



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统

Operating Systems

刘川意 教授

哈尔滨工业大学 (深圳)

2025年9月

Module 2-2: 线程 (Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块 (TCB) -进阶
4. 用户级线程及实现 (Posix 线程)
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶

线程的引入

进程具有二个基本属性：

- 拥有资源的独立单位：它可独立分配虚地址空间、主存和其它
- 可独立调度和分派的基本单位

由于进程是一个资源的拥有者，因而在进程创建、撤销、调度切换时，系统需要付出较大的时空开销，**限制了并发程度**。

所以，**将进程的两个基本属性分开，可解决并发问题**

- 对于拥有资源的基本单位，不对其进行频繁切换
 - 对于调度的基本单位，不作为拥有资源的单位，“轻装上阵”
- 故引入线程的目的是简化线程间的通信，以小的开销来提高进程内的并发程度。

Module 2-2: 线程 (Thread)

1. 线程的引入-基础
2. 线程概念, 线程与进程的区别和联系-基础
3. 线程控制块 (TCB) -进阶
4. 用户级线程及实现 (Posix 线程)
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶

传统视角看进程

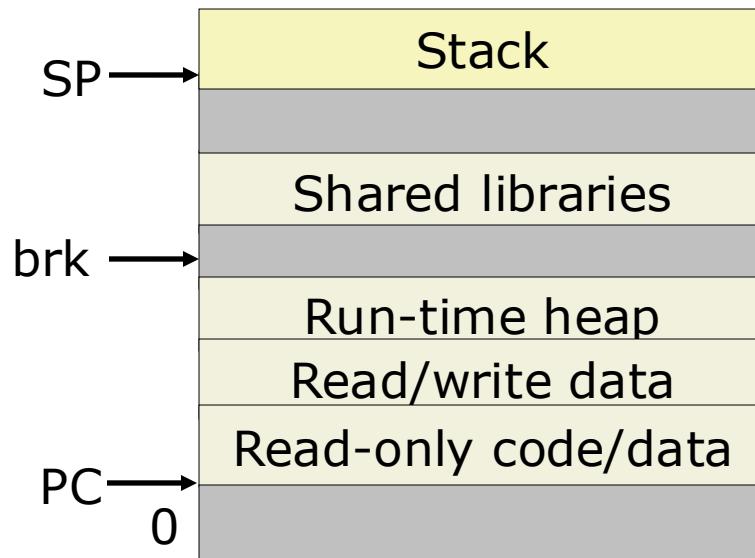
- Process = process context(进程上下文) + code, data, and stack

Process context

Program context:
Data registers
Condition codes
Stack pointer (SP)
Program counter (PC)

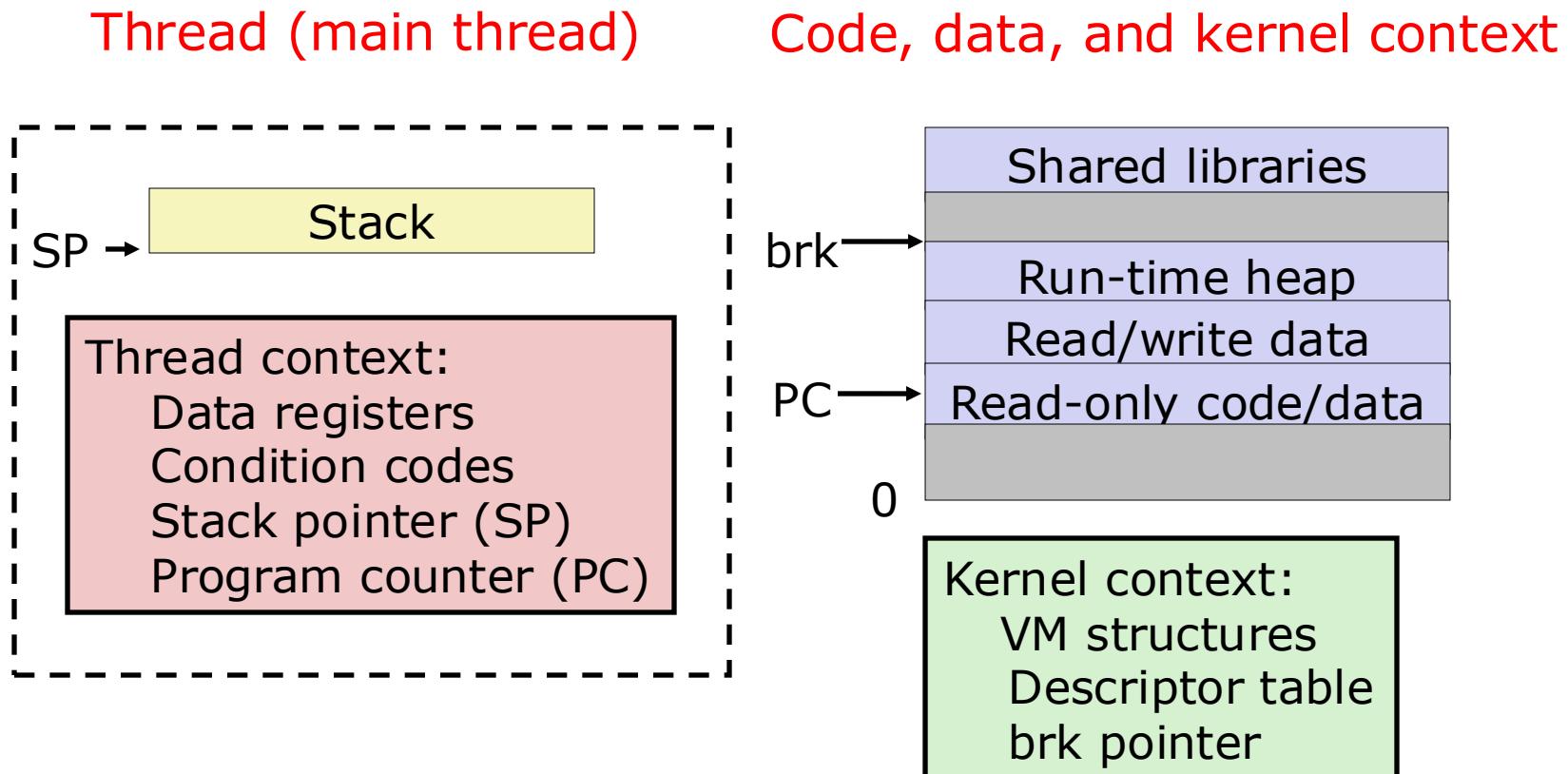
Kernel context:
VM structures
Descriptor table
brk pointer

