中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

进程与线程II

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

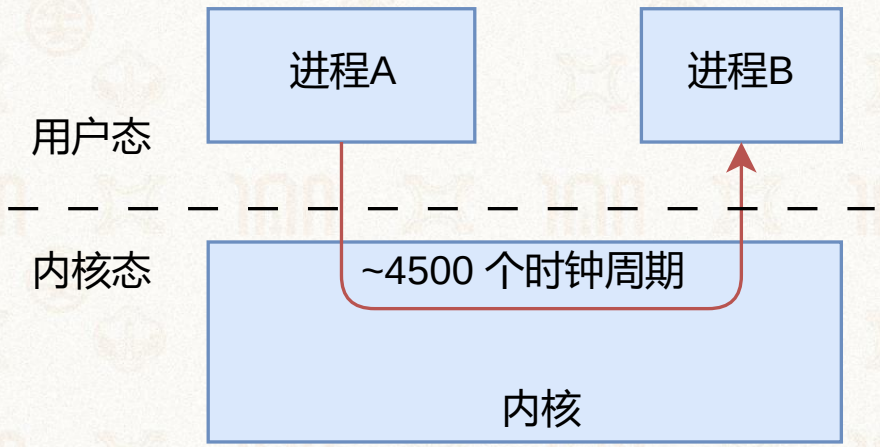
大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作
- 线程
  - 线程的概念
  - 线程模型
  - 相关数据结构
  - 基本操作
  - 上下文切换
- 纤程
  - 纤程的概念
  - 编程模型
  - Windows和编程语言支持
  - Go语言简介与协程

为什么需要线程?

- 创建进程的开销较大
  - 包括了数据、代码、堆、栈等
- 进程的隔离性过强
  - 进程间交互：可以通过进程间通信（IPC），但开销较大
- 进程内部无法支持并行

同步进程间调用



线程：更加轻量级的运行时抽象

- 线程只包含运行时的状态
  - 静态部分由进程提供
  - 包括了执行所需的最小状态（主要是寄存器和栈）

- 一个进程可以包含多个线程
  - 每个线程共享同一地址空间（方便数据共享和交互）
  - 允许进程内并行

进阶模型：多线程的进程

- 一个进程可以包含多个线程

- 一个进程的多线程可以在不同处理器上同时执行
  - 调度的基本单元由进程变为了线程
  - 每个线程都有状态
  - 上下文切换的单位变为了线程

多线程进程的地址空间

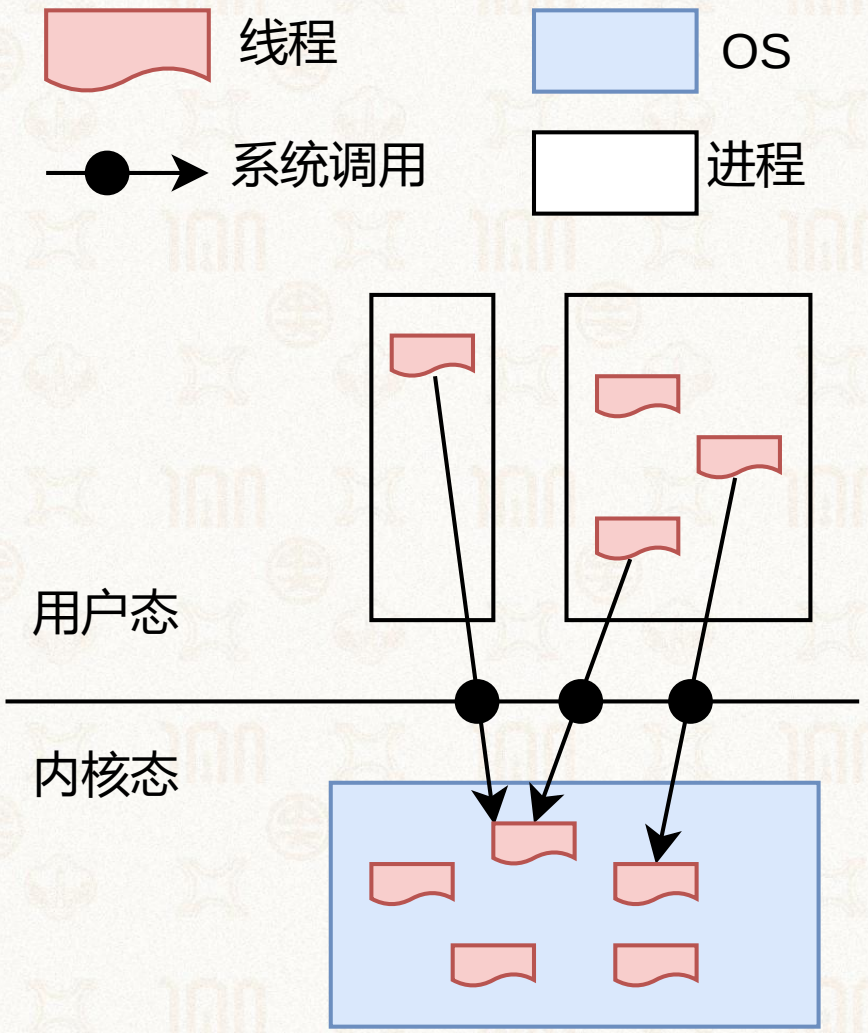
- 每个线程拥有自己的栈
- 内核中也有为线程准备的内核栈
- 其它区域共享
  - 数据、代码、堆……

