void deposit(float amount) {
        balance += amount;
    }
    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
}
```

۱- هیچ گاه دو نخ مختلف نمی‌توانند متد deposit را همزمان اجرا کنند

۲- اگر یک نخ درحال اجرای deposit است، نخ دیگری نمی‌تواند اجرای withdraw را آغاز کند

۳- اگر یک نخ روی شیء X متد deposit را اجرا می‌کند،
نخ دیگری نمی‌تواند اجرای deposit روی همان شیء (X) را آغاز کند

۴- اگر یک نخ روی شیء X متد deposit را اجرا می‌کند،
نخ دیگری نمی‌تواند اجرای withdraw روی همان شیء (X) را آغاز کند

گزینه‌های ۳ و ۴

- در موارد فوق منظور از همزمان، همروند است (یعنی قبل از پایان یکی، دیگری شروع شود)

ادمه مبحث synchronized

بلوک synchronized

- دیدیم یک متد می‌تواند synchronized باشد
  یعنی هر نخ باید قبل از ورود به متد، قفل this را بدست آورد و در انتها آزاد کند
- امکان ایجاد بخش بحرانی با کمک قفلی به جز this هم وجود دارد
- این کار با ایجاد بلوک synchronized و ذکر یک شیء ممکن است
- مثال:

```
List<String> names;
...
synchronized(names){
  names.add("ali");
}
```

بلوک synchronized

- یعنی دو نخ مختلف به شرطی می‌توانند همزمان وارد این بلوک شوند
