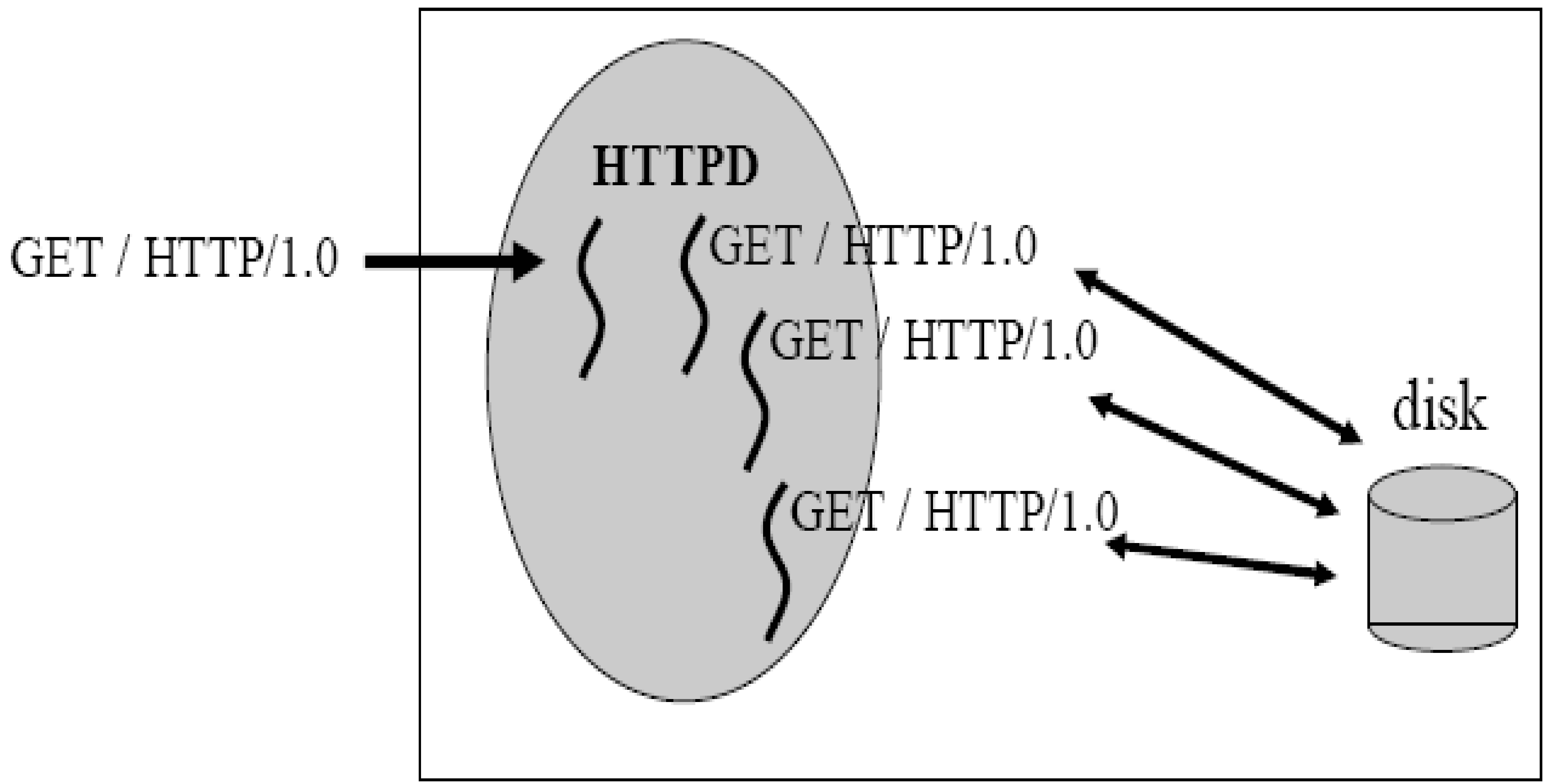
Concurrency
Introduction & Thread API

Liu yufeng

Fx_yfliu@163.com

Hunan University

Why we need threads?