用户态线程与内核态线程

- 根据线程是否受内核管理，可以将线程分为两类
  - 内核态线程：内核可见，受内核管理
  - 用户态线程：内核不可见，不受内核直接管理

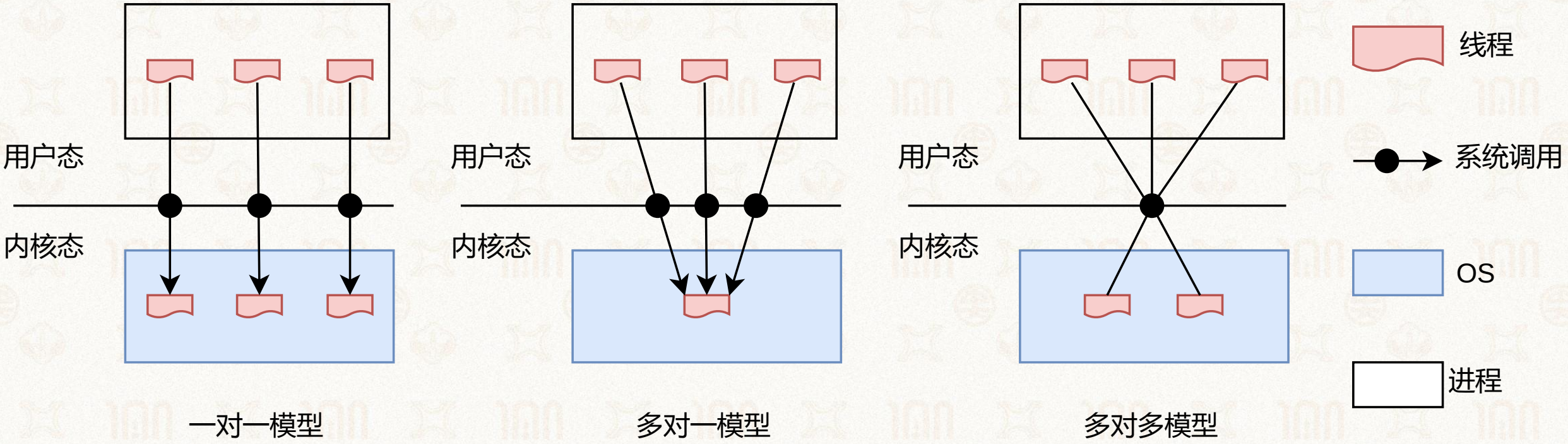
- 内核态线程
  - 由内核创建，线程相关信息存放在内核中

- 用户态线程（纤程）
  - 在应用态创建，线程相关信息主要存放在应用数据中



线程模型

- 线程模型表示了用户态线程与内核态线程之间的联系
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程

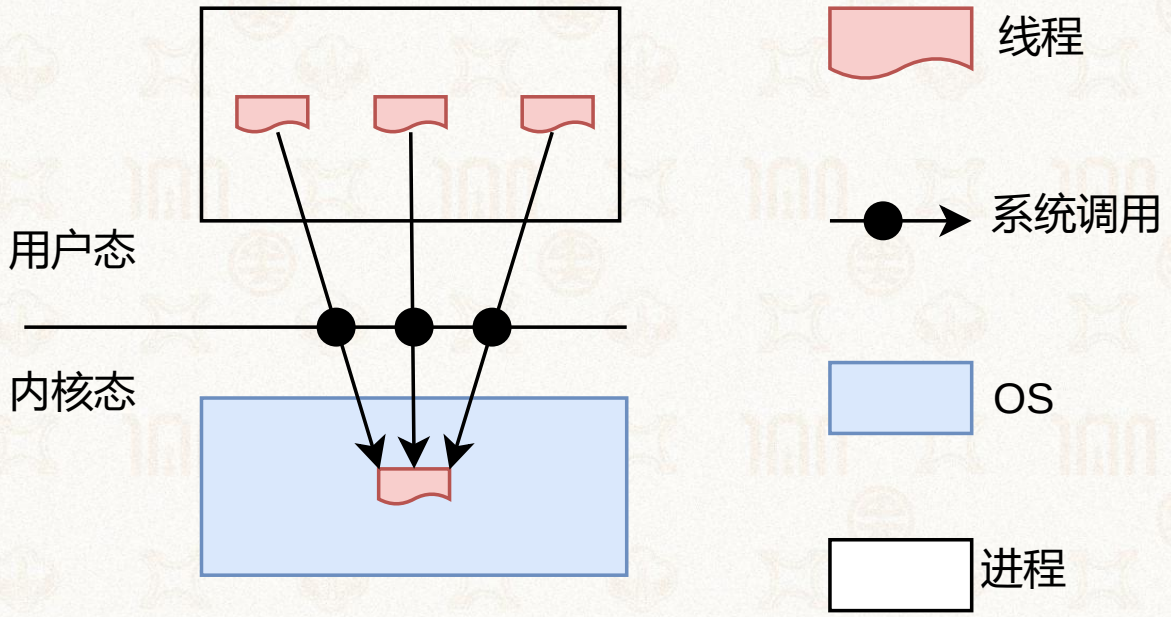


一对一模型　　多对一模型　　多对多模型

多对一模型

- 将多个用户态线程映射给单一的内核线程
- 优点：内核管理简单
- 缺点：可扩展性差，无法适应多核机器的发展

- 在主流操作系统中被弃用
- 用于各种用户态线程库中

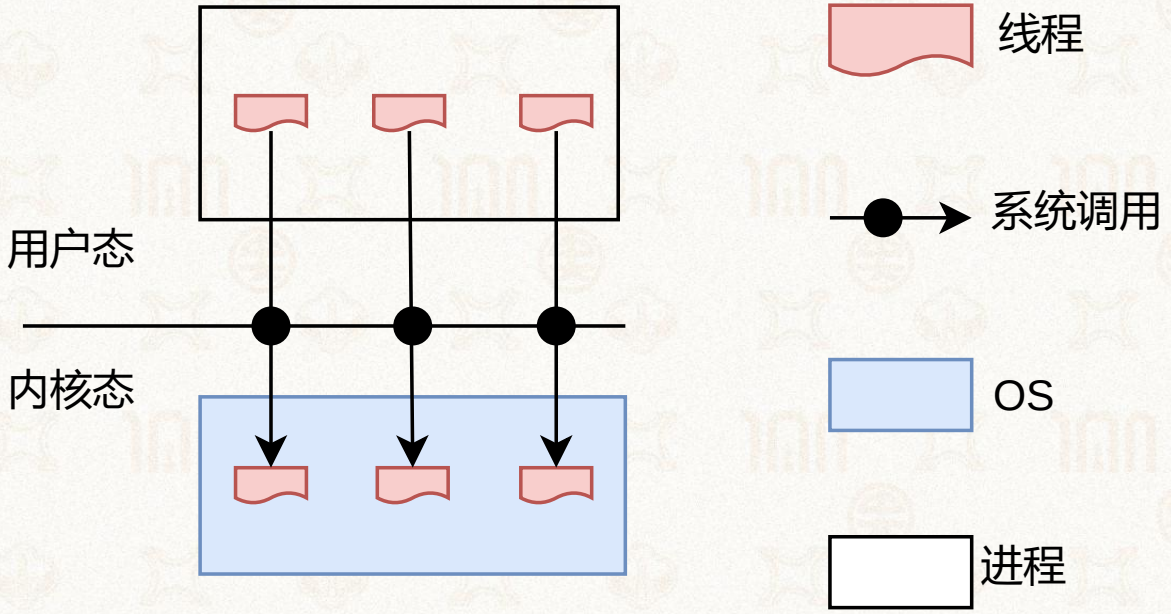


多对一模型

一对一模型

- 每个用户线程映射单独的内核线程
  - 优点：解决了多对一模型中的可扩展性问题
  - 缺点：内核线程数量大，开销大

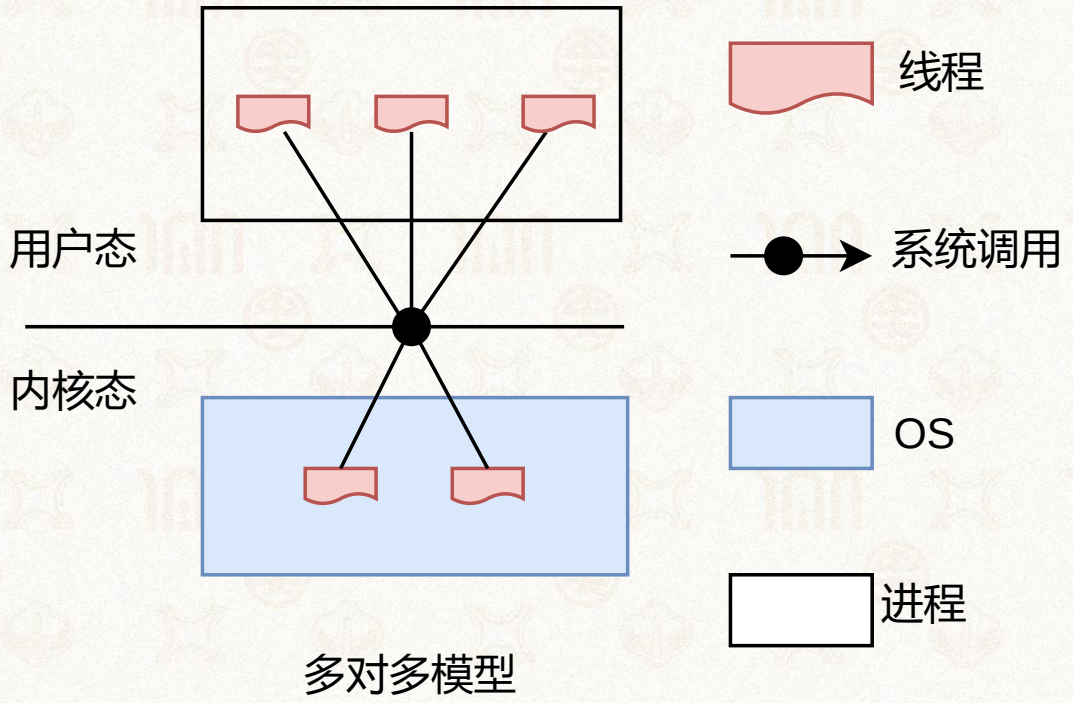
- 主流操作系统都采用一对一模型
  - Windows、Linux、OS X……



一对一模型

多对多模型（又叫Scheduler Activation）

- N个用户态线程映射到M个内核态线程（N > M）
  - 优点：解决了可扩展性问题（多对一）和线程过多问题（一对一）
  - 缺点：管理更为复杂
- Solaris在9之前使用该模型
  - 9之后改为一对一
- 在虚拟化中得到了广泛应用



多对多模型

线程的相关数据结构：TCB

- 一对一模型的TCB可以分为两部分
- 内核态：与PCB结构类似
  - Linux中进程与线程使用的是同一种数据结构（task_struct）
  - 上下文切换中会使用
- 应用态：可以由线程库定义
  - Linux：pthread结构体
  - Windows：TIB（Thread Information Block）
  - 可以认为是内核TCB的扩展