  که شیء names در آن دو نخ متفاوت باشد

چند نکته

- این دو تعریف برای متد g تقریباً هم‌معنی هستند:

```
void g() {
  synchronized(this){
     h();
  }
}
```

```
synchronized void g() {
  h();
}
```

- اگر یک متد استاتیک synchronized شود:

یعنی هر نخ برای ورود به متد باید قفل کلاس را بگیرد (به جای قفل یک شیء)

- یعنی هیچ دو نخی همزمان نمی‌توانند این متد را اجرا کنند
- یک متد غیراستاتیک synchronized را ممکن است دو نخ همزمان اجرا کنند، به شرطی که روی دو شیء مختلف فراخوانی شوند

تعامل بین چند نخ
Inter-thread Communication

متدهای wait و notify

- گاهی لازم است دو نخ با هم تعامل داشته باشند
  - گاهی یک نخ، صبر کند (wait) تا نخی دیگر به آن خبر دهد (notify)
- مثلاً فرض کنید دو نخ داریم: ۱- نمایش‌دهنده ویروس‌ها ۲- جستجوگر ویروس‌ها
- نخ اول متوقف می‌شود، هرگاه نخ دوم ویروسی پیدا کند، به نخ اول خبر می‌دهد هربار نخ اول باخبر می‌شود، به اجرا (نمایش ویروس) ادامه می‌دهد و سپس دوباره متوقف می‌شود
- متدهای wait و notify برای برقراری تعامل بین نخ‌ها استفاده می‌شوند
  - این متدها در کلاس Object تعریف شده‌اند، final هستند
    - از پیاده‌سازی سطح پایین (native) استفاده می‌کنند