Code, data, and stack



线程视角看进程

- Process = thread(线程) + code, data, and kernel context



多线程的地址空间布局

□ 分离的内核栈与用户栈

- 由于线程独立执行，故进程为每个线程都准备了不同的栈，供存放临时数据
- 内核中每个线程也有对应的内核栈，当线程切换到内核执行时，其栈指针就会切换成对应的内核栈

□ 共享的其他区域

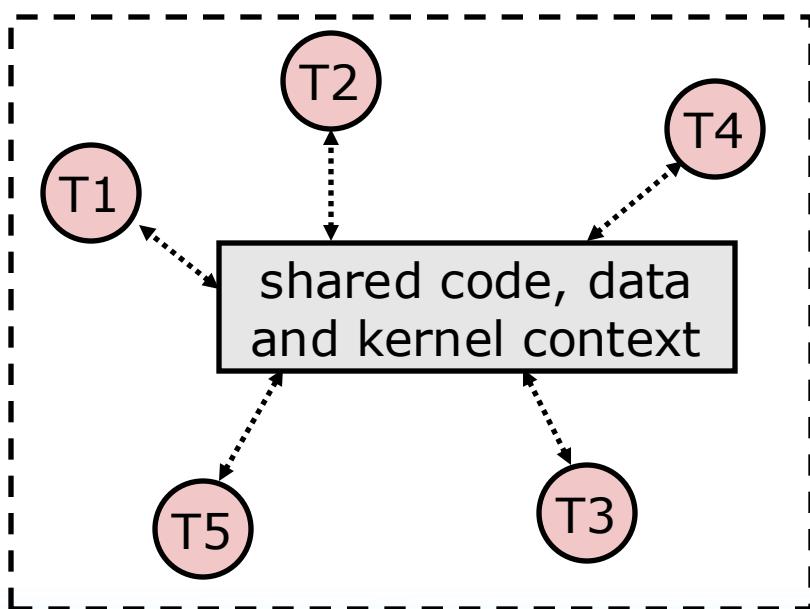
- 除了栈以外的其他区域，由该进程的所有线程共享
 - 包括：堆、数据段、代码段等
- 当同一进程的多个线程要动态分配内存时，它们的内存分配操作都是在同一个堆上完成
 - 因此malloc的实现需要使用**同步原语**



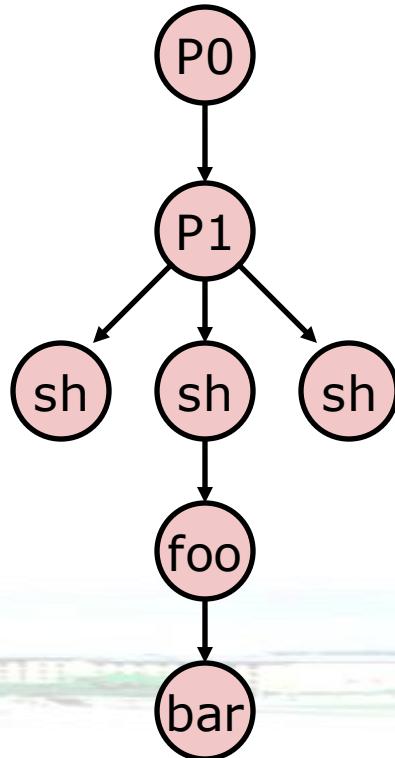
一个进程的多线程组织成池化结构(*pool of peers*)，而非像多进程之间形成的层次结构

- Threads associated with process form a pool of peers (线程池)
 - Unlike processes which form a tree hierarchy

Threads associated with process



Process hierarchy



Threads(线程) vs. Processes(进程)

- How threads and processes are similar
 - Each has its own logical control flow
 - Each can run concurrently with others (possibly on different cores)
 - Each is context switched(上下文切换)
- How threads and processes are different
 - Threads share all code and data (except local stacks)
 - ▶ Processes (typically) do not
 - Threads are somewhat less expensive than processes
 - ▶ Process control (creating and reaping) twice as expensive as thread control
 - ▶ Linux numbers:
 - ~20K cycles to create and reap a process
 - ~10K cycles (or less) to create and reap a thread



Threads(线程) vs. Processes(进程)

【例题】 在下面的叙述中正确的是()。

- A. 线程是比进程更小的能独立运行的基本单位
- B. 引入线程可提高程序并发执行的程度，可进一步提高系统效率
- C. 系统级线程和用户级线程的切换都需要内核的支持
- D. 一个进程一定包含多个线程

答案：B

解析：

- A. 线程不能独立运行，线程需要进程所获得的资源。
- B. 在用户级线程中，有关线程管理的所有工作都由应用程序完成，无需内核的干预，内核意识不到线程的存在。
- C. 一个进程至少包含一个主线程（线程数量大于等于1）。



Module 2-2: 线程(Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块（TCB）-进阶
4. 用户级线程及实现（Posix 线程）
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶



线程控制块TCB 用于保存线程信息

用户空间线程的操作函数如pthread_create、pthread_exit, 均由glibc实现

- x86_64体系结构的线程控制块 sysdeps/x86_64/nptl/tls.h

```
42  typedef struct
43  {
44      void *tcb;      /* Pointer to the TCB. Not necessarily the
45      | | | thread descriptor used by libpthread. */
46      dtv_t *dtv;
47      void *self;    /* Pointer to the thread descriptor. */
48      int multiple_threads;
49      int gscope_flag;
50      uintptr_t sysinfo;
51      uintptr_t stack_guard;
52      uintptr_t pointer_guard;
53      unsigned long int unused_vgetcpu_cache[2];
54      /* Bit 0: X86_FEATURE_1_IBT.
55      | Bit 1: X86_FEATURE_1_SHSTK.
56      */
57      unsigned int feature_1;
58      int __glibc_unused1;
59      /* Reservation of some values for the TM ABI. */
60      void *__private_tm[4];
61      /* GCC split stack support. */
62      void *__private_ss;
63      /* The lowest address of shadow stack, */
64      unsigned long long int ssp_base;
65      /* Must be kept even if it is no longer used by glibc since programs,
66      | | like AddressSanitizer, depend on the size of tcbhead_t. */
67      __128bits __glibc_unused2[8][4] __attribute__ ((aligned (32)));
68
69      void *__padding[8];
70  } tcbhead_t;
```

