Operating Systems
سیستمهای عامل

مجموعه اسلایدهای شماره ۶

دکتر خانمیرزا

h.khanmirza@kntu.ac.ir

دانشکده کامپیوتر

دانشگاه صنعتی خواجه نصیرالدین طوسی

فرآیندها و ریسمانهای همکار

- مستقل (independent)
  - هیچ اشتراک داده‌ای بین آنها نیست
  - نتایج اجرا با ورودیهای معین قطعی (deterministic) است
  - نتایج و تمامی شرایط اجرا تا رسیدن به نتایج قابل بازتولید (re-producible) است

- همکار (cooperative)
  - فرآیندها و یا ریسمانها دارای داده مشترک با سایرین هستند
  - نتایج اجرا با ورودیهای مشخص قطعی نیست
  - نتایج و برخی شرایط به سختی قابل بازتولید هستند. چرا که بسته به اینکه کدام ریسمان یا فرآیند زودتر اجرا شود نتایج متفاوت می‌شود

فرآیندها و ریسمانهای همکار

- چرا نیاز به همکاری وجود دارد؟

- اشتراک منابع
  - مصرف بهینه حافظه

- افزایش سرعت اجرا
  - اجرای موازی
  - همپوشانی IO و پردازش – زمانی که یکی در حال اجراست دیگری می‌تواند داده‌های مورد نیاز را از سخت‌افزار قبل از زمان استفاده لود کند

- ماجولار بودن
  - شکستن یک برنامه بزرگ به قطعات کوچکتر برای قابل توسعه بودن و راحتی تست و بروز رسانی

ارتباط بین فرآیندها

Inter Process Communication (IPC)

ارتباط بین فرآیندها

- فرآیندها در آدرسهای مجزای حافظه قرار دارند و به صورت سخت‌افزاری از دسترسی به بقیه آدرسهای حافظه منع می‌شوند
- با این اوصاف فرآیندهای همکاری چطور می‌توانند بین خود داده رد و بدل کنند

- روشهای ارتباط بین فرآیندها
  - سیگنال
  - حافظه مشترک
  - ارسال پیام

سیگنال (signal)

- یک روش با قابلیتهای محدود برای ارتباط بین فرآیندهاست
- سیگنال در اصل یک نوع مکانیزم ارسال اعلان (notification) به فرآیندها است.

- سیگنال بسته به **منشا سیگنال** و **دلیل آن** می‌تواند همگام و یا ناهمگام باشد.
  - سیگنال همگام همواره توسط یک رویداد داخل فرآیند تولید می‌شود مثلا تقسیم بر صفر و یا دسترسی به محل غیرمجاز حافظه.
  - عموماً خطاهای سخت‌افزاری سیستمی توسط خود کرنل رسیدگی می‌شوند
    - در برخی موارد کرنل اطلاعات کافی برای رسیدگی صحیح به خطا را دارد مثل page-fault
    - در برخی موارد دیگر کرنل اطلاعات کافی ندارد تا هوشمندانه مساله را رفع کند مثل تقسیم بر صفر و دسترسی غیرمجاز به حافظه که در این حالت با ارسال اعلان به فرآیند مربوطه رسیدگی به آن را به خودش محول می‌کند

- سیگنال غیرهمگام منشا خارجی دارد و در اثر یک رویداد خارج از فرآیند ایجاد می‌شود مثل دریافت اعلان Ctrl+C و یا اعلان KILL می‌باشد.

سیگنال (signal)

- زمانی که یک سیگنال تولید می‌شود سیستم عامل روند اجرای فرآیند را متوقف و تابع رسیدگی (handler) آن سیگنال را اجرا می‌کند.
- هر سیگنال عموما یک تابع رسیدگی پیش فرض دارد که توسط هسته سیستم عامل پیاده‌سازی شده است.
- تابع رسیدگی بیشتر سیگنالها را می‌توان در خود فرآیند بازنویسی کرد و اصطلاحا تابع handler پیش‌فرض را override کنند.
  - دوسیگنال SIGKILL و SIGSTOP تماما توسط هسته رسیدگی شده و قابل بازنویسی در فرآیند نیست

تفاوت با وقفه

- روش عملکرد سیگنالها و وقفه‌ها هستند بسیار شبیه بهم است اما ...

- وقفه‌ها توسط تجهیزات جانبی تولید و به پردازنده ارسال می‌شود و سپس توسط کرنل رسیدگی میشوند
- سیگنالها در پردازنده تولید شده و به هسته سیستم عامل ارسال می‌شود و سپس توسط فرآیندها رسیدگی میشوند.
  - اجرای سیگنال در زمینه همان فرآیند (یا ریسمان) و در حالت کاربری اجرا می‌شود مثل اینکه آن فرآیند یک فراخوانی سیستمی انجام داده باشد.
  - گاهی اوقات یک وقفه به شکل سیگنال به فرآیند تحویل داده می‌شود

انواع سیگنال

- سیگنالهای تعریف شده در استاندارد POSIX موارد متعددی هستند برای مثال
  - SIGCHLD
    - زمانی فرآیند فرزند تمام می‌شود به فرآیند والد اطلاع‌رسانی می‌شود
  - SIGFPE
    - در زمان تقسیم بر صفر و یا سرریز
  - SIGINT
    - زمانی که ترکیب Ctrl+C فراخوانی می‌شود
  - SIGKILL
    - برای اتمام ناگهانی فرآیند به فرآیند ارسال می‌شود و باعث اتمام و بسته‌شدن فرآیند می‌شود
  - SIGTERM
    - همانند SIGKILL است اما می‌توان آنرا در فرآیند handle کرد.
  - SIGSTOP
    - به سیستم عامل اعلام میکند که فرآیند را تا زمان بیدار کردن بعدی متوقف کند
  - SIGUSR1, SIGUSR2
    - سیگنالهای قابل تعریف شدن توسط کاربر که در برخی موارد می‌توان یک رشته را همراه با سیگنال ارسال کرد (در برخی سیستم عاملها)

ارسال سیگنال

- برای ارسال از shell از دستورات kill و killall استفاده می‌شود

- `kill 1200 157`
  - ارسال سیگنال TERM به صورت پیش‌فرض به فرآیندهای با شناسه 1200 و 157

- `kill -s KILL 507`
  - ارسال سیگنال KILL که سیگنال غیر قابل پوشاندن (mask) است به فرآیند با شناسه 507

- `killall SIGUSR1 my-process`
  - ارسال سیگنال قابل تعریف توسط کاربر به فرآیندی با نام my-process

ارسال سیگنال

- برای ارسال سیگنال از برنامه از توابع زیر استفاده می‌شود

- `kill(pid t pid, int signal)`

- `pthread_kill(pthread t tid, int signal)`

بازنویسی رسیدگی به سیگنال

```c
#include <stdlib.h>
#include <stdio.h>

#include <signal.h> /*for signal() and raise()*/

void hello(int signum){
  printf("Hello World!\n");
}

int main(){

  //execute hello() when receiving signal SIGUSR1
  signal(SIGUSR1, hello);

  //send SIGUSR1 to the calling process
  raise(SIGUSR1);
}
```

```
#> ./hello_signal
Hello World!
```

https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html

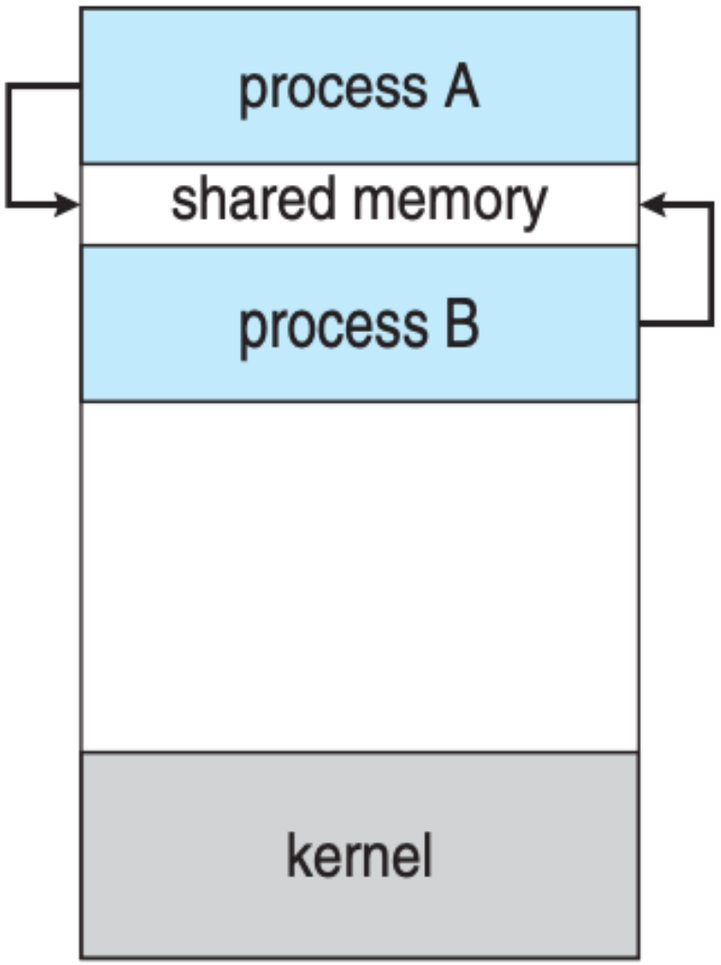
سیگنال و چند ریسمانی

- اگر یک فرآیند دارای چند ریسمان باشد سیگنال باید به کدامیک ارسال شود؟
- برخی سیگنالها مخصوصا سیگنالهای همگامها به همان ریسمانی که باعث سیگنال شده ارسال می‌شود
- برخی سیگنال‌ها به تمامی ریسمانها ارسال می‌شود مثل سیگنال SIGKILL
- با بازنویسی توابع می‌توان مشخص کرد که چه ریسمانهایی چه سیگنالی را دریافت کنند
- یک ریسمان برای دریافت تمامی سیگنالها مشخص می‌شود

سیگنال و ارتباط بین فرآیندها

- سیگنالها از جهت مصرف حافظه و منابع پردازشی بسیار بهینه و کم مصرف هستند
-
- اما عموما با سیگنالها نمی‌توان داده‌ای ارسال کرد و صرفا می‌توان یک اعلان به فرآیند دیگر ارسال کرد.
- ترتیب دریافت سیگنالها توسط یک فرآیند مشخص نیست و ضمانتی در رعایت ترتیب وجود ندارد.

حافظه مشترک (shared memory)



- می‌توان بخشهایی از حافظه را بین دو فرآیند به اشتراک گذاشت.
- بدلیل وجود واحد مدیریت حافظه (MMU) می‌توان به نحوی عمل کنیم که در ترجمه آدرسها آدرس‌های مجازی تولید شده در دو فرآیند به یک محل فیزیکی نگاشته شود.
- با این روش هر دو فرآیند می‌توانند همانند خواندن و نوشتن یک متغیر از آن محل استفاده کنند.
- این روش برای انتقال داده‌های بزرگ مناسب بوده و سریعترین روش انتقال اطلاعات بین دو فرآیند است
  - در این روش باید مکانیزمهایی برای نوشتن صحیح و خواندن اطلاعات بین دو فرآیند بکار گرفت.
- در این روش فقط زمانی که حافظه مشترک ایجاد می‌شود نیاز به فراخوانی سیستمی هست و پس از ایجاد تمامی کارها در حالت کاربری انجام میشود

حافظه مشترک (shared memory)

- بخش نویسنده

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

حافظه مشترک (ed memory

- بخش گیرنده

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

حافظه مشترک (d memory

```java
public class MemoryMappedReader {
    public static void main(String[] args) {
        final File file = new File( pathname: "test_mem_map");

        if (!file.exists()) {
            System.out.println("No test_mem_map file found");
            System.exit( status: 1);
        }

        try {
            final Path path = file.toPath();
            readFromFile(path);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void readFromFile(Path path) {
        try {
            MappedByteBuffer mbb;
            try (FileChannel fc = (FileChannel.open(path, StandardOpenOption.READ))) {

                long position = 0;
                long size = 1024;

                mbb = fc.map(FileChannel.MapMode.READ_ONLY, position, size);

                for (int i = 0; i < 10; i++) {
                    double value = mbb.getDouble();
                    System.out.println("value read from buffer = " + value);
                    Thread.sleep( millis: 1200);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

حافظه مشترک

```java
public class MemoryMappedWriter {
    public static void main(String[] args) {
        final File file = new File( pathname: "test_mem_map");

        if (file.exists()) file.delete();

        try {
            file.createNewFile();
            final Path path = file.toPath();
            writeToFile(path);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void writeToFile(Path path) {
        try {
            MappedByteBuffer mbb;
            try (FileChannel fc = (FileChannel.open(path, StandardOpenOption.WRITE, StandardOpenOption.READ))) {
                long position = 0;
                long size = 1024;

                mbb = fc.map(FileChannel.MapMode.READ_WRITE, position, size);

                double value = 4.44;
                for (int i = 0; i < 10; i++) {
                    System.out.println("value written to shared buffer = " + value);
                    mbb.putDouble(value);
                    mbb.force();
                    value *= 2;
                    Thread.sleep( millis: 1000);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

ارسال پیام (message passing)

- در این روش فرآیندها با ارسال پیام به همدیگر داده‌ها را با هم به اشتراک می‌گذارند
- طبیعی است که در این روش داده‌ها در حافظه دو فرآیند به شکل تکراری وجود دارد
- در این روش عموما دو دسته تابع وجود دارد یک دسته برای ارسال و یک دسته برای دریافت
- `send(msg)`
- `recv(msg)`

- هر دو تابع فراخوانی سیستمی هستند. بنابراین برای ارسال و دریافت هر پیام نیاز به رفتن به حالت هسته وجود دارد.
- ارسال و دریافت پیام نیازمند یک بستر ارتباطی است.
- بسترهای شناخته شده
  - انتقال از طریق مکانیزمهای شبکه (Network)
  - پایپ (pipe)

ارسال پیام (message passing)

- پیاده‌سازی ارسال پیام می‌تواند مستقیم (direct) و یا غیر مستقیم (indirect) باشد

  - در حالت مستقیم ارسال کننده پیام باید شناسه گیرنده را در اختیار داشته باشد
  - ارتباط می‌تواند یکطرفه و یا دوطرفه باشد

  - در حالت غیرمستقیم یک واسط بین ارسال کننده و دریافت کننده پیام وجود دارد
  - این موجود واسط با نام mailbox شناخته می‌شود

  - ارسال کننده پیام را با مشخصه‌ای از دریافت کننده به mailbox ارسال می‌کند.
  - سپس mailbox پیام را در زمان مقتضی به دریافت کننده ارسال می‌کند (push) و یا ممکن است دریافت کننده خودش به صورت منظم از mailbox پیامهای خود را طلب کند (pull).
  - در این حالت پیاده‌سازی ارسال دسته‌ای (ارسال به چندین گیرنده) بسیار ساده است.