- وقتی یک نخ متد wait را روی یک شیء دلخواه فراخوانی می‌کند، متوقف می‌شود تا این که یک نخ دیگر، روی همان شیء متد notify را فراخوانی کند

نکته

- متد wait یا notify فقط در صورتی روی شیء X قابل فراخوانی هستند که در یک بلوک synchronized(X) قرار گرفته باشد
  - یک نخ برای فراخوانی wait یا notify روی یک شیء باید قفل آن شیء را گرفته باشد
- وگرنه خطای IllegalMonitorStateException پرتاب می‌شود
- البته با فراخوانی X.wait ، بلافاصله قفل X آزاد می‌شود
  - تا نخ‌های دیگر بتوانند وارد بلوک synchronized(X) شوند و X.notify را صدا بزنند تا این نخ از حالت انتظار (waiting) خارج شود
- مثال:

```
synchronized void f() {
  wait();
}
```

```
synchronized (obj) {
  obj.notify();
}
```

چند نکته درباره wait

- روی هر شیء، تعدادی نخ wait کرده‌اند
  - هر شیء، فهرستی از نخ‌های منتظر دارد
- با هر فراخوانی notify روی یک شیء، یکی از این نخ‌ها بیدار می‌شود
  - یکی از نخ‌هایی که روی آن شیء منتظر هستند، اجرایش را ادامه می‌دهد
- متد notifyAll همه‌ی نخ‌های منتظر روی آن شیء را بیدار می‌کند
- نکته: متد wait می‌تواند حداکثر مهلت انتظار را مشخص کند
  - مثلاً ;(wait(100 یعنی بعد از ۱۰۰ میلی‌ثانیه از انتظار خارج شود

