



哈爾濱工業大學 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第二章：线程

陈芳林 副教授

哈尔滨工业大学（深圳）

2024年秋

Email: chenfanglin@hit.edu.cn

■ 线程

- 线程的引入与介绍
- 多线程模型
- Posix 线程

线程的基本概念--线程的引入

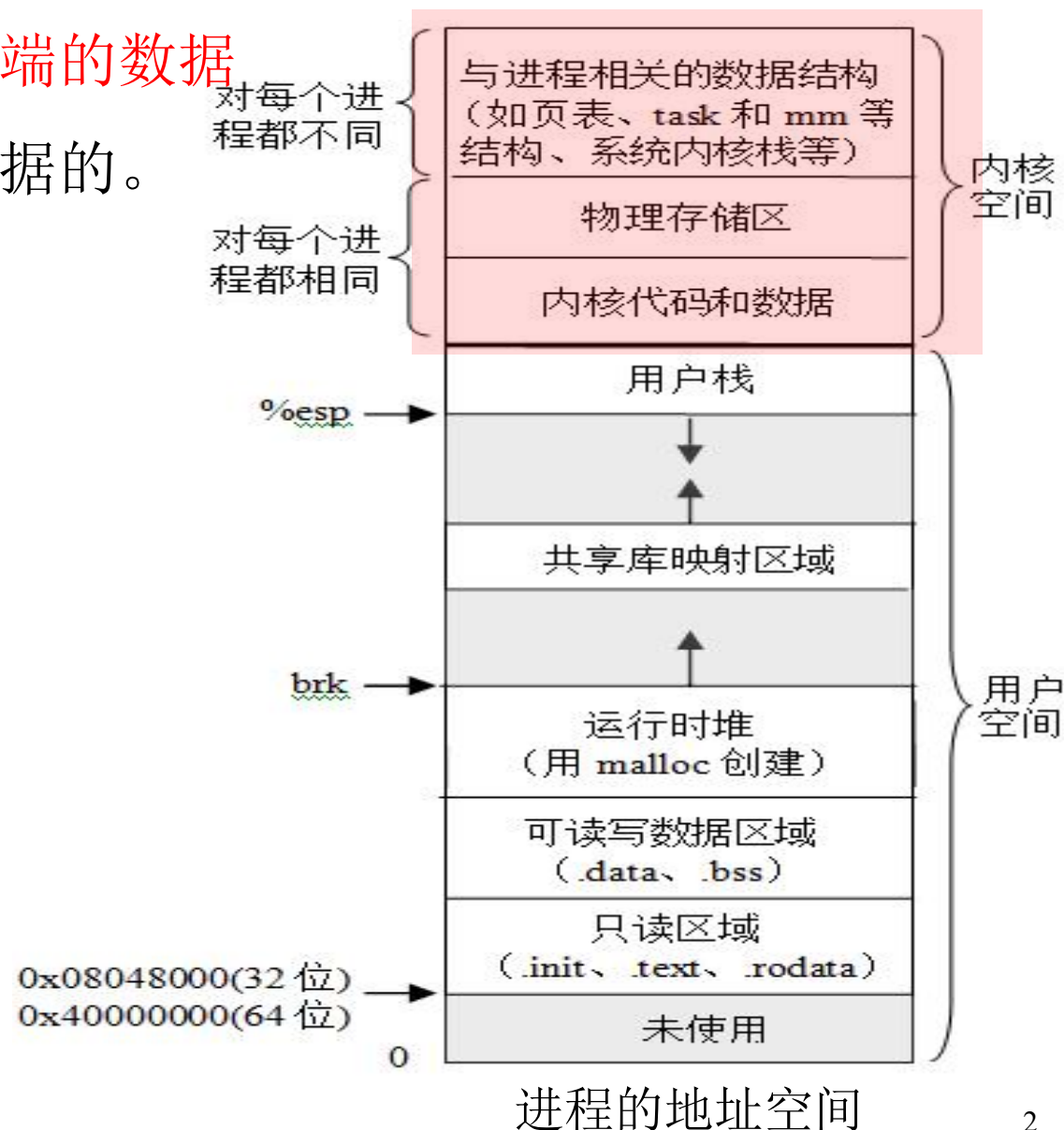
■ 例：数据库服务器如何同时处理来自多个客户端的数据查询请求，这些请求都是针对同一数据库的数据的。

