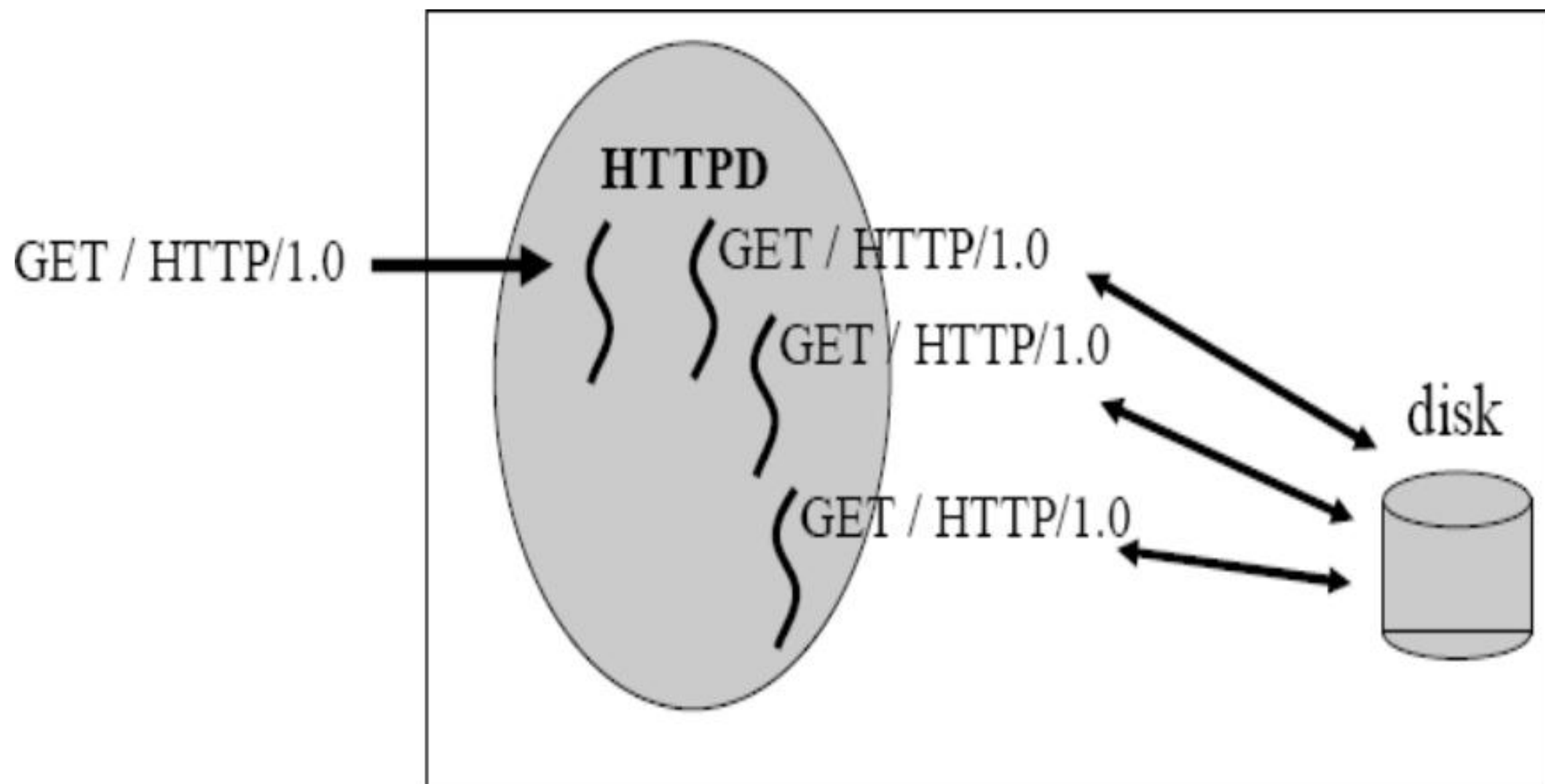


第二十六章——并发 & 线程 API 简介

刘 玉 峰

Fx_163.com 湖南
大学

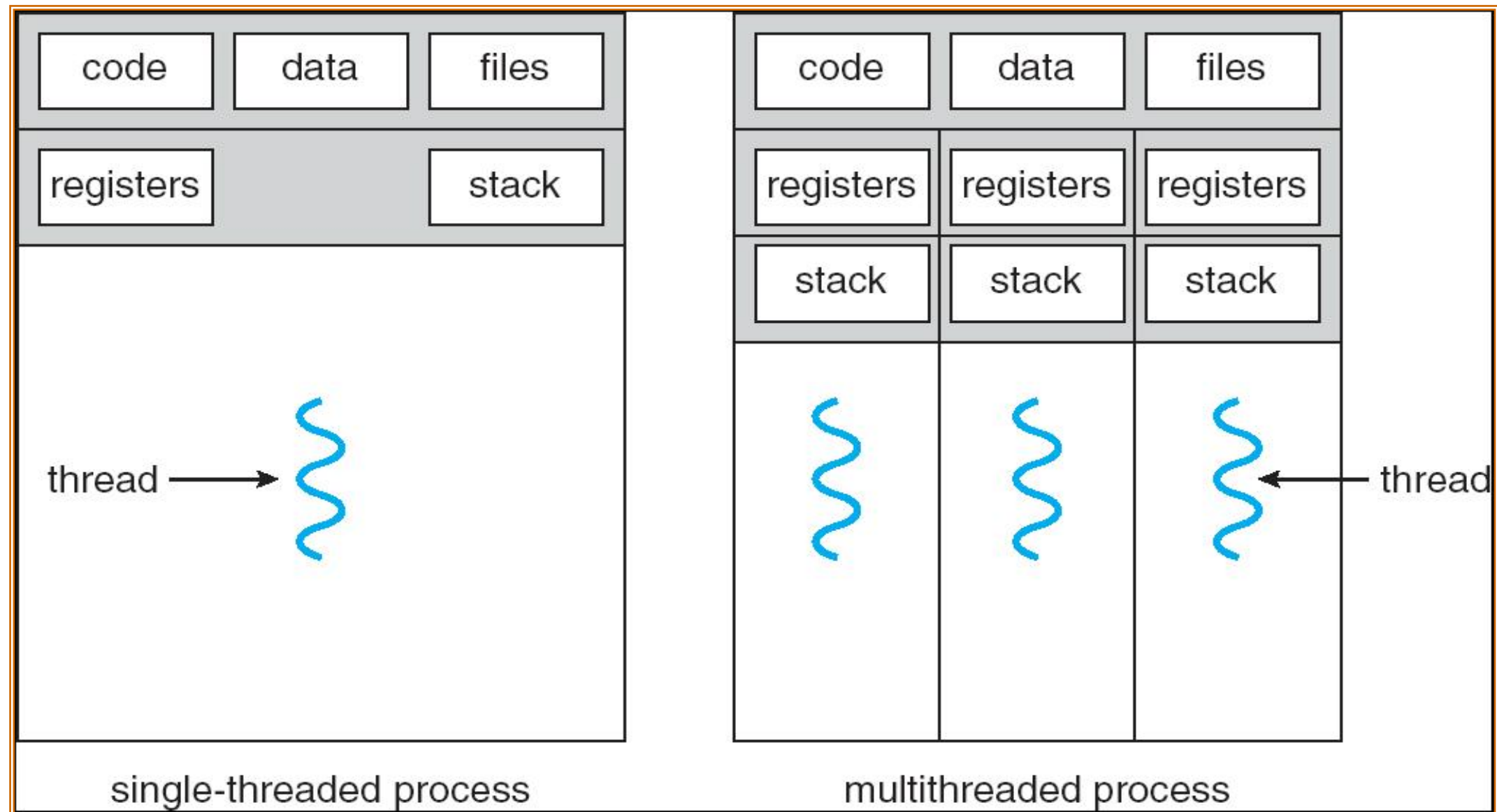
为什么 我们需要 线程？



线程

- 多线程程序具有多于一个的执行点（即，多台PC，其中每台PC都从其获取和执行）
- 每个线程都非常像一个独立的进程，除了一个区别：它们**共享相同的地址空间**，因此可以访问**相同的数据**。
- 在上下文中，与进程相比，线程之间的切换：**地址空间保持相同**（即，不需要切换我们正在使用的页表）。