- arm体系结构的线程控制块
sysdeps/arm/nptl/tls.h

```
41  typedef struct
42  {
43      dtv_t *dtv;
44      void *private;
45  } tcbhead_t;
```

线程控制块由tcbhead_t数据
结构定义



Module 2-2: 线程(Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块（TCB）-进阶
4. 用户级线程及实现（Posix 线程）

2.1 Posix Threads Interface-基础

2.2 线程的创建与终止-基础

2.3 向线程传递参数-基础

5. 内核级线程及实现

3.1 内核级线程的引入与介绍-基础

3.2 内核级线程的切换-进阶

3.3 内核级线程的创建-进阶

Posix Threads Interface (Pthreads 接口)

- *Pthreads*: Standard interface for ~60 functions that manipulate threads from C programs (C程序中处理进程的标准接口)

- Creating threads (创建线程)

- ▶ `pthread_create()`

- Terminating threads (终止线程)

- ▶ `pthread_cancel()`
 - ▶ `pthread_exit()`
 - ▶ `exit()` [terminates all threads] , `RET` [terminates current thread]

- Joining and Detaching threads (结合和分离线程)

- ▶ `pthread_join()`
 - ▶ `pthread_detach()`

Module 2-2: 线程 (Thread)

1. 线程的引入-基础
2. 线程概念, 线程与进程的区别和联系-基础
3. 线程控制块 (TCB) -进阶
4. 用户级线程及实现 (Posix 线程)
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础**
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶

Creating Threads(创建线程)

- `int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg)`
 - `tid` : TID for the new thread returned by the subroutine.
 - `attr`(属性) : set thread attributes. You can specify a thread attributes object, or NULL for the default values.
 - `f` : the C routine that the thread will execute once it is created.
 - `arg`(参数) : A single argument that may be passed to `start_routine`. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.
- **Thread Attributes(线程属性):**
 - By default, a thread is created with certain attributes
 - Some of these attributes can be changed by the programmer via the thread attribute object.
 - `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object.

Terminating Threads (终止线程)

- `void pthread_exit(void *thread_return)`
- There are several ways in which a thread may be terminated:
 - The thread terminates *implicitly* when its top-level thread routine returns.
 - The thread terminates *explicitly* by calling the `pthread_exit` function.
 - If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate and then terminates the main thread and the entire process with a return value of `thread_return`.
 - The entire process is terminated due to making a call to the `exit()`
- `int pthread_cancel(pthread_t tid)`
 - calling the `pthread_cancel` function with the ID of the current thread.

Example: 线程的创建与终止

```
1. void *PrintHello(void *threadid)
2. {
3.     long tid;
4.     tid = (long)threadid;
5.     printf("Hello World! It's me, thread #%ld!\n", tid);
6.     pthread_exit(NULL);    回收当前线程
7. }
8. int main(int argc, char *argv[])
9. {
10.    pthread_t threads[NUM_THREADS]; /* NUM_THREADS = 5 */
11.    int rc;
12.    long t;
13.    for(t=0;t<NUM_THREADS;t++){
14.        printf("In main: creating thread %d\n", t);
15.        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
16.        if (rc){
17.            printf("ERROR; return code from pthread_create() is %d\n", rc);
18.            exit(-1);
19.        }
20.    }
21.    /* Last thing that main() should do */
22.    pthread_exit(NULL);    回收主线程
23. }
```

hello.c

Example: 线程的创建与终止

第一次运行:

```
os@os:~/OS_Course/lecture2-2$ ./hello
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
Hello World! It's me, thread #1!
Hello World! It's me, thread #0!
```

第二次运行:

```
os@os:~/OS_Course/lecture2-2$ ./hello
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
Hello World! It's me, thread #3!
Hello World! It's me, thread #1!
Hello World! It's me, thread #0!
```

Example: 单核 VS 多核

- 当一个程序创建了多个线程以后，其在多核机和单核机上面运行会有什么差别呢？多线程的执行速度一定比顺序执行快吗？
- 顺序执行多个task VS 并行执行多个task
- 单核 VS 多核

```
1. /* A task that takes some time to complete. The ID identifies
2. distinct tasks for printed messages. */
3.
4. void *task(void *ID) {
5.     long id = (long)ID;
6.     printf("Task %d started\n", id);
7.     int i;
8.     double result = 0.0;
9.     for (i = 0; i < 10000000; i++) {
10.         result = result + sin(i) * tan(i);
11.     }
12.     printf("Task %d completed with result %e\n", id, result);
13. }
```

pthreads_test.c

Example: 单核 VS 多核

```
1. void *print_usage(int argc, char *argv[]) {
2.     printf("Usage: %s serial|parallel num_tasks\n", argv[0]);
3.     exit(1);
4. }
5.
6. int main(int argc, char *argv[]) {
7.     if (argc != 3) {print_usage(argc, argv);}
8.     int num_tasks = atoi(argv[2]);
9.
10.    if (!strcmp(argv[1], "serial")) {
11.        serial(num_tasks);
12.    } else if (!strcmp(argv[1], "parallel")) {
13.        parallel(num_tasks);
14.    }
15.    else {
16.        print_usage(argc, argv);
17.    }
18.
19.    printf("Main completed\n");
20.    pthread_exit(NULL);
21.}
```

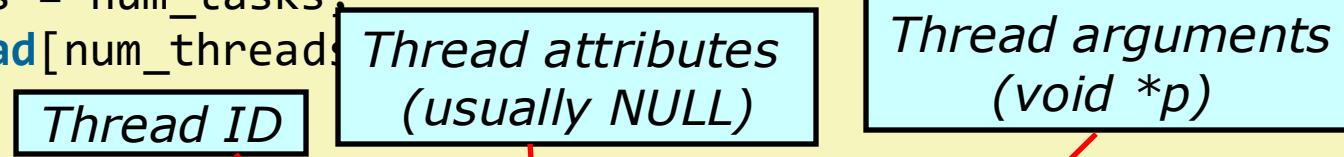
pthreads_test.c

Example: 单核 VS 多核

```

/* Parallel(并行): Run 'task' num_tasks times, creating a separate thread
for each call to 'task'. */
1. void *parallel(int num_tasks)
2. {
3.     int num_threads = num_tasks;
4.     pthread_t thread[num_threads];
5.     int rc;
6.     long t;
7.     for (t = 0; t < num_threads; t++) {
8.         printf("Creating thread %d\n", t);
9.         rc = pthread_create(&thread[t], NULL, task, (void *)t);
10.        if (rc) {
11.            printf("ERROR: return code from pthread_create() is %d\n", rc);
12.            exit(-1);