线程本地存储（TLS）

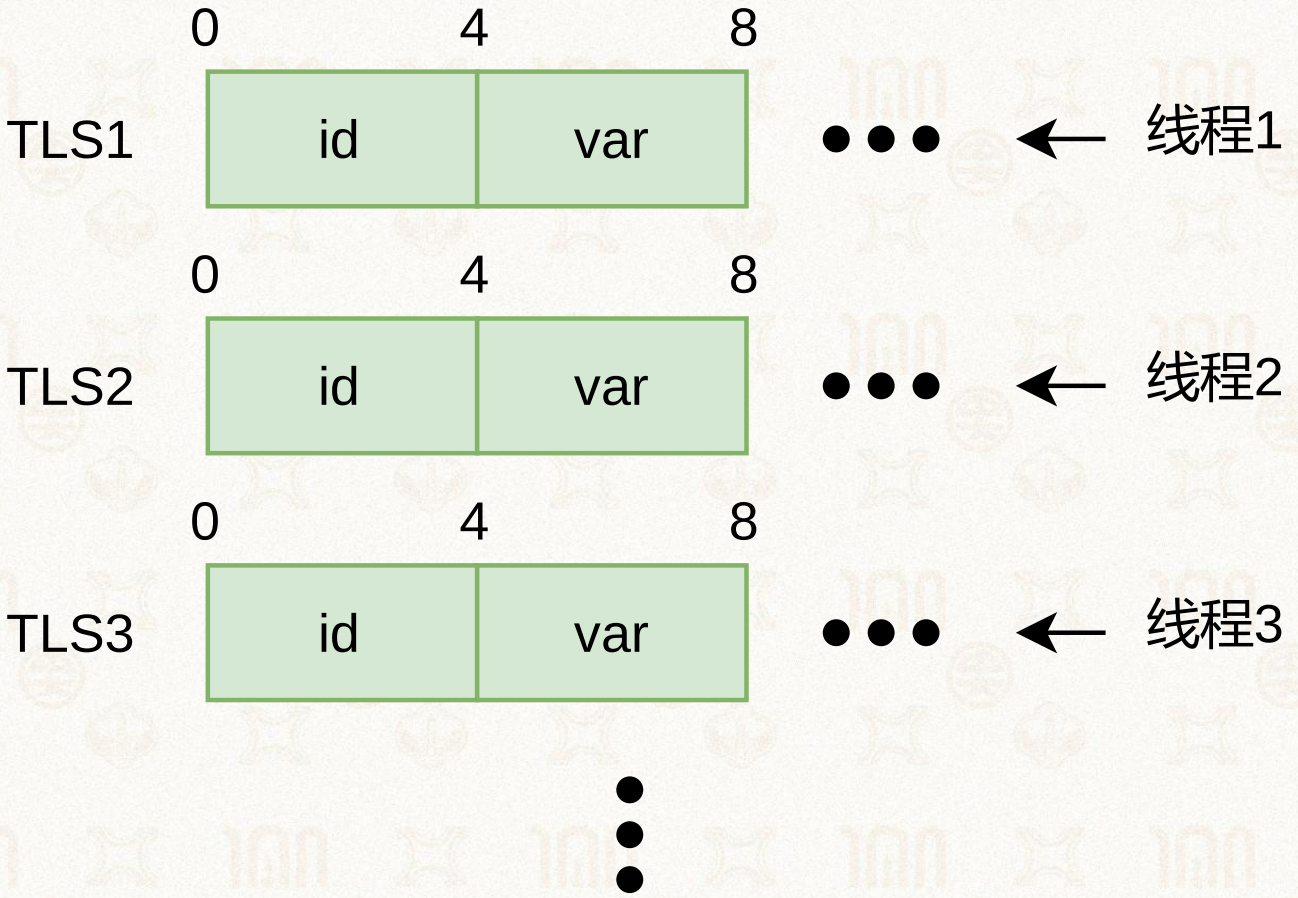
- 不同线程可能会执行相同的代码
  - 线程不具有独立的地址空间，多线程共享代码段

- 问题：对于全局变量，不同线程可能需要不同的版本
  - 举例：用于标明系统调用错误的errno

- 解决方案：线程本地存储（Thread Local Storage）

线程本地存储（TLS）

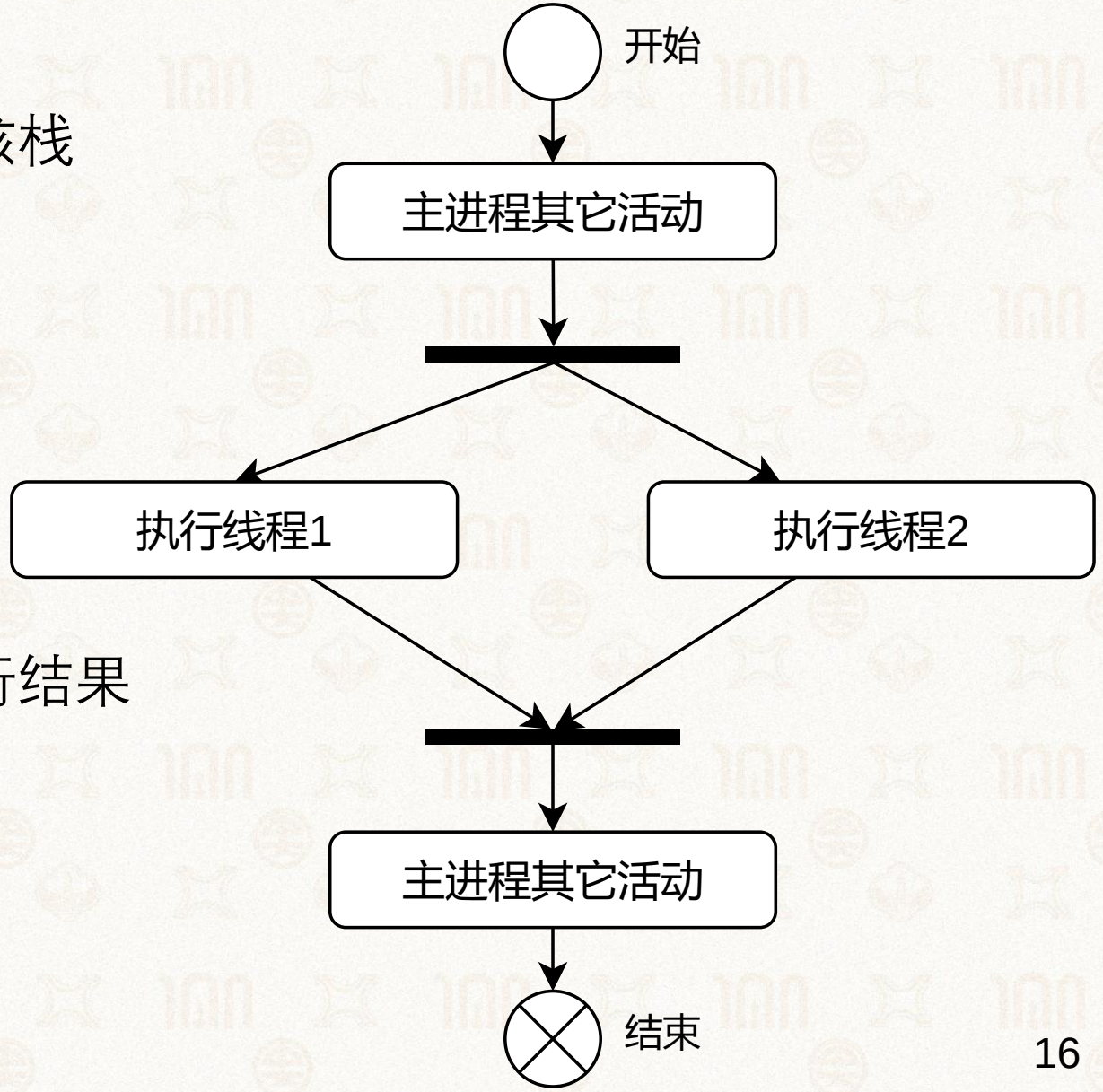
- 线程库允许定义每个线程独有的数据
- __thread int id;　会为每个线程定义一个独有的id变量



- 每个线程的TLS结构相似
  - 可通过TCB索引
- TLS寻址模式：基地址 + 偏移量
  - X86: 段页式 (fs寄存器)
  - AArch64: 特殊寄存器tpidr_el0