单线程 和多线程 进程



- 属于同一个进程的线程共享代码段、数据段和其他操作系统资源，但它有自己的线程ID、程序计数器、寄存器和栈。
- **线程是CPU使用的基本单位。**

- 线程和进程之间的另一个主要区别涉及堆栈。

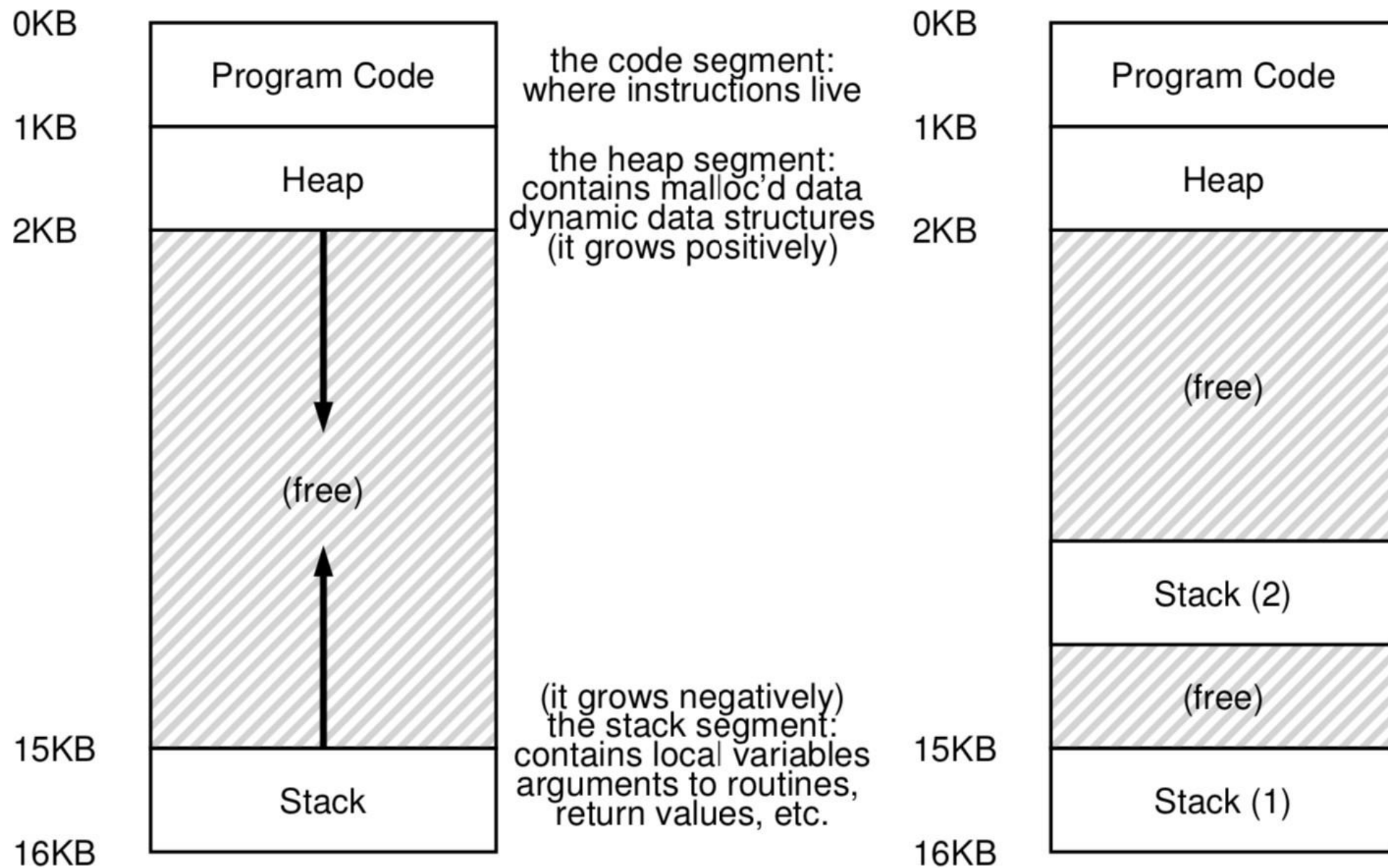


Figure 26.1: Single-Threaded And Multi-Threaded Address Spaces

线程

线程与进程的比较

1 调度

同一进程的多线程间调度时，不引起进程的切换

不同进程的线程间调度，需要进程切换

线程上下文切换和进程上下文切换一个最主要的区别是线程的切换虚拟内存空间依然是相同的，但是进程切换是不同的。因此，进程切换需要切换页表以使用新的地址空间。而线程切换不需要，