ارسال پیام (message passing)

- پیاده‌سازی ارسال پیام می‌تواند همگام (synchronous) و یا ناهمگام (asynchronous) باشد

- در ارسال همگام، ریسمان‌های اجرایی در توابع ارسال و دریافت بلاک می‌شوند تا عمل ارسال و یا دریافت بطور کامل انجام شود

- در ارسال ناهمگام ریسمان تابع را فراخوانی کرده و باقی کارها را به سیستم عامل می‌سپرد. ممکن است یک تابع callback هم معرفی شود تا پس از ارسال کامل سیستم عامل این تابع را فراخوانی کند

```
{
    ...
    send_asyc(msg, send_completed)
    printf(“send done!”);
}
void send_completed(){
    printf(“sending message completed”);
}
```

- در دریافت ناهمگام ریسمان در سیستم عامل برای دریافت پیام ثبت نام کرده و یک تابع callback معرفی می‌کند تا در هنگام دریافت بسته متناسب این تابع فراخوانی شده و پیام به ریسمان تحویل شود.

ارسال پیام (essage passing

- مثال پایپ

```c
#define BUFFER SIZE 25
#define READ END 0
#define WRITE END 1
int main(void){
  char write msg[BUFFER SIZE] = "Greetings";
  char read msg[BUFFER SIZE];
  int fd[2];
  pid t pid;
  /* create the pipe */
  if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
  }
  pid = fork();
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  if (pid > 0) { /* parent process */
    close(fd[READ END]);  /* close the unused end of the pipe */
    write(fd[WRITE END], write msg,   strlen(write msg)+1);
    close(fd[WRITE END]); /* close the write end of the pipe */
  }
  else { /* child process */
    close(fd[WRITE END]); /* close the unused end of the pipe */
    read(fd[READ END], read msg, BUFFER SIZE);
    printf("read %s",read msg);
    close(fd[READ END]);  /* close the write end of the pipe */
  }
  return 0;
}
```

ارسال پیام (message passing)

- در shell هم از پایپ استفاده می‌شود. برای این کار از '|' استفاده می‌شود

```
FreshAir:www hamed$ cat index.html | tail -n 10
                    <button id="btnImportCustomBlockedZones" type="submit" class="btn btn-primary
" data-loading-text="Importing..." onclick="return importCustomBlockedZones();">Import</button>
                    <button type="button" class="btn btn-default" data-dismiss="modal">Close</but
ton>
                </div>
            </div>
        </div>
    </div>

    <div id="footer"></div>
</body>
</html>



FreshAir:www hamed$ cat index.html | tail -n 10 | grep -i footer
    <div id="footer"></div>
```

همگام‌سازی ریسمانها

Thread Synchronization

همگام‌سازی ریسمانها

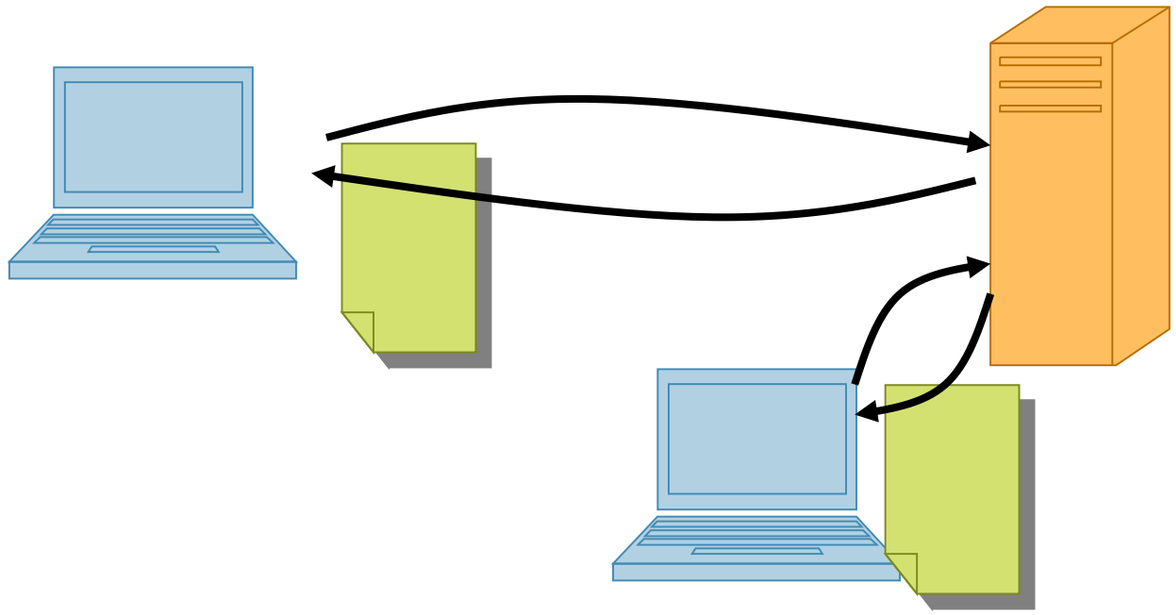
- همانطور که مشاهده شد فرآیندها بدلیل داشتن حفاظت و فضای آدرس جداگانه براحتی نمی‌توانند داده‌ای را به اشتراک بگذارند مگر آنکه به شکل مشخص با استفاده از مکانیزمهای معرفی شده به تبادل داده بپردازند.

- ریسمانها بطور ذاتی محافظت ندارند و براحتی می‌توانند داده‌های موجود در برخی بخشهای فرآیند را به صورت مشترک استفاده کنند.

- اما در استفاده مشترک از داده‌ها مشکلاتی پیش می‌آید ...

مثال وب سرور

- یک سرور وب را در نظر بگیرید ... در این سرور در لحظه ممکن است چندین درخواست به سرور انجام گیرد



- اگر درخواستها یکی یکی اجرا شود بدیهی است که سرور فقط تعداد محدودی از درخواستها را می‌تواند پردازش کند
- مثلا زمانی که یک ریسمان مشغول IO است نمی‌توان ریسمان دیگر را که نیاز به پردازش دارد اجرا کرد

مثال وب سرور

- مدل پیاده‌سازی مستقل می‌تواند چیزی شبیه به این شبه کد باشد

```
serverLoop() {
      con = AcceptConnection();

      ProcessFork(ServiceWebPage(),con);
  }
```

- در این پیاده‌سازی چه مشکلاتی وجود دارد؟
  - هزینه ایجاد فرآیند برای هر درخواست
  - عدم اشتراک داده‌های مهمی نظیر cache
  - مصرف حافظه بیشتر

مثال وب سرور

- مدل پیاده‌سازی مستقل می‌تواند چیزی شبیه به این شبه کد باشد

```
serverLoop() {
      con = AcceptConnection();

      ProcessFork(ServiceWebPage(),con);
  }
```

- در این پیاده‌سازی چه مشکلاتی وجود دارد؟
  - هزینه ایجاد فرآیند برای هر درخواست
  - عدم اشتراک داده‌های مهمی نظیر cache
  - مصرف حافظه بیشتر

مثال وب سرور

- مدل پیاده‌سازی می‌تواند چند ریسمانی باشد

```
serverLoop() {
      con = AcceptConnection();

      ThreadFork(ServiceWebPage(),con);
  }
```

- در این پیاده‌سازی
  - هزینه ایجاد ریسمان برای پاسخ‌دهی به هر درخواست کاهش یافته است
  - قابلیت اشتراک داده‌های مهمی نظیر cache
  - مصرف حافظه کمتر

مثال وب سرور

- مشکل پیاده‌سازی چند ریسمانی ایجاد تعداد بی‌نهایت ریسمان است.
  - هر چقدر تعداد درخواستها افزایش یابد برای هر درخواست یک ریسمان ایجاد می‌شود.
  - با این کار منابع سیستم کاملا پر شده و ممکن است سرور از کار بیفتد
- برای رفع مشکل می‌توان از استخر ریسمان استفاده کرد تا درجه موازی‌سازی محدود باشد

```
master() {
   allocThreads(worker,queue);

   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```

مثال وب سرور

- اگر این سرور مربوط به یک بانک باشد می‌توان توابعی نظیر کدهای زیر برای این سرور در نظر گرفت

```
Deposit(acctId, amount) {
    acct = GetAccount(acctId);      /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct);             /* Involves disk I/O */
}

Withdraw(acctId, amount) {
    acct = GetAccount(acctId);      /* may use disk I/O */
    acct->balance -= amount;
    StoreAccount(acct);             /* Involves disk I/O */
}
```

مثال وب سرور

- باید دقت کنیم که این کد سطح بالاست و این برنامه کامپایل شده و به دستورات اسمبلی و زبان ماشین تبدیل می‌شود.
- فرض کنید که ترجمه زیر اتفاق افتاده باشد

```
Deposit(acctId, amount) {
    acct = GetAccount(acctId);      /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct);             /* Involves disk I/O */
  }

_deposite:
  ……
  load r1, acct->balance
  add r1, amount1
  store r1, acct->balance
   ……
```

مثال وب سرور

- میدانیم که در اجرای ریسمانها ممکن است در هر لحظه پردازنده از آنها گرفته شده و به ریسمان دیگر داده شود.
- اگر دو ریسمان بطور همزمان در حال اجرای تابع deposit بر روی یک حساب باشند چه اتفاقی میفتد؟

Thread 1:

```
//r1 = 1000
load r1, acct->balance

// r1 = 1000 + 100
add r1, amount1
// account = 1100!!
store r1, acct->balance
```

Thread 2:

```
load r1, acct->balance //r1=1000
add r1, amount1        //r1=1500
store r1, acct->balance
```

مثال وب سرور

- دلایل بروز چنین تداخلی عبارتند از
  - مکانیزم اجرایی همزمانی
  - مشترک بودن داده بین ریسمانها و دسترسی بدون هماهنگی به آنها
  - ترجمه شدن هر خط برنامه در زبانهای سطح بالا به چندین دستور اسمبلی و یا زبان ماشین

Thread 1:

```
//r1 = 1000
load r1, acct->balance

// r1 = 1000 + 100
add r1, amount1
// account = 1100!!
store r1, acct->balance
```

Thread 2:

```
load r1, acct->balance //r1=1000
add r1, amount1        //r1=1500
store r1, acct->balance
```

مثال دیگری از تداخل اجرای ریسمانها

- در کد زیر اگر مقدار اولیه y=12 باشد x, y در انتهای اجرا چه مقادیری ممکن است داشته باشند

Thread A	Thread
x = 1;	y = 2;
x = y+1;	y = y*2;

thA: run completely, x=1 y=12 → x=13

thA: x=1 → thB: y=2, y=4 → thA: x= 4+1 = 5

thA: x=1 → thB: y=2 → thA: x= 2+1 = 3

X can be one of {13, 5, 3}

مثال دیگری از تداخل اجرای ریسمانها

- در کد زیر کدام ریسمان در نهایت برنده خواهد شد؟
- آیا ممکن است اجرای این دو ریسمان هیچگاه تمام نشود؟

int i=0;

Thread A