➤ 解决方法：

- ✓ (1) 设置一个进程顺序处理所有请求；
- ✓ (2) 设置多个进程分别处理多个请求；

➤ 存在的问题：

- ✓ (1) 进程同步复杂；
- ✓ (2) 进程切换的系统开销大。



■ 1、线程定义：

➤ 线程是进程内一个相对独立的**可调度的执行单元**。

■ 2、线程属性：

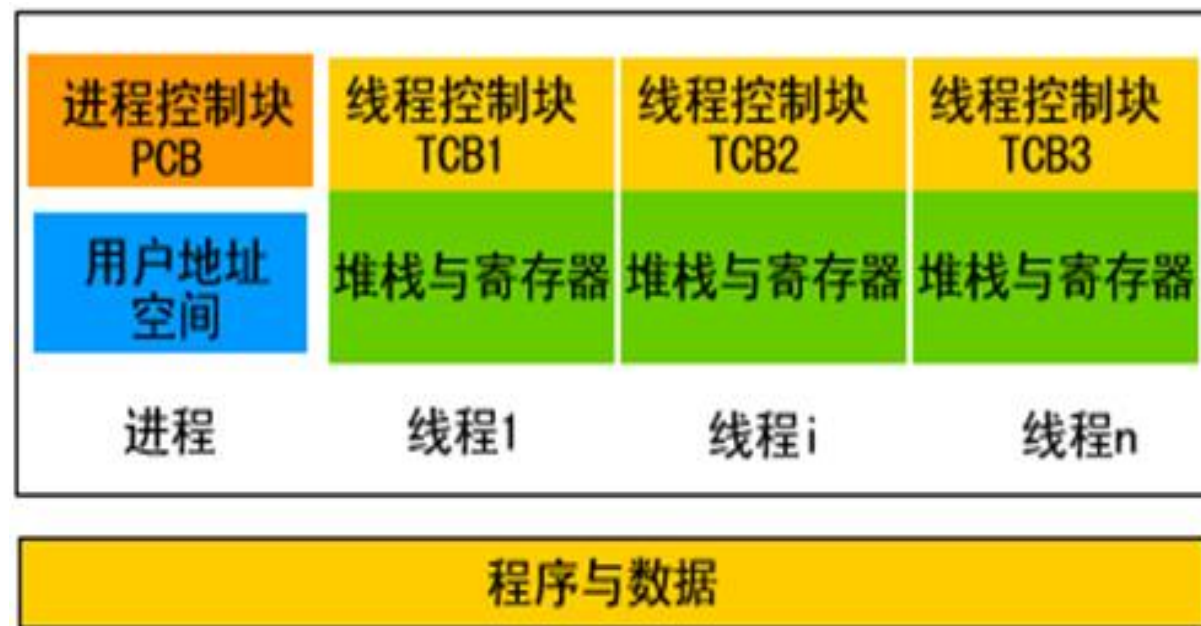
➤ (1) 轻型实体：TCB+堆栈+一组寄存器

➤ (2) 独立调度和分派的基本单位；

➤ (3) 可并发执行；

➤ (4) 共享进程资源；

➤ (5) 线程有生命周期，在生命期中有状态变化。



■ 3、线程基本状态:

- (1) 就绪态;
- (2) 运行态;
- (3) 阻塞态;

■ Windows2000/xp的线程状态:

- | | | |
|------------|-----------|-----------|
| (1) 初始化状态; | (2) 就绪状态; | (3) 备用状态; |
| (4) 运行状态; | (5) 等待状态; | (6) 转换状态 |
| (7) 终止状态。 | | |

■ 进程 = 系统上下文+ code, data, and stack

系统级上下文

寄存器上下文:

寄存器 (Data registers)

条件码 (condition codes)

栈指针 (Stack pointer)

程序计数器 (PC)

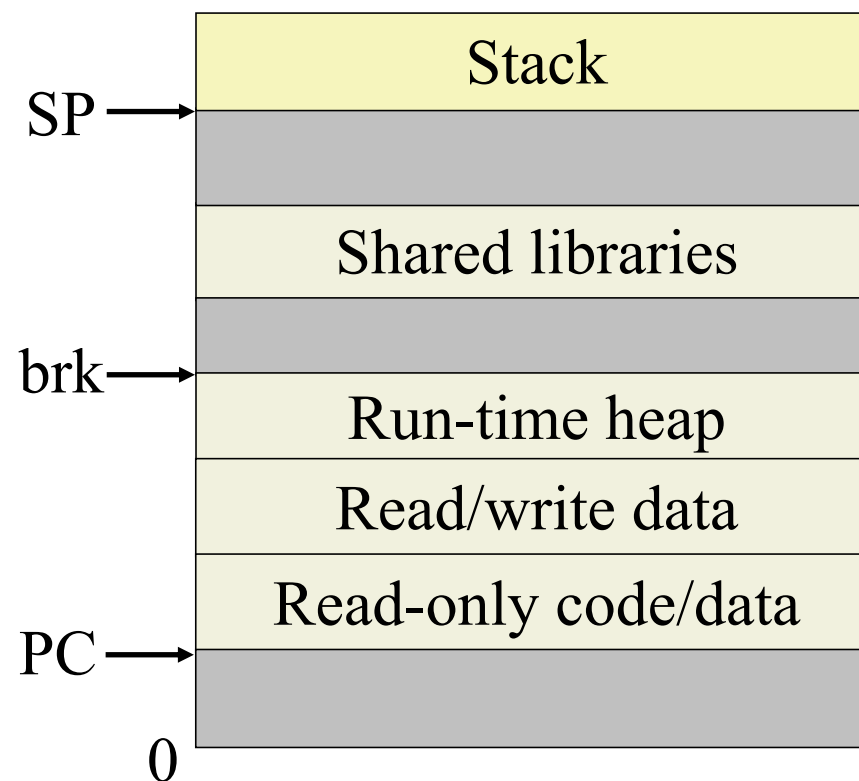
kernel context:

VM structures

描述符表 (Descriptor table)