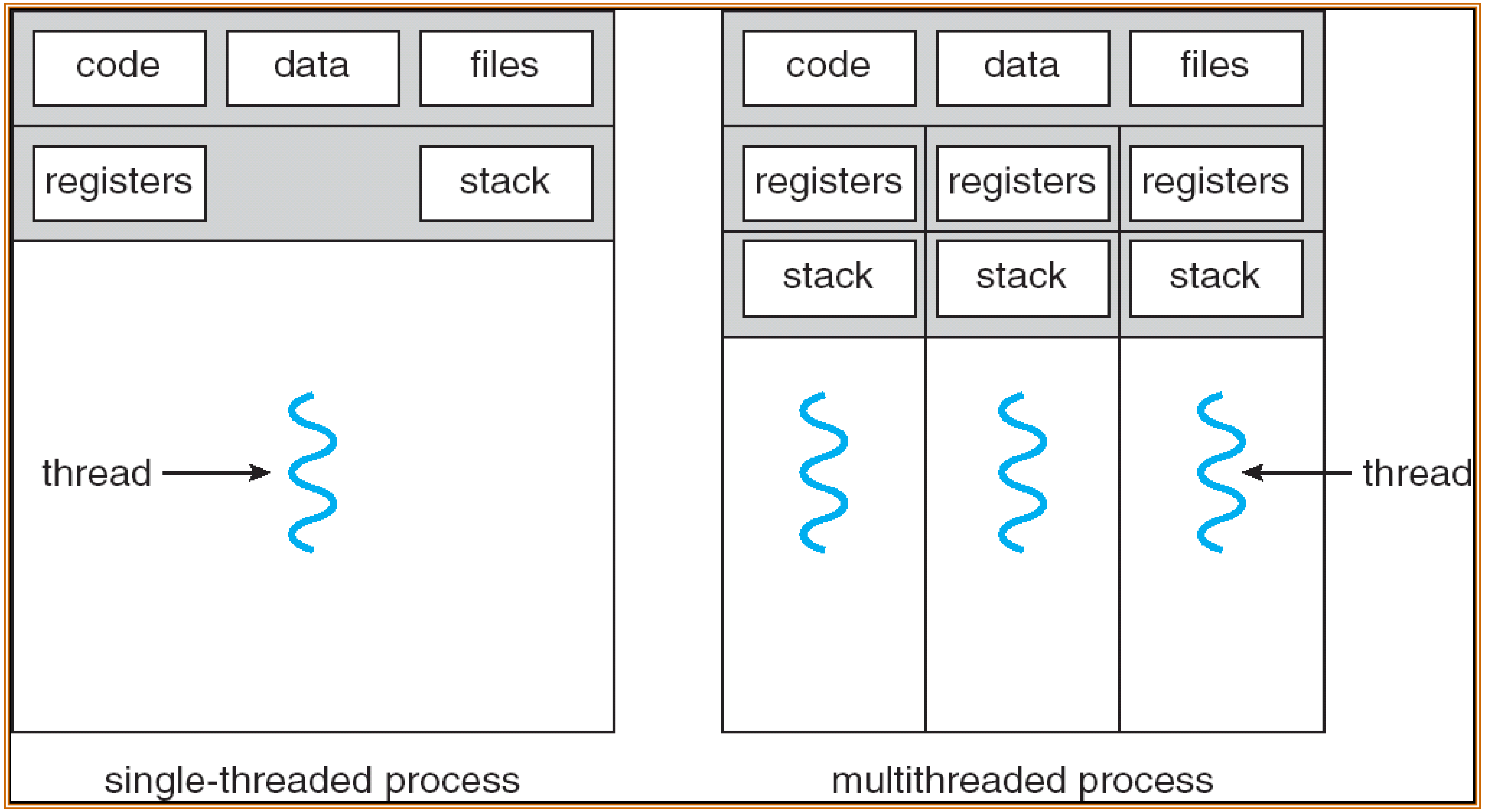
Thread

- A **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from)

- Each thread is very much like a separate process, except for one difference: they **share the same address space** and thus can access **the same data**.