(حتی اگر در این مدت، متد notify توسط نخ دیگری روی این شیء فراخوانی نشود)

متد interrupt

- گاهی یک نخ منتظر است و اجرای آن متوقف شده است
- مثلاً به خاطر فراخوانی wait یا join یا sleep به حالت انتظار رفته است
- در این حالت اگر متد interrupt روی شیء این نخ فراخوانی شود:
  - نخ منتظر، از حالت انتظار خارج می‌شود
    و یک InterruptedException دریافت می‌کند
- به همین دلیل است که متدهای wait و joint و sleep این خطا را پرتاب می‌کنند

مثال

```
public class Interrupting extends Thread {
    public static void main(String[] a)
            throws InterruptedException{
        Interrupting t = new Interrupting();
        t.start();
        Thread.sleep(1000);
        t.interrupt();
    }
    @Override
    public synchronized void run() {
        try {
            wait();
            System.out.println("After wait");
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("Resume");
    }
}
```

Interrupted
Resume

مثال

```
System.out.println("Main Starts.");
Scan scan = new Scan();
Object obj = scan.obj = new Object();
scan.start();
synchronized (obj) {    obj.wait();    }
System.out.println("Main other jobs");
```

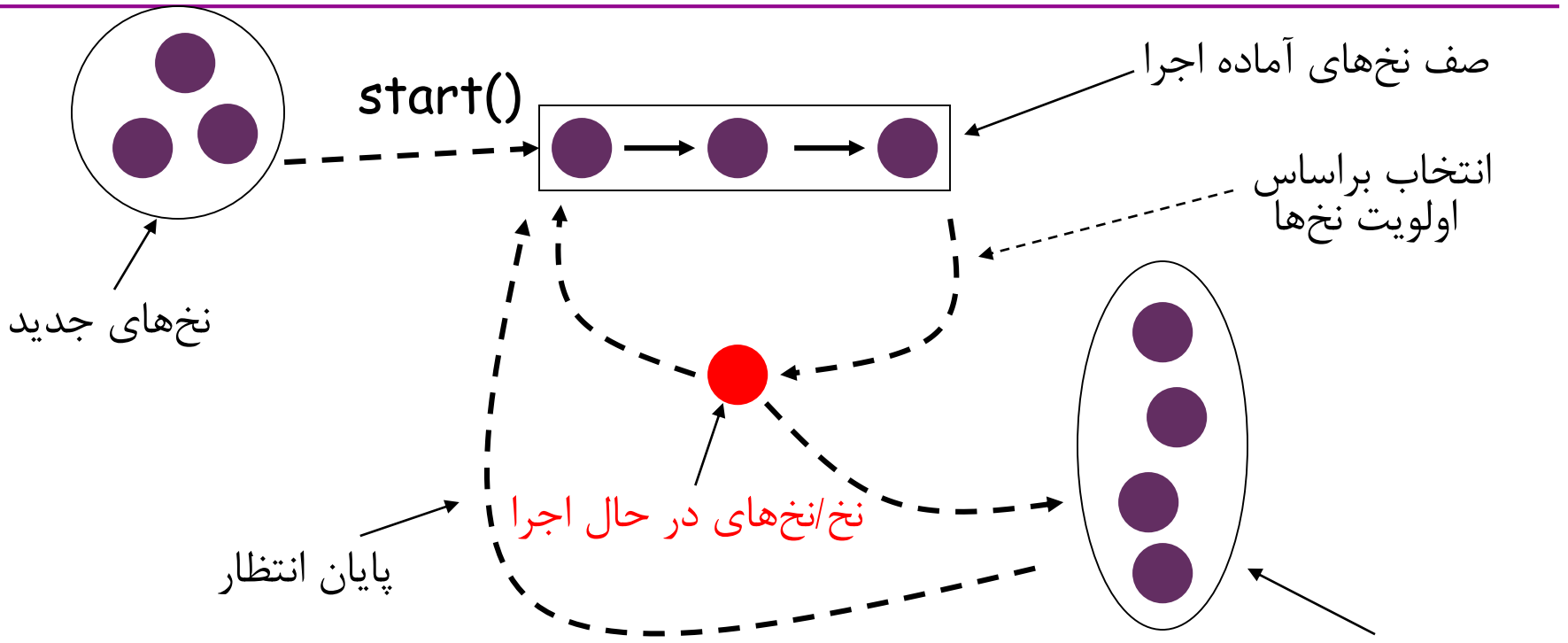
- نتیجه:

تا زمانی که Scan چاپ نشود، Main other jobs چاپ نخواهد شد

```
class Scan extends Thread {
 public Object obj;
 public void run() {
  try { Thread.sleep(1000); } catch (InterruptedException e) {//...}
  System.out.println("Scan");
  synchronized (obj) {      obj.notify();      }
  System.out.println("Scan other jobs");
 }
}
```

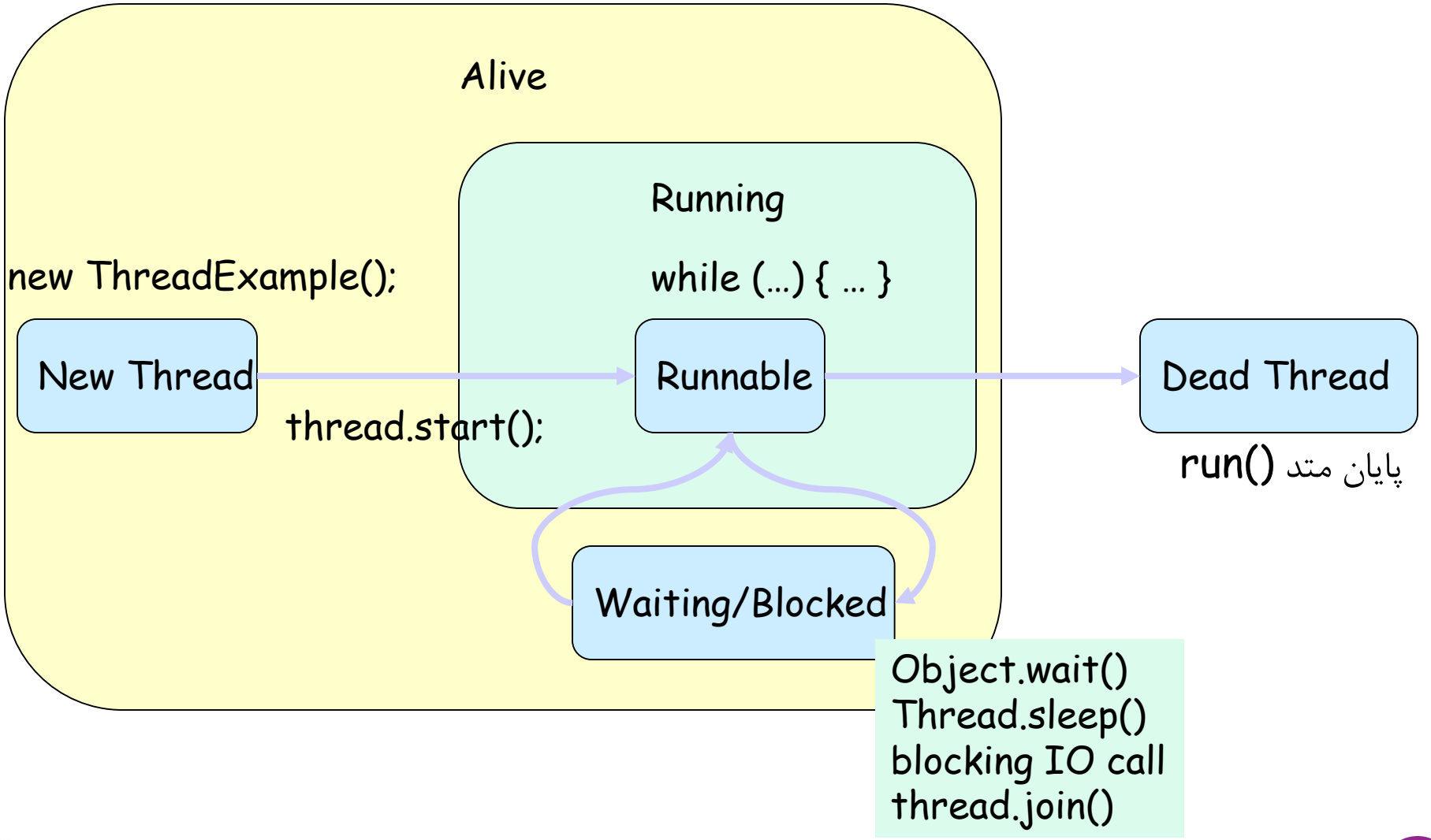
حالت‌های هر نخ

داستان زندگی یک نخ

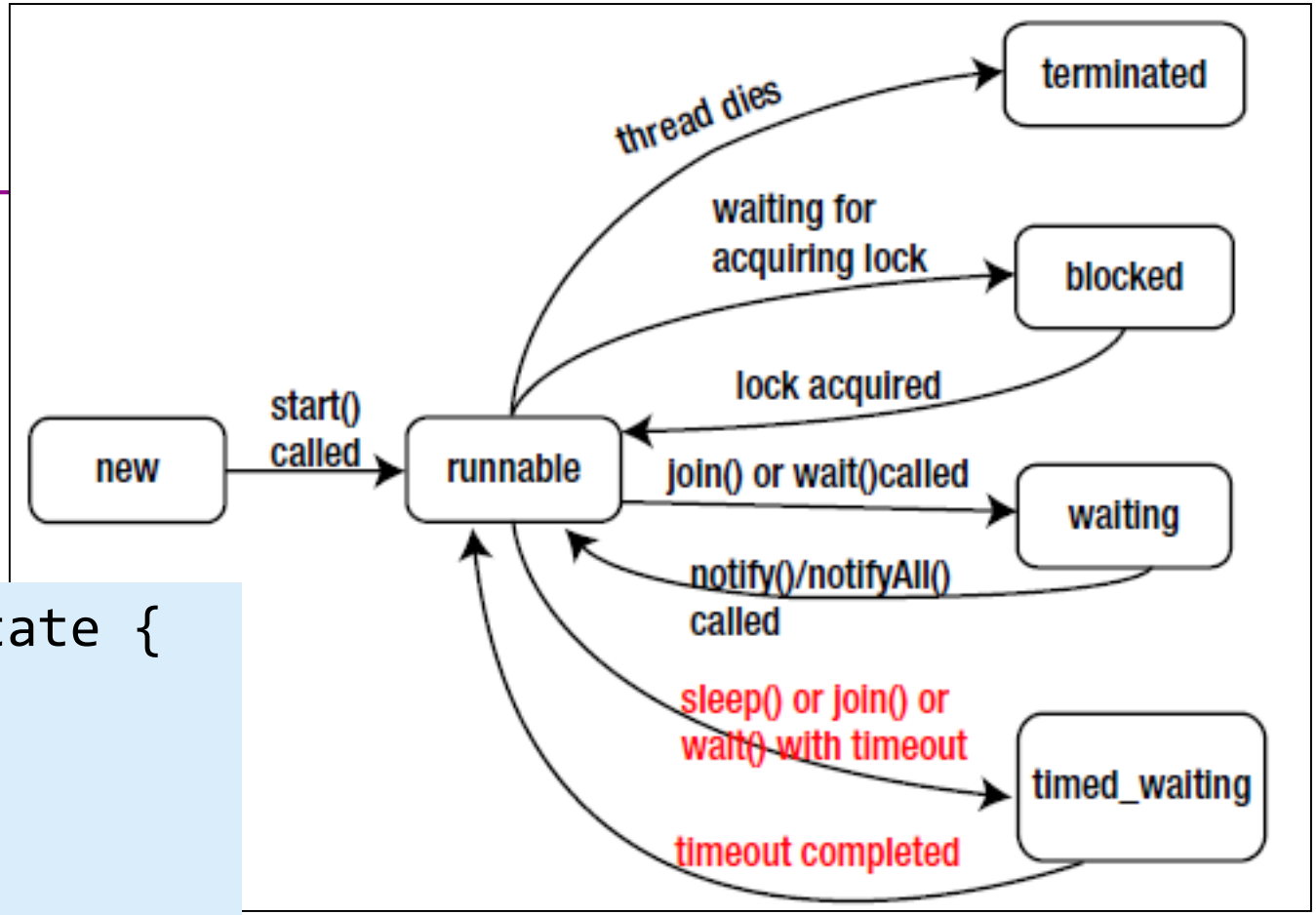


- نخ به خواب رفته (sleep)
- نخی که متد wait یا join را فراخوانی کرده
- انتظار برای ورود به یک بخش synchronized
- در انتظار برای تکمیل فرآیند ورودی/خروجی (IO)
- ...

نگاهی دیگر به حالت‌های نخ



حالت‌های نخ




```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED;
}
```

- متد getState() برای هر شیء از نوع Thread وضعیت آن نخ را برمی‌گرداند

کوییز

سؤال

- تفاوت فراخوانی sleep و wait و join چیست؟

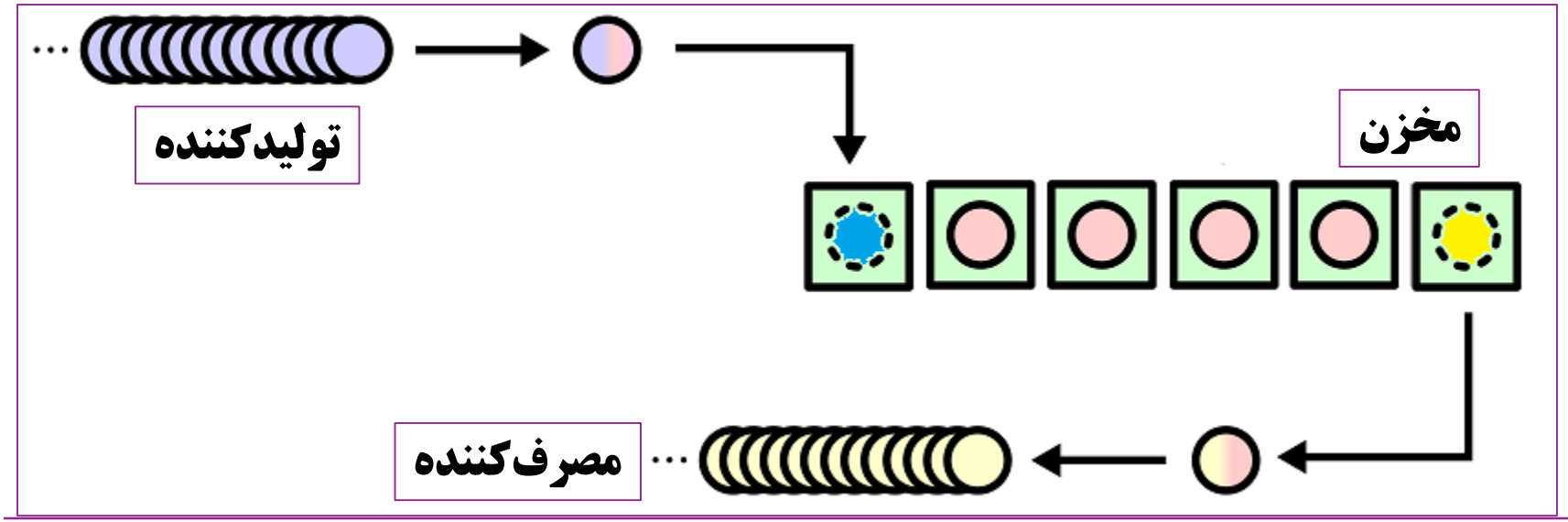
- **پاسخ:**
- sleep : برای مدت مشخصی متوقف می‌شود و سپس به اجرا ادامه می‌دهد