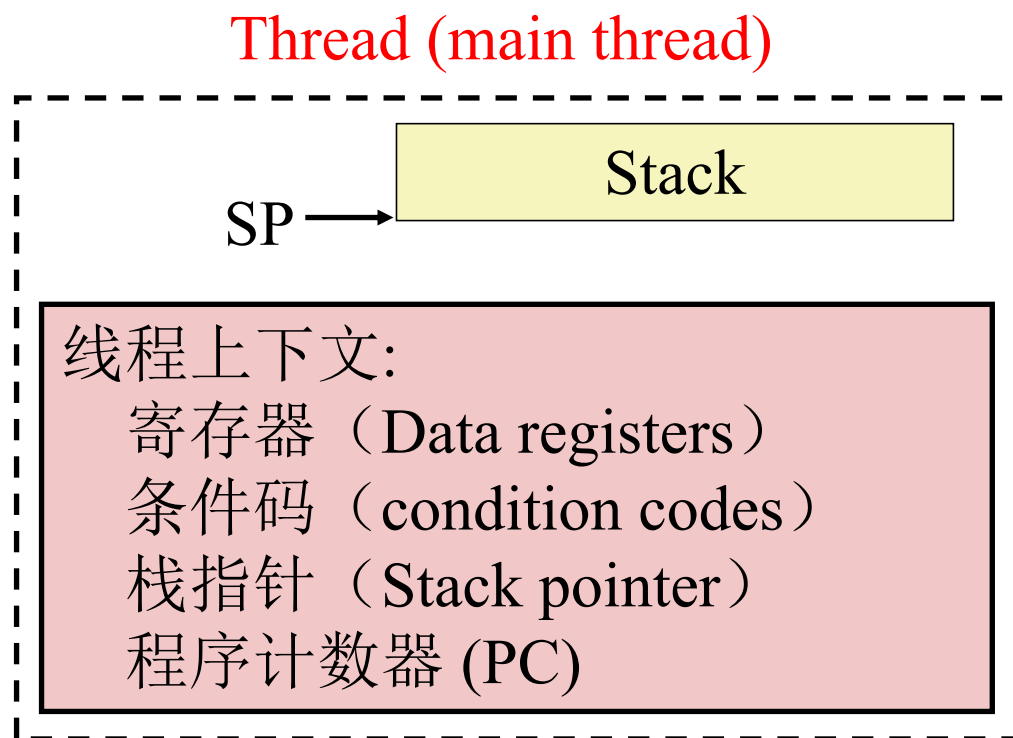
brk pointer

code, data, and stack

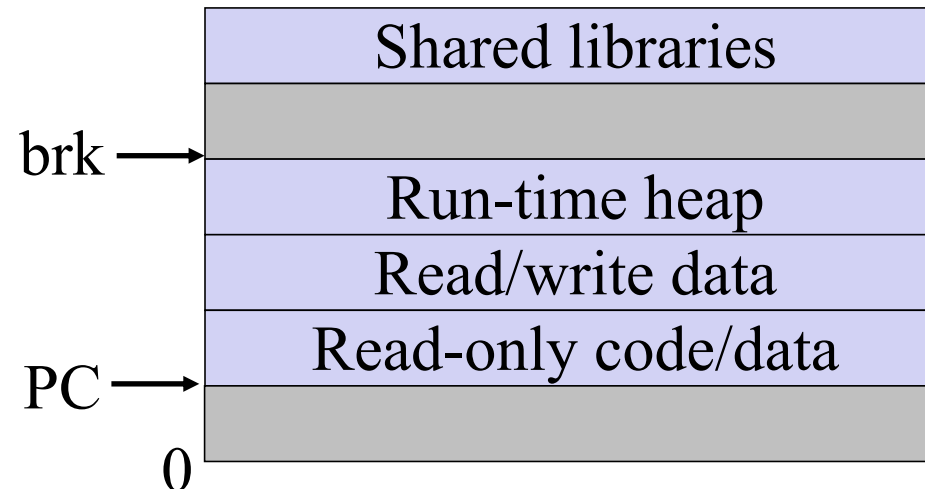


■ 进程的内涵发生了改变：进程作为除CPU外的系统资源的分配单元，线程作为CPU的分配单元。

■ 进程 = 线程 + code, data, and kernel context



Code, data, and kernel context



kernel context:

- VM structures
- 描述符表 (Descriptor table)
- brk pointer

Thread 1 (main thread主线程)

stack 1

线程1上下文:

寄存器 (Data registers)

条件码 (condition codes)

栈指针1 (Stack pointer 1)

程序计数器1 (PC 1)

Thread 2 (peer thread对等线程)

stack 2

线程2上下文:

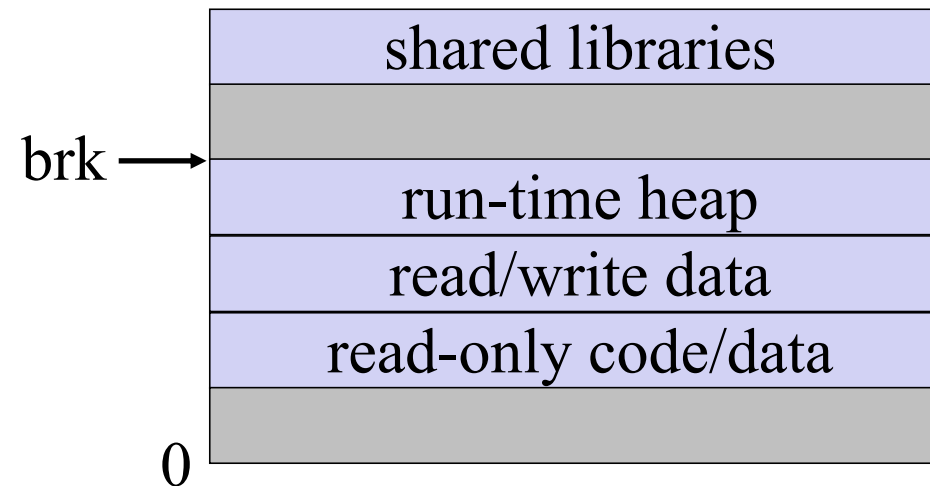
寄存器 (Data registers)

条件码 (condition codes)

栈指针2 (Stack pointer 2)