```



```

1. void *serial(int num_tasks) {
2.     long i;
3.     for (i = 0; i < num_tasks; i++)
4.     {
5.         task((void *)i);
6.     }
7. } /*Serial (顺序) : Run 'task' num_tasks times serially.*/

```

pthreads_test.c

pthreads_test.c

Example: 单核 VS 多核

```
[os@os:~/OS_Course/lecture2-2$ ./pthreads_test
Usage: ./pthreads_test serial|parallel num_tasks
[os@os:~/OS_Course/lecture2-2$ ./pthreads_test serial 2
Task 0 started
Task 0 completed with result -3.153838e+06
Task 1 started
Task 1 completed with result -3.153838e+06
Main completed
[os@os:~/OS_Course/lecture2-2$ ./pthreads_test parallel 2
Creating thread 0
Creating thread 1
Main completed
Thread 1 started
Task 1 started
Thread 0 started
Task 0 started
Task 1 completed with result -3.153838e+06
Thread 1 done
Task 0 completed with result -3.153838e+06
Thread 0 done
```

Example: 单核 VS 多核——观察执行时间

VMware中设置处理器为单核：



Example: 单核 VS 多核——观察执行时间

VMware中单核处理器:

顺序执行:

```
os@os:~/OS_Course/lecture2-2$ time ./pthreads_test serial 4
Task 0 started
Task 0 completed with result -3.153838e+06
Task 1 started
Task 1 completed with result -3.153838e+06
Task 2 started
Task 2 completed with result -3.153838e+06
Task 3 started
Task 3 completed with result -3.153838e+06
Main completed

real    0m0.118s
user    0m0.115s
sys     0m0.001s
```

Example: 单核 VS 多核——观察执行时间

VMware中单核处理器：

并行执行： os@os:~/OS_Course/lecture2-2\$ time ./pthreads_test parallel 4

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Main completed
Thread 3 started
Task 3 started
Thread 2 started
Task 2 started
Thread 1 started
Task 1 started
Thread 0 started
Task 0 started
Task 2 completed with result -3.153838e+06
Task 1 completed with result -3.153838e+06
Thread 1 done
Task 3 completed with result -3.153838e+06
Thread 3 done
Task 0 completed with result -3.153838e+06
Thread 0 done
Thread 2 done
```

real	0m0.113s
user	0m0.104s
sys	0m0.005s

单核的顺序执行和并发执行在时间上差别不大

Example: 单核 VS 多核——观察执行时间

VMware中设置为4核处理器：



Example: 单核 VS 多核——观察执行时间

VMware中设置4核处理器:

顺序执行:

```
os@os:~/OS_Course/lecture2-2$ time ./pthreads_test serial 4
Task 0 started
Task 0 completed with result -3.153838e+06
Task 1 started
Task 1 completed with result -3.153838e+06
Task 2 started
Task 2 completed with result -3.153838e+06
Task 3 started
Task 3 completed with result -3.153838e+06
Main completed

real    0m0.113s
user    0m0.102s
sys     0m0.012s
```

4核顺序执行和单核顺序执行在时间上差别不大

Example: 单核 VS 多核——观察执行时间

VMware中设置4核处理器:

并行执行: [os@os:~/OS_Course/lecture2-2\$ time ./pthreads_test parallel 4

```
Creating thread 0
Creating thread 1
Thread 0 started
Task 0 started
Creating thread 2
Creating thread 3
Main completed
Thread 3 started
Task 3 started
Thread 2 started
Task 2 started
Thread 1 started
Task 1 started
Task 0 completed with result -3.153838e+06
Thread 0 done
Task 3 completed with result -3.153838e+06
Thread 3 done
Task 1 completed with result -3.153838e+06
Thread 1 done
Task 2 completed with result -3.153838e+06
Thread 2 done
```

real	0m0.056s
user	0m0.195s
sys	0m0.010s

4核并行执行明显比单核并行执行在时间上快

Module 2-2: 线程 (Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块 (TCB) -进阶
4. 用户级线程及实现 (Posix 线程)
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶

Passing Arguments to Threads (向线程传递参数)

- `pthread_create()` 允许程序员将一个参数传递给线程并启动例程。对于必须传递多个参数的情况，通过创建一个包含所有参数的结构体，然后在 `pthread_create()` 中传递一个指向该结构体的指针，可以轻松克服此限制
- 所有参数必须通过引用传递并强制转换为 `(void *)`
- 考虑到不确定的启动和调度，如何将数据安全地传递给新创建的线程？

```
1. void *PrintHello(void *threadid)
2. {
3.     long taskid;
4.     sleep(1);
5.     taskid = *(long *)threadid;
6.     printf("Hello from thread %ld\n", taskid);
7.     pthread_exit(NULL);
8. }
```

Passing Arguments to Threads (向线程传递参数)

```

1. int main(int argc, char *argv[])
2. {
3.     pthread_t threads[NUM_THREADS]; /* 设置NUM_THREADS为8 */
4.     int rc;
5.     long t;
6.
7.     for(t=0;t<NUM_THREADS;t++) {
8.         printf("Creating thread %ld\n", t);
9.         rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
10.        if (rc) {
11.            printf("ERROR; return code from pthread_create() is %d\n", rc);
12.            exit(-1);
13.        }
14.    }
15.    pthread_exit(NULL);
16. }
```

hello_arg2.c

所有线程共用变量

rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);

os@os:~/OS_Course/lecture2-2\$./hello_arg2
 Creating thread 0
 Creating thread 1
 Creating thread 2
 Creating thread 3
 Creating thread 4
 Creating thread 5
 Creating thread 6
 Creating thread 7
 Hello from thread 8
 Hello from thread 8



Passing Arguments to Threads (向线程传递参数)