- In the **context switch** between threads as compared to processes: the **address space remains the same** (i.e., no need to switch which page table we are using).

Single and Multithreaded Processes



- 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。

- One other major **difference** between threads and processes concerns **the stack**.

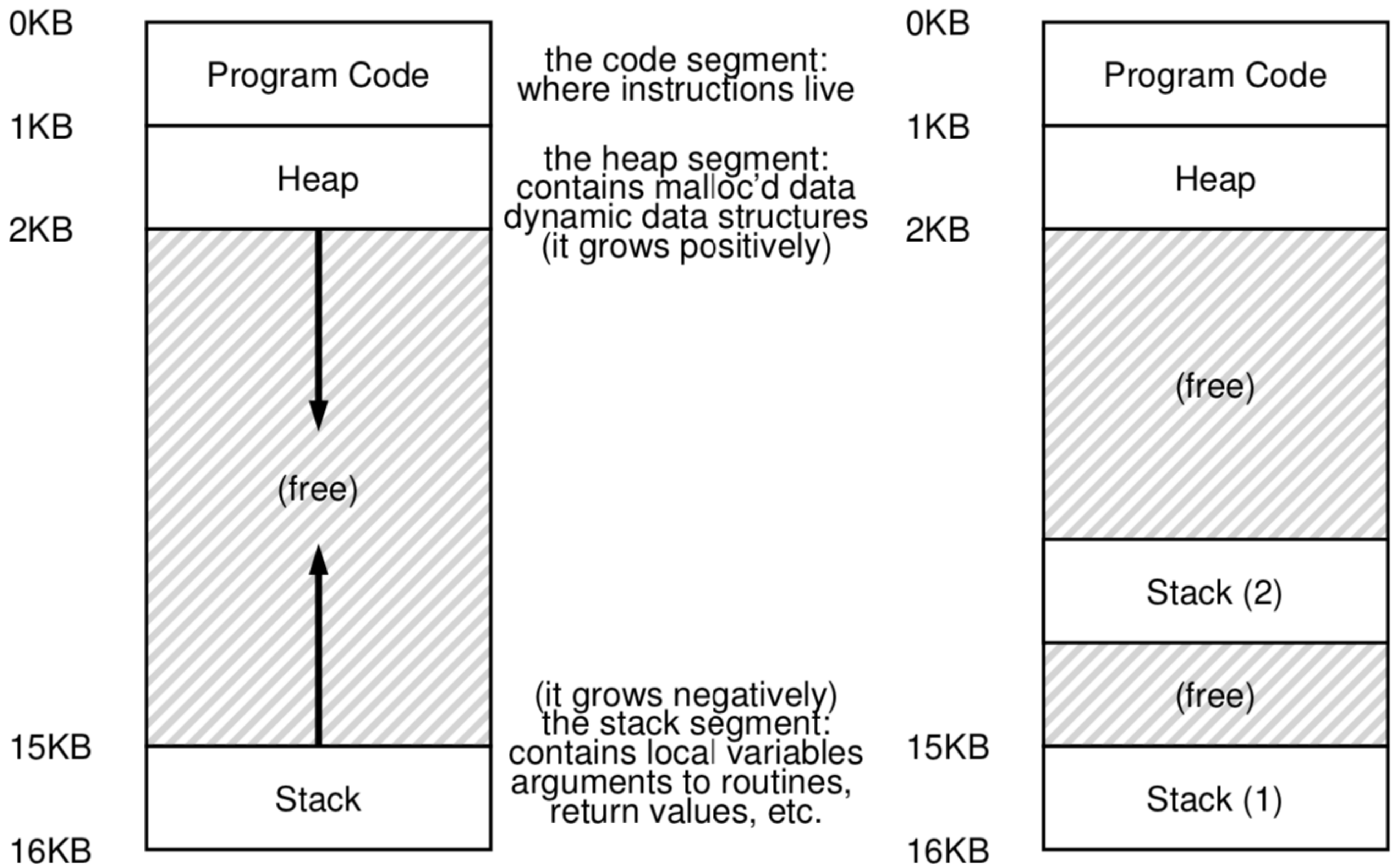


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

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问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。

Thread API

Thread Creation

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

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

```
#include <pthread.h>
int pthread_create (pthread_t *tid, const pthread_attr *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

Thread Creation (Cont.)

- If `start_routine` instead required another type argument, the declaration would look like this:
  - An integer argument:

```
int
pthread_create(…, // first two args are the same
               void*   (*start_routine)(int),
               int     arg);
```

  - Return an integer:

```
int
pthread_create(…, // first two args are the same
               int   (*start_routine)(void*),
               void*    arg);
```

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    return NULL;
}

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

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

Thread 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。

Example: Waiting for Thread Completion

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Waiting for Thread Completion (Cont.)

```
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
         // waiting inside of the pthread_join() routine.
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Example: Dangerous code

- Be careful with <u>how values are returned</u> from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

- When the variable `r` returns, it is automatically de-allocated.

Thread API

pthread_self Function

- A thread fetches the thread ID for itself

```
#include <pthread.h>
pthread_t pthread_slef (void);
                         Return: thread ID of calling thread
```

Process		Thread
getpid	⟷	pthread_self

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

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 *立即返回，线程 1 已完成* 等待线程 2 *立即返回，线程 2 已完成* 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 *立即返回，线程 2 已完成* 打印“main:end”		

- Threads make life complicated: it is already hard to tell what will run when!

- Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it simply gets worse. **Much worse**.

Another Example

```c
#include <stdio.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

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：20000000。

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)
```

Why It Gets Worse: Shared Data

- Computers are supposed to produce **deterministic** results.