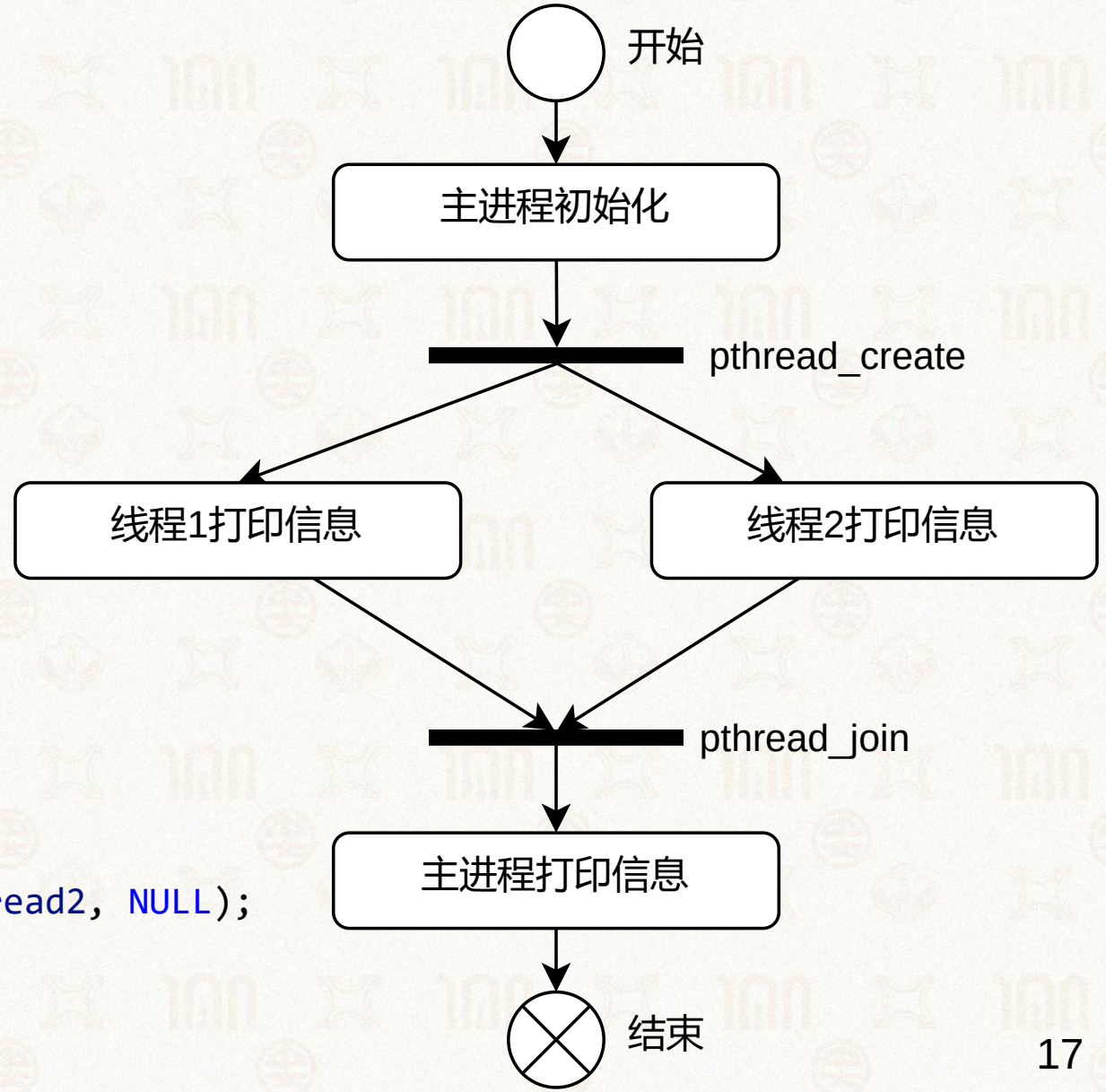
线程的基本操作：以pthreads为例

- 创建：pthread_create
  - 内核态：创建相应的内核态线程及内核栈
  - 应用态：创建TCB、应用栈和TLS
- 合并：pthread_join
  - 等待另一线程执行完成，并获取其执行结果
  - 可以认为是fork的“逆向操作”



pthread_create示例

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr ) {
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
void main() {
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

     iret1 = pthread_create( &thread1, NULL,
        print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL,
        print_message_function, (void*) message2);

     pthread_join( thread1, NULL);  pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
```



填空题 4分

设置

```
volatile int balance = 0;

void *mythread(void *arg) {
    int i;
    for (i = 0; i < 200; i++) {
        balance++;
    }
    printf("Balance is % d\n", balance);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3;
    pthread_create(&p1, NULL, mythread, (void *)"A");
    pthread_join(p1, NULL);
    pthread_create(&p2, NULL, mythread, (void *)"B");
    pthread_join(p2, NULL);
    pthread_create(&p3, NULL, mythread, (void *)"C");
    pthread_join(p3, NULL);
    printf("Final Balance is % d\n", balance);
}
```

假设线程创建都会成功，且不会触发其它无关的系统调用:

(a) 运行代码，线程p1、p2、p3打印的结果分别是什么？p1: [填空1] p2: [填空2] p3: [填空3]

(b) 运行代码，main函数最后输出的结果是什么？ [填空4]

作答

填空题 4分

设置

```
volatile int balance = 0;

void *mythread(void *arg) {
    int i;
    for (i = 0; i < 200; i++) {
        balance++;
    }
    printf("Balance is % d\n", balance);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3;
    pthread_create(&p1, NULL, mythread, (void *)"A");
    pthread_create(&p2, NULL, mythread, (void *)"B");
    pthread_create(&p3, NULL, mythread, (void *)"C");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
    printf("Final Balance is % d\n", balance);
}
```

假设线程创建都会成功，且不会触发其它无关的系统调用:

(a) 运行代码，线程p1、p2、p3打印的结果分别是什么？p1: [填空1] p2: [填空2] p3: [填空3]

(b) 运行代码，main函数最后输出的结果是什么？ [填空4]

作答

填空题 4分

设置

```
volatile int balance = 0;

void *mythread(void *arg) {
    int i;
    for (i = 0; i < 20000; i++) {
        balance++;
    }
    printf("Balance is % d\n", balance);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2, p3;
    pthread_create(&p1, NULL, mythread, (void *)"A");
    pthread_create(&p2, NULL, mythread, (void *)"B");
    pthread_create(&p3, NULL, mythread, (void *)"C");
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);
    printf("Final Balance is % d\n", balance);
}
```

假设线程创建都会成功，且不会触发其它无关的系统调用:

(a) 运行代码，线程p1、p2、p3打印的结果分别是什么？p1: [填空1] p2: [填空2] p3: [填空3]

(b) 运行代码，main函数最后输出的结果是什么？ [填空4]

作答

线程的基本操作：以pthreads为例

- 退出：pthread_exit
  - 可设置返回值（会被pthread_join获取）

- 暂停：pthread_yield
  - 立即暂停执行，出让CPU资源给其它线程
  - 好处：可以帮助调度器做出更优的决策

pthread_yield 示例

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

void *thread(void *arg) {
    while (1) {
        puts((char *)arg);
        pthread_yield(NULL);
    }
}
void main() {
    pthread_t t1, t2, t3;
    if (pthread_create(&t1, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error"); exit(1);
    }
    if (pthread_create(&t2, NULL, thread, "thread 2") != 0) {
        perror("pthread_create() error"); exit(2);
    }
    if (pthread_create(&t3, NULL, thread, "thread 3") != 0) {
        perror("pthread_create() error"); exit(3);
    }
    sleep(1);
    exit(0);
}
```

循环每执行一次，就主动换其它线程执行

主进程过1秒之后会结束，同时终止所有线程

线程的上下文切换：以ChCore为例

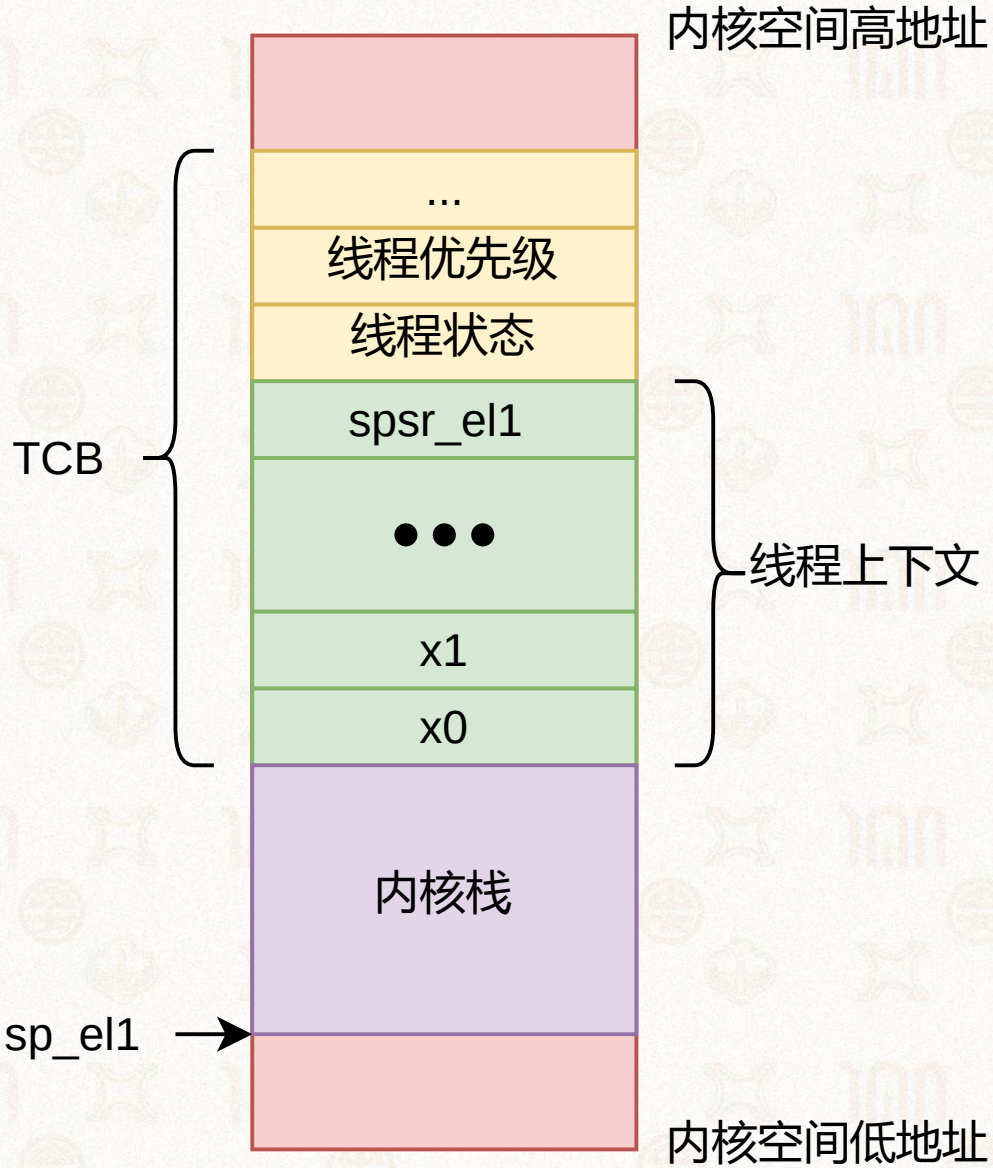
- 线程的上下文即重要的寄存器信息
  - 常规寄存器：x0-x30
  - 程序计数器（PC）： elr_el1
  - 栈指针：sp_el0
  - CPU状态（如条件码）：spsr_el1