```
while (i < 10)
    i = i + 1;
printf("A wins!");
```

Thread B

```
while (i > -10)
        i = i – 1;
printf("B wins!");
```

عملیات تجزیه‌ناپذیر (atmoic)

- برای داشتن درک صحیح از نحوه‌ی اجرای کدهای همزمان باید بدانیم که در پردازنده چه دستوراتی به شکل تجزیه‌ناپذیر اجرا می‌شوند

- عملیات تجزیه‌ناپذیر (atmoic operation)
- عملیاتی است که یا تا کامل شدن اجرا می‌شود و یا اینکه اصلا اجرا نمی‌شود

- تجزیه‌ناپذیر بودن شامل یکی از دو شرط زیر است:
  - اجرای عملیات قابل تقسیم به چند مرحله اجرا نباشد
  - وضعیت اجرای دستور توسط هیچ موجود دیگری در وسط عملیات قابل تغییر نباشد

- اگر دستورات تجزیه‌ناپذیر در پردازنده‌ها نباشد نمی‌توان بین ریسمانهای همکار هماهنگی ایجاد کرد
  - فرض بر این است که دستورات خواندن و نوشتن در حافظه به شکل تجزیه‌ناپذیر اجرا می‌شود

مساله شیر زیادی

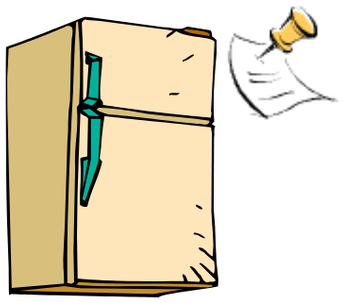
- دو دانشجو در یک خوابگاه مشترک زندگی می‌کنند.
- بعد از اتمام کلاس هر کس زودتر به خانه برگشت اگر در یخچال شیر نبود به سوپری سر کوچه رفته و یک شیشه شیر می‌خرد
- اما ممکن است زمانی که نفر اول برای خرید شیر رفته باشد، نفر دوم سر برسد و او هم برای خرید برود. با این کار دو شیشه شیر خریداری خواهد شد که اضافی است

- شرایط صحت پاسخ الگوریتم «شیر زیادی»
  - اگر لازم است شیر خریداری شود
  - فقط یک نفر باید شیر بخرد

- نکته: لازم است در این نوع مسائل حتما قبل از نوشتن کد در مورد راه‌حل فکر کنیم، چون debug کردن این نوع مسائل بسیار سخت‌تر است

مساله شیر زیادی

- راه‌حل اول: از یک یادداشت استفاده شود
  - قبل از رفتن برای خرید یک یادداشت گذاشته می‌شود
  - بعد از برگشتن یادداشت را برمی‌دارد
  - اگر یک نفر یادداشت را دید برای خرید نمی‌رود



```
if (noMilk) {
   if (noNote) {
         leave Note;
         buy milk;
         remove note;
   }
}
```

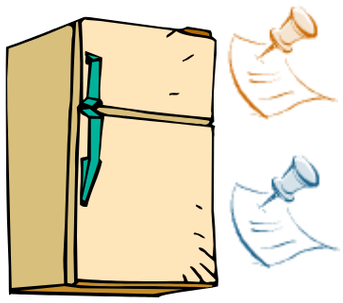
مساله شیر زیادی

- راه‌حل اول: از یک یادداشت استفاده شود
- اگر بعد از بررسی شیر و یادداشت پردازنده گرفته شود باز هم شیر زیادی خریداری خواهد شد

```
THREAD-A
if (noMilk) {
      if (noNote) {
            leave Note;
            buy milk;
            remove note;
      }
}
```

```
THREAD-B

if (noMilk) {
      if (noNote) {
            leave Note;
            buy milk;
            remove note;
      }
}
```

مساله شیر زیادی



- راه‌حل دوم: استفاده از یادداشت برچسب‌دار
  - هر شخص قبل از رفتن برای خرید یک یادداشت با نام خودش می‌گذارد
  - بعد از برگشتن یادداشت را برمی‌دارد
  - اگر یک نفر یادداشت را دید برای خرید نمی‌رود

```
Thread A                    Thread B
leave note A;               leave note B;
if (noNote B) {             if (noNoteA) {
   if (noMilk) {               if (noMilk) {
      buy Milk;                   buy Milk;
   }                           }
}                           }
remove note A;              remove note B;
```

مساله شیر زیادی

- راه‌حل سوم:
- در این راه‌حل یکی از طرفین با دیدن یادداشت هم اتاقی‌اش بجای تصمیم سریع منتظر می‌ماند تا ببیند که چه اتفاقی خواهد افتاد
- این راه‌حل بر اساس شرایط تعیین شده درست کار می‌کند

Thread A

```
leave note A;
while (note B);
if (noMilk)
        buy milk;

remove note A;
```

Thread B

```
leave note B;
if (no note A)
        if (noMilk)
                buy milk;

remove note B
```

مساله شیر زیادی

- راه‌حل سوم دارای معایبی است
  - پیاده‌سازی نامتقارن است.
    - اگر تعداد ریسمانها (هم اتاقی‌ها) بیشتر شود پیاده‌سازی به چه نحوی باید باشد
  - این راه‌حل دارای صبر بی‌خود (Busy waiting) است
    - یکی از ریسمانها زمانی که پردازنده را می‌گیرد هیچ کاری نمی‌کند

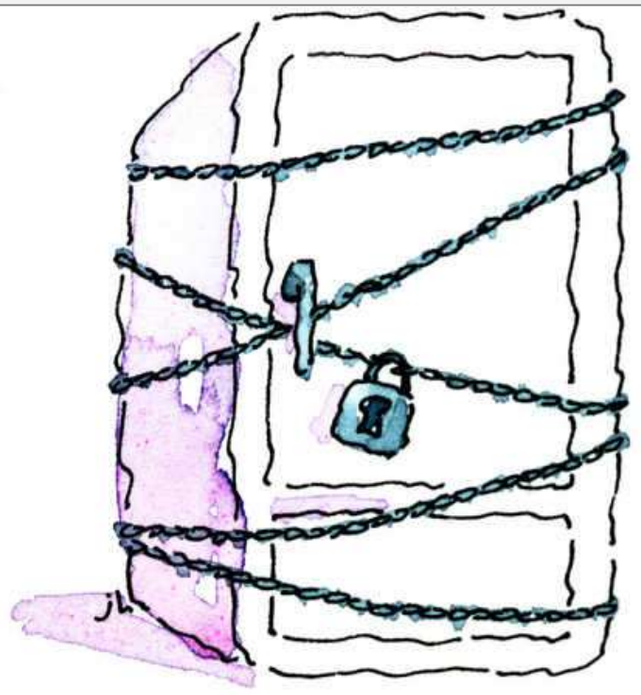
Thread A

```
leave note A;
while (note B);
if (noMilk)
        buy milk;

remove note A;
```

Thread B

```
leave note B;
if (no note A)
        if (noMilk)
                buy milk;

remove note B
```

مساله شیر زیادی



- راه‌حل چهارم: استفاده از قفل
- شخص قبل از بررسی یخچال قفل یخچال را برداشته و بعد وجود شیر را بررسی می‌کند و اگر شیر نبود آنرا می‌خرد
- نفر بعدی برای بررسی یخچال باید قفل را داشته باشد

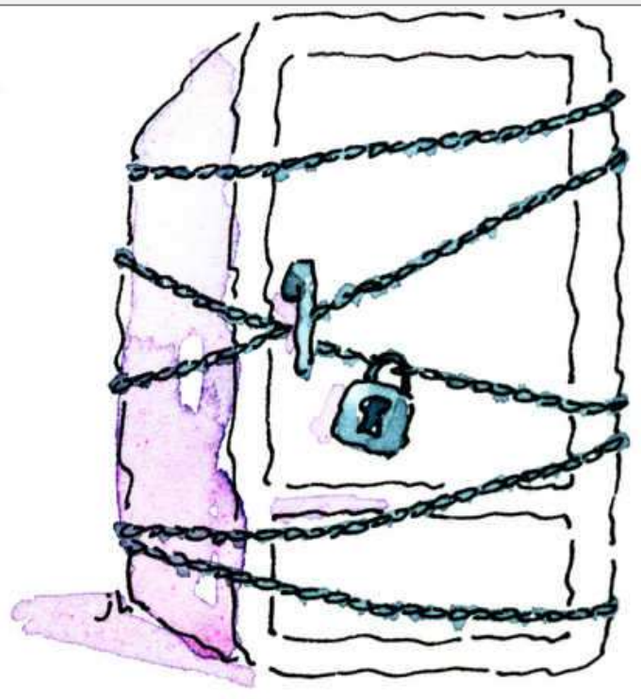
Thread A

```
milklock.Acquire();
  if (nomilk)
       buy milk;
milklock.Release();
```

Thread B

```
milklock.Acquire();
  if (nomilk)
       buy milk;
milklock.Release();
```

مفهوم قفل



- در واقع اگر در سیستم عامل یک مفهومی مانند قفل وجود داشت پیاده‌سازی بسیار راحت و قابل فهم بود
- در این راه‌حل بخش حساس پیاده‌سازی که موجب مشکل می‌شود توسط یک نفر انجام می‌شود
- در این مدت هیچ کس دیگری نمی‌تواند آن کار را انجام دهد و یا اجرای شخص دیگر را دچار خدشه کند
- نفرات بعدی تا زمان انجام کار توسط نفر اول باید صبر کنند

Thread A

```
milklock.Acquire();
  if (nomilk)
       buy milk;
milklock.Release();
```

- در واقع مفهوم قفل به ما کمک می‌کند تا کارهایی را که به صورت تجزیه‌ناپذیر (atomic) اجرا نمی‌شوند به صورت تجزیه‌ناپذیر اجرا شوند

تعاریف و اصطلاحات همگام سازی

- همگام سازی (synchroniztion):
- با استفاده از عملیات اتمیک همکاری را بین ریسمانها ممکن می‌کند

- طرد متقابل – انحصار متقابل (Mutual Exclusion):
- اطمینان از اینکه فقط یک ریسمان در یک زمان یک کار خاص را انجام میدهد.
- در حقیقت یک ریسمان در حال اجرا ریسمان دیگر را از اجرای یک بخش طرد میکند

- ناحیه بحرانی (Critical Section):
- قطعه‌ای از کد که در هر زمان فقط یک ریسمان می‌تواند آن را اجرا کند.
- این ناحیه در اثر طرد متقابل ایجاد می‌شود

تعاریف و اصطلاحات همگام سازی

- قفل (lock):
- ممانعت از انجام یک کار توسط یک ریسمان
- یک ریسمان با بدست آوردن قفل، انحصار متقابل ایجاد می‌کند
- برای پیاده‌سازی ناحیه بحرانی می‌توان ریسمانها را وادار کرد در ابتدای ناحیه قفل مربوط به ناحیه بحرانی را بدست آورده و در انتهای ناحیه قفل را آزاد کرد

```
lock.acquire();
  // critical section
lock.release();
```

- بقیه ریسمانها برای ورود به ناحیه بحرانی باید صبر کنند تا قفل مربوط به آن ناحیه آزاد شود

قفل

- شرایط یک قفل که به صورت صحیح پیاده‌سازی شده است

- انحصار متقابل
  - در هر لحظه فقط یک ریسمان در حال اجرای ناحیه بحرانی باشد

- پیشرفت (Progress)
  - اگر هیچ ریسمان دیگری منتظر ناحیه بحرانی نیست، ریسمان درخواست دهنده بدون صبر کردن (یا با صبر محدود) وارد ناحیه بحرانی شود

- صبر محدود (Bounded Waiting)
  - در صورتی که برای ناحیه بحرانی بین ریسمانها رقابت وجود دارد همه ریسمانهای درخواست دهنده بالاخره وارد ناحیه بحرانی شوند
  - هیچ ریسمانی نباشد که باندازه نامحدود صبر کند

پیاده‌سازی‌های قفل

Lock Implementations

پیاده‌سازی‌های قفل

- پیاده‌سازی ۱ – با استفاده از وقفه‌ها
- ریسمان در حال اجرا چه زمانی پردازنده را از دست می‌دهد؟ بعبارت دیگر چه‌چیزی باعث می‌شود که در یک سیستم تک‌پردازنده ریسمان کنترل اجرا را از دست داده و یک قطعه کد به صورت اتمیک اجرا نشود؟
  - قبلا صحبت شده که IO، وقفه و دستور yield پردازنده را از ریسمان می‌گیرد
- برای اینکه یک ریسمان بخشی از کد را به شکل اتمیک اجرا کند کافی است که پردازنده در طول اجرای ناحیه بحرانی در اختیار همان ریسمان بماند!
- اگر در ناحیه بحرانی وقفه‌ها غیر فعال شود و ریسمان کاری IO انجام ندهد عملا هیچ رویدادی نیست که پردازنده را از یک ریسمان بگیرد

پیاده‌سازی‌های قفل

- پیاده‌سازی ۱ – با استفاده از وقفه‌ها

```
// using locks
lock_acquire();

do_critical_section();

lock_release();
```

```
// lock implementation
lock_acquire() {
        disable_interrupts();
}

lock_release() {
        enable_interrupts();
}
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۱ – با استفاده از وقفه‌ها
- این پیاده‌سازی دارای مشکلاتی جدی است
  - اگر کاربر پیاده‌سازی داشته باشد که ناحیه بحرانی طولانی شود و یا بی‌نهایت باشد، تمام سیستم از کار میفتد چون وقفه‌ها کار نمی‌کنند
  - با غیرفعال شدن وقفه‌ها عملاْ هیچ تجهیزی نمی‌توانند کار کند و ممکن برخی داده‌ها از دست برود

```
// using locks
lock_acquire();

while(true);

lock_release();
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۲ –استفاده از وقفه‌ها و یک متغیر
- برای رفع مشکلات پیاده‌سازی قبلی بهتر است که ناحیه بحرانی کوچک و در کنترل سیستم عامل باشد

```
int lock_var = FREE;

lock_acquire() {
        disable_interrupts();
        if (lock_var == BUSY) {
                put thread on wait queue;
                go to sleep;
        }
        else {
                lock_var = BUSY;
        }
        enable_interrupts();
}
```

