

# 线程

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

# 版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
  - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

# 为什么需要线程

线程的概念 - 线程模型 - 相关数据结构 - 基本操作

# 背景：单台设备计算资源逐渐丰富

- **多核处理器**

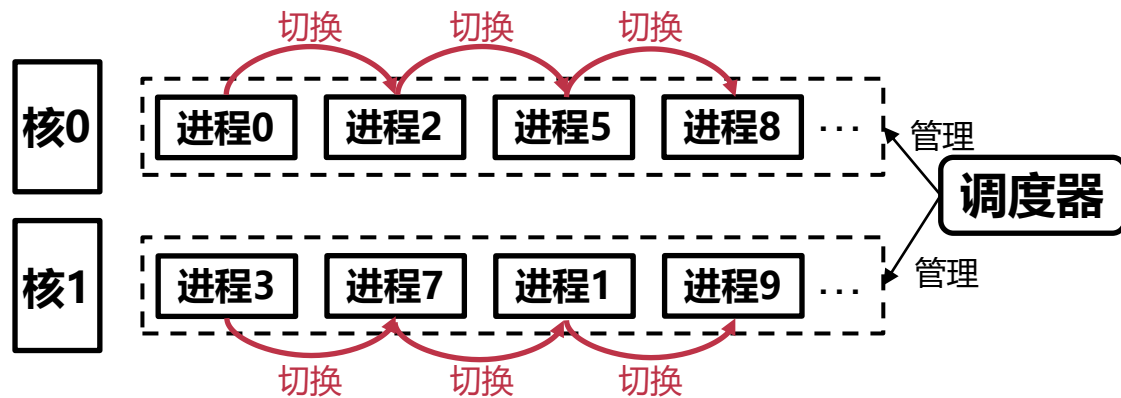
- 每个“核”（core）都能独立运行程序
- Apple M1：8核；M1 Pro：10核

- **大型计算机包含大量多核处理器**

- 我国的神威·太湖之光：包含40960颗神威处理器（6核）
- 每个处理器260核，共10,649,600核

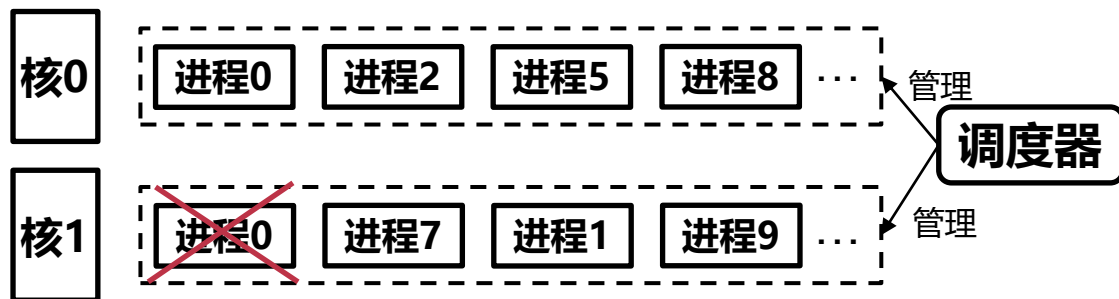
# 简单方法：进程+调度

- 进程数量一般远超过CPU核数
  - 通过简单分配，使每个核至少分得一个进程
- 调度器通过分时复用增加计算资源利用率
  - 通过调度策略，在进程需要等待时切换到其他进程执行