- wait : متوقف می‌شود تا یک نخ دیگر آن را باخبر (notify) کند
- join : متوقف می‌شود تا یک نخ دیگر پایان یابد

تمرین عملی

مسأله تولیدکننده/مصرف‌کننده (Producer / Consumer)

یک مسأله سنتی و پرکاربرد در زمینه همروندی

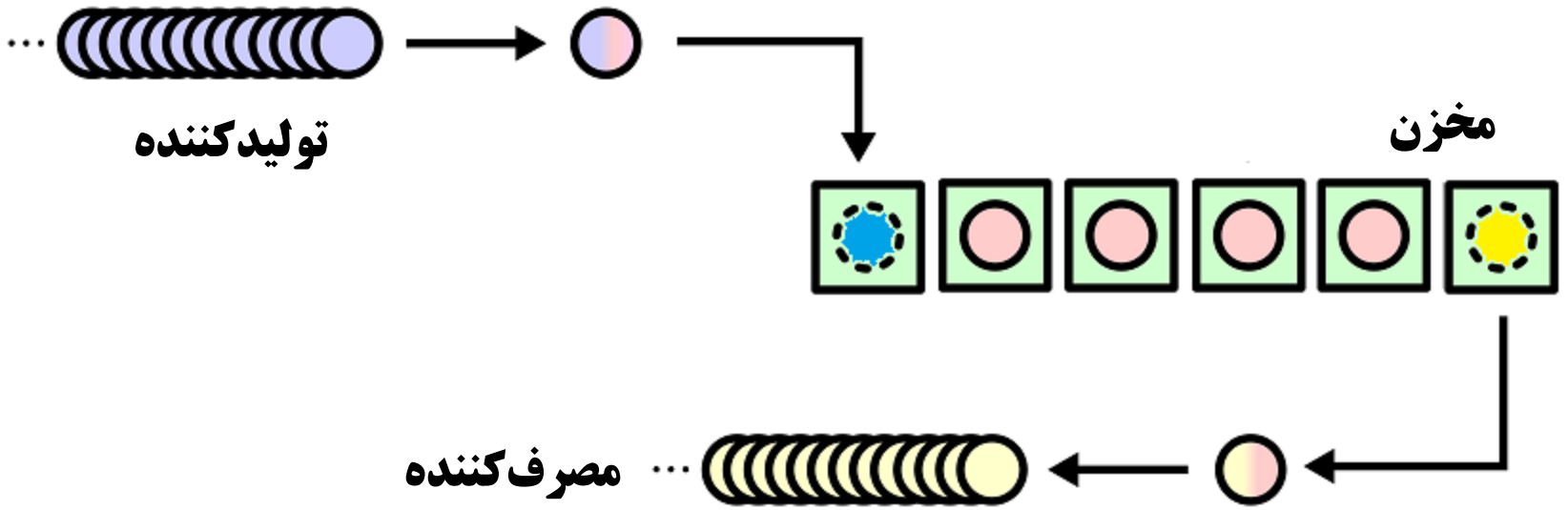
- یک یا چند نخ مشغول تولید داده هستند
- یک یا چند نخ مشغول خواندن داده‌ها هستند
- داده‌ها را در یک مخزن مشترک (مثلاً یک صف) قرار می‌دهند
- تعداد تولیدکننده‌ها، تعداد مصرف‌کننده‌ها و اندازه مخزن ممکن است محدود شود



چند نکته درباره مسأله تولیدکننده/مصرف‌کننده

- دو نخ مختلف همزمان نباید با مخزن کار کنند
  - اگر یکی مشغول خواندن یا نوشتن از مخزن است، نخ دیگری وارد نشود
- اگر مخزن خالی است، نخ مصرف‌کننده باید منتظر بماند تا یک تولیدکننده، داده تولید کند
- در صورتی که اندازه مخزن محدود است:

اگر مخزن پر است، نخ تولیدکننده باید منتظر بماند تا یک مصرف‌کننده، داده مصرف کند



تمرین

- مرور یک پیاده‌سازی اولیه برای پیاده‌سازی ProducerConsumer
- در حالتی که:
  - چند نخ تولیدکننده داریم
  - چند نخ مصرف‌کننده داریم
  - اندازه مخزن محدودیت ندارد (مخزن پر نمی‌شود)

جمع‌بندی

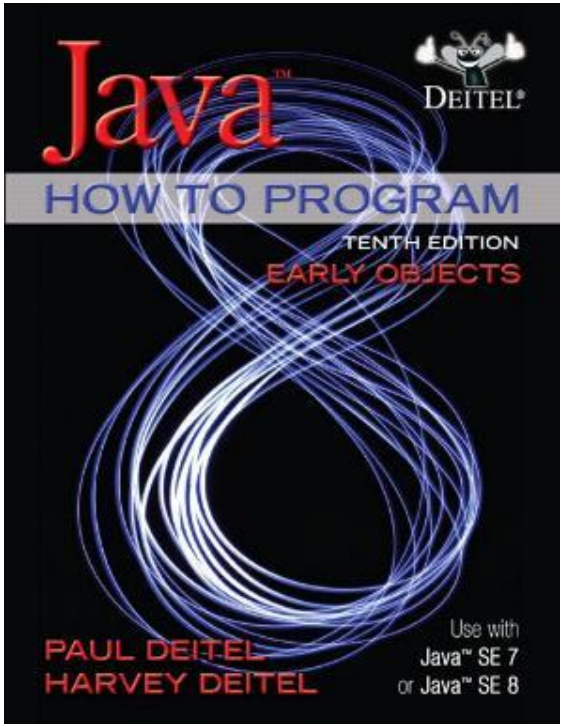
جمع‌بندی