```
lock_release() {
        disable_interrupts();
        if (anyone on wait queue) {
                take thread off wait queue
                place on ready queue;
        }
        else {
                lock_var = FREE;
        }
        enable_interrupts();
}
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۲ – استفاده از وقفه‌ها و یک متغیر
- در اینجا مقدار متغیر در یک ناحیه بحرانی بررسی و تغییر می‌یابد چراکه مقدار آن توسط چند ریسمان مورد دسترسی قرار می‌گیرد
- خود این تابع‌ها برای پیاده‌سازی قفل بکار می‌روند.
- ناحیه بحرانی برنامه‌نویسان می‌تواند طولانی باشد و این در رفتار کلی سیستم تاثیری ندارد

```
int lock_var = FREE;

lock_acquire() {
        disable_interrupts();
        if (lock_var == BUSY) {
                put thread on wait queue;
                go to sleep;
        }
        else {
                lock_var = BUSY;
        }
        enable_interrupts();
}
```

```
lock_release() {
        disable_interrupts();
        if (anyone on wait queue) {
                take thread off wait queue
                place on ready queue;
        }
        else {
                lock_var = FREE;
        }
        enable_interrupts();
}
```

Critical Sections

پیاده‌سازی‌های قفل

- پیاده‌سازی ۲ – استفاده از وقفه‌ها و یک متغیر
- مشکلات:
  - بهر حال وقفه‌ها برای مدتی باید غیرفعال شوند و برخی کارها با تاخیر مواجه می‌شوند
    - دقت شود که این کار باید برای همه نواحی بحرانی تمامی فرآیندها اجرا شود
  - در سیستم‌های چند پردازنده باید وقفه‌ها در تمامی پردازنده‌ها غیرفعال شوند که هزینه ارسال پیام به تمام پردازنده‌ها برای توقف وقفه‌ها را در بردارد

پیاده‌سازی‌های قفل

- چطور می‌توانیم بدون استفاده از وقفه‌ها مقدار متغیر مرتبط با قفل را تغییر دهیم؟
- در پیاده‌سازی ۲ قسمتی که باید در بین فعال‌سازی و غیرفعال سازی وقفه‌ها قرار گیرد این بخش است:

```
if (lock_var == FREE)
      lock_var = BUSY
```

- اگر بتوانیم چرخه read-modify-write متغیر lock_var را بدون کمک وقفه‌ها به صورت اتمیک اجرا کنیم بطور کلی از معایب غیرفعال‌سازی وقفه‌ها خلاص می‌شویم
  - در واقع مشکل اصلی اینجاست که خواندن مقدار یک متغیر و بعد بررسی شرط و نوشتن متغیر اگر به شکل تجزیه‌ناپذیر انجام نشود مشکل مساله‌ی شیر زیادی بروز می‌کند

- راه‌حل داشتن دستورات سخت‌افزاری read-modify-write است که در سخت‌افزار و نه با کمک نرم‌افزار به شکل اتمیک کار کند

پیاده‌سازی‌های قفل

- **شبه کد** برخی از پرکاربردترین دستورات سخت‌افزاری که در پردازنده‌های مختلف پیاده‌سازی شده است.

```
testAndset (&address) {
      result = M[address];
      M[address] = 1;
      return result;
}

swap (&address, value) {
      temp = M[address];
      M[address] = value;
      return temp;
}

compareAndSwap (&address, expected_value, new_value) {
      if (M[address] == expected_value) {
         M[address] = new_value;
         return success;
      } else {
         return failure;
      }
}
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۳- با دستورات اتمیک سخت‌افزاری
- دستور testAndSet مقدار یک خانه حافظه را 1 کرده و مقدار قبلی را برمی‌گرداند و تمام این فرآیند به شکل یک دستور تجزیه‌ناپذیر یا همان اتمیک انجام می‌گیرد

- یک پیاده‌سازی ساده می‌تواند به شکل مقابل باشد:
- تا زمانی که متغیر 1 است یعنی قفل در اختیار یک ریسمان دیگر است دستور 1 برمی‌گرداند.
- زمانی که آزاد شد مقدار value=1 شده و مقدار 0 برگردانده می‌شود که نشان می‌دهد قفل آزاد بوده است

```
int value = 0; // 0:Free, 1:Busy

lock_acquire() {
   //busy waiting
   while (test&set(value));
}

lock_release() {
   value = 0;
}
```

معادل

```
int value = 0; // 0:Free, 1:Busy

lock_acquire() {
   while(value == 1);
   value = 0;
}

lock_release() {
   value = 0;
}
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۳ - با دستورات اتمیک سخت‌افزاری
- دستور testAndSet ساده‌ترین دستورات اتمیک است و براحتی با سایر دستورات هم می‌توان این نوع قفل را پیاده‌سازی کرد
- در این پیاده‌سازی وقفه‌ها فعال هستند و براحتی بر روی چند سیستمهای چند پردازنده کار می‌کند

```
int value = 0; // 0:Free, 1:Busy

lock_acquire() {
   while (swap (value, 1));
}

lock_release() {
   value = 0;
}
```

```
int value = 0; // 0:Free, 1:Busy

lock_acquire() {
   while (!cas(value, 0, 1));
}

lock_release() {
   value = 0;
}
```

پیاده‌سازی‌های قفل

- پیاده‌سازی ۳ - با دستورات اتمیک سخت‌افزاری
- مشکل اصلی پیاده‌سازی ۳ داشتن حلقه انتظار (busy waiting) است.
- در سیستمهای چند پردازنده و یا چند هسته‌ای نیز مشکل دارد.
  - دستورات اتمیک سخت‌افزاری دستورات نوشتن هستند و نه خواندن صرف.
  - نوشتن مرتب در یک متغیر باعث میشود که نتوان برای متغیر قفل از cache استفاده کرد و پردازنده‌ها مقدار این متغیر را دائما باید از حافظه اصلی بخوانند که در برابر سرعت cache بسیار زیاد است

پیاده‌سازی‌های قفل

- پیاده‌سازی ۴- قفلهای spin-lock
- در برخی موارد که طول ناحیه بحرانی کوتاه است busy waiting برای گرفتن قفل می‌تواند مفید باشد چرا که ریسمان گرفتار سربار تعوض زمینه و یا زمان‌بندی مجدد نمی‌شود (cache ریسمان دست نخورده می‌ماند)
- در این کد اول انتظار از نوع فقط خواندنی است در نتیجه می‌توان از مقدار متغیر در cache استفاده کرد.
- بعد که اطمینان از 0 بودن متغیر حاصل شد ریسمانها وارد رقابت برای گرفتن قفل به صورت اتمیک می‌شوند

```
int mylock = 0; // Free

spinlock_acquire() {
    do {
        while(mylock);  // Wait until might be free

    } while(test&set(&mylock)); // exit if get lock
}

spinlock_release() {
    mylock = 0;
}
```

- در هسته سیستم عاملها از این قفل زیاد استفاده شده است.

پیاده‌سازی‌های قفل

- پیاده‌سازی ۵ – با دستورات اتمیک سخت‌افزاری و یک متغیر
- در این پیاده‌سازی دو متغیر در نظر گرفته می‌شود. متغیر اول برای ورود به ناحیه بحرانی بین ریسمانها برای بررسی متغیر قفل است
  - این ناحیه بحرانی کوتاه است و متغیر guard زود آزاد می‌شود بنابراین انتظار هم کوتاه است
- متغیر دوم همان قفل اصلی است برای ناحیه بحرانی برنامه

```
int guard = 0;
int value = FREE;

lock_acquire() {
        // Short busy-wait time
        while (test&set(guard));
        if (value == BUSY) {
                put thread on wait queue;
                go to sleep() & guard = 0;
        }
        else {
                value = BUSY;
                guard = 0;
        }
}
```

```
lock_release() {
        // Short busy-wait time
        while (test&set(guard));
        if anyone on wait queue {
                take thread off wait queue
                place on ready queue;
        }
        else {
                value = FREE;
        }
        guard = 0;
}
```

قفل‌های سطح بالا

HighLevel Locks

قفل‌های سطح بالا

- نوشتن برنامه‌های چند ریسمانی با ریسمانهای همکار کار دشواری است. بنابراین نیاز به مکانیزمهای سطح بالاتری وجود دارد تا نوشتن برنامه‌ها و تست و توسعه آنها را ساده‌تر کند

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

سمافور

- سمافور به نوعی یک lock عمومی‌تر است که توسط Dijkestra معرفی شد
  - بسیاری از سیستم عاملها در آن دوره (دهه ۶۰) از این مکانیزم استفاده میکردند.

- سمافور در واقع یک متغیر از نوع عدد صحیح (integer) در درون خود دارد که منفی نمی‌شود
- دارای دو تابع است که هر دو به شکل اتمیک اجرا می‌شوند
  - P یا wait:
  - اگر متغیر سمافور مثبت باشد یک واحد از آن کم می‌شود و گرنه ریسمان فراخواننده در حالت انتظار می‌ماند.
  - V و یا signal:
  - مقدار متغیر سمافور را یک واحد زیاد می‌کند و ریسمانهای در حال انتظار را بیدار می‌کند.

- به غیر از این دو تابع هیچ راه‌حل دیگری برای دسترسی خواندن و یا نوشتن به متغیر درون سمافور وجود ندارد مگر در زمان ساختن که مقدار اولیه می‌توان به آن نسبت داد

سمافور

- پیاده‌سازی سطح بالای سمافور

```
int sem = initial_value;

wait() {
       while (sem <= 0);

       sem--;
}

signal(){
       sem++;
}
```

- پیاده‌سازی بدون انتظار

```
int sem = initial_value;

wait() {
       sem--;
       if(sem <= 0)
              add thread to waiting list;
              go to sleep
}
```

```
int sem = initial_value;

signal() {
       sem++;
       if(sem > 0)
              remove a thread from waiting
                     list;
              wakeup the thread
}
```

سمافور

- سمافور با مقدار اولیه 1 یا semaphore(1) همان قفل است که با نام سمافور باینری (Binary Semaphore) هم شناخته می‌شود و می‌توان از آن برای حفاظت از ناحیه بحرانی استفاده کرد

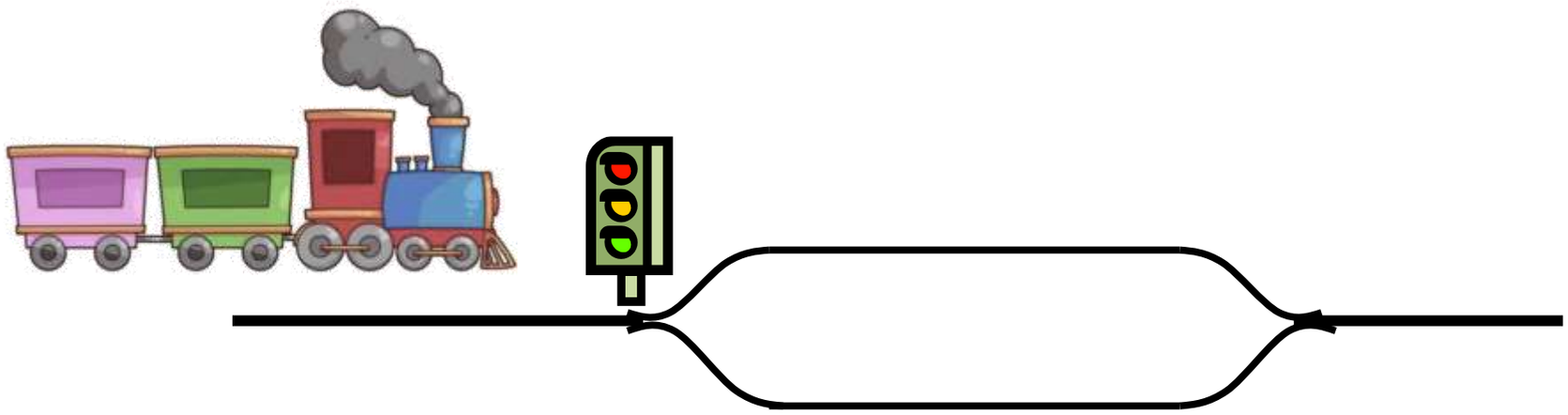
```
lock:Semaphore(1,1);
lock.wait();
      do_critical_section();
lock.signal();
```

مقدار اولیه

اندازه سمافور

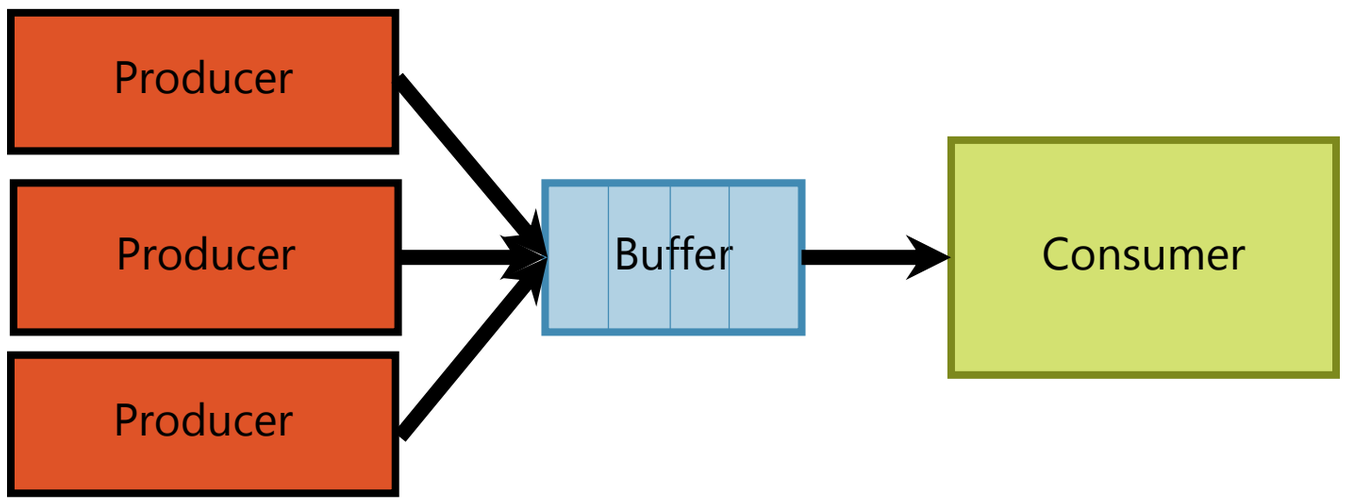
- در مثال زیر مقدار سمافور ۲ است و بهمین جهت دو قطار می‌تواند عبور کند ولی قطار سوم باید منتظر بماند

```
light:Semaphore(2,2);
train(){
      light.wait()
            cross();
      light.signal();
}
```