```

1. int main(int argc, char *argv[])
2. {
3.     pthread_t threads[NUM_THREADS];
4.     long taskids[NUM_THREADS];
5.     int rc, t;
6.
7.     for(t=0;t<NUM_THREADS;t++) {
8.         taskids[t] = t;
9.         printf("Creating thread %d\n", t);
10.        rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
11.        if (rc) {
12.            printf("ERROR; return code from pthread_create() is %d\n", rc);
13.            exit(-1);
14.        }
15.    }
16.    pthread_exit(NULL);
17. }
```

hello_arg1.c

向每个线程单独传递一个变量

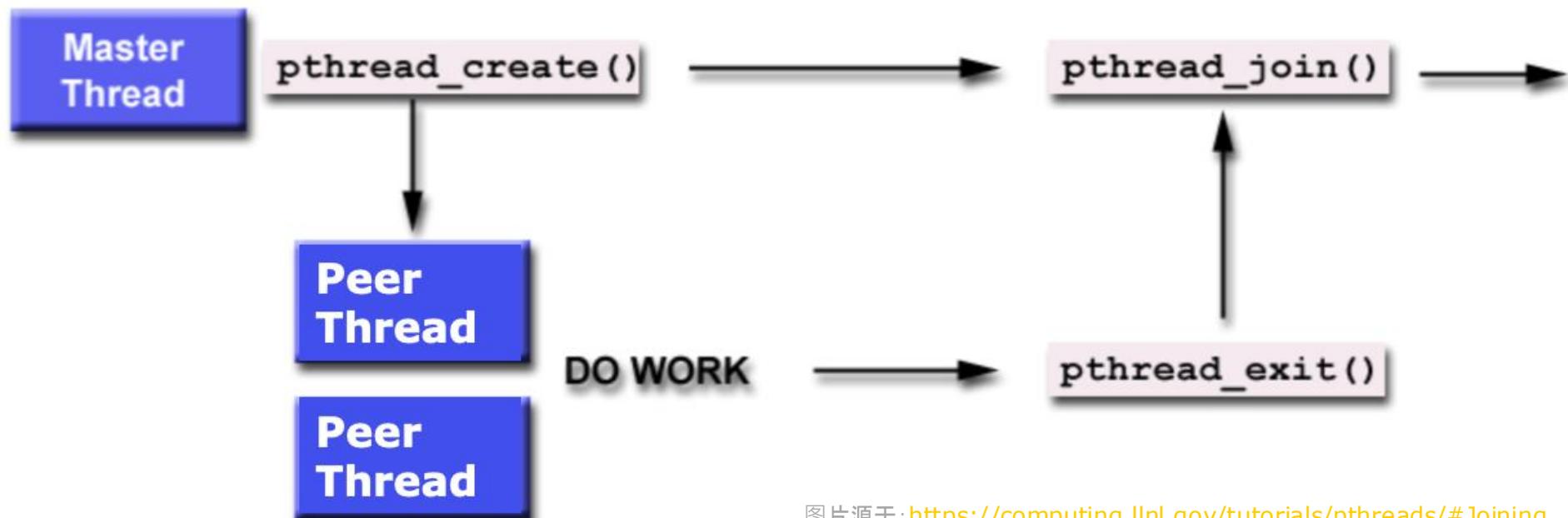
./hello_arg1

os@os:~/OS_Course/lecture2-2\$./hello_arg1

Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Hello from thread 3
Hello from thread 1
Hello from thread 5
Hello from thread 7
Hello from thread 4
Hello from thread 2
Hello from thread 6
Hello from thread 0

Joining(结合) and Detaching(分离) Threads

- `int pthread_join(pthread_t tid, void **thread_return)`
 - The `pthread_join` function blocks until thread `tid` terminates
 - unlike the Linux `wait` function, the `pthread_join` function can only wait for a specific thread to terminate
 - A joining thread can match one `pthread_join()` call. It is a logical error to attempt multiple joins on the same thread.



图片源于:<https://computing.llnl.gov/tutorials/pthreads/#Joining>

Joining(结合) and Detaching(分离) Threads

- `int pthread_detach(pthread_t tid)`
 - The `pthread_detach` function detaches the joinable thread `tid`.
 - By default, threads are created joinable.
 - each joinable thread should be either explicitly reaped by another thread or detached by a call to the `pthread_detach` function.

- To explicitly create a thread as joinable or detached, the `attr` argument in the `pthread_create()` routine is used. The typical 4 step procedure is:
 - Declare a pthread attribute variable of the `pthread_attr_t` data type
 - Initialize the attribute variable with `pthread_attr_init()`
 - Set the attribute detached status with `pthread_attr_setdetachstate()`
 - When done, free library resources used by the attribute with `pthread_attr_destroy()`

Joining(结合) and Detaching(分离) Threads

```
1. void *BusyWork(void *t)
2. {
3.     int i;
4.     long tid;
5.     double result=0.0;
6.     tid = (long)t;
7.     printf("Thread %ld starting...\n",tid);
8.     for (i=0; i<1000000; i++)
9.     {
10.         result = result + sin(i) * tan(i);
11.     }
12.     printf("Thread %ld done. Result = %e\n",tid, result);
13.     pthread_exit((void*) t);
14. }
```

join.c

Joining(结合) and Detaching(分离) Threads