- 主要分为三步
  - 进入内核态，保存上下文
  - 切换页表与内核栈
  - 恢复上下文，返回用户态

ChCore的TCB结构

- 上半部分：线程的相关信息
- 下半部分：线程上下文

- TCB下面为线程的内核栈
  - 刚进入内核时的线程内核栈为空
  - sp_el1指向栈顶



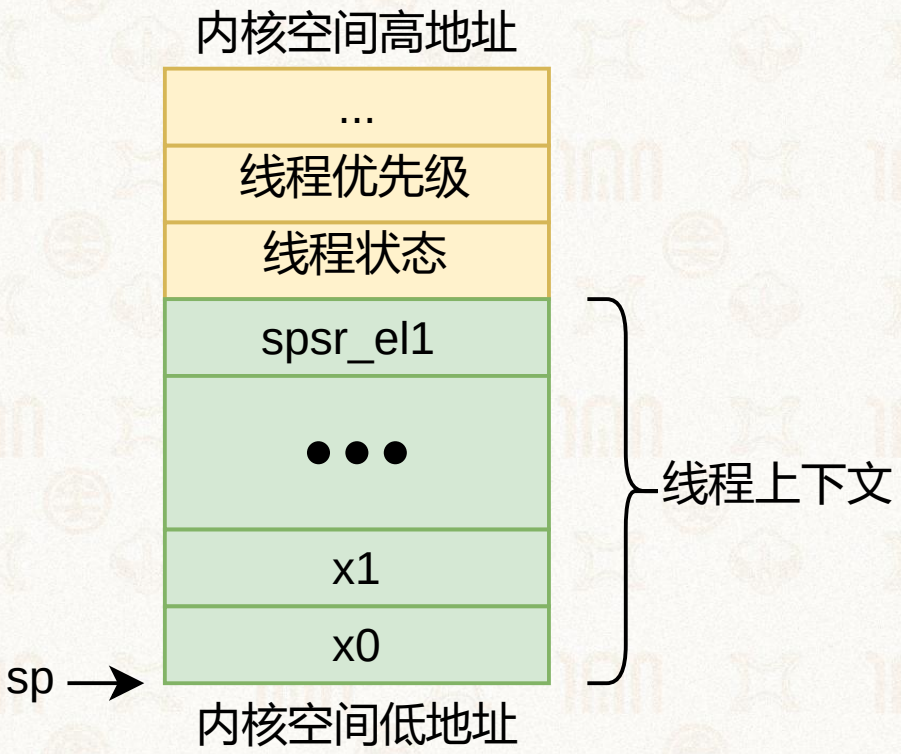
第一步：进入内核态、保存上下文

- 应用线程可通过异常、中断或系统调用进入内核态
  - 运行状态将切换到内核态（EL1）
  - 开始使用sp_el1作为栈指针（用户栈切换到内核栈）
  - 保存应用线程的PC（elr_el1）
  - 保存应用线程的CPU状态（spsr_el1）
  - 以上均由硬件自动完成

第一步：进入内核态、保存上下文

- 保存上下文

```
sub sp, sp, #ARCH_EXEC_CONT_SIZE

// 保存常规寄存器(x0-x29)
stp x0, x1, [sp, #16 * 0]
stp x2, x3, [sp, #16 * 1]
stp x4, x5, [sp, #16 * 2]

// ...
stp x28, x29, [sp, #16 * 14]

// 保存x30和三个特殊寄存器：sp_el0, elr_e1, spsr_el1
mrs x21, sp_el0
mrs x22, elr_el1
mrs x23, spsr_el1
stp x30, x21, [sp, #16 * 15]
stp x22, x23, [sp, #16 * 16]
```



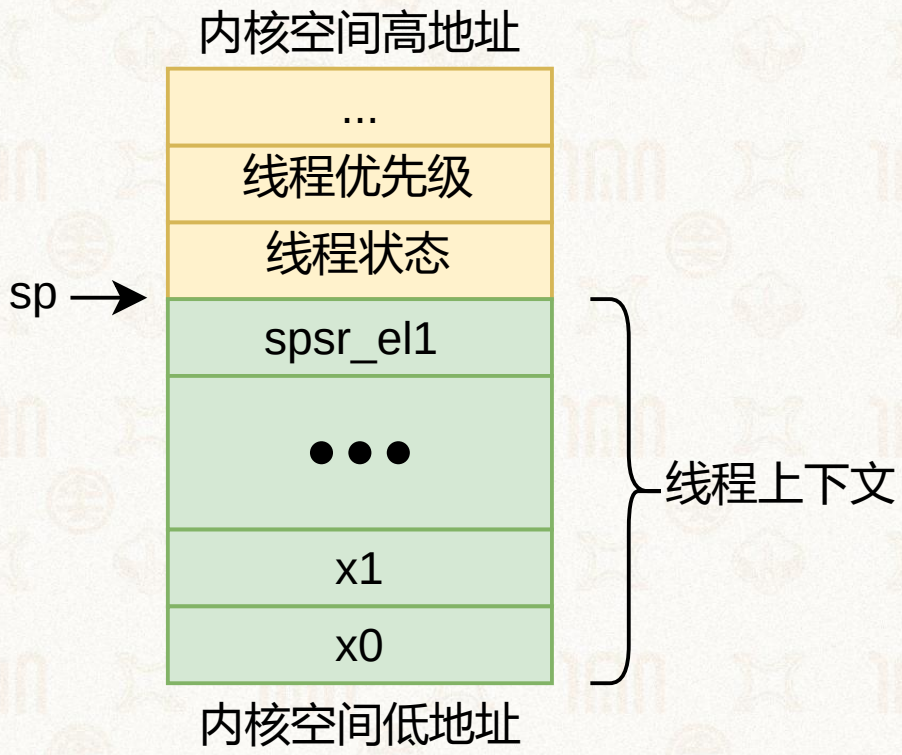
第二步：切换页表和内核栈

- 操作系统确定下一个被调度的线程（调度器决定）
- 切换页表
  - 将页表相关寄存器的值置为目标线程的页表基地址
- 切换内核栈
  - 找到目标内核栈的栈顶指针（目标线程的TCB）
  - 修改sp_el1的值至目标内核栈
  - 可以认为是线程执行的分界点（切换之后变为目标线程执行）

第三步：上下文恢复，返回用户态

- 上下文恢复：取出栈上的值并存回寄存器

```
ldp x22, x23, [sp, #16 * 16]
ldp x30, x21, [sp, #16 * 15]

// 恢复三个特殊寄存器
msr sp_el0, x21
msr elr_el1, x22
msr spsr_el1, x23

// 恢复常规寄存器X0-X29
ldp x0, x1, [sp, #16 * 0]
// ......
ldp x28, x29, [sp, #16 * 14]
add sp, sp, #ARCH_EXEC_CONT_SIZE
```