مساله تولید کننده – مصرف کننده

- مساله تولیدکننده-مصرف کننده (Producer-Consumer)
- در این مساله یک یا چند ریسمان داده‌ای را تولید می‌کنند که توسط یک یا چند ریسمان مصرف می‌شود.
- در بین این ریسمانها یک صف (بافر) وجود دارد که داده‌های تولید شده در آن قرار داده شده و مصرف‌کننده‌ها داده‌ها را از سر صف بر میدارند
- زمانی که صف پر می‌شود ریسمانهای تولید کننده باید صبر کنند تا صف خالی شود
- زمانی که صف خالی می‌شود ریسمانهای مصرف کننده باید صبر کنند تا داده‌ای در صف قرار داده شود



مساله تولید کننده – مصرف کننده

```
fullBuffer:Semaphore(n, 0);   // initial number of data in queue
emptyBuffers:Semaphore(n, n); // initial num of empty slots in queue
mutex:Semaphore(1, 1);        // lock for using shared queue


Producer(item) {
     emptyBuffers.wait();     // Wait until space
     mutex.wait();            // Wait until buffer free
     Enqueue(item);
     mutex.signal();
     fullBuffers.signal();    // Tell consumers there is an item
}


Consumer() {
     fullBuffers.wait();      // Check if there's a data
     mutex.wait();            // Wait until machine free
     item = Dequeue();
     mutex.signal();
     emptyBuffers.signal();   // tell producer need more
     return item;
}
```

مساله تولید کننده – مصرف کننده

- چه می‌شود اگر جای دو خط اول جابجا شود؟

```
Producer(item) {
        mutex.wait();                   // Wait until buffer free
        emptyBuffers.wait();            // Wait until space
        Enqueue(item);
        mutex.signal();
        fullBuffers.signal();    // Tell consumers there is an item
}
```

- ممکن است کد در خط دوم تا ابد بماند!
- ترتیب waitها مهم است ولی ترتیب signalها خیر

- آیا این پیاده‌سازی برای بیش از یک تولید کننده و مصرف کننده کار می‌کند؟

مساله قبیله آدمخواران

- یک قبیله آدمخوار غذا را از یک دیگ که برای M نفر جا دارد میخورند.
- هرگاه دیگ خالی شد آنها آشپز را بیدار کرده و آشپز M غذا را در دیگ پخته و بعد به آنها خبر میدهد.
- این مساله شبیه به مساله producer-consumer است منتها باید دقت کرد که زمانی تولید کننده تولید می‌کند مصرف کننده منتظر است و زمانی که مصرف کننده در حال مصرف است، تولید کننده به خواب میرود.

- حل مساله:
- آیا داده مشترک وجود دارد؟ بله دیگ بین آدمخواران مشترک است و همه از آن غذا می‌خورند
  - بنابراین یک قفل معمولی لازم است
- چه شرایطی در مساله برقرار است؟ برای هر شرط یک سمافور لازم است
  - پر بودن دیگ
  - خالی شدن دیگ

مساله قبیله آدمخواران

```
int servings=0 ;// up to M
mutex: Semaphore(1)
emptyPot:Semaphore(1,1);
fullPot :Semaphore(1,0);

Cook() {
      while(true)
            emptyPot.wait();

            makeFood();

            fullPot.signal();
}
```

```
Savage(){
      while(true)
            mutex.wait();

            if(servings == 0)
                  emptyPot.signal();
                  fullPot.wait();
                  servings=M

            servings--;
            getServingFromPot();

            mutex.signal();

            eat();
}
```

مساله قبیله آدمخواران

- چرا در این مساله مانند مساله تولیدکنندگان و مصرف کنندگان از یک سمافور M تایی استفاده نکردیم؟

- چون M غذا یکباره به سیستم وارد می‌شود نه یکی یکی

مساله فیلسوفهای دور میز شام

- دور یک میز ۵ (یا n) فیلسوف نشسته‌اند که مدتی فکر می‌کنند و بعد با استفاده از دو چوب یا چنگال که در سمت و چپ خود دارند شروع به غذا خوردن می‌کنند.
- بعد از مدت نامعلومی چنگالها را گذاشته و بعد به فکر میروند

- حل مساله:
- آیا داده مشترک وجود دارد؟ بله چنگالها
- آیا چنگالها یکی یکی برداشته می‌شوند؟ بله
- چه شرایطی در مساله برقرار است؟ برای هر شرط یک سمافور لازم است
  - برای خوردن باید فیلسوفها هر دو چوب سمت راست و چپشان را باید داشته باشند

مساله فیلسوفهای دور میز شام

- راه‌حل اولیه ساده

```
chopstick[5]: Semaphore(1, 1);

philsopher(int i){
    do {

        think();

        chopstick[i].wait()
        chopStick[(i + 1) % 5].wait()

        eat();

        chopstick[i].signal()
        chopStick[(i + 1) % 5].signal()


    } while (TRUE);
}
```

- این راه‌حل دارای مشکل بن بست (deadlock) است
- اگر همه فیلسوفها در یک زمان تصمیم به خوردن بکنند همه چنگالهای سمت راست را برداشته و منتظر چنگال سمت چپ می‌شوند
- این در حالی است که چنگالهای سمت چپ، سمت راستی فیلسوف بغل دستی بوده است
- بن‌بست:
- برای انجام کار ریسمان منتظر منبعی است که آن منبع در اختیار ریسمانی است که برای ادامه کار نیاز به منبعی دارد که در اختیار همین ریسمان است.

مساله فیلسوفهای دور میز شام

- راه‌حل دوم
- فیلسوفهای با اندیس زوج از سمت راست و فیلسوفهای فرد از سمت چپ شروع کنند.

```
chopstick[5]: Semaphore(1, 1);

philsopher(int i){
    do {

       think();

       if(i%2 == 0)
          chopstick[i].wait()
          chopStick[(i + 1) % 5].wait()
       else
          chopStick[(i + 1) % 5].wait()
          chopstick[i].wait()

       eat();

       chopstick[i].signal()
       chopStick[(i + 1) % 5].signal()


    } while (TRUE);
}
```

مساله فیلسوفهای دور میز شام

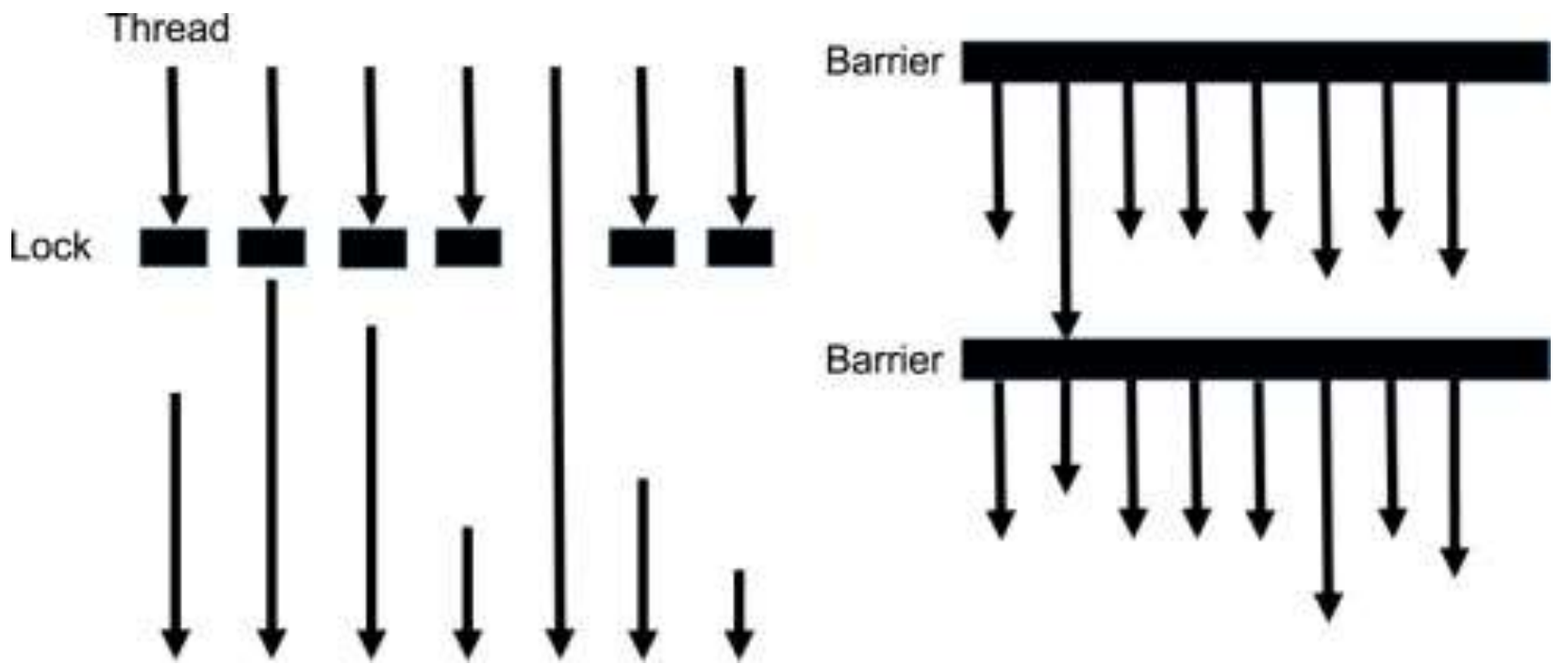
- راه‌حل سوم
- گرفتن هر دو چنگال در ناحیه بحرانی باشد

```
chopstick[5]: boolean;
mutex: Semaphore(1,1);

philsopher(int i){
   do {

      think();

      boolean flag = FALSE;
      while(!flag)
         mutex.wait()
         if(chopstick[i] && chopstick[(i+1)%5])
            chopstick[i] = 0;
            chopstick[(i+1)%5] = 0;
             flag= true;
         mutex.signal()

      eat();

      mutex.wait()
      chopstick[i] = chopstick[(i+1)%5] = 1
      mutex.signal()

      flag = false
   } while (TRUE);
}
```

مساله مانع (Barrier)

- چند ریسمان به صورت موازی در حال اجرای یک برنامه هستند. مکانیزم مانع مکانیزمی است که یک عدد صحیح مثبت گرفته و در یک جای کد فراخوانی می‌شود.
- تمامی ریسمانها در آن خط باید صبر کنند تا تعداد ریسمانها به عدد داده شده برسد
- یعنی اجرای آن تعداد ریسمان باید به آن خط رسیده باشد تا همه با هم دوباره به اجرا ادامه بدهند

```
b:Barrier(5);

multi_thread_run(){

     do_first_phase();

     b.wait();

     do_second_phase();
}
```



https://images.app.goo.gl/zRUVKX2pxHvDuR9j6

مساله مانع (Barrier)

- حل مساله:
- آیا داده مشترک وجود دارد؟ بله تعداد ریسمانهایی که تابع wait مکانیزم مانع را فراخوانی می‌کنند.
- آیا شرطی وجود دارد؟ بله. تا زمانی که تعداد ریسمانها به حد نصاب نرسیده باید ریسمانهایی که تا این خط رسیده‌اند منتظر بمانند

مساله مانع (Barrier)

- راه‌حل

```
mutex: Semaphore(1,1);
int counter = 0;
barrier: Semaphore(1,0)

barrier_wait(){

    mutex.wait()
    counter++;
    if(counter == N)
       barrier.signal()
    mutex.signal()

    // threads must wait here, how??
}
```

مساله مانع (Barrier)

- راه‌حل

```
mutex: Semaphore(1,1);
int counter = 0;
barrier: Semaphore(1,0)

barrier_wait(){

    mutex.wait()
    counter++;
    if(counter == N)
       barrier.signal()
    mutex.signal()

    barrier.wait()
    barrier.signal()
}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

- در یک برنامه چند ریسمانی یک داده مشترک بین ریسمانها وجود
  - درصد بالایی از ریسمانها فقط مقدار متغیر را می‌خوانند
  - تعداد محدودی از ریسمان‌ها و با سرعت بمراتب کمتری مقدار متغیر را تغییر می‌دهند
- ورود چندین ریسمان که مقدار متغیر را تغییر نمی‌دهند به ناحیه بحرانی مشکلی ندارد ولی در آن واحد فقط یک ریسمان تغییر دهنده باید وارد ناحیه بحرانی شود.

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

- روال نوشتن تقریبا مشخص و ساده است چون فقط یک ریسمان باید داخل ناحیه بحرانی باشد

```
w_mutex: Semaphore(1,1);

writer (){
   while(TRUE)
      w_mutex.wait()

      modify_data();

      w_mutex.signal()