2 并发性

一个进程的多个线程之间可并发执行

3 资源的拥有

线程不拥有系统资源，不拥有代码段、数据段。

线程 API

Thread Creation

- When a program is started, a single thread (called **initial thread** or **main thread**) is created
- Additional threads are created by

```
#include <pthread.h>
```

Return 0 if OK, positive *Exxx* value on error

```
int pthread_create (pthread_t *tid, const pthread_attr_t *attr,  
                    void *(*func)(void *), void *arg);
```

tid: pointer to ID of the created thread
attr: pointer to the attribute of the thread; **NULL** to take the default
func: pointer to the function for the created thread to execute
arg: pointer to the data passed to *func*

第一个参数 `thread` 是指向 `pthread_t` 结构类型的指针，将来可以利用这个结构与该线程交互。

第二个参数 `attr` 用于指定该线程可能具有的属性。在大多数情况下，默认值传入 `NULL`。

第三个参数 函数名称（`start_routine`），它被传入一个类型为 `void *` 的参数（`start_routine` 后面的括号表明了这一点），并且它返回一个 `void *` 类型的值（即一个 `void` 指针）。

第四个参数 `arg` 就是要传递给线程开始执行的函数的参数

线程 创建 （续）

- 如果 `start_routine` 需要另一个类型参数，则声明将如下所示：
 - 整数 参数：

整数

```
pthread_create (... , //前两个参数相同  
                void * (*start_routine)  
                (int) , int arg) ;
```

- 返回 一个整数：

整数

```
pthread_create (... , //前两个参数相同  
                int (*start_routine)  
                (void*) , void* arg) ;
```


示例： 创建线程

```
#include<pthread.h>

inta_ ; int b;

                                int b;
                                } int t;

void *mythread ( void *arg)
{ myarg_t *m = (myarg_t *)
arg;printf ("%d %d\n", m-> a,
m-> b) ;returnNULL;
}

int main ( int argc, char*argv[])
{ pthread_t p;
int c;

myarg_t args;
args.a =10;
int b = 20;
return pthread_create (&p, NULL, mythread,
&args) ;
...
}
```

线程 API

pthread_join Function

- Wait for a given thread to terminate

Return 0 if OK, positive *Exxx* value on error

```
#include <pthread.h>
```

```
int pthread_join (pthread_t tid, void **status);
```

tid: thread ID

status: if non-null, the return value from the thread (which is a void pointer) is stored in the location pointed to by *status*

Process

Thread

fork	↔	pthread_create
waitpid	↔	pthread_join

当A线程调用线程B并 pthread_join() 时，A线程会处于阻塞状态，直到B线程结束后，A线程才会继续执行下去。

第一个是 pthread_t 类型，用于指定要等待的线程。这个变量是由 线程创建函数初始化的（当你将一个指针作为参数传递给 pthread_create() 时）。如果你保留了它，就可以用它来等待该线程终止。

第二个参数是一个指针，指向你希望得到的返回值。

示例： 等待线程 完成

```
1  #include<stdio.h>
2  #include<pthread.h>
3  #include<assert.h>
4#include<stdlib.h> 5
6  typedef struct ____myargt {
7      int a;
8      int b;
9} myarg_t; 10
11 typedef struct ____myret_t {
12     整数 x;
13     int i;
14} myret_t;
15
16 public void onDestinyCode ( void
    onDestiny) {
17myarg_t *m = (myarg_t *) arg;18
    printf ( "%d %d\n", m-> a, m->
b) ;
19    myret_t =malloc (sizeof (myret_t) ) ;
20    r-> x =1;
21    r-> y =2;
22    return ( void *)
r; 23}
24
```

示例： 等待 线程完成 (续)

```
25  public int findDuplicate ( int []  
    nums) {  
26      int c;  
27      pthread_ t;  
28myret_t *m;29  
30      myarg_t args;  
31      args.a = 10;  
32      int b = 20 ;  
33      pthread_create (&p, NULL, mythread, &args) ;  
34      pthread_join (p, ( void **) & m) ;//此线程已被  
        // 在pthread_join () 例程中等待。  
35      printf ( "返回%d %d\n", m-> x, m-> y) ;  
36返回 0 ; 37      }
```

示例： 危险 代码

- 注意 线程返回值的方式 。

```
1  public void onDestinyCode
   ( voidonDestiny) {
2      myarg_t *m= (myarg_t*)    arg;
3      printf ( "%d %d\n", m-> a, m-> b) ;
4      myret_t r; //分配          堆叠 :          糟糕!
5      r.x = 1 ;
6      r.y = 2 ;
7      return ( void*)    &r;
8  }
```