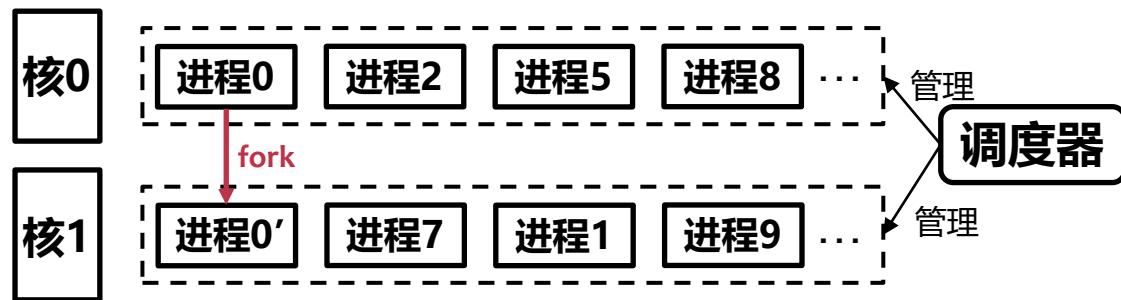
# 简单方法：进程+调度

- 存在局限：单一进程无法利用多核资源
  - 一个进程同一时刻只能被调度到其中一个核上运行
  - 如果一个程序想同时利用多核，该怎么办呢？



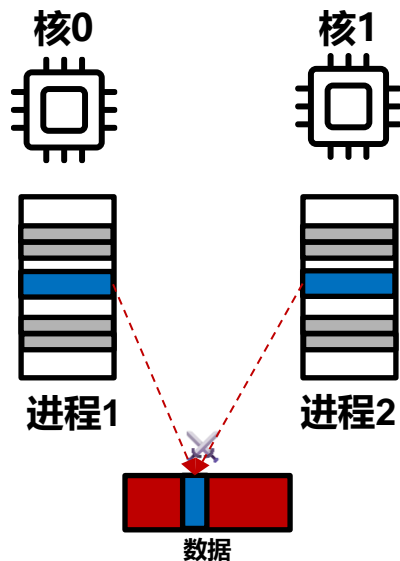
# 简单方法：进程+调度

- **存在局限：单一进程无法利用多核资源**
  - 一个进程一同时刻只能被调度到其中一个核上运行
- **解决思路：用fork创建相似进程**
  - Fork创建的进程与原进程行为类似，可用于其他核上执行



# Fork方法存在的局限

- **进程间隔离过强，共享数据困难**
  - 各进程拥有独立的地址空间，共享需以页为粒度
  - 协调困难，需要复杂的通信机制
- **进程管理开销较大**
  - 创建：地址空间复制
  - 切换：页表切换





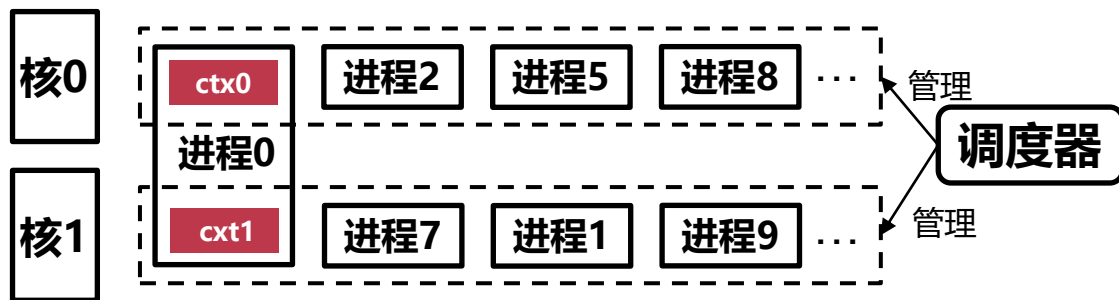
# 解决方案：能否使单一进程跨核执行？

- 优势：无需用fork创建新进程

- 降低进程管理开销
- 同一地址空间数据共享/同步方便

- 需要什么支持？

- 处理器上下文：不同核执行状态不同，需要独立处理器上下文
- 接下来的问题：如何在进程内部为多个处理器上下文提供支持？



# 线程：更加轻量级的运行时抽象

- 线程只包含运行时的状态
  - 代码和数据等由**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

# 程序员如何使用线程

pthread接口 – 线程本地存储

# 在程序中如何使用线程？

- **常用库：POSIX threads (pthreads)**
  - 包含约60+标准接口
  - 实现的功能与进程相关系统调用相似
    - 创建：pthread\_create
    - 回收：pthread\_join
    - 退出：pthread\_exit
    - .....
- **注意：一个线程执行系统调用，可能影响该进程的所有线程**
  - 如exit会使所有线程退出

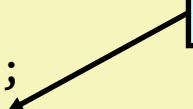
# 使用线程的程序示例（创建）

```
/* pthreads "hello, world" program */
#include <pthread.h>
#include <stdio.h>

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}
```

创建线程接口



# 使用线程的程序示例（创建）

```
/* pthreads "hello, world" program */
#include <pthread.h>
#include <stdio.h>

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}
```