}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

- در روال خواندن اولین و آخرین ریسمان خواننده اجازه ورود به ریسمان نوشتن را می‌دهند
- دقت کنید که مرحله خواندن مقدار، ناحیه بحرانی نیست و همین باعث افزایش سرعت خواندن مقدار مشترک می‌شود.

```
w_mutex: Semaphore(1,1);
mutex: Semaphore(1,1);
readers:int

reader (){
   while(TRUE)
       mutex.wait();
       readers++;
       if(readers==1)
              w_mutex.wait()
       mutex.signal()

       read_data();

       mutex.wait()
       readers--;
       if(readers==0)
            w_mutex.signal()

       mutex.signal()


}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

- کد قبلی برای ریسمانهای نویسنده منصفانه نیست چون تا زمانی که یک ریسمان خواننده در حال خواندن مقدار متغیر است ریسمان نویسنده حق ورود ندارد

- پاسخ را طوری تغییر می‌دهیم که زمانی که یک ریسمان نویسنده وارد شد دیگر هیچ ریسمان خواننده قادر به ورود نباشد

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

- راه‌حل عادلانه

```
access : Semaphore (1,1);
rmutex : Semaphore (1,1);
service_mutex : Semaphore (1,1);
readers:int

writer() {
    service_mutex.wait();
     access.wait();
    service_mutex.signal();

    write();

    access.signal();
}
```

```
reader(){
     service_mutex.wait();
     rmutex.wait();
     if(readers == 0)
       access.wait();

     readers++;
     rmutex.signal();

     service_mutex.signal();

     //critical section
     read_shared_data();

     rmutex.wait();
     readers--;
     if(readers == 0)
      access.signal();
     rmutex.signal();
}
```

مانیتور (Monitor)

- مساله‌ی اصلی سمافور این است که این ساختار برای دو منظور به کار می‌رود.
  - هم برای ایجاد ناحیه بحرانی و انحصار متقابل
  - هم برای ایجاد شرط به منظور صبر کردن

- این مشکل باعث می‌شود که نوشتن و آزمون کدها سخت و پیچیده شود

مانیتور (Monitor)

- بدون وجود شرط می‌توان مانیتور را یک شیء یا ماجول دانست که شامل یک قفل (mutex) است و امکان دسترسی امن خودکار را به متغیرها و متدها فراهم می‌کند.
- در تعریف اولیه مثلا در مورد یک class تمامی متدهای public کلاس بطور خودکار توسط قفل مانیتور محافظت می‌شوند

```
1  monitor class Account {
2    private int balance := 0
3
4    public boolean withdraw(int amount) {
5      if (balance < amount) {
6        return false
7      } else {
8        balance := balance - amount
9        return true
10     }
11   }
12
13   public deposit(int amount){
14     balance := balance + amount
15   }
16 }
```

- در واقع هدف از مانیتور تعریف چهارچوبی است که در آن برنامه‌نویس مجبور به انجام کارهای کمتری شود و در نتیجه احتمال اشتباه و یا فراموشی در برنامه‌نویسی چند ریسمانی کاهش یابد

مانیتور (Monitor)

- زبان جاوا به طور ذاتی از مکانیزم مانیتور پشتیبانی می‌کند.
- در جاوا می‌توان برای تعریف یک متد امن برای چند ریسمانی (thread-safe) از کلمه کلیدی synchronized استفاده کرد.

```
1  public class Account {
2
3      private int balance = 0;
4
5      public synchronized boolean withdraw(int amount) {
6          if (balance < amount) {
7              return false;
8          }
9          else {
10             balance = balance - amount;
11             return true;
12         }
13     }
14
15     public synchronized void deposit(int amount) {
16         balance = balance + amount;
17     }
18 }
```

مانیتور (Monitor)

- در مانیتور قفل (mutex) فقط برای ایجاد ناحیه بحرانی بکار میرود

- چندین ریسمان می‌توانند برای یک شرط خاص منتظر بمانند که این شرط منتسب به قفل مورد استفاده در مانیتور است.
  - نوع داده شرطها با قفل متفاوت است. این جداسازی باعث می‌شود
    - کدها شفاف و واضح باشند
    - برخی کارها به شکل خودکار انجام شده و برنامه‌نویسی تسهیل شود

- در ابتدا به نظر می‌رسد که می‌توان از یک حلقه انتظار برای پیاده‌سازی شرطها استفاده کرد

مانیتور (Monitor)

```java
public class Producer<T> implements Runnable{

    private final Queue<T> queue;
    private final int capacity;

    public Producer(Queue<T> queue, int capacity) {
        this.queue = queue;
        this.capacity = capacity;
    }

    public synchronized void run() {


        // busy waiting
        while(queue.size() == capacity);

        T data = produceData();
        queue.offer(data);

    }
```

مانیتور (Monitor)

- در این پیاده‌سازی مشکل اصلی این است که ریسمانی که قفل را در اختیار دارد در حلقه می‌ماند و قفل را رها نمی‌کند
- در این پیاده‌سازی امکان بن‌بست وجود دارد

- در واقع باید برای پیاده‌سازی صحیح شرطها و انتظار ریسمانها برای به وقوع پیوستن یک شرط ریسمانی که هنوز شرط مورد نظرش بوقوع نپیوسته باید:
  - قفل را به صورت موقت رها کند
  - دوباره برای بررسی شرط قفل را بدست آورد

- با این ترفند هم بررسی متغیر مشترک در داخل ناحیه بحرانی انجام شده و هم امکان بن‌بست از بین رفته

مانیتور (Monitor)

No synchronized keyword

Using explicit Lock

lock/unlock until condition is satisfied

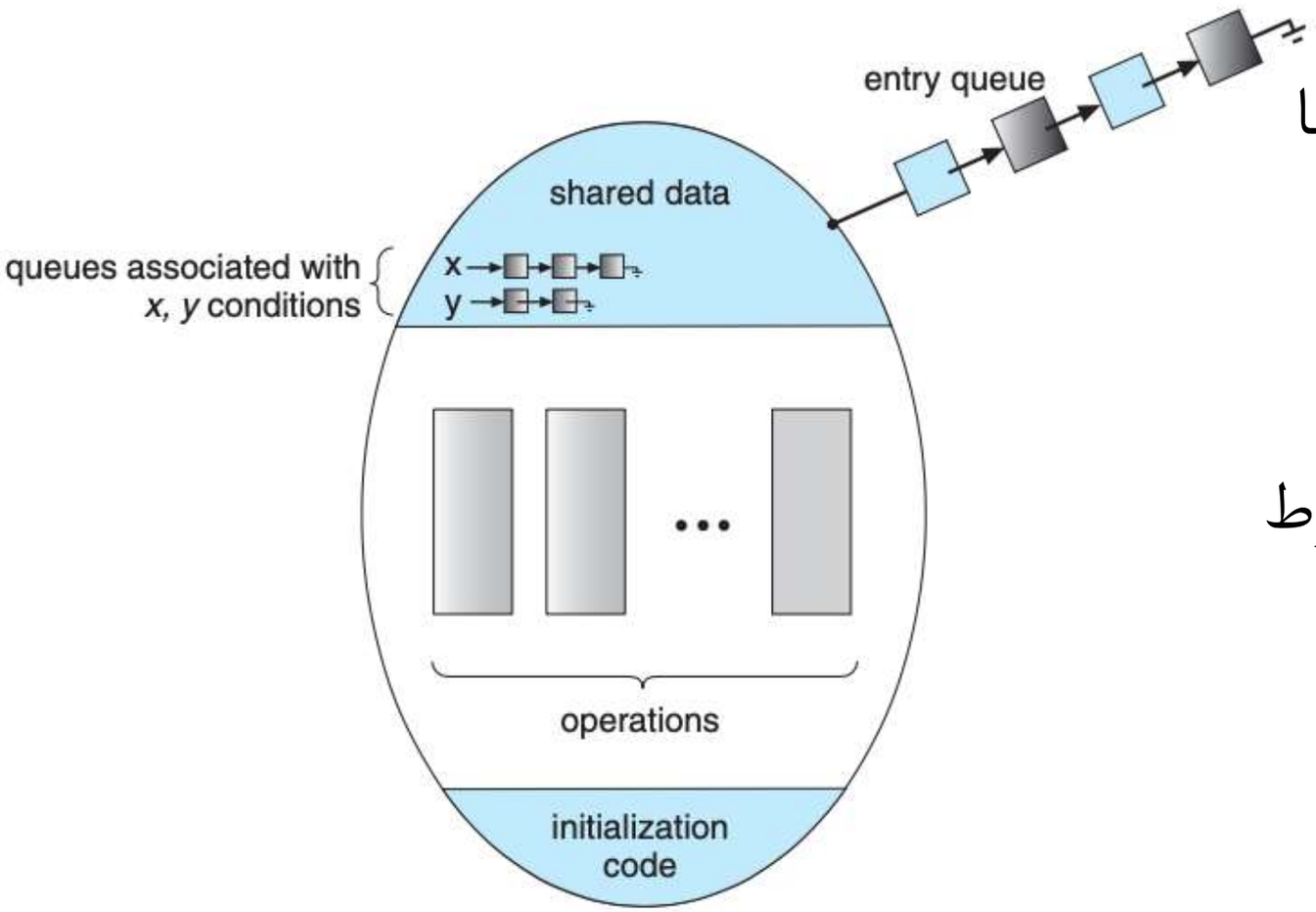
```
11      public void run() {
12
13          while (true) {
14              lock.lock();
15              try {
16                  // busy waiting
17                  while (queue.size() == capacity){
18                      lock.unlock();
19
20                      Thread.yield();
21
22                      lock.lock();
23                  }
24
25                  T data = produceData();
26                  queue.offer(data);
27              }
28              finally {
29                  lock.unlock();
30              }
31          }
32      }
```

مانیتور (Monitor)

- برای اینکه کل این فرآیند انتظار برای یک شرط به صورت ساده‌تری قابل پیاده‌سازی باشد و نیز بخشی از کار به شکل خودکار انجام گیرد یک نوع داده جدید شرط معرفی شده است.

- هر متغیر شرطی در واقع یک صف از ریسمانهایی است که منتظر هستند آن شرط برقرار شود و رویدادی بوقوع بپیوندد

- در زمان برقرار نبودن شرط ریسمان به شکل خودکار و اتمیک و بدون دخالت یا آگاهی برنامه‌نویس به صف انتظار میرود و قفل را رها می‌کند.



- در بیدار شدن هم بدون دخالت کاربر برای گرفتن مجدد قفل رقابت می‌کند و دوباره شرط را چک می‌کند

مانیتور (Monitor)

- متغیرهای شرط دارای سه تابع هستند
  - wait
  - signal
  - signalAll – در برخی پیاده‌سازیها وجود ندارد
- بر اساس شرایط بر روی هر کدام از متغیرهای شرط می‌توان wait و signal انجام داد
- قبل از فراخوانی توابع شرط‌ها، باید قبلا قفل متناظر اخذ شود
- عموما قالب استفاده از مانیتور به این شکل است:

```
lock: Lock
condition1: lock.Condition()

function1(){
    lock.aqcuire();

    while(my_conditions)
       condition1.wait();

    do_something();

    lock.release();
}
```

```
function2(){
    lock.aqcuire();

    do_something();

    condition1.signal();

    lock.release();
}
```

مانیتور (Monitor)

- در واقع پیاده‌سازی تابع wait برای شرط مانند شبه کد زیر است.
- این کار کاملا خودکار انجام می‌شود

```
lock: Lock
condition1: lock.Condition()

function1(){
   lock.aqcuire();

   while(my_conditions)
      condition1.wait();

   do_something();

   lock.release();
}
```

```
lock: Lock
condition1: lock.Condition()

function1(){
   lock.aqcuire();

   while(my_conditions){
      lock.release();

      go_to_sleep();

      lock.aqcuire();
   }

   do_something();

   lock.release();
}
```

When thread is resumed, starts from here

مساله تولیدکننده-مصرف کننده

```
1  public class Queue{
2
3  Lock lock = new ReentrantLock();
4  Condition notEmpty = lock.newCondition();
5  Condition notFull = lock.newCondition();
6  int count = 0;
7
8  public E take() throws InterruptedException {
9    lock.lockInterruptibly();
10   try {
11     while (count == 0)
12       notEmpty.await();
13
14     Object item = dequeue();
15     notFull.signal();
16     return item;
17   } finally {
18     lock.unlock();
19   }
20 }
```

```
1  public void put(E e) throws
   InterruptedException {
2    lock.lockInterruptibly();
3    try {
4      while (count == items.length)
5        notFull.await();
6      enqueue(e);
7
8      notEmpty.signal();
9    } finally {
10     lock.unlock();
11   }
12 }
```

مساله مانع (Barrier)

```
class Barrier {
    private int currentCount;

    private Lock lock = new ReentrantLock();
    private Condition barrierLift = lock.newCondition();

    Barrier(int friendsCount) {
        this.currentCount = friendsCount;
    }

    void waitForFriends() {
        lock.lock();
        try {
            currentCount--;

            while (currentCount > 0)
                barrierLift.await();

            barrierLift.signal();
        } catch (Exception e) { e.printStackTrace(); }
        finally { lock.unlock(); }
    }
}
```

مساله مانع (Barrier) – مثال نحوه استفاده

```
Barrier barrier = new Barrier(friendsCount);
final ExecutorService executorService = Executors.newFixedThreadPool(friendsCount);

try {
    for (int i = 0; i < friendsCount; i++) {
        int finalI = i;
        executorService.submit(() ->
        {
            System.out.println("Thread " + finalI + " started....");

            sleep(100);

            System.out.println("Thread " + finalI + " before barrier....");

            barrier.waitForFriends();

            System.out.println("Thread " + finalI + " after barrier...");