- 当变量r 返回时，它被自动释放。

线程 API

pthread_self Function

- A thread fetches the thread ID for itself


```
#include <pthread.h>
```

```
pthread_t pthread_self (void);
```

Return: thread ID of calling thread

Process

Thread

getpid  pthread_self

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <pthread.h>
4
5  void *mythread(void *arg) {
6      printf("%s\n", (char *) arg);
7      return NULL;
8  }
9
10 int
11 main(int argc, char *argv[]) {
12     pthread_t p1, p2;
13     int rc;
14     printf("main: begin\n");
15     rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
16     rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
17     // join waits for the threads to finish
18     rc = pthread_join(p1, NULL); assert(rc == 0);
19     rc = pthread_join(p2, NULL); assert(rc == 0);
20     printf("main: end\n");
21     return 0;
22 }

```

图 26.2 简单线程创建代码 (t0.c)

- 线程创建有点像进行函数调用。
- 然而，并不是首先执行函数然后返回给调用者，而是为被调用的例程创建一个新的执行线程，它可以独立于调用者运行。

表 26.1

线程追踪 (1)

主程序	线程 1	线程 2
开始运行 打印 “main:begin” 创建线程 1 创建线程 2 等待线程 1		
	运行 打印 “A” 返回	
等待线程 2		
		运行 打印 “B” 返回
打印 “main:end”		

表 26.2

线程追踪 (2)

主程序	线程 1	线程 2
开始运行 打印 “main:begin” 创建线程 1		
	运行 打印 “A” 返回	
创建线程 2		
		运行 打印 “B” 返回
等待线程 1 <i>立即返回, 线程 1 已完成</i> 等待线程 2 <i>立即返回, 线程 2 已完成</i> 打印 “main:end”		

表 26.3

线程追踪 (3)

主程序	线程 1	线程 2
开始运行 打印 “main:begin” 创建线程 1 创建线程 2		
		运行 打印 “B” 返回
等待线程 1		
	运行 打印 “A” 返回	
等待线程 2 <i>立即返回, 线程 2 已完成</i> 打印 “main:end”		

- 线程 使生活变得复杂： 已经很难说什么时候会运行！
- 如果没有并发性，计算机就很难理解。不幸的是，随着**并发性**，它只是变得更糟。**更糟**。

另一个例子

```

1  #include <stdio.h>
2  #include <pthread.h>
3  #include "common.h"
4  #include "common_threads.h"
5
6  static volatile int counter = 0;
7
8  // mythread()
9  //
10 // Simply adds 1 to counter repeatedly, in a loop
11 // No, this is not how you would add 10,000,000 to
12 // a counter, but it shows the problem nicely.
13 //
14 void *mythread(void *arg) {
15     printf("%s: begin\n", (char *) arg);
16     int i;
17     for (i = 0; i < 1e7; i++) {
18         counter = counter + 1;
19     }
20     printf("%s: done\n", (char *) arg);
21     return NULL;
22 }
23
24 // main()
25 //
26 // Just launches two threads (pthread_create)
27 // and then waits for them (pthread_join)
28 //
29 int main(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("main: begin (counter = %d)\n", counter);
32     Pthread_create(&p1, NULL, mythread, "A");
33     Pthread_create(&p2, NULL, mythread, "B");
34
35     // join waits for the threads to finish
36     Pthread_join(p1, NULL);
37     Pthread_join(p2, NULL);
38     printf("main: done with both (counter = %d)\n",
39           counter);
40     return 0;
41 }

```

volatile告诉编译器对该变量不做优化，都会直接从变量内存地址中读取数据，从而可以提供对特殊地址的稳定访问。

向共享变量计数器添加一个数字，并在循环中执行 1000 万（ 10^7 ）次。因此，预期的最终结果是：200000000。

期望的结果

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

可能的结果

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

为什么 它变得更糟： 共享数据

- 计算机 应该 产生 确定性的结果。
- 在 本例中，输入很小，我们得到了确定性结果。但是大的输入总是 导致错误的 结果。
- 不仅 每次 运行都是错误的， 而且 产生不同的结果！
- 一个大 问题仍然存在： 为何 会出现这种情况？

问题的核心：不受控制的调度

- 要理解为什么会发生这种情况，我们必须理解编译器为更新 **计数器而生成的代码序列**。

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

竞态 条件

- 双 线程示例
 - counter = counter + 1 （默认 值为 50）
 - 我们 希望结果是52。 然而

操 作 系 统	阅 读 1	T read2	(在指示之后) PC %eax 计数器		
	临界区前		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	添加\$0x1, %eax		108	51	50
中断	保存T1的状态				
	恢复T2的状态		100	0	50
		移动0x8049a1c, %eax	105	50	50
		添加\$0x1, %eax	108	51	50
		移动%eax, 0x8049a1c	113	51	51
中断	保存T2的状态				
	恢复T1的状态		108	51	50
		mov %eax, 0x8049a1c	113	51	51

- 我们 在这里演示 的内容 称为 **竞争 条件**（或者更具体地说，**数据 竞争**）：结果取决于代码 的执行 时间。
- 运气 不好的话（即，在执行中的不合时宜的点 发生的上下文切换），我们 得到错误的 结果。事实上，我们可能每次都**得到不同的结果**；
- 因此，我们称这个结果为不确定的，而 不是 一个 很好的**确定性** 计算（我们习惯于从计算机），其中 不知道 输出 将是什么，并且它 确实 可能 在运行中不同。

原子性的 愿望

- 假设此指令将值添加到存储器位置，并且硬件保证其 **原子地执行**。