子线程ID

子线程起始执行点

# 使用线程的程序示例（创建）

```
/* pthreads "hello, world" program */
#include <pthread.h>
#include <stdio.h>

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}
```

子线程属性  
(通常可设为NULL)

子线程参数

# 使用线程的程序示例（创建）

```
/* pthreads "hello, world" program */
```

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

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

```
/* thread routine */
```

```
void *thread(void *vargp) {  
    printf("Hello, world!\n");  
    return NULL;
```

```
}
```

```
int main() {
```

```
    pthread_t tid;
```

```
    pthread_create(&tid, NULL, thread, NULL);
```

```
    exit(0);
```

```
}
```

预期运行结果:

```
~/test$ ./a.out  
Hello, world!  
~/test$
```

子线程

主线程



# 使用线程的程序示例（创建）

```
/* pthreads "hello, world" program */
```

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

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

```
/* thread routine */
```

```
void *thread(void *vargp) {  
    printf("Hello, world!\n");  
    return NULL;  
}
```

```
int main() {  
    pthread_t tid;  
    pthread_create(&tid, NULL, thread, NULL);  
    exit(0);  
}
```

可能的实际运行结果：

```
~/test$ ./a.out  
~/test$
```

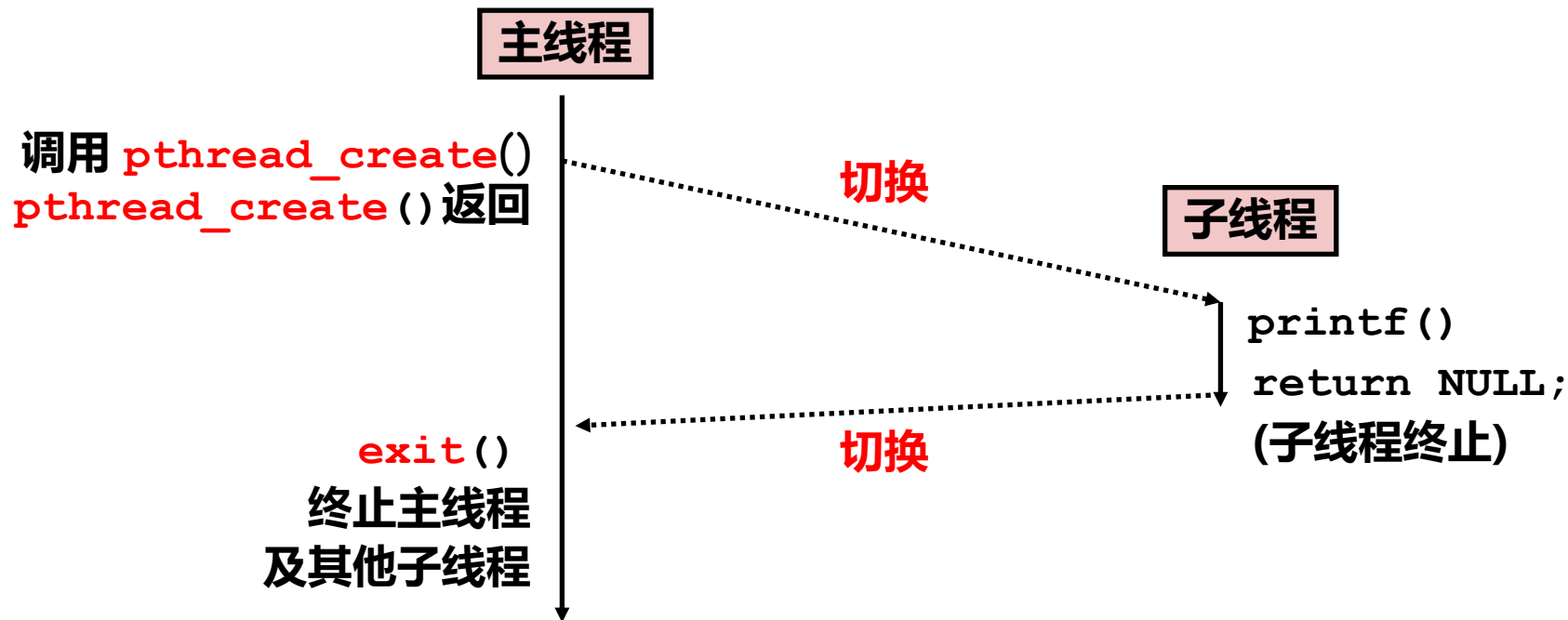
子线程

为什么没有输出？

主线程

# 程序控制流分析

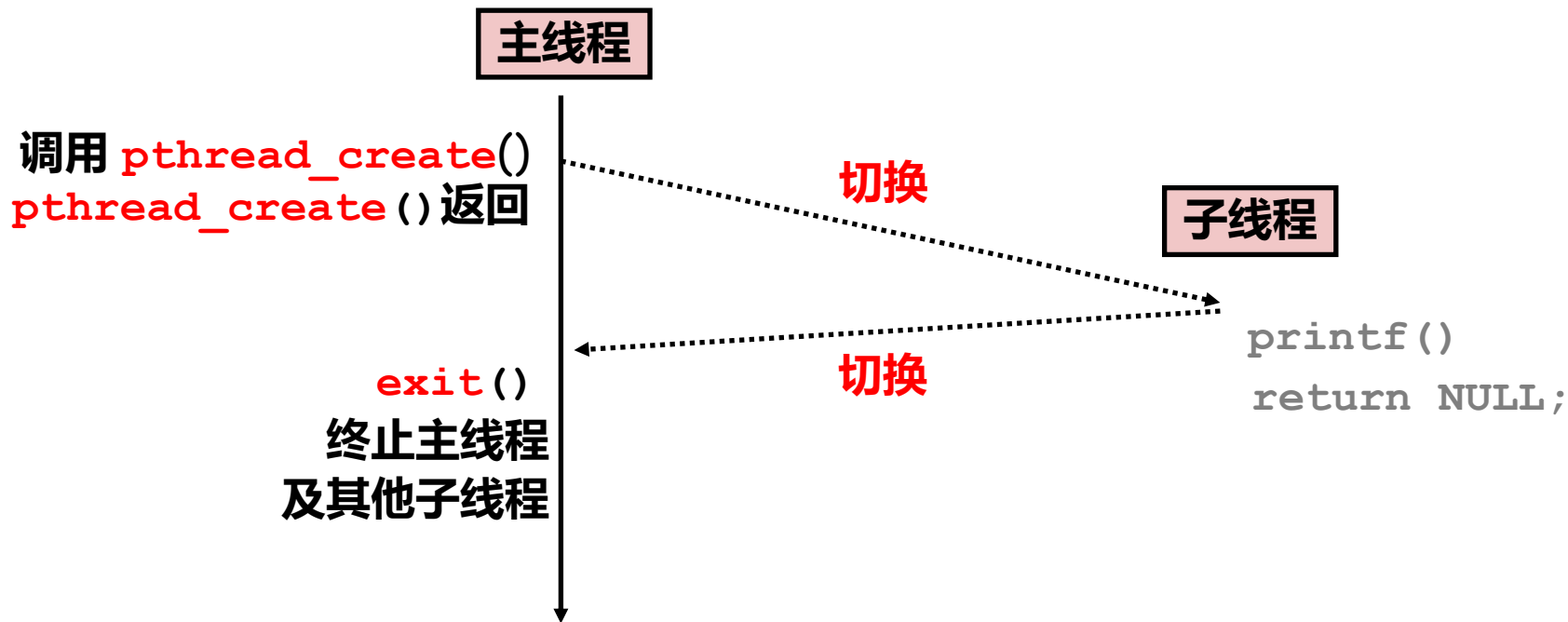
- 主线程创建子线程后，两线程独立执行
  - 若子线程先执行，则printf顺利输出