```
1. /* Initialize and set thread detached attribute */
2. pthread_attr_init(&attr);
3. pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
4.
5. for(t=0; t<NUM_THREADS; t++) {
6.     printf("Main: creating thread %ld\n", t);
7.     rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
8.     if (rc) {
9.         printf("ERROR; return code from pthread_create() is %d\n", rc);
10.        exit(-1);
11.    }
12. }
13. /* Free attribute and wait for the other threads */
14. pthread_attr_destroy(&attr);
15. for(t=0; t<NUM_THREADS; t++) {
16.     rc = pthread_join(thread[t], &status);
17.     if (rc) {
18.         printf("ERROR; return code from pthread_join() is %d\n", rc);
19.         exit(-1);
20.     }
21.     printf("Main: completed join with thread %ld having a status of %ld\n", t,
22. (long)status);
23. }
24. printf("Main: program completed. Exiting.\n");
25. pthread_exit(NULL);
```

join.c

Module 2-2: 线程 (Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块 (TCB) -进阶
4. 用户级线程及实现 (Posix 线程)
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶

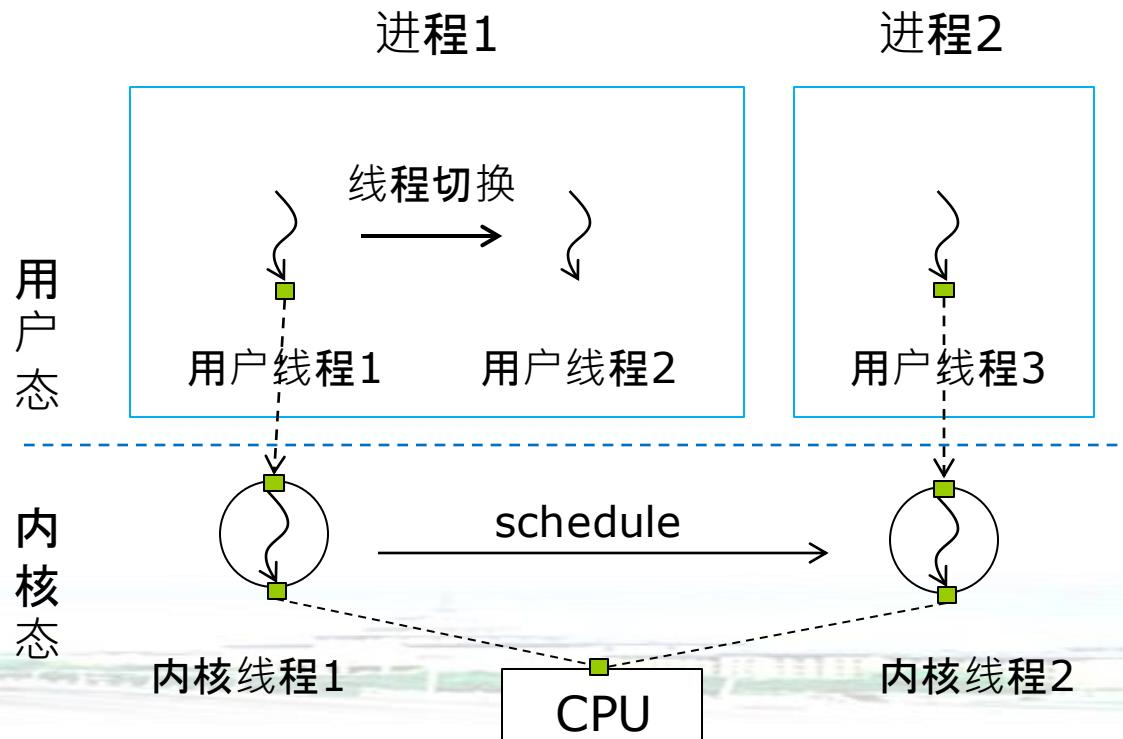


内核级线程的引入与介绍

为什么要引入内核级线程？与内核态相关的操作（如系统调用）需要内核态线程的协作才能完成

假设用户线程1的任务是通过网卡下载数据。由于网卡驱动是由OS内核管理，因此下载数据请求必须进入内核完成。用户线程1进入内核后，可能要等待网络连接成功，**这段时间CPU完全可以切换出去执行其他任务。**

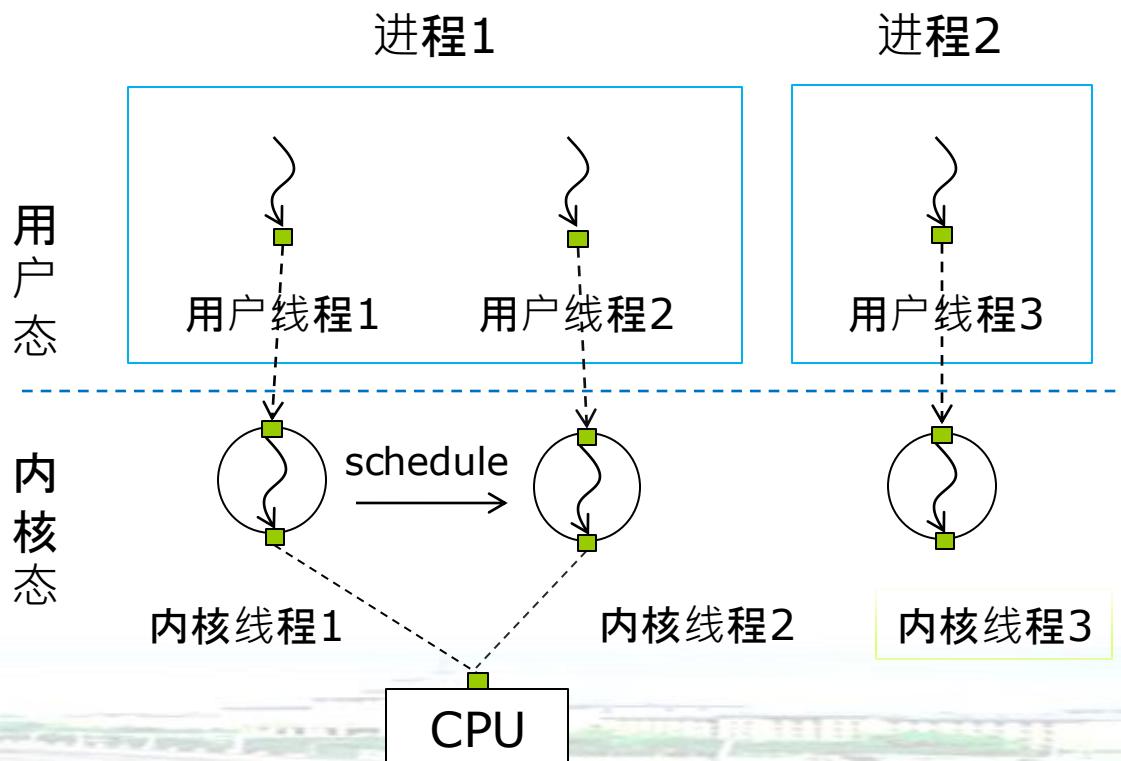
但由于OS无法感知到用户线程2的存在（假设同一个进程中多用户线程跟内核线程采用多对一模型），所以OS内核不能切换到用户线程2去执行，即：如果一个用户级线程在内核中阻塞，则这个进程的所有用户级线程将**全部阻塞**



内核级线程的引入与介绍

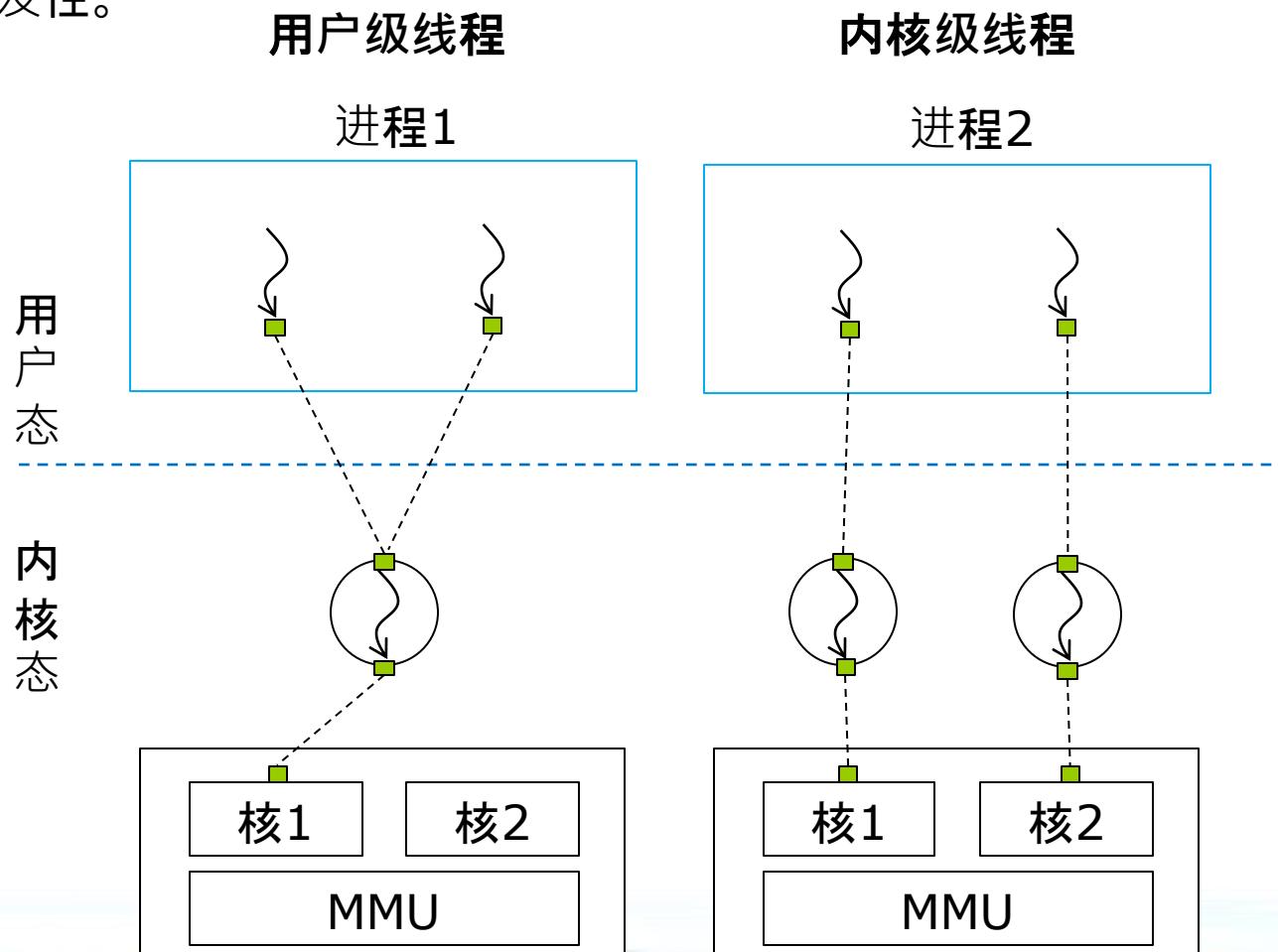