程序计数器2 (PC 2)

Shared code and data



kernel context:

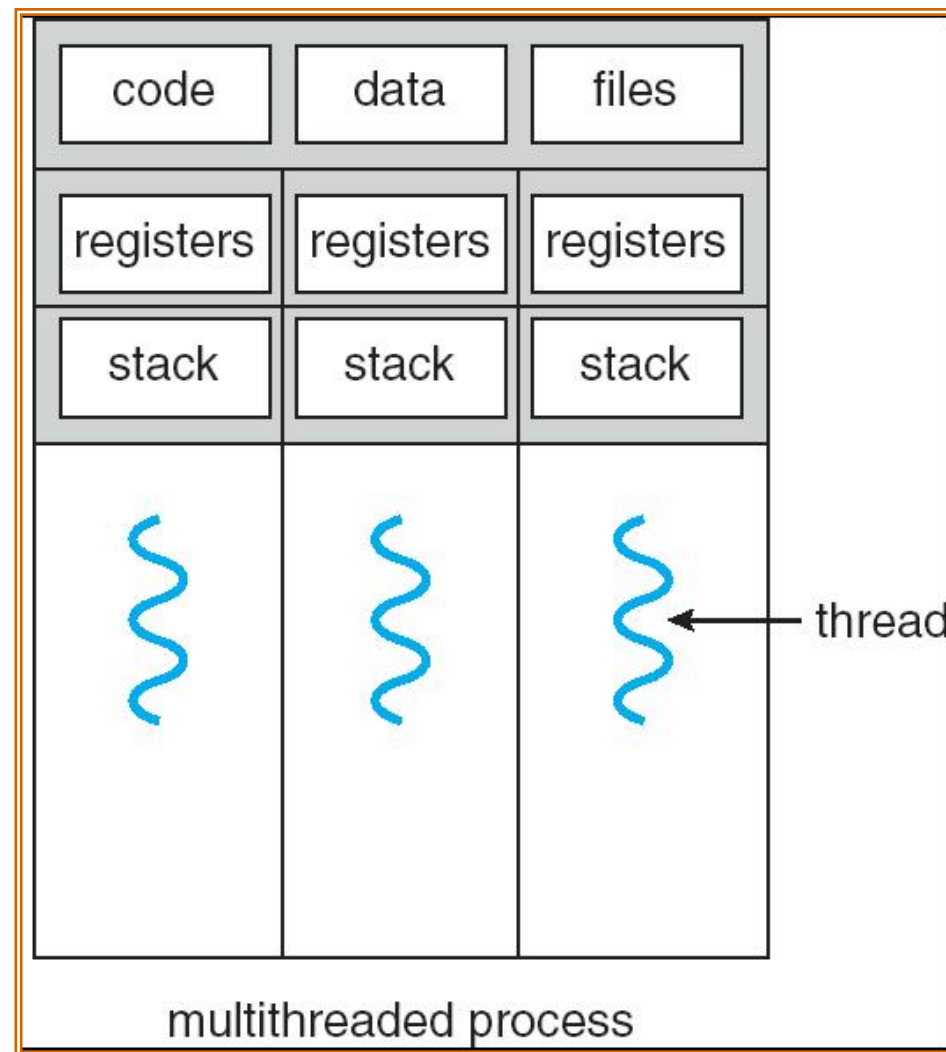
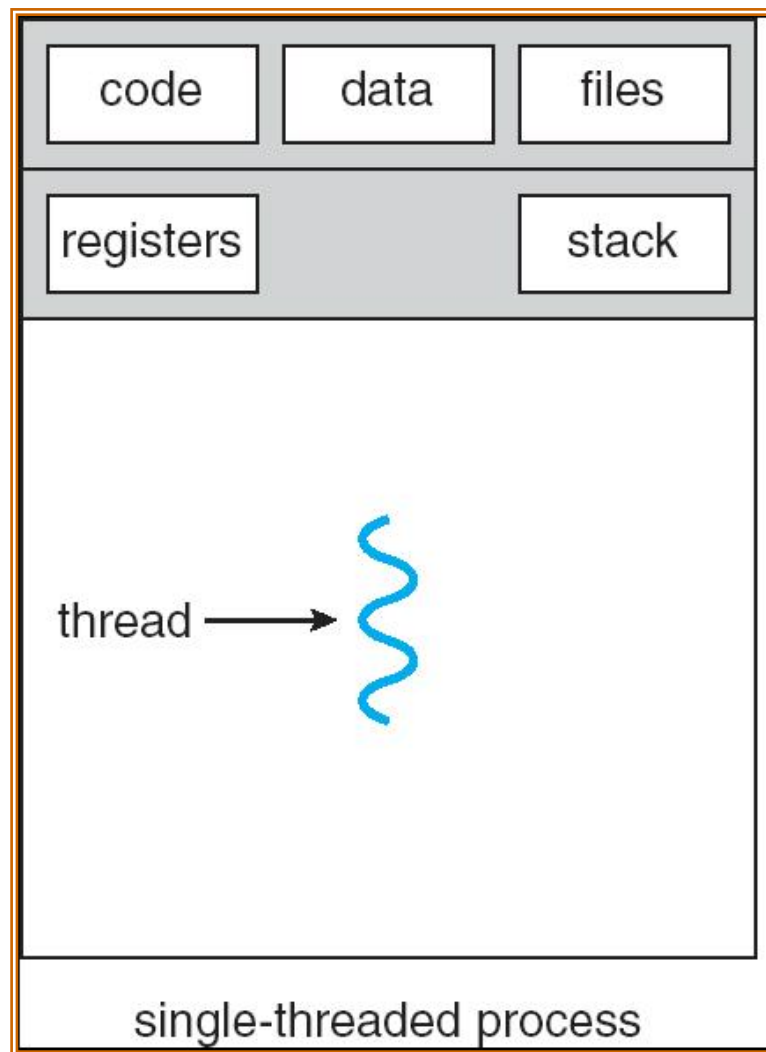
VM structures