# 程序控制流分析

- 主线程创建子线程后，两线程独立执行
  - 若子线程先执行，则exit被调用，子线程直接被终止

如何避免？



# 解决方案：加入join操作

```
/* pthreads "hello, world" program */
#include <pthread.h>
#include <stdio.h>

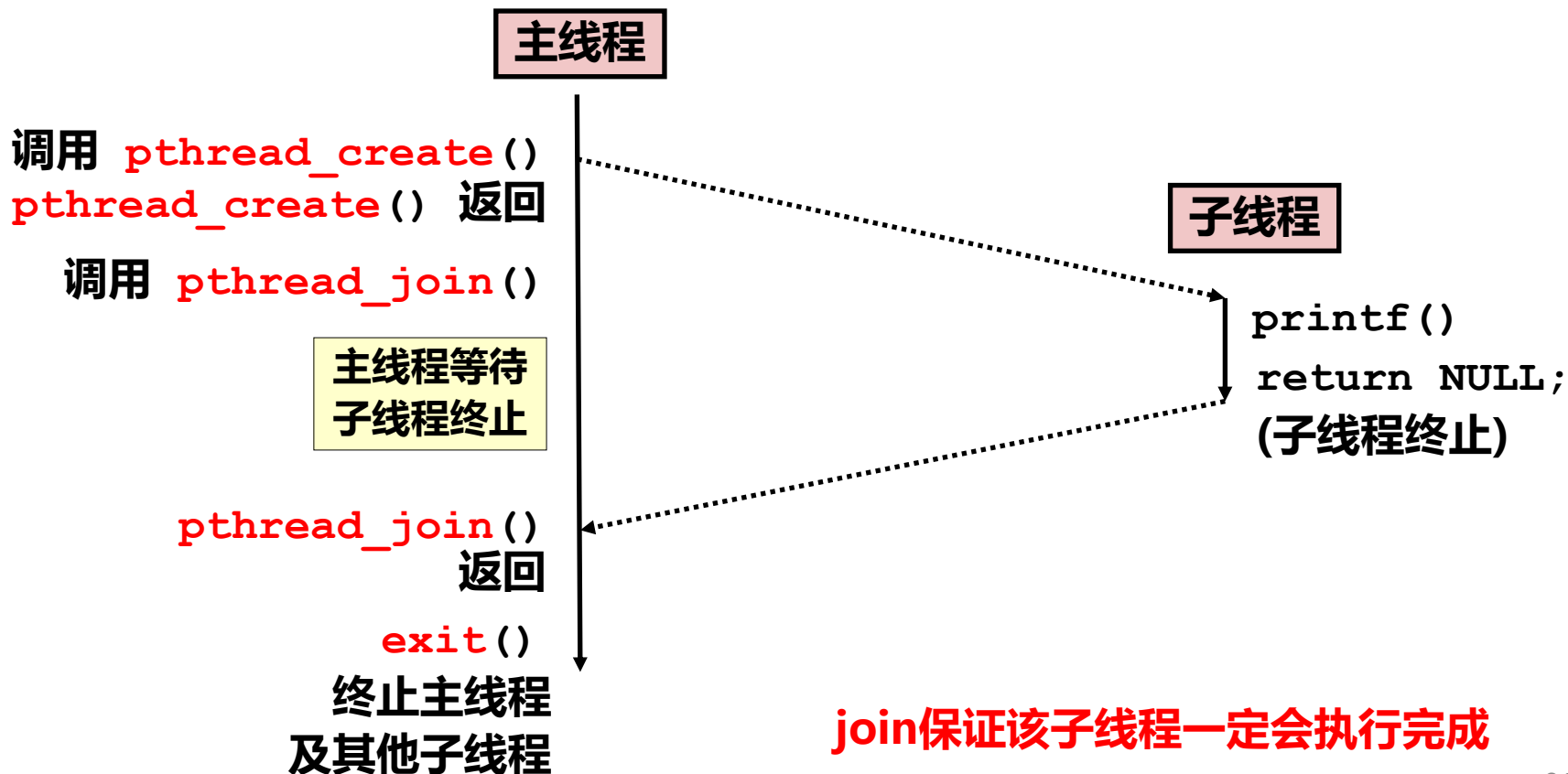
/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}
```

join操作：等待tid对应线程退出并回收

线程返回值

# 加入join之后的程序控制流



# 潜在的资源泄露

- 需要主线程手动调用回收资源

- 若主线程未调用则可能出现资源溢出

能否实现自动资源管理?