    });
}
```

مساله قبیله آدمخواران

```
private static final int MAX_FOOD = 10;
private Lock lock = new ReentrantLock();
private Condition emptyCondition = lock.newCondition();
private Condition fullCondition = lock.newCondition();
private int food = MAX_FOOD;

class Cook implements Runnable {
    @Override
    public void run() {
        while (running) {
            try {
                lock.lock();
                while (food > 0)
                    emptyCondition.await();

                //cooking delay
                sleep(100);
                food = MAX_FOOD;

                System.out.println("Food prepared...");
                fullCondition.signalAll();
            }
            catch (Exception e) { e.printStackTrace(); }
            finally { lock.unlock(); }
        }
    }
```

مساله قبیله آدمخواران

```
class Cannibals implements Runnable {
    private int index;
    public Cannibals(int cannibalIndex) {
        this.index = cannibalIndex;
    }

    @Override
    public void run() {
        while (running) {
            lock.lock();
            try {
                while (food == 0) {
                    emptyCondition.signal();
                    fullCondition.await();
                }

                food -= 1;
            }
            catch (Exception e) {e.printStackTrace();}
            finally {lock.unlock();}

            // go and eat the food
            Thread.yield();
        }
    }
}
```

مساله فیلسوفهای دور میز شام

```
private Lock lock = new ReentrantLock();
/**
 * One condition for each Philosopher
 */
private Condition[] conditions = new Condition[TOTAL_COUNT];
/**
 * Availability of chopsticks
 */
private boolean[] chopsAvail = new boolean[TOTAL_COUNT];
/**
 * A boolean array to observe which Philosophers are eating simultaneously.
 * This array is not necessary for solving the problem.
 */
private boolean[] eating = new boolean[TOTAL_COUNT];
```

مساله فیلسوفهای دور میز شام

```
public void run() {
    while (running) {
        lock.lock();
        try {
          while (!chopsAvail[getPrevNo()] || !chopsAvail[no]) {
                conditions[no].await();
            }

            chopsAvail[getPrevNo()] = chopsAvail[no] = false;
            eating[no] = true;
        }
        catch (Exception e) { break; }
        finally { lock.unlock(); }

        sleep(200);//eating


        lock.lock();
        try {
            chopsAvail[getPrevNo()] = chopsAvail[no] = true;
            // signal both sides of a Philosopher to wake-up and eat
            conditions[getNextNo()].signal();
            conditions[getPrevNo()].signal();
            eating[no] = false;
        }
        finally { lock.unlock(); }
}}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

```
private Lock lock = new ReentrantLock();
private Condition condition = lock.newCondition();
private int readerCount = 0;
private int sharedData = 0;


private void writer() throws InterruptedException {
    while (!finish) {
        lock.lock();
        try {
            while (readerCount > 0)
                condition.await();

            sharedData = new Random().nextInt(1000);
            condition.signalAll();

        }
        finally { lock.unlock(); }

        sleep(1000);
    }
}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem)

```
private void reader() {
    while (!finish) {
        lock.lock();
        try {
            readerCount++;
        }
        finally { lock.unlock(); }

        // read value
        System.out.println(Thread.currentThread().getName() + "::value = " + sharedData);
        sleep(1000);

        lock.lock();
        try {
            readerCount--;

            if (readerCount == 0)
                condition.signalAll();
        }
        finally { lock.unlock(); }
    }
}
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem) راه حل تقدم نویسنده

```
private void writer() throws InterruptedException {
    while (!finish) {
        lock.lock();
        try {
            while (readerCount > 0) {
                writer = true;
                condition.await();
            }

            value = new Random().nextInt(1000);
            System.out.println(Thread.currentThread().getName() + "::value = " + value);
            sleep(100);

            writer = false;
            condition.signal();
        }
        finally { lock.unlock(); }
```

مساله خوانندگان و نویسندگان (Readers-Writers Problem) راه حل تقدم نویسنده

```
private void reader() throws InterruptedException {
    while (!finish) {
        lock.lock();
        try {
            while (writer)
                condition.await();

            readerCount++;
        }
        finally { lock.unlock(); }

        // read value
        System.out.println(Thread.currentThread().getName() + "::value = " + value);
        sleep(100);

        lock.lock();
        try {
            readerCount--;

            if (readerCount == 0)
                condition.signal();
        }
        finally { lock.unlock(); }
    }
}
```

مساله حوزه دید بین ریسمانها

Visibility

مثال اتمام کار ریسمانها

- فرض کنید که میخواهیم در زمان مشخص و یا با رویداد خاصی کار ریسمانهایی که به صورت while(true) بودند را متوقف کنیم.

- در زبان جاوا و یا برخی زبانهای دیگر توابع suspend و stop وجود دارد. هر دو این توابع منسوخ (deprecated) شده‌اند
  - در بیان علت آن گفته شده است که استفاده از آنها می‌تواند برنامه و حتی ماشین مجازی جاوا را در حالت غیر پایدار قرار دهد
  - یک علت مهم این است که فراخوانی این توابع می‌تواند زمانی انجام گیرد که ریسمان در حالت مناسبی برای اتمام کارش نیست.
  - مثلا ممکن است منابعی از سیستم در اختیار داشته باشد نظیر فایل باز و یا قفل

مثال اتمام کار ریسمانها

- یک راه‌حل ابتدایی این است که از یک متغیر استفاده کنیم

```
boolean finish = false;
```

```
public void run(){

    while(!finish){

      //do something

    }
}

public void stop(){
    finish = true;
}
```

Thread 1

```
public void run(){

    while(!finish){

      //do something

    }
}

public void stop(){
    finish = true;
}
```

Thread 2

```
public void run(){

    while(!finish){

      //do something

    }
}

public void stop(){
    finish = true;
}
```

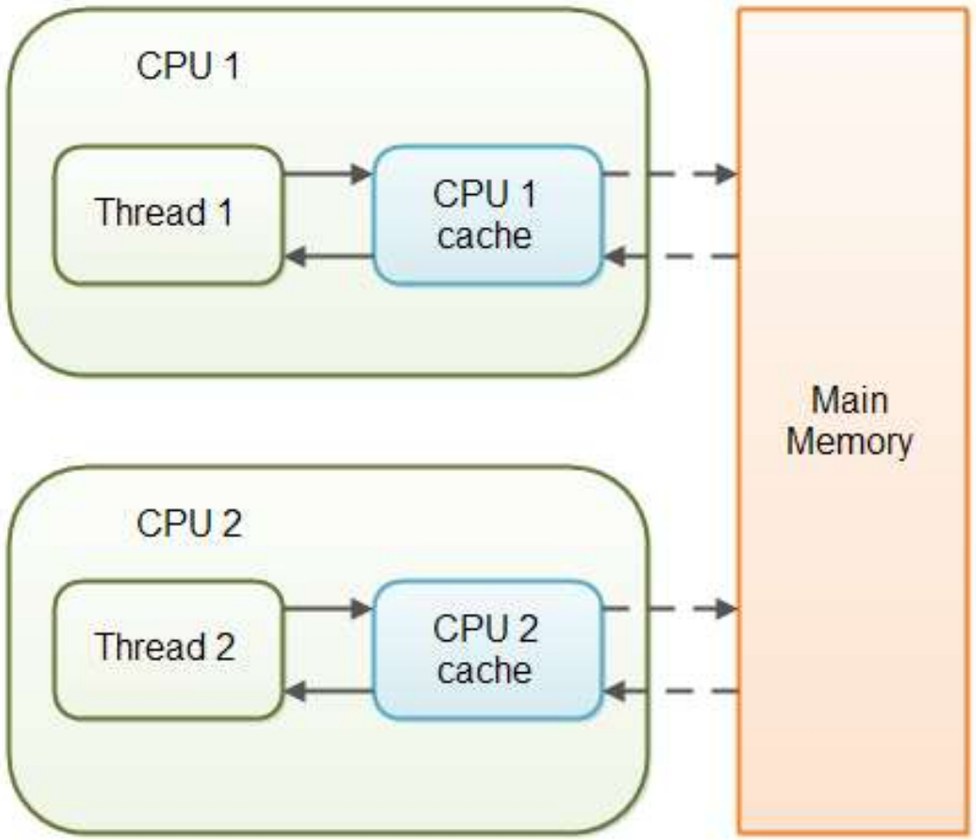
Thread 3

مثال اتمام کار ریسمانها

- این راه‌حل خصوصا در سیستمهای چند پردازنده و یا چند ریسمانی درست کار نخواهد کرد

- دلیل آن این است که ما به هیچ روشی به کامپایلر و یا محیط اجرایی اعلام نکرده‌ایم که این متغیر مشترک است و قرار است توسط چندین ریسمان تغییر یابد.

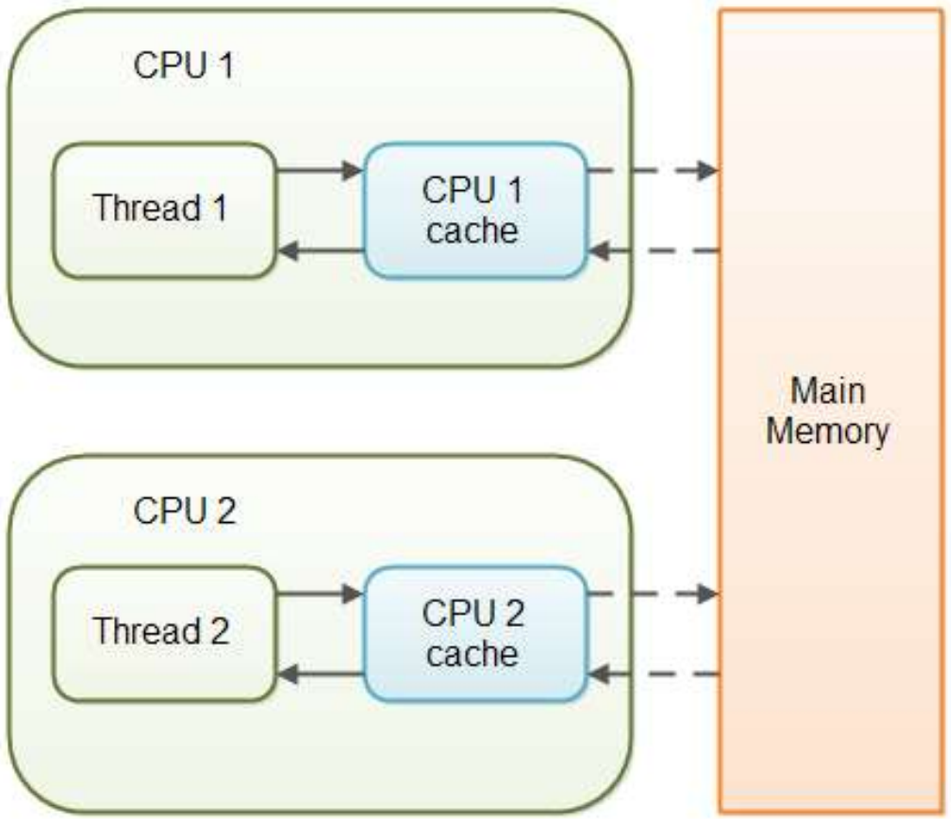
مثال اتمام کار ریسمانها



- بطور مثال معماری یک پردازنده چند هسته‌ای را در نظر بگیرید.
  - در حالت عادی ریسمانها مقدار متغیر finish را یکبار از حافظه اصلی خوانده و در cache پردازنده کپی می‌کنند
  - در اجرای حلقه مقدار همیشه از cache خوانده می‌شود.
  - اگر مقدار finish در یک ریسمان تغییر یابد در ریسمانهای دیگر تغییر منعکس نخواهد شد
- برای اجرای صحیح باید به کامپایلر اعلام شود تا در زمان کامپایل کدی تولید کند که ریسمان مجبور شود هر بار به حافظه اصلی رفته و آخرین مقدار متغیر finish را بخواند.