描述符表 (Descriptor table)

brk pointer

单线程进程与多线程进程

- 创建多个轻量级的进程处理多个客户端的请求。

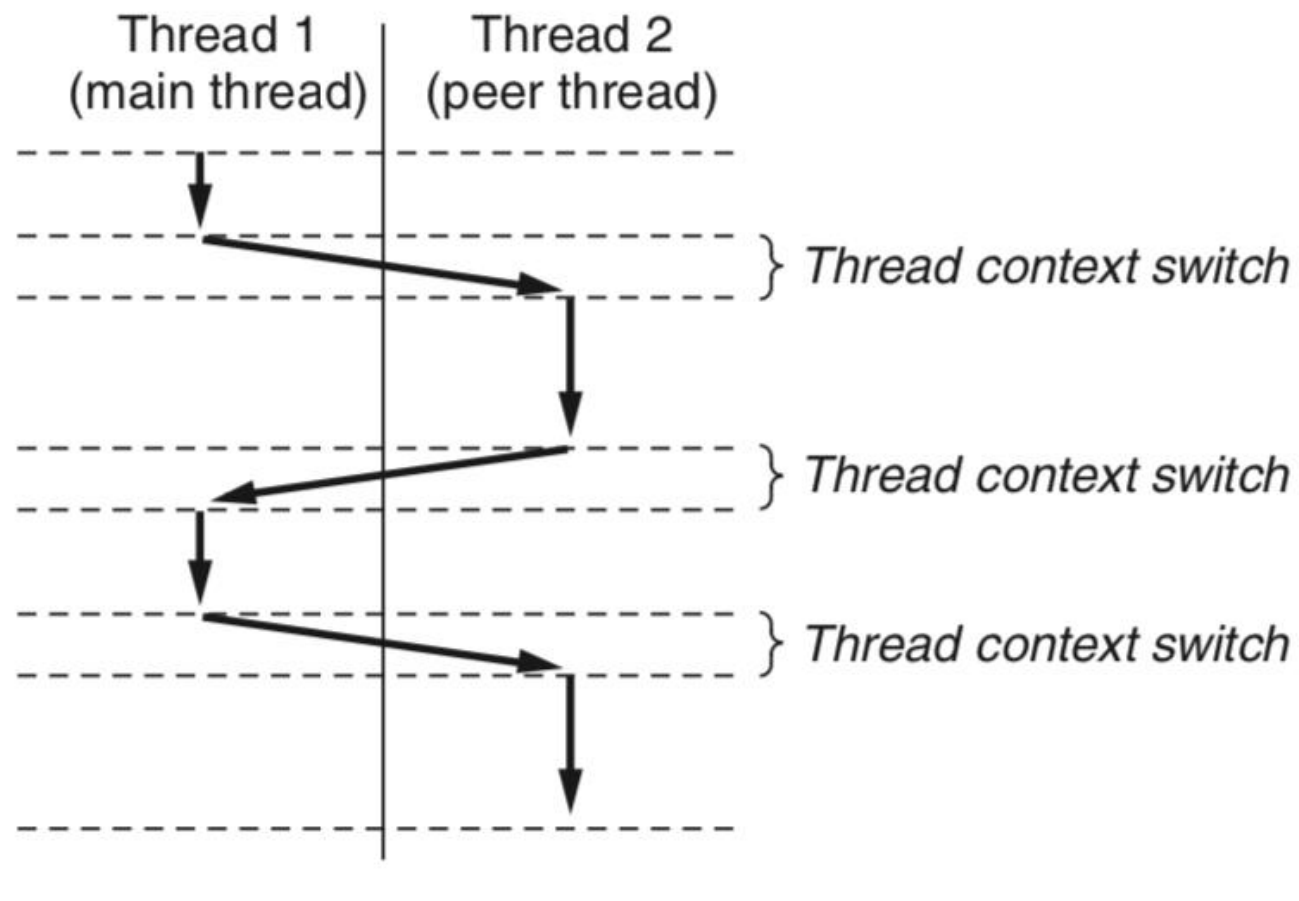


- 线程是轻量级进程，基本的CPU执行单元。
- 不单独拥有系统资源，只拥有其在运行中必不可少的资源，如线程ID、程序计数器、寄存器集合和堆栈
- 一个进程可以关联多个线程
- 每个线程有着自己的控制流
- 属于同一个进程下的线程共享所属进程的地址空间
- 每个线程有着自己的局部变量的堆栈
 - ✓ 但对于其他线程来说并不是被保护的