实际运行结果:

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

void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}

int main() {
    pthread_t tid;
    while(1) pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}
```

```
~/test$ ./a.out
Hello, world!
Hello, world!
...
Hello, world!
(stuck; errno:11)
```

# 解决方案：加入detach操作

- 调用detach可使线程进入“分离”状态
  - 分离线程不能被其他线程杀死或回收，退出时资源自动回收

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp) {
    pthread_detach(pthread_self());
    printf("Hello, world!\n");
    return NULL;
}
int main() {
    pthread_t tid;
    while(1) pthread_create(&tid, NULL, thread, NULL);
    exit(0);
}
```

返回自身线程ID

实际运行结果:

```
~/test$ ./a.out
Hello, world!
Hello, world!
...
```

# 如果detach和join一起使用会怎么样?

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp) {
    printf("Hello, world!\n");
    while(1);
    return NULL;
}
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_detach(tid);
    printf("%d\n", pthread_join(tid, NULL));
    exit(0);
}
```

一种实际运行结果:

```
~/test$ ./a.out
22
~/test$
```

22代表illegal arguments, 因为这个线程无法join



# detach代表线程完全独立了吗?

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp) {
    printf("Hello, world!\n");
    while(1);
    return NULL;
}
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_detach(tid);
    return 0;
}
```

一种实际运行结果:

```
~/test$ ./a.out
~/test$
```

**子线程仍被终结**

**} ← 核心原因: 主函数返回后隐式调用exit, 强制终止所有线程**

# 解决方案：加入pthread\_exit

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp) {
    printf("Hello, world!\n");
    while(1);
    return NULL;
}
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_detach(tid);
    pthread_exit(0);
}
```

实际运行结果：