内核级线程就是要让内核态内存和用户态内存合作创建一个指令执行序列，内核级线程的TCB等信息是创建在OS内核中的，OS通过这些数据结构感知和操控内核级线程，如下图所示。

引入内核级线程后，用户线程1进入内核阻塞后，OS可以执行用户线程2，进程1可以继续执行。



内核级线程的引入与介绍

内核级线程可以有效支持多核处理器，共用地址空间(共享MMU和缓存)，可以进一步提高并发性。





Module 2-2: 线程(Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块（TCB）-进阶
4. 用户级线程及实现（Posix 线程）
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶



内核级线程的切换

回顾用户级线程的切换，主要分为三步：**TCB切换**、根据TCB中存储的栈指针完成**用户栈切换**、根据用户栈中压入的函数返回地址完成**PC指针切换**。

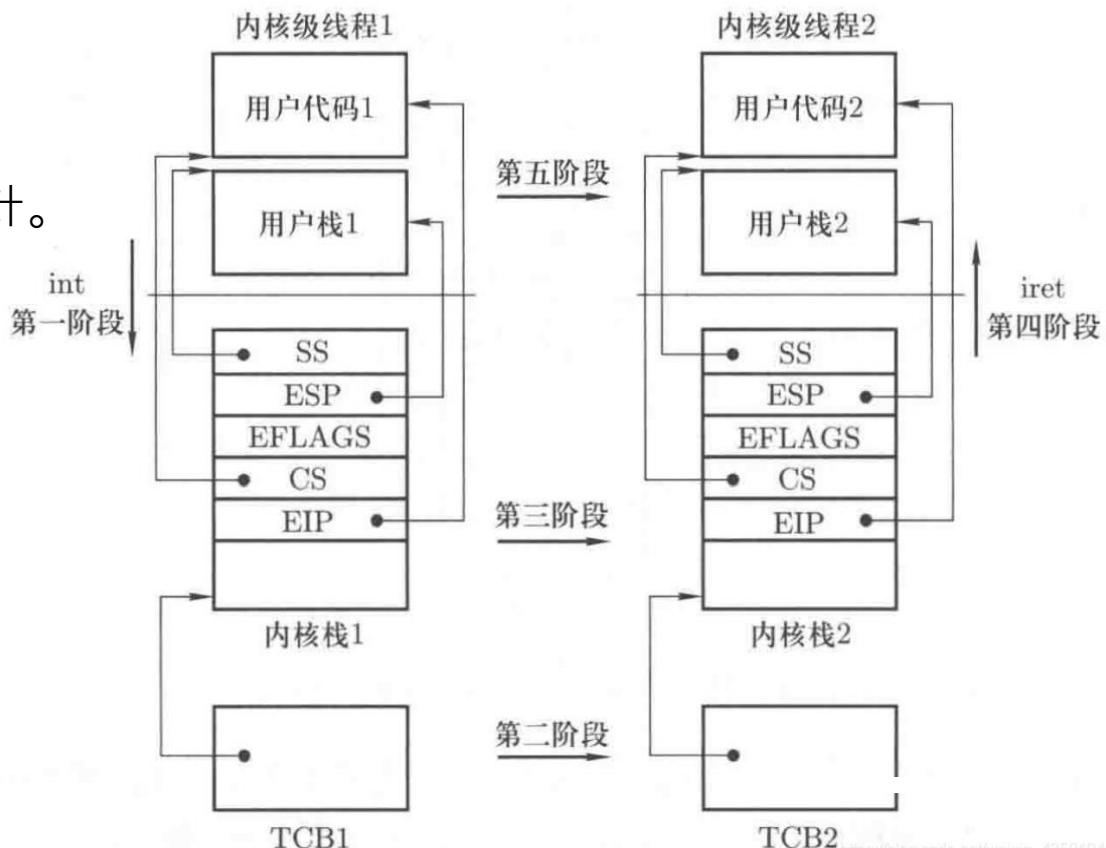
内核级线程的切换也要分为TCB切换、用户栈切换、PC指针切换这三个步骤。两者有两个重要区别：

1. 用户级线程通过调用用户态函数完成切换，而内核级线程必须进入内核才能引起切换，所以内核级线程的切换应该**从中断开始**（中断会导致从用户态到内核态的切换）。
2. 与用户级线程相比，内核级线程切换栈时要**同时切换用户栈和内核栈**。

内核级线程的切换

内核级线程的切换可以分为5个阶段：

1. 中断进入。
2. 调用schedule函数，引起TCB切换。
3. 内核栈的切换。
4. 中断返回。
5. 用户栈的切换，并切换PC指针。





Module 2-2: 线程(Thread)

1. 线程的引入-基础
2. 线程概念，线程与进程的区别和联系-基础
3. 线程控制块（TCB）-进阶
4. 用户级线程及实现（Posix 线程）
 - 2.1 Posix Threads Interface-基础
 - 2.2 线程的创建与终止-基础
 - 2.3 向线程传递参数-基础
5. 内核级线程及实现
 - 3.1 内核级线程的引入与介绍-基础
 - 3.2 内核级线程的切换-进阶
 - 3.3 内核级线程的创建-进阶



内核级线程的创建

创建内核线程的关键是初始化TCB、内核栈和用户栈。具体步骤如下：

1. 创建一个TCB，主要存放内核栈的栈指针；
2. 分配一个内核栈，其中存放用户态程序的PC指针、用户栈地址以及执行现场；
3. 分配用户栈，主要存放进入用户态函数时用到的参数等。

Linux kernel创建内核线程：(linux/kernel/kthread.c)

1. 创建kthread_create_info结构体，其中包含TCB信息
2. 调用kthread_create_on_node函数，将kthread_create_info添加到kthread_create_list链表，唤醒kthreadd线程
3. kthreadd线程从kthread_create_list链表遍历每个kthread_create_info结构体
4. 调用create_kthread()函数创建一个为kthread的线程，在kthread线程中执行需要的逻辑



内核级线程的创建

■ kthread_create_info结构体

kernel/kthread.c