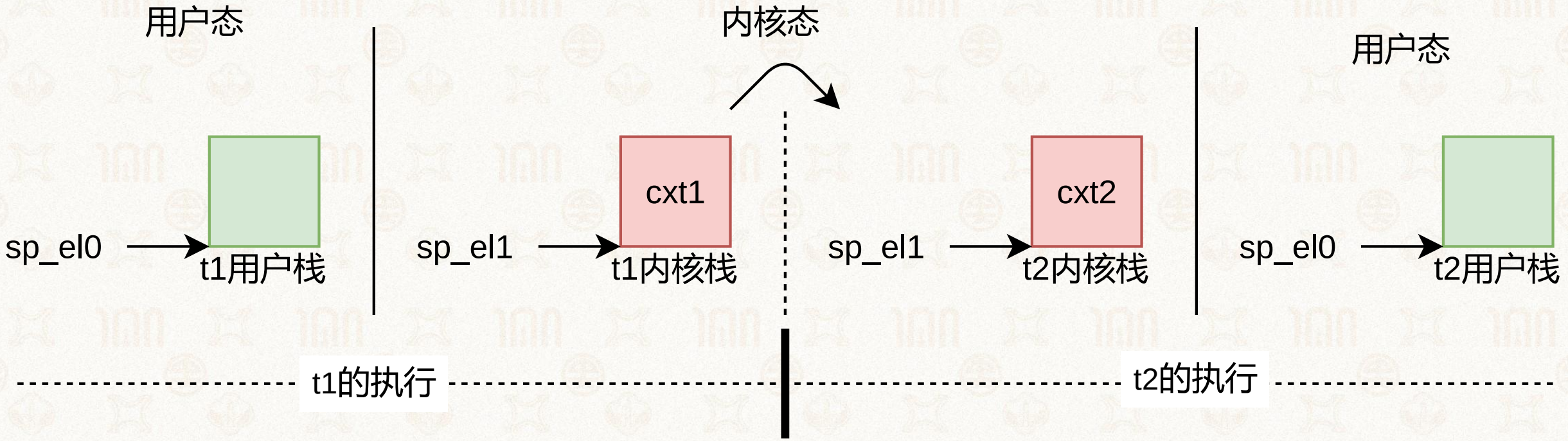


第三步：上下文恢复，返回用户态

- 返回用户态：调用eret，由硬件执行一系列操作
  - 将elr_el1中的返回地址存回PC
  - 改为使用sp_el0作为栈指针（内核栈切换到用户栈）
  - 将CPU状态设为spsr_el1中的值
  - 运行状态切换为用户态（EL0）

上下文切换小结

- 共涉及两次权限等级切换、三次栈切换
- 内核栈的切换是线程切换执行的“分界点”



大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作
- 线程
  - 线程的概念
  - 线程模型
  - 相关数据结构
  - 基本操作
  - 上下文切换
- 纤程
  - 纤程的概念
  - 编程模型
  - Windows和编程语言支持
  - Go语言简介与协程

一对一线程模型的局限

- 复杂应用：对调度存在更多需求
  - 生产者消费者模型：生产者完成后，消费者最好马上被调度
  - 内核调度器的信息不足，无法完成及时调度

- “短命”线程：执行时间亚毫秒级（如处理web请求）
  - 内核线程初始化时间较长，造成执行开销
  - 线程上下文切换频繁，开销较大

纤程（用户态线程）

- 比线程更加轻量级的运行时抽象
  - 不单独对应内核线程
  - 一个内核线程可以对应多个纤程（多对一）

- 纤程的优点
  - 不需要创建内核线程，开销小
  - 上下文切换快（不需要进入内核）
  - 允许用户态自主调度，有助于做出更优的调度决策

Linux对于纤程的支持：ucontext

- 每个ucontext可以看作一个用户态线程
- makecontext: 创建新的ucontext
- setcontext:纤程上下文切换
- getcontext:保存当前的ucontext

setcontext / getcontext 示例

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

int x = 0;
ucontext_t context, *cp = &context;

void func(void) {
    x++;
    setcontext(cp);
}

int main(void) {
    getcontext(cp);
    if (!x) {
        printf("getcontext has been called\n");
        func();
    } else {
        printf("setcontext has been called\n");
    }
}
```

纤程的例子：生产者 - 消费者

生产者

```
void produce() {
    buf[++cnt] = rand();
    setcontext(&cxt2);
}
```

消费者

```
void consume() {
    process(buf[cnt]);
    setcontext(&cxt1);
}
```

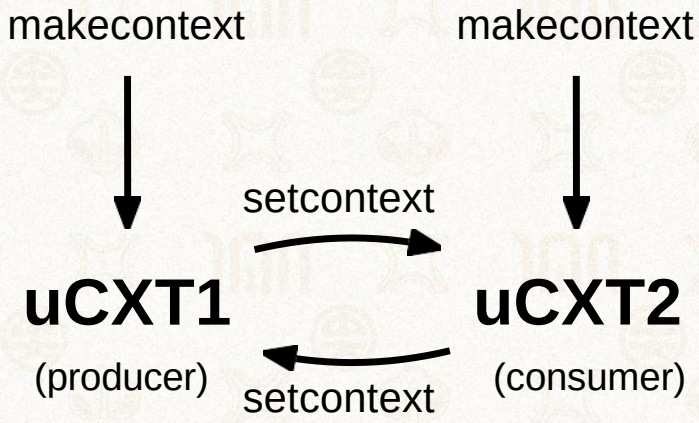
主纤程

```
makecontext(&cxt1, produce, …);
makecontext(&cxt2, consume, …);
setcontext(&cxt1);
```

从例子看纤程的优势

- 纤程切换及时
  - 当生产者完成任务后，可直接用户态切换到消费者
  - 对该线程来说是最优调度（内核调度器和难做到）

- 高效上下文切换
  - 切换不进入内核态，开销小
  - 即时频繁切换也不会造成过大开销



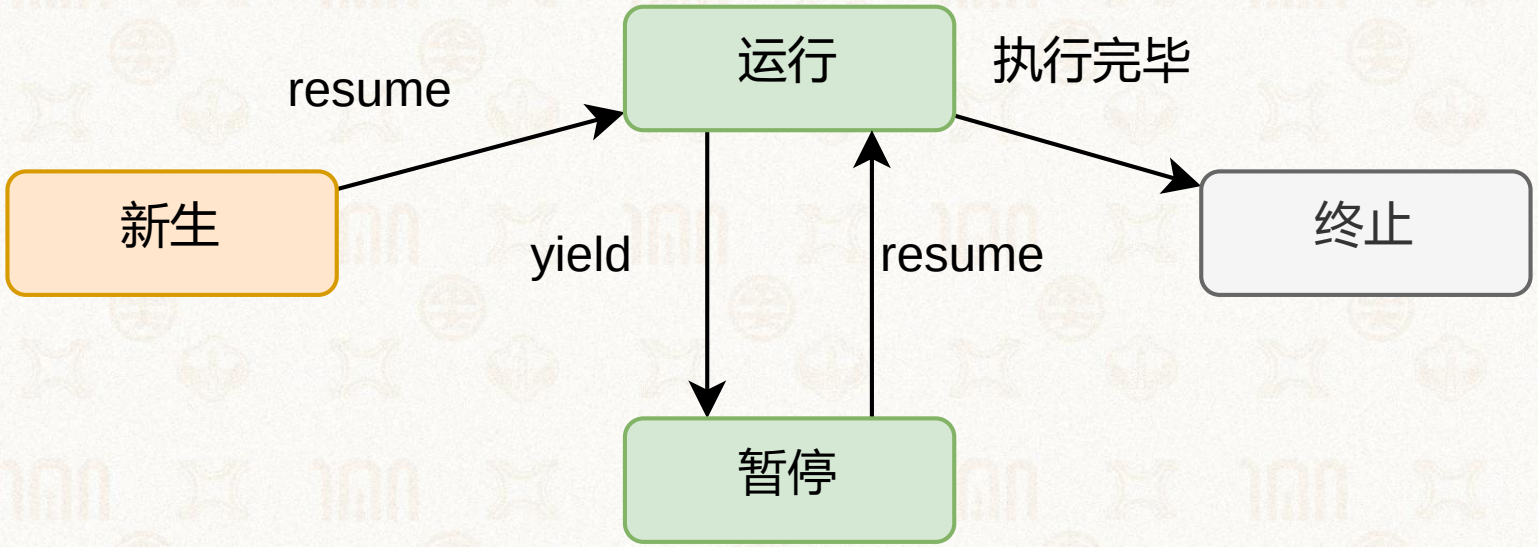
Windows对于纤程的支持：Fiber库

- 与ucontext类似的编程模型
  - createFiber: 创建新的纤程
  - SwitchToFiber: 纤程切换

- 支持纤程本地存储（FLS）
  - 当一个内核线程对应单个纤程时，FLS与TLS结构相同
  - 当一个内核线程对应多个纤程时，TLS可分裂为多个FLS

程序语言中对纤程的支持：协程

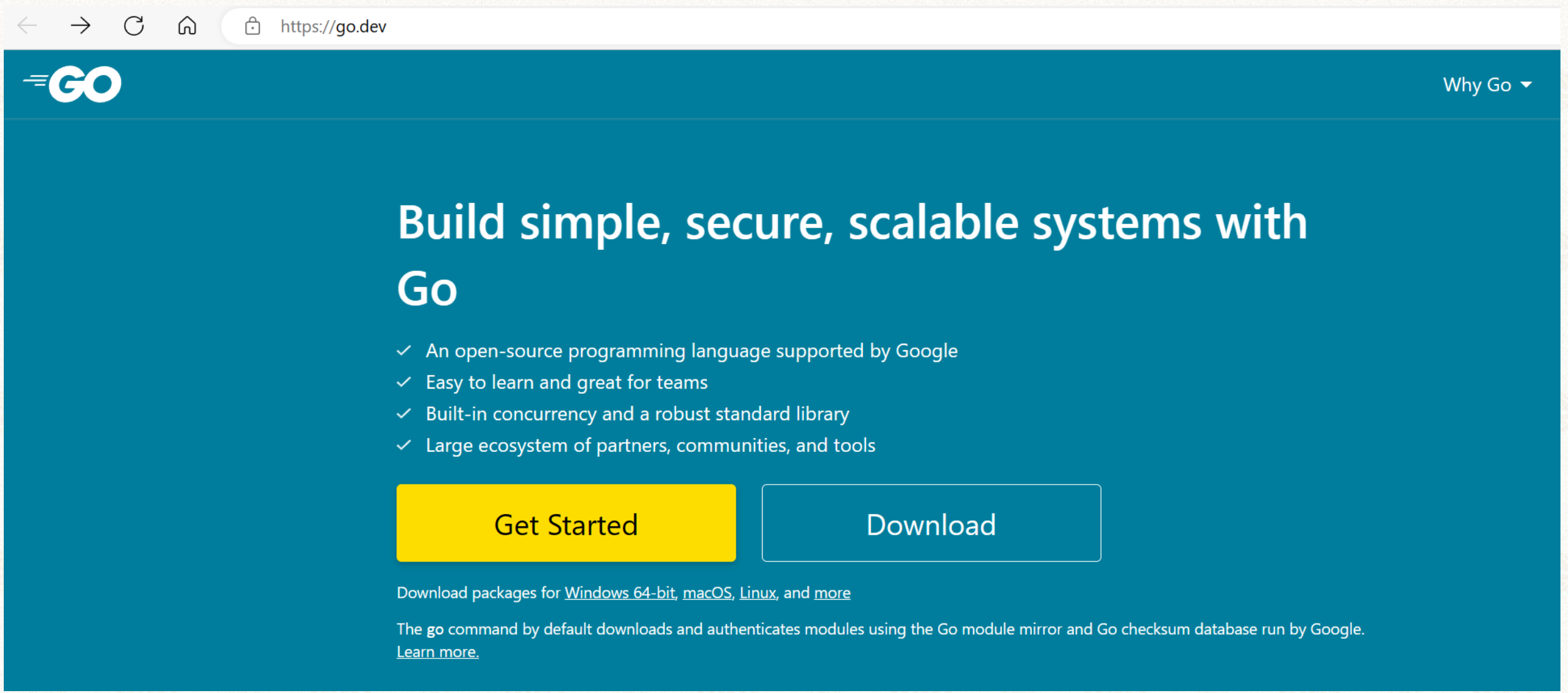
- 许多高级程序语言都对协程提供了支持
  - go、python、lua……
  - C++自20开始也支持了协程
- 协程也拥有状态（新生 / 暂停 / 终止 / 执行）
  - 核心操作：yield（使协程暂停执行）、resume（继续执行）



大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作
- 线程
  - 线程的概念
  - 线程模型
  - 相关数据结构
  - 基本操作
  - 上下文切换
- 纤程
  - 纤程的概念
  - 编程模型
  - Windows和编程语言支持
  - Go语言简介与协程

Go官网


https://go.dev
GO
Why Go
Build simple, secure, scalable systems with Go
An open-source programming language supported by Google
Easy to learn and great for teams
Built-in concurrency and a robust standard library
Large ecosystem of partners, communities, and tools
Get Started
Download
Download packages for Windows 64-bit, macOS, Linux, and more
The go command by default downloads and authenticates modules using the Go module mirror and Go checksum database run by Google.
Learn more.

版权信息

- 课件中代码示例主要来自于：
  - 作者：MarvinZhang
  - 链接：https://juejin.cn/post/6943524790636544036
  - 来源：稀土掘金

Go第一印象

- 静态强类型编程语言，和C/C++相同
- 但会自动执行类型推断:

```
// 自动类型推断
func main() {
    valInt := 1        // 自动推断 int 类型
    valStr := "hello" // 自动推断为 string 类型
    valBool := false  // 自动推断为 bool 类型
}
```

- 有C语言基础，Go可以很快上手

Go第一印象

```go
// 定义一个 struct 类
type SomeClass struct {
    PublicVariable  string // 公共变量
    privateVariable string // 私有变量
}
// 公共方法
func (c *SomeClass) PublicMethod() (result string) {
    return "This can be called by external modules"
}
// 私有方法
func (c *SomeClass) privateMethod() (result string) {
    return "This can only be called in SomeClass"
}
func main() {
    // 生成实例
    someInstance := SomeClass{
        PublicVariable:  "hello",
        privateVariable: "world",
    }
}
```

Go第一印象：面向接口编程，不是面向对象编程

```
// 定义 Animal 接口
interface Animal {
    Eat()  // 声明 Eat 方法
    Move()  // 声明 Move 方法
}

// ==== 定义 Dog Start ====
// 定义 Dog 类
type Dog struct {
}

// 实现 Eat 方法
func (d *Dog) Eat() {
fmt.Printf("Eating bones")
}

// 实现 Move 方法
func (d *Dog) Move() {
fmt.Printf("Moving with four legs")
}
// ==== 定义 Dog End ====
```

内置并发：最友好的协程体验

```go
package main

import (
    "fmt"
    "time"
)

func asyncTask() {
    fmt.Printf("This is an asynchronized task")
}

func syncTask() {
    fmt.Printf("This is a synchronized task")
}

func main() {
    go asyncTask() // 异步执行，不等待
    syncTask()     // 同步执行，等待
    go asyncTask() // 等待前面 syncTask 完成之后，再异步执行，不等待
    time.Sleep(time.Second * 1)
}
```

内置并发：最友好的协程体验

```go
// deadloop.go
package main

import (
    "fmt"
    "time"
)

func deadloop() {
    for {

    }
}

func main() {
    deadloop()
    for {
        time.Sleep(time.Second * 1)
        fmt.Println("I got scheduled!")
    }
}
```

- 关键字go表示运行一个协程
- 编译运行： go run deadloop.go

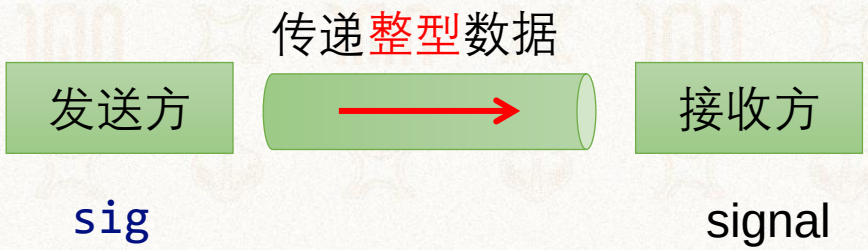
试一下在这里加与不加go，
程序运行会产生什么样的差别

协程之间的通信：管道(channel)

```go
package main
import (
    "fmt"
    "time"
)
func longTask(signal chan int) {
    // 不带参数的 for
    // 相当于 while 循环
    for {
        fmt.Println("longTask is running")
        // 接收 signal 通道传值
        v := <-signal
        // 如果接收值为 1，停止循环
        if v == 1 {
            break
        }
        time.Sleep(1 * time.Second)
    }
    fmt.Println("longTask is finihsed")
}
```

这个函数接收的参数是一个int类型的管道

```go
func main() {
    // 声明通道
    sig := make(chan int)
    // 异步调用 longTask
    go longTask(sig)
    // 等待 1 秒钟
    time.Sleep(3 * time.Second)
    // 向通道 sig 传值
    sig <- 1
    // 然后 longTask 会接收 sig 传值，终止循环
    time.Sleep(1 * time.Second)
}
```



协程用于处理网络请求

```
func handler(c net.Conn) {
    c.Write([]byte("ok"))
    c.Close()
}

func main() {
    l, err := net.Listen("tcp", ":8000")
    if err != nil {
        panic(err)
    }
    for {
        c, err := l.Accept()
        if err != nil {
            continue
        }
        go handler(c)
    }
}
```

- 程序运行后，另外开一个终端运行如下命令会有什么反应：
  - telnet 127.0.0.1 8000
  - 以上命令尝试与本机8000端口建立tcp连接

简单的管道操作

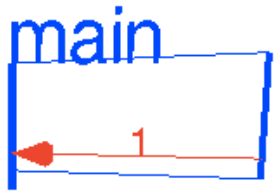
- 两个协程：main 和 go#19 (内部的编号，不用理会)
  - 蓝色表示协程，红色表示管道通信