```
memory-add 0x8049a1c, $0x1
```

- 原子，在这种情况下，意味着“**作为一个单位**“，有时我们采取的“**全部或没有**。”我们想要的是原子地执行三个指令序列。

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

- What we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**（同步原语）。

提示：使用原子操作

原子操作是构建计算机系统的最强大的基础技术之一，从计算机体系结构到并行代码（我们在这里研究的内容）、文件系统（我们将很快研究）、数据库管理系统，甚至分布式系统[L+93]。

将一系列动作原子化（atomic）背后的想法可以简单用一个短语表达：“全部或没有”。看上去，要么你希望组合在一起的所有活动都发生了，要么它们都没有发生。不会看到中间状态。有时，将许多行为组合为单个原子动作称为事务（transaction），这是一个在数据库和事务处理世界中非常详细地发展的概念[GR92]。

在探讨并发的主题中，我们将使用同步原语，将指令的短序列变成原子性的执行块。但是我们会看到，原子性的想法远不止这些。例如，文件系统使用诸如日志记录或写入时复制等技术来自动转换其磁盘状态，这对于在系统故障时正确运行至关重要。如果不明白，不要担心——后续某章会探讨。

我们能 学到什么？

竞赛 条件

- Race condition（竞争条件）：The situation where **several processes access – and manipulate shared data concurrently.**
共享数据的最终值 取决于哪个进程最后完成。
- 当竞争条件存在时，进程处理结果可能失效甚至发生错误
- 在同一个应用程序中运行多个线程（进程）本身并不会引起问题。当多个线程访问相同的资源时才会出现问题。比如多个线程访问同一块内存区域（变量、数组、或对象）、系统（数据库、web 服务等）或文件。事实上，**只有一个或多个线程（进程）改写这些资源时才会出现问题**。多个线程（进程）只读取而不会改变这些相同的资源时是安全的。

临界区问题

- **critical section(临界区)** 访问共享资源的一段代码，资源通常是一个变量或数据结构,或者说导致竞争条件发生的代码片段就叫临界区。
- 问题 - 确保 **当一个进程在其临界区执行时，不允许其他进程在其临界区执行。**
- **进程同步**指两个以上进程基于**某个条件**来协调它们的活动

第二十七章——并发问题

- 访问共享变量，因此需要为 临界区支持原子性。
- 另一种常见的交互，即一个线程在继续之前必须等待另一个线程完成某些操作。

解决问题的初步 尝试

- 只有 2 个进程， P_0 和 P_1
- 进程 P_i (其他进程 P_j) 的一般结构
做什么

入口 段

临界 截面

出口 截面

提醒 段

} while (1) ;

- 进程 可以 共享 一些 公共 变量 来同步它们的
动作。

洛 克 斯

- 确保 任何 这样的临界区 都 像 单个原子指令一样执行（以原子方式执行一系列指令）。