مثال اتمام کار ریسمانها

- یکی از راه‌حلهای اعلام وجود یک متغیر مشترک بین ریسمانها استفاده از قفل است
- در واقع در کد قبل مشکل اصلی این است که یک متغیر مشترک بین ریسمانها وجود دارد ولی برای خواندن و نوشتن آن از قفل استفاده نشده است.




```
Lock lock;
boolean finish = false;
public void run() {

   while (true) {
       lock.lock();
       try {
           if (finish)
               break;
       }
       finally {
           lock.unlock();
       }

   }
}

public void stop() {
   lock.lock();
   try {
       finish = true;
   }
   finally {
       lock.unlock();
   }
}
```

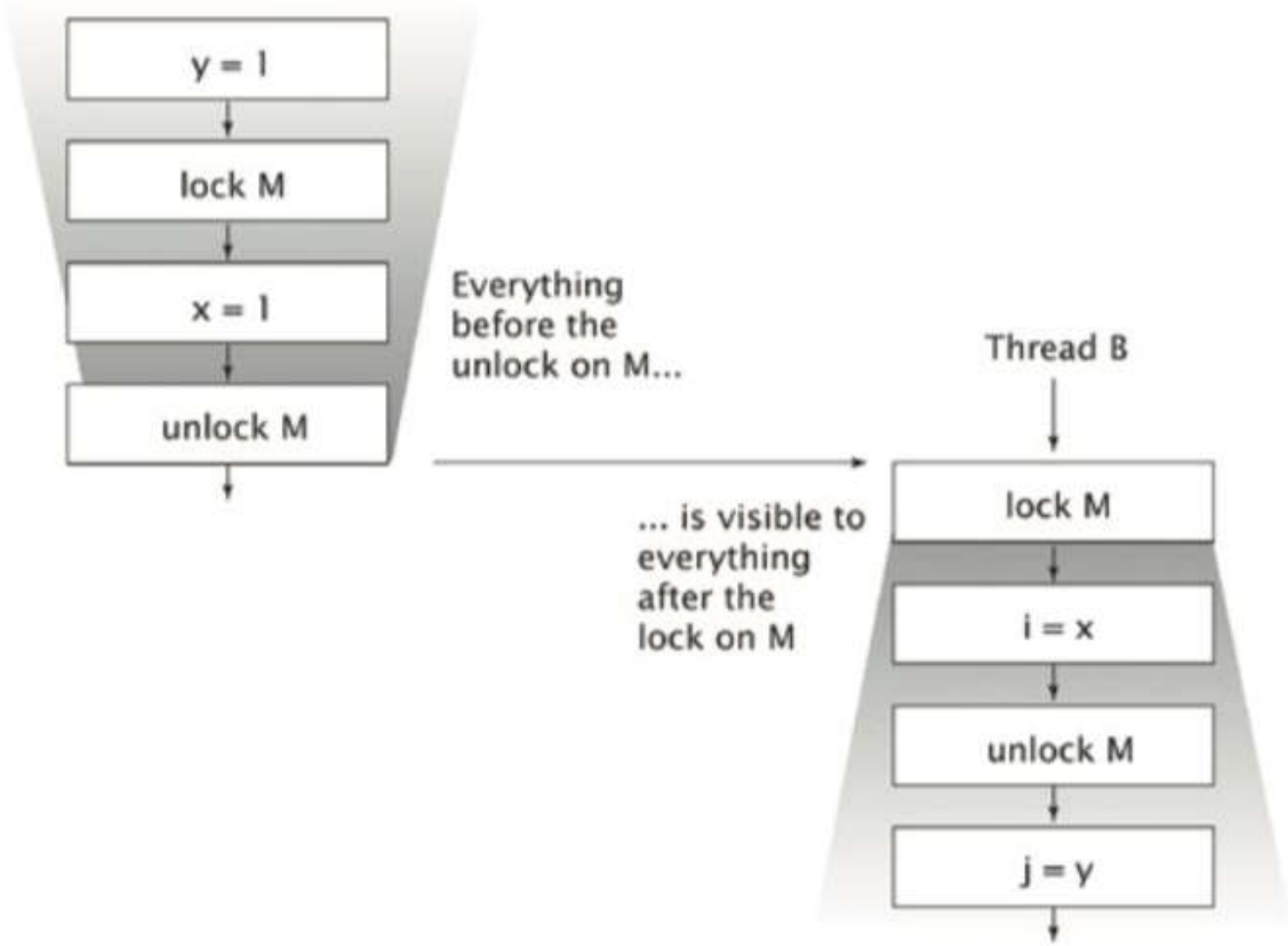
مساله حوزه دید

- در واقع هر قفل دارای دو مشخصه است
  - اجرای تجزیه‌ناپذیر یک قطعه کد بین lock و unlock
  - حوزه دید متغیرها (visibility)

- حوزه دید
- با اجرای lock و ورود یک ریسمان به محدوده ناحیه بحرانی مقادیر تمام متغیرهایی که در ناحیه بحرانی هستند (و یا همه متغیرها) بروز رسانی می‌شوند یعنی آخرین مقادیر آنها از حافظه اصلی خوانده شده و در cache قرار داده می‌شود.

- بعد از اجرای unlock مقادیر تمامی متغیرهای استفاده شده در ناحیه بحرانی در حافظه اصلی نوشته می‌شود

مساله حوزه دید



متغیرهای volatile

- در بیشتر زبانهای برنامه‌نویسی می‌توان متغیرهایی از نوع volatile تعریف کرد.
- در لغت volatile به معنای فرّار است.
- در زبانهای برنامه‌نویسی هدف از تعریف این متغیرها این است که مشخص شود
  - نباید به مقادیر cache این متغیرها اعتماد کرد و همیشه مقدار واقعی باید از حافظه اصلی خوانده شود.
  - موقع بروز رسانی مقدار، مقدار جدید باید در حافظه اصلی هم نوشته شود.

- بسته به معماری پردازنده این دو کار با مکانیزمهای مختلفی ممکن است پیاده‌سازی شود

متغیرهای volatile

- آیا متغیرهای volatile فقط در سیستمهای چند پردازنده و یا چند هسته مصرف دارد؟
- اگر یک متغیر به صورت مشترک بین چند ریسمان تعریف شود ولی هیچ نشانه‌ای برای این متغیر تعریف نشود که این متغیر مشترک است عملا دست کامپایلر و پردازنده در اینکه مقادیر درست بین ریسمانها چه زمانی در حافظه اصلی نوشته و در cacheها بروز رسانی شود باز است
- توضیح اینکه کامپایلرها و پردازنده‌ها برای اجرای سریع دستورات خواندن مقادیر و نوشتن مقدار بروز شده را در حافظه اصلی نمی‌نویسند و یا اینکه مثلا یک ریسمان از ابتدا تا انتها اجرا شده و بعد مقادیر متغیرها در حافظه اصلی نوشته می‌شود. در نتیجه ریسمان دیگر که از مقدار متغیر استفاده می‌کند ممکن است با مقدار قبلی کار کند و مقدار جدید را نبیند و یا بلعکس ممکن است مفهوم برنامه به هم بریزد.
- حتی برای بهینه‌سازی زمان اجرا برخی دستورات خارج از ترتیب کد اجرا می‌شوند

متغیرهای volatile

- آیا متغیرهای volatile فقط در سیستمهای چند پردازنده و یا چند هسته مصرف دارد؟

- با تعریف یک متغیر به صورت volatile و یا در استفاده از قفل کامپایلرها با گذاشتن دستورات خاص مانع از اجرای بدون ترتیب می‌شوند

- همیشه نوشتن مقدار باید قبل از خواندن مقدار متغیر باید انجام گیرد.
  - بعبارت دیگر آنچه توسط ریسمان تغییر دهنده اعمال می‌شود پس از نوشته شدن حتما توسط ریسمانهایی که بعدا مقدار را می‌خوانند دیده می‌شود و این تضمین شده است.

- مساله حوزه دید در سیستمهای توزیع شده و تحت عنوان مبحث consistency مطرح می‌شود

متغیرهای volatile

- ویژگیهای متغیر volatile

- زمان خواندن این متغیرها بیشتر از متغیرهای عادی است
  - چرا که در cache ذخیره نمی‌شوند

- در خواندن مقدار متغیرهای volatile همواره مقدار نهایی متغیر خوانده می‌شود.

- در خواندن مقدار متغیرهای volatile ریسمان اجرا کننده هیچگاه بلاک نمی‌شود
  - مثل استفاده از قفل ریسمان به صف بلاک شده‌ها نمی‌رود

- کارآیی استفاده از این متغیرها بیشتر از قفل است چرا که نیازی به فراخوانی سیستمی و تعویض زمینه نیست

متغیرهای volatile

- بنابراین برای متوقف ساختن ریسمانها می‌توان از کد ساده‌تر (در مقایسه با قفل) زیر استفاده کرد

```
volatile boolean finish = false;
Lock lock;

public void run(){

    while(!finish){
      //do something
    }
}

public void stop(){
    finish = true;
}
```

- بهترین استفاده از این نوع متغیر در تعریف فیلدهای status, flag, ... است.

متغیرهای volatile

- تفاوت متغیرهای volatile و قفل‌ها
- متغیرهای volatile دارای ویژگی اجرای تجزیه‌ناپذیر (اتمیک) نیستند

- اگر چند ریسمان می‌خواهند مقدار یک متغیر مشترک را تغییر دهند این کار به شکل اتمیک کار نمی‌کند.
  - اصلاح مقدار متغیر شامل سه مرحله است خواندن و بعد تغییر و نوشتن آن می‌شود
  - اگر تعداد نویسنده‌ها بیش از یکی باشد احتمال خراب شدن مقدار متغیر وجود دارد چرا که در وسط اجرای کد بعد از خواندن مقدار می‌توان تصور کرد که پردازنده از ریسمان گرفته شده و به ریسمان دیگری داده شود.
  - اگر اندازه متغیر از معماری پردازنده بزرگتر باشد مشکل وجود دارد چرا که مثلا در معماری ۳۲ بیتی خواندن یک متغیر ۶۴ بیتی از نوع long در دو مرحله انجام می‌شود و اصلا اتمیک نیست.

متغیرهای volatile

- بهترین موارد استفاده از متغیرهای volatile

- زمانی که نوشتن مقدار متغیر به خواندن مقدار فعلی آن وابسته نیست
  - چرخه read-modify-write ندارند
  - مثل مثال finish که در متد stop تغییر مقدار به true وابسته به مقدار فعلی آن نیست.

- زمانی که فقط یک ریسمان مقدار متغیر را تغییر می‌دهد و باقی ریسمانها همواره مقدار متغیر را می‌خوانند.

- در غیر از این دو حالت بهتر است از
  - از قفل استفاده شود
  - از متغیرهای Atmoic استفاده شود

متغیرهای Atomic

- این کلاسها مستقیما از دستورات سطح پایین پردازنده نظیر compareAndSwap استفاده می‌کنند
- عمدتا برای پیاده‌سازی الگوریتمهای non-blocking و lock-free مورد استفاده قرار می‌گیرند.

- این کلاسها اجرای اتمیک یک چرخه کامل read-modify-write ممکن می‌کنند.
- در جاوا کلاسهای زیر وجود دارند
  - AtomicInteger
  - AtomicLong
  - AtomicRefrenece
  - AtomicIntegerArray
  - AtomicLongArray
- این کلاسها دارای متدهایی مثل getAndIncrement و یا incrementAndGet هستند که به شکل اتمیک اجرا می‌شوند.

متغیرهای Atomic

- در شرایط غیر رقابتی اجرای یک دستور با کلاسهای Atomic تقریبا نصف زمان اجرای همان کار با پیاده‌سازی قفل است
  - طبق آزمایشهای انجام شده در شرایط بار متوسط و کم این کلاسها کارایی ۲ تا ۳ برابری نسبت به قفل دارند.

- بنابراین متغیرهایی از نوع *Atomic هر دو خاصیت lock را با هزینه کمتر دارا هستند.

- با وجود این کلاسها چرا به متغیرهای قفل نیاز داریم؟
- زمانی که قرار است یک کار از چندین متغیر استفاده کند و یک روند و یا یک قطعه کد است نه فقط بروز رسانی یک متغیر باید از قفل استفاده شود.

استفاده از مکانیزم Thread Interruption

- اگر در مثال اتمام کار ریسمان، اجرای ریسمان مثلا در یک شرط و یا یک تابع blocking در حالت waiting باشد ریسمان هیچگاه به ابتدای حلقه نمی‌رسد که مقدار finish را بخواند.

```
volatile boolean finish = false;
Lock lock;

public void run(){

    while(!finish){

      lock.lock();
      while(counter == 0)
           condition.await();

      ....
    }
}
```

استفاده از مکانیزم Thread Interruption

- در زبان جاوا برخی متدهای blocking در کتابخانه ریسمان از interruption پشتیبانی می‌کنند.
  - interruption یک راهکار همکارانه برای بیدار کردن ریسمان از وضعیتی است که هست و بیان این نکته است که ریسمان باید کار فعلی را خاتمه دهد.
  - البته لزوما interruption برای خاتمه دادن و یا لغو کردن کار فعلی نیست ولی برای این کار قابل استفاده است.
- هر ریسمان دارای یک فیلد volatile است که نشان میدهد ریسمان interrupt شده است یا خیر و با متد isInterrupted قابل خواندن است.
- برای ارسال وقفه به یک ریسمان می‌توان از متد interrupt استفاده کرد.
- برای پاک کردن حالت interrupt هم می‌توان از متد interrupted استفاده کرد که مقدار قبلی را برگردانده و flag را پاک می‌کند.

استفاده از مکانیزم Thread Interruption

- متدهای sleep, join و wait (در Object) و await در condition از جمله متدهایی هستند که از interruption پشتیبانی می‌کنند.
- استاندارد JVM مشخص نکرده است که این کار به چه نحو و با چه سرعتی انجام می‌شود ولی عموما سریع اتفاق می‌افتد.
- ریسمان از حالت بلاک خارج شده،flag را پاک کرده و InterruptedException را پرتاب می‌کند.
- اگر ریسمان در حال اجرا باشد یعنی در حالت wait نباشد فراخوانی تابع interrupt کاری نمی‌کند ولی مقدار flag را true میکند.
- برای استفاده از این چنین شرایطی برنامه‌نویس باید بطور مشخص در بخشهایی از کد از isInterrupted استفاده کند.
- اگر از ThreadPool استفاده می‌کنید در Future هم تابع cancel وجود دارد که دقیقا از همین روش برای لغو کردن اجرای یک کار استفاده می‌کند.

استفاده از مکانیزم Thread Interruption

```
public class PrimeProducer extends Thread {
    private final BlockingQueue<BigInteger> queue;

    PrimeProducer(BlockingQueue<BigInteger> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            BigInteger p = BigInteger.ONE;
            while (!Thread.currentThread().isInterrupted()) //not clears flag
                queue.put(p = p.nextProbablePrime());

            //do something

            if(Thread.interrupted()) //clears interrupt flag
                return;

            //do other things
        }
        catch (InterruptedException consumed) {
            /*  Allow thread to exit  */
        }
    }

    public void cancel() {
        interrupt();
    }
}
```