```go
package main

func main() {
    ch := make(chan int)

    go func() {
        ch <- 42
    }()

    close(ch)
}
```

交互示例：https://divan.dev/demos/hello/

计时器

- 蓝色表示协程，红色表示管道通信




```
func tick(d time.Duration) <-chan int {
    c := make(chan int)
    go func() {
        time.Sleep(d)
        c <- 1
    }()
    return c
}

func main() {
    for i := 0; i < 24; i++ {
        c := tick(1 * time.Second)
        result := <-c
        fmt.Println(i, " : ", result)
    }
}
```

交互示例：https://divan.dev/demos/timers/

乒乓

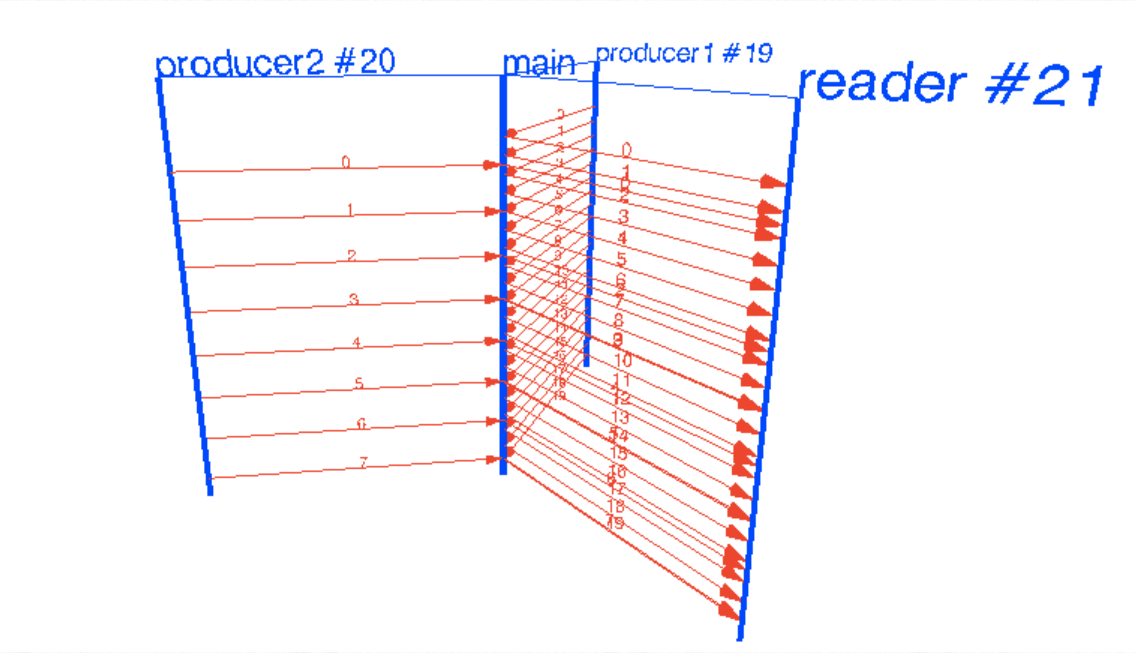
- 蓝色表示协程，红色表示管道通信


```
type Ball struct {
    hits int
}
func player(name string, table chan *Ball) {
    for {
        ball := <-table
        ball.hits++
        fmt.Println(name, ball.hits)
        time.Sleep(1 * time.Second)
        table <- ball
    }
}
func main() {
    table := make(chan *Ball)
    go player("ping", table)
    go player("pong", table)
    table <- new(Ball) // 游戏开始，发球
    time.Sleep(10 * time.Second)
    <-table
    close(table) // 游戏结束
}
```

交互示例：https://divan.dev/demos/pingpong/

多条管道联合使用：渐入效果(fade-in)

- 两个生产者向管道写入数据
- 一个消费者读取数据




```go
func producer(ch chan int, d time.Duration) {
    i := 0
    for {
        i++
        ch <- i
        time.Sleep(d)
    }
}
func reader(out chan int) {
    for {
        result := <-out
        fmt.Println("read: ", result)
    }
}
func main() {
    ch := make(chan int)
    out := make(chan int)
    go producer(ch, 1*time.Second)
    go producer(ch, 3*time.Second)
    go reader(out)
    for {
        out <- <-ch
    }
}
```

交互示例： https://divan.dev/demos/fanin/

主观题 10分

设置

SUN YAT-SEN UNIVERSITY

问题：给定自然数n，找出前n个质数

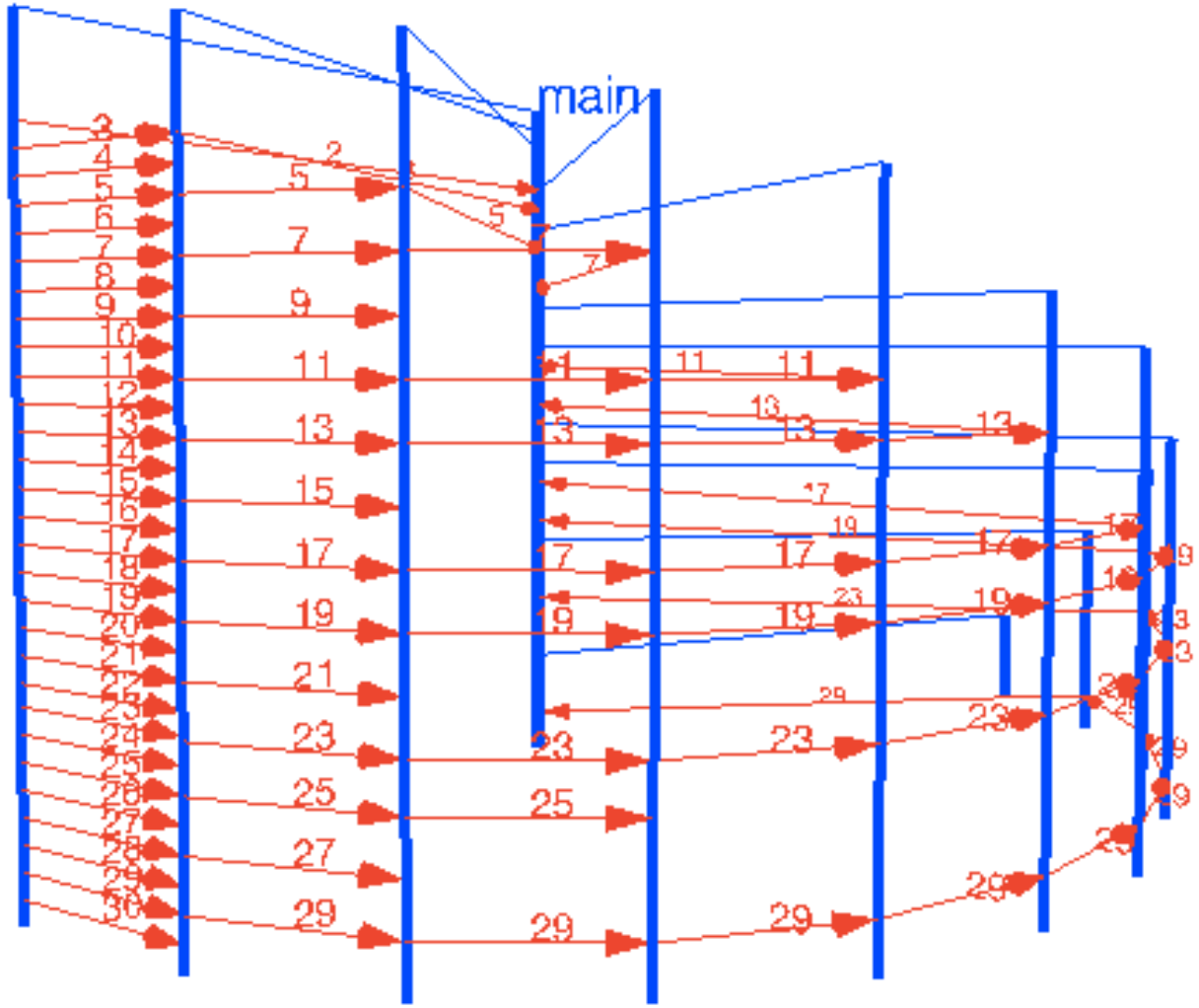
算法描述：

- 先用最小的质数2去筛，把2的倍数筛除；第一个未被筛除的数就是质数（这里是3）
- 再用这样的质数3去筛，筛除3的倍数...
- 不断重复下去，直到筛完为止

如何使用协程和管道实现以上算法，给出大意即可

作答

协程：实现质数筛选算法



交互示例：https://divan.dev/demos/primesieve/

大纲

- 进程
  - 进程的诞生和概念
  - 进程的状态
  - 数据结构
  - 基本操作
- 线程
  - 线程的概念
  - 线程模型
  - 相关数据结构
  - 基本操作
  - 上下文切换
- 纤程
  - 纤程的概念
  - 编程模型
  - Windows和编程语言支持
  - Go语言简介与协程

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course
匿名提问箱：https://suask.me/ask-teacher/106/苏玉鑫