线程切换
开销小

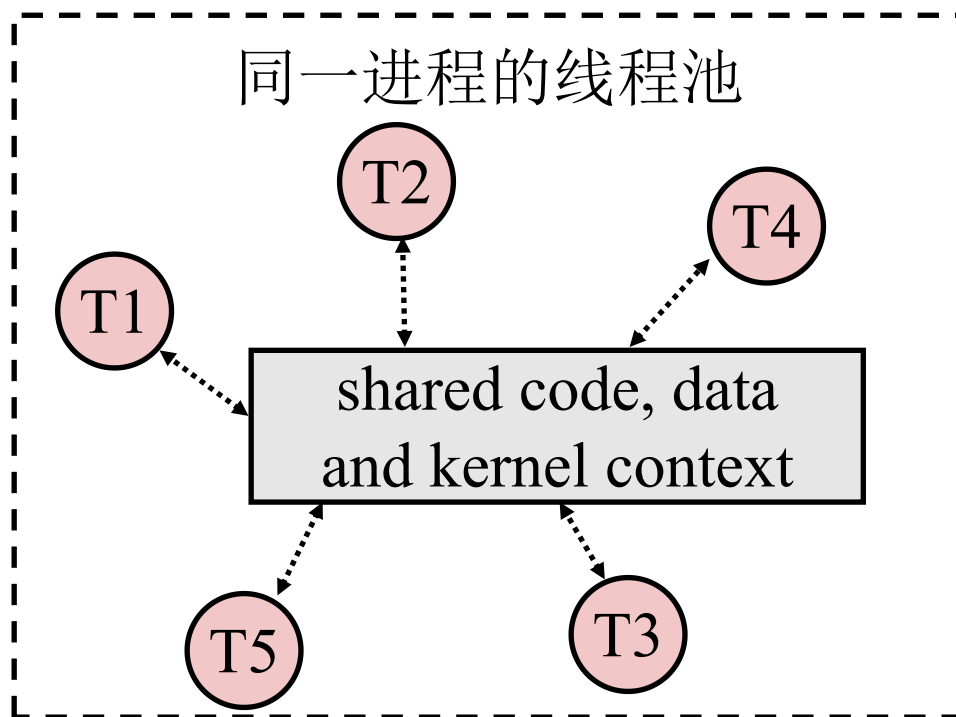
- 每个进程都是作为一个单一线程启动的，也被称为主线程
- 在之后的某个时间点，主线程创建了对等线程，然后两个线程并发的执行
- 并发线程执行