```
~/test$ ./a.out
Hello, world!
(stuck)
```

只退出当前线程  
(参数为返回值)

# pthread使用小结

- 常用接口：创建、合并、分离、退出
- 线程资源默认需要显示回收
  - 主线程可使用合并（pthread\_join）回收其他线程
  - 线程也可主动调用分离函数（pthread\_detach）以自动回收
- 主线程退出默认会终结所有线程
  - 可改为调用退出（pthread\_exit），只退出主线程



# 线程

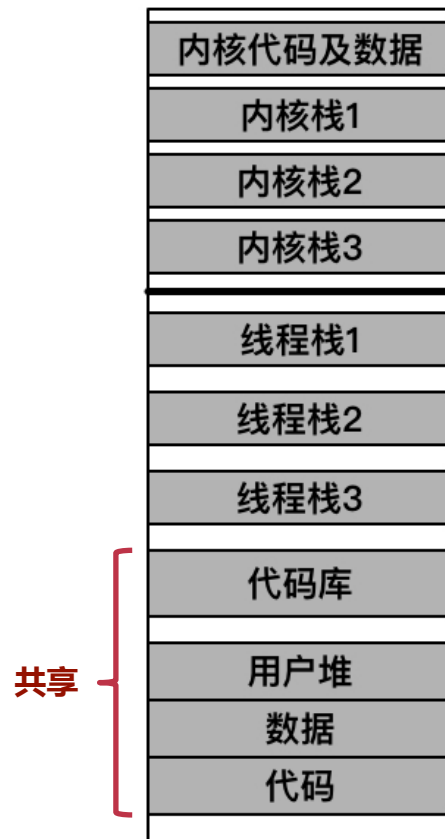
线程的概念 - 线程模型 - 相关数据结构 - 基本操作

# 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
  - 每个线程都有自己的**执行状态**

# 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
  - 可选（取决于内核实现）
- 其他区域共享
  - 数据、代码、堆.....

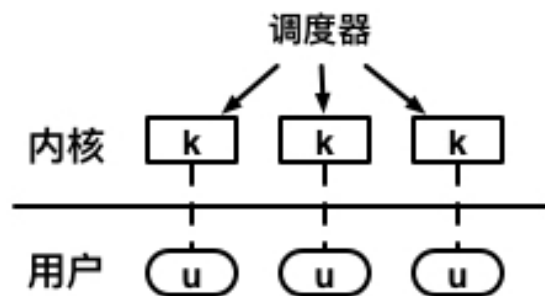


# 用户态线程与内核态线程

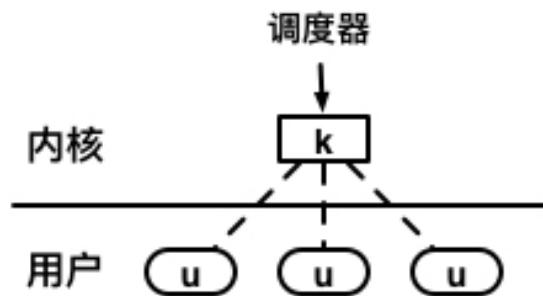
- **根据线程是否受内核管理，可以将线程分为两类**
  - 内核态线程：内核可见，受内核管理
  - 用户态线程：内核不可见，不受内核直接管理
- **内核态线程**
  - 由内核创建，线程相关信息存放在内核中
- **用户态线程**
  - 在应用态创建，线程相关信息主要存放在应用数据中

# 线程模型

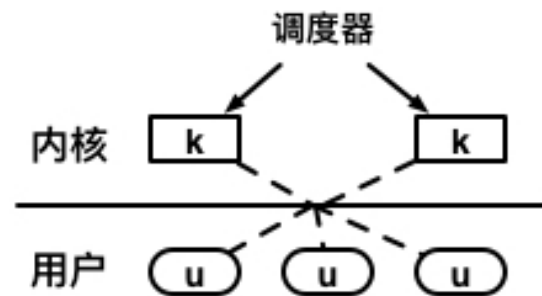
- 线程模型表示了用户态线程与内核态线程之间的联系
  - 多对一模型：多个用户态线程对应一个内核态线程
  - 一对一模型：一个用户态线程对应一个内核态线程
  - 多对多模型：多个用户态线程对应多个内核态线程



一对一模型



多对一模型

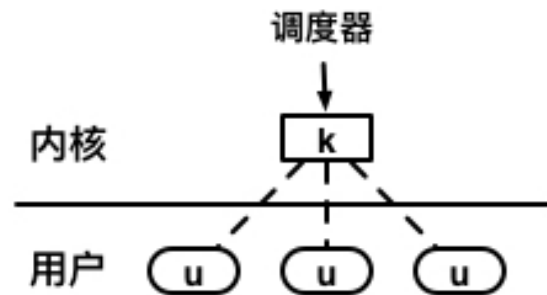


多对多模型



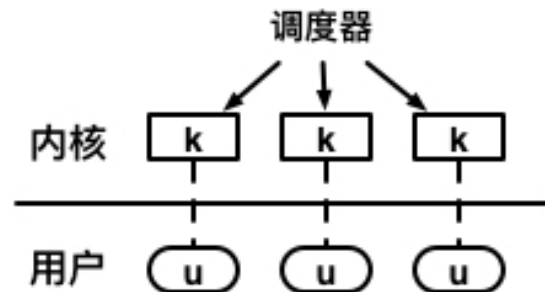