```
1    lock_tmutex;  
2    . . 我 的天  
3    public void  
4    incr count = count + 1; → 临界 截面  
5    unlock (&) ;
```

锁变量（简称锁）保存了锁在某一时刻的状态。它要么是可用的（available，或 unlocked，或 free），表示没有线程持有锁，要么是被占用的（acquired，或 locked，或 held），表示有一个线程持有锁，正处于临界区。

- 线程调用 `lock()` 尝试获取锁，如果没有其他线程持有锁（即它是可用的），该线程会获得锁，进入临界区。这个线程有时被称为锁的持有者（owner）。
- 如果另外一个线程对相同的锁变量（本例中的 `mutex`）调用 `lock()`，因为锁被另一线程持有，该调用不会返回。这样，当持有锁的线程在临界区时，其他线程就无法进入临界区。
- 锁的持有者一旦调用 `unlock()`，锁就变成可用了。如果没有其他等待线程（即没有其他线程调用过 `lock()` 并卡在那里），锁的状态就变成可用了。如果有等待线程（卡在 `lock()` 里），其中一个会（最终）注意到（或收到通知）锁状态的变化，获取该锁，进入临界区。

补充：关键并发术语

临界区、竞态条件、不确定性、互斥执行

这 4 个术语对于并发代码来说非常重要，我们认为有必要明确地指出。请参阅 Dijkstra 的一些早期著作[D65, D68]了解更多细节。

- 临界区（critical section）是访问共享资源的一段代码，资源通常是一个变量或数据结构。
- 竞态条件（race condition）出现在多个执行线程大致同时进入临界区时，它们都试图更新共享的数据结构，导致了令人惊讶的（也许是不希望的）结果。
- 不确定性（indeterminate）程序由一个或多个竞态条件组成，程序的输出因运行而异，具体取决于哪些线程在何时运行。这导致结果不是确定的（deterministic），而我们通常期望计算机系统给出确定的结果。
- 为了避免这些问题，线程应该使用某种互斥（mutual exclusion）原语。这样做可以保证只有一个线程进入临界区，从而避免出现竞态，并产生确定的程序输出。

关键问题：如何实现同步

为了构建有用的同步原语，需要从硬件中获得哪些支持？需要从操作系统中获得什么支持？如何正确有效地构建这些原语？程序如何使用它们来获得期望的结果？

洛 克 斯

- POSIX 库将锁称为互斥量（mutex），因为它被用来提供线程之间的互斥。即当一个线程在临界区，它能够阻止其他线程进入直到本线程离开临界区。
- 接口

```
int pthread_mutex_lock (pthread_mutex_t  
*mutex) ; int pthread_mutex_unlock  
(pthread_mutex_t*mutex) ;
```

- 用法（无锁初始化和错误检查）

```
pthread_mutex_t 锁;  
pthread_mutex_lock (&lock) ;  
x = x + 1; //或者你的临界区是 pthread_mutex_unlock  
(&lock) ;
```

这段代码有两个重要的问题。第一个问题是缺乏正确的初始化（lack of proper initialization）。所有锁必须正确初始化，以确保它们具有正确的值，并在加锁和解锁时按照需要工作。

第二个问题是在调用获取锁和释放锁时没有检查错误代码。就像 UNIX 系统中调用的任何库函数一样，这些函数也可能会失败！

Locks (初始化)

- 所有 锁必须正确初始化。
- 静态初始化: 使用 `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
```

- 动态初始化: 调用 `pthread_mutex_init()`

```
int rc =pthread_mutex_init (& lock,  NULL) ;  
assert (rc ==0 ) ; //总是检查 成功!
```

锁（续）

- 调用 锁定 和 解锁时检查错误代码
- 示例 包装器

```
// 使用这个来保持代码干净，但检查 失败
// 只有在退出程序失败时才可以使用void
pthread_mutex_lock (pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock
        (mutex); return (0) ;
}
```

- 这两个调用用于锁获取

```
int pthread_mutex_trylock (pthread_mutex_t*mutex) ;
int pthread_mutex_timelock (pthread_mutex_t*mutex,
                            struct timespec *abs_timeout) ;
```

Trylock和timelock这两个调用用于获取锁。如果锁已被占用，则 trylock 版本将失败，或者说`pthread_mutex_trylock ()` 是 `pthread_mutex_lock ()` 的非阻塞版本。获取锁的 timedlock 定版本会在超时或获取锁后返回，以先发生者为准。因此，具有零超时的 timedlock 退化为 trylock 的情况。

结束