Time

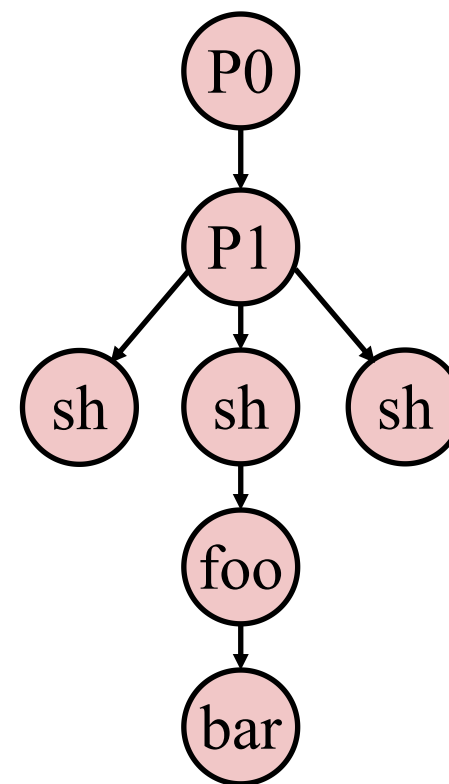


■ 与进程关联的线程形成一个线程池

➤ 并不像进程一样有一个树状的继承关系



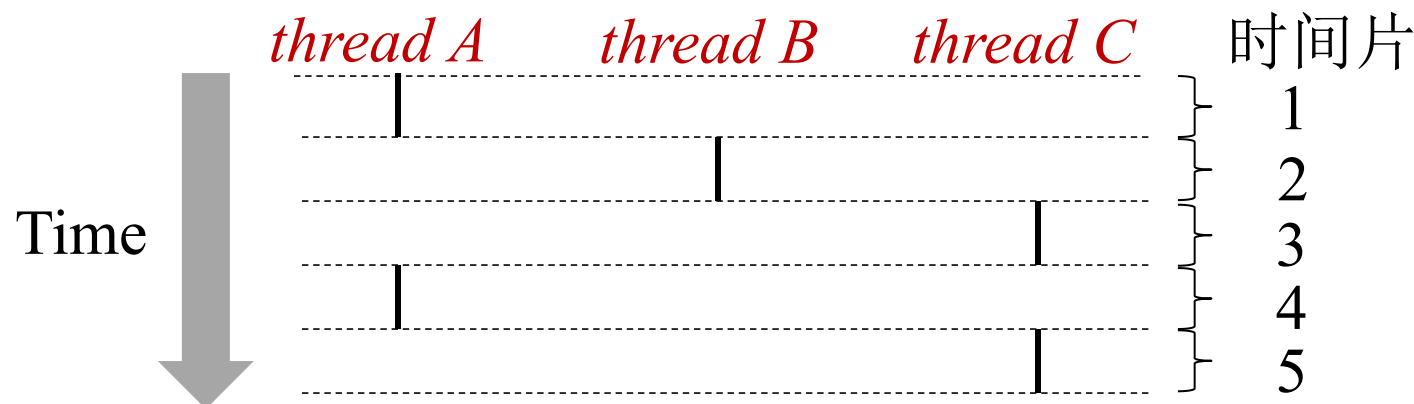
进程继承树



- 在时间线上控制流有重叠的线程是并发的
- 否则，他们是顺序执行的

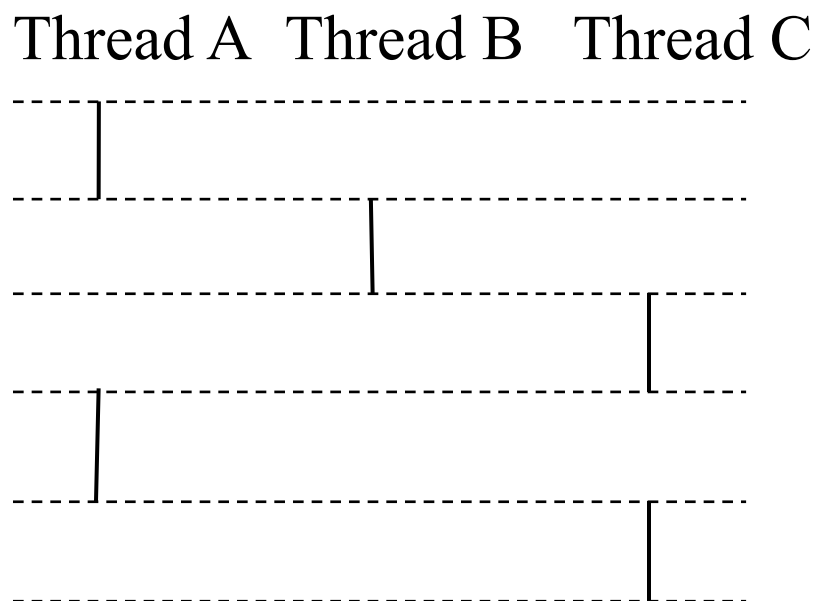
■ 例如:

- 并发: A & B, A & C
- 顺序: B & C



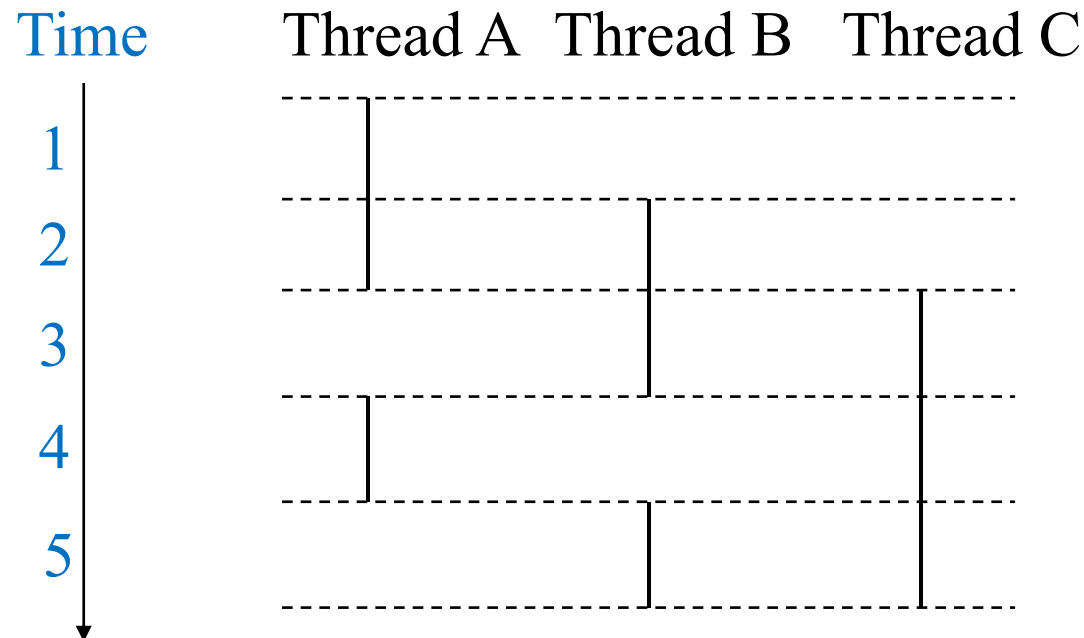
■ 单核处理器

➤ 通过时间片模拟并行



■ 多核处理器

➤ 能够实现真正的并行



在2个核心上运行3个线程

■ 线程和进程的相似点:

- 都有着自己的控制流
- 都能够并发的执行 (可能在不同的核心上)
- 都需要上下文切换的操作

■ 线程与进程的不同点:

- 线程共享进程的code、data区域 (除了局部的堆栈部分)
- 进程不共享, 不同的进程有着自己独立的区域
- 线程的开销比进程小
- 进程的控制开销 (创建、回收) 是两倍于线程的控制开销

✓ Linux 环境下:

- ~20K 个时钟周期创建、回收进程
- ~10K 个时钟周期 (或更少) 创建、回收线程

■ 线程

- 线程的引入与介绍
- 多线程模型
- Posix 线程

■ 有两种不同方法来提供线程支持

➤ 用户层的用户线程

✓ 用户线程位于内核之上，它的管理无需内核支持；

➤ 内核层的内核线程

✓ 内核线程由操作系统来直接支持与管理

■ 用户线程和内核线程之间关系

➤ 多对一模型

➤ 一对一模型

➤ 多对多模型

多线程模型—多对一模型

■ 映射多个用户级线程到一个内核级线程

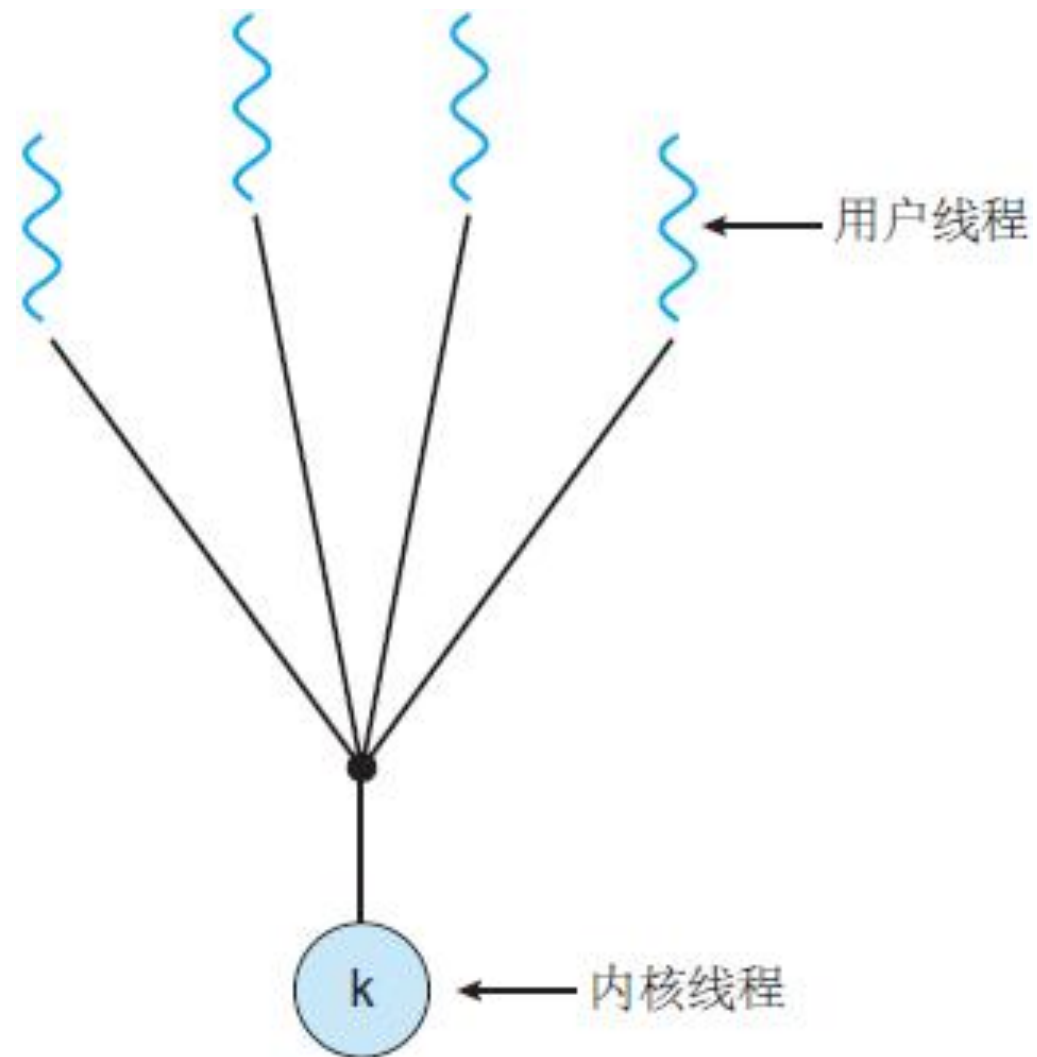
■ 优点:

➢ 线程管理是由用户空间的线程库来完成的, 因此效率更高。

■ 缺点:

➢ 如果一个线程执行阻塞系统调用, 那么整个进程将会阻塞。

➢ 因为任一时间只有一个线程可以访问内核, 所以多个线程不能并行运行在多处理核系统上。



现在几乎没有系统继续使用这个模型, 因为它无法利用多个处理核。

多线程模型——一对一模型

■ 映射每个用户级线程到一个内核级线程

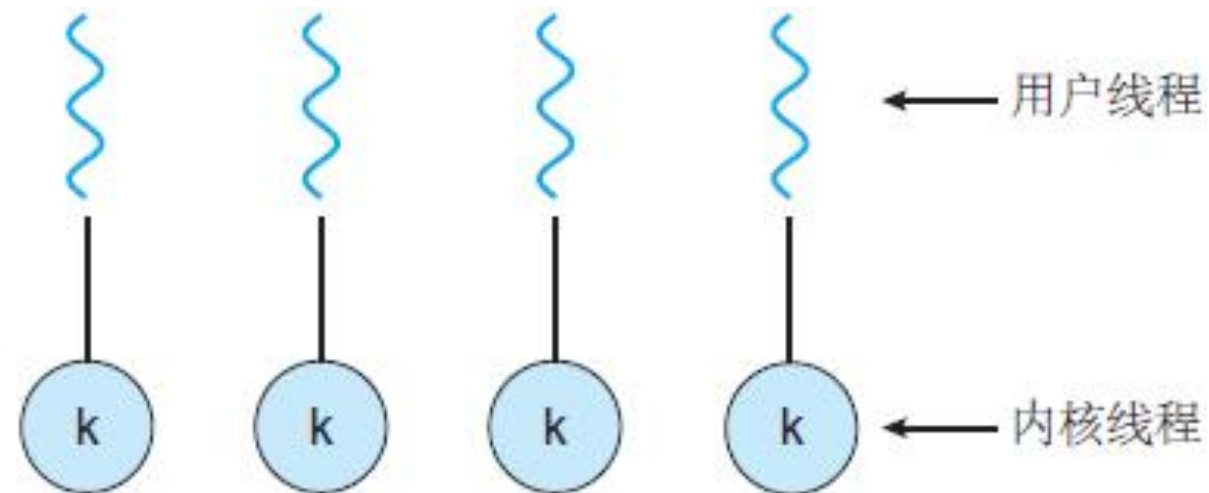
■ 优点:

➤ 一个线程执行阻塞系统调用时，能够允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能

➤ 允许多个线程并行运行在多处理器系统上

■ 缺点