# 多对一模型

- 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（协程）



# 一对一模型

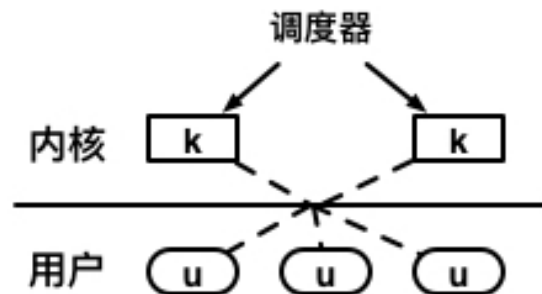
- 每个用户线程映射单独的内核线程
  - 优点：解决了多对一模型中的可扩展性问题
  - 缺点：线程切换需要经过内核
- 主流操作系统都采用一对一模型
  - Windows、Linux、OS X.....



# 多对多模型

- **N个用户态线程映射到M个内核态线程 ( $N > M$ )**
  - 优点：解决可扩展性问题（多对一）和线程切换时延问题（一对一）
  - 缺点：管理更为复杂

- **典型场景：虚拟机配置VCPU数量**



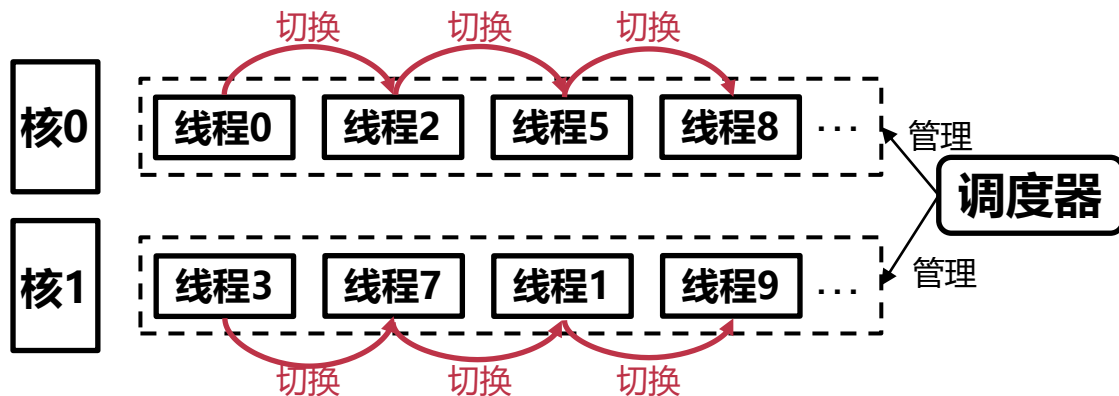
# 线程的相关数据结构：TCB

- **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）
  - 线程切换中会使用
- **应用态：可以由线程库定义**
  - Linux：pthread结构体
  - 可以认为是内核TCB的扩展

# 对比：进程vs. 线程

- 线程和进程的相似之处：

- 都可以与其他进程/线程并发执行（可能在不同核心上）
- 都可以进行切换
  - 引入线程后，调度管理单位由进程变为线程
    - 内核变量 `curr_proc` → `curr_thread`



# 对比：进程vs. 线程

- **线程和进程的不同之处：**

- 同一进程的不同线程共享代码和部分数据
  - 不同进程不共享虚拟地址空间
- 线程与进程相比开销较低
  - 进程控制（创建和回收）通常比线程更耗时
  - Linux的数据：
    - 创建和回收进程：~ 20K cycle
    - 创建和回收线程：~ 10K cycle（或更少）



# 线程的实现

数据结构 – 相关接口实现

# 回顾：进程的内核“户口”：PCB

- 包含处理器上下文、虚拟地址空间、内核栈等内容
- 如果需要支持线程，要如何修改？
  - 单个处理器上下文 -> 多个处理器上下文
  - 内核需为每个线程维护相应数据结构（方便以线程为粒度调度）