- In the example, **with small input**, we have deterministic results. But **large input always** results in **wrong** results.

- Not only is each run wrong, but also yields a **different** result!

- **A big question remains: why does this happen**?

The Heart of the Problem: Uncontrolled Scheduling

- To understand why this happens, we must understand the code sequence that the compiler generates for the update to **counter**.

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

Race condition

- Example with two threads
  - counter = counter + 1 (default is 50)
  - We expect the result is 52. However,

OS	Thread1	Thread2	PC	%eax	counter
			(after instruction)		
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	50	50
	add $0x1, %eax		108	51	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	50	50
		add $0x1, %eax	108	51	50
		mov %eax, 0x8049a1c	113	51	51
interrupt					
save T2's state					
restore T1's state			108	51	50
	mov %eax, 0x8049a1c		113	51	**51**

- What we have demonstrated here is called a **race condition** (or, more specifically, a **data race**): the results depend on the timing execution of the code.

- With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may **get a different result each time**;

- thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

The Wish For Atomicity

- Assume this instruction adds a value to a memory location, and the hardware guarantees that it executes **atomically**.

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

- Atomically, in this context, means "**as a unit**", which sometimes we take as "**all or none**." What we'd like is to execute the three instruction sequence atomically.

```
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。

What We Can Learn?

Race Condition

- Race condition（竞争条件）: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- 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。

The Critical-Section Problem

- 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。
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- 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

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。

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Locks

- Ensure that any such critical section executes as if it were a single atomic instruction (**execute a series of instructions atomically**).

```
lock_t mutex;
. . .
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Critical section

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？

Locks

- 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。
- Interface

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

- Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
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)

- All locks must be properly initialized.
  - 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); // always check success!
```

Locks (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition

```
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。

End