```
38 struct kthread_create_info
39 {
40     /* Information passed to kthread() from kthreadd. */
41     int (*threadfn)(void *data);
42     void *data;
43     int node;
44
45     /* Result passed back to kthread_create() from kthreadd. */
46     struct task_struct *result;
47     struct completion *done;
48
49     struct list_head list;
50 };
```



内核级线程的创建

make qemu CPUS=1

```
xv6 kernel is booting

init: starting sh
$ threadtest
Main thread: starting 3 threads.
Main thread: all threads created, waiting for them to finish.
Thread 0: started.
Thread 0: counter 0
Thread 1: started.
Thread 1: counter 0
Thread 2: started.
Thread 2: counter 0
Thread 0: counter 1
Thread 1: counter 1
Thread 2: counter 1
Thread 0: counter 2
Thread 1: counter 2
Thread 2: counter 2
Thread 0: counter 3
Thread 1: counter 3
Thread 2: counter 3
Thread 0: counter 4
Thread 1: counter 4
Thread 2: counter 4
Thread 0: finished.
Thread 1: finished.
Thread 2: finished.
Main thread: all threads finished.
```



内核级线程的创建

【例题】下列关于线程的描述中，错误的是()。

- A. 内核级线程的调度由操作系统完成
- B. 用户级线程间的切换比内核级线程间的切换效率高
- C. 用户级线程可以在不支持内核级线程的操作系统上实现
- D. 操作系统为每个用户级线程建立一个线程控制块

答案:D

D选项，用户级线程的管理完全由用户态的线程库完成。



内核级线程的创建

【例题】关于线程的以下描述中，哪一个是正确的()。

- A. 线程共享进程的地址空间和资源
- B. 每个线程用有独立的进程上下文
- C. 线程的切换比进程的切换代价更大
- D. 线程是独立于进程的执行单元

答案:A



A central graphic features the words "Thank You" in English and Chinese (謝謝). Surrounding this center are numerous other "Thank You" messages in various global languages, each with a small icon representing a different culture or setting.

- Top Left: "obrigado" (Portuguese) and "ありがとうございます" (Japanese)
- Middle Left: "Köszi" (Hungarian)
- Bottom Left: "merci bien" (French), "Danke" (German), "Danku" (Swedish), and "Děkuji" (Czech)
- Bottom Center: "terima kasih" (Indonesian)
- Bottom Right: "תודה" (Hebrew)
- Right Side: "спасибо" (Russian), "sagol" (Arabic), "Kiitos" (Finnish), "хвала" (Ukrainian), "Gracias" (Spanish), and "cảm ơn cô" (Vietnamese)
- Center: "谢 谢" (Chinese), "شكرا" (Arabic), "谢谢" (Chinese), "ευχαριστώ" (Greek), and "Tak" (Malay)