```
1 enum exec_status {NEW, READY, RUNNING,  
2                   ZOMBIE, TERMINATED};  
3  
4 struct process_v5 {  
5     // 处理器上下文  
6     struct context *ctx;  
7     // 虚拟地址空间 (包含页表基地址)  
8     struct vmSPACE *vmSPACE;  
9     // 内核栈  
10    void *stack;  
11    // 进程标识符  
12    int pid;  
13    // 退出状态  
14    int exit_status;  
15    // 子进程列表  
16    pcb_list *children;  
17    // 执行状态  
18    enum exec_status exec_status;  
19 };
```



# 线程对应数据结构：TCB（线程控制块）

- 将PCB中部分内容移入TCB中

- 每个线程TCB保存自己的处理器上下文、内核栈、退出/执行状态
- 进程PCB仍维护共享的地址空间
- PCB/TCB间相互引用，便于管理

```
1 // 支持线程之后的 PCB 结构实现
2 struct process_v6 {
3     // 虚拟地址空间
4     struct vmpace *vmpace;
5     // 进程标识符
6     int pid;
7     // 子进程列表
8     pcb_list *children;
9     // 包含的线程列表
10    tcb_list *threads;
11    // 包含的线程总数
12    int thread_cnt;
13 };
```

```
1 // 一种简单的 TCB 结构实现
2 enum exec_status {NEW, READY, RUNNING,
3                   ZOMBIE, TERMINATED};
4
5 struct tcb_v1 {
6     // 处理器上下文
7     struct context *ctx;
8     // 所属进程
9     struct process *proc;
10    // 内核栈
11    void *stack;
12    // 退出状态 (用于与 exit 相关的实现)
13    int exit_status;
14    // 执行状态
15    enum exec_status exec_status;
16 };
```

# Linux中线程创建的实现

- 在Linux中，线程创建由clone（**本用于创建进程**）实现
- 为创建线程，需要为clone配置多个特殊标记
  - CLONE\_VM: 创建的两个“进程”共享同一地址空间
  - CLONE\_THREAD: 新“进程”与原进程从属于同一进程
  - .....

```
1 const int clone_flags = (CLONE_VM | CLONE_FS
2                          | CLONE_FILES | CLONE_SYSVSEM
3                          | CLONE_SIGHAND | CLONE_THREAD
4                          | CLONE_SETTLS | ...);
5
6 ARCH_CLONE (&start_thread, STACK_VARIABLES_ARGS,
7             clone_flags, ...);
```

# 小结

Next: CPU调度, 进程间通信