- مفهوم نخ (Thread)
- برنامه‌نویسی چندنخی (Multi-thread)
- کنترل همزمانی (Synchronization)
- حالت‌های یک نخ (Thread State)
- متدهای wait و notify

- مفاهیم پیشرفته‌تر در همروندی: موضوع یک جلسه دیگر

مطالعه کنید

- فصل ۲۳ کتاب دایتل (Deitel & Deitel) Java How to Program



23 Concurrency 957

- تمرین‌های همین فصل از کتاب دایتل

تمرین

- یک متد چندنخی بنویسید که حجم کل فایل‌های یک شاخه را محاسبه کند
  - آدرس شاخه و تعداد نخ‌ها را به عنوان پارامتر بگیرد و حجم کل شاخه را برگرداند
- مسأله تولیدکننده/مصرف‌کننده را در حالتی پیاده‌سازی کنید
  که اندازه بافر (مخزن) هم محدود باشد
  - در این حالت اگر مخزن پر باشد و یک تولیدکننده بخواهد تولید کند، باید منتظر شود تا یک مصرف‌کننده، یک خانه مصرف کند
  - چند نخ تولیدکننده و چند نخ مصرف‌کننده ایجاد کنید
  - نخ‌های تولیدکننده یک عدد تصادفی به مخزن اضافه کنند
  - نخ‌های مصرف‌کننده هم یک عدد از مخزن بردارند و چاپ کنند

جستجو کنید و بخوانید

- امکانات سیستم‌عامل‌ها برای برنامه‌نویسی چندنخی
  - نحوه مدیریت و زمان‌بندی نخ‌ها توسط سیستم عامل
- مزایا و معایب برنامه‌نویسی چندنخی
  - در چه مواردی، چندنخی باعث افت کارایی برنامه می‌شود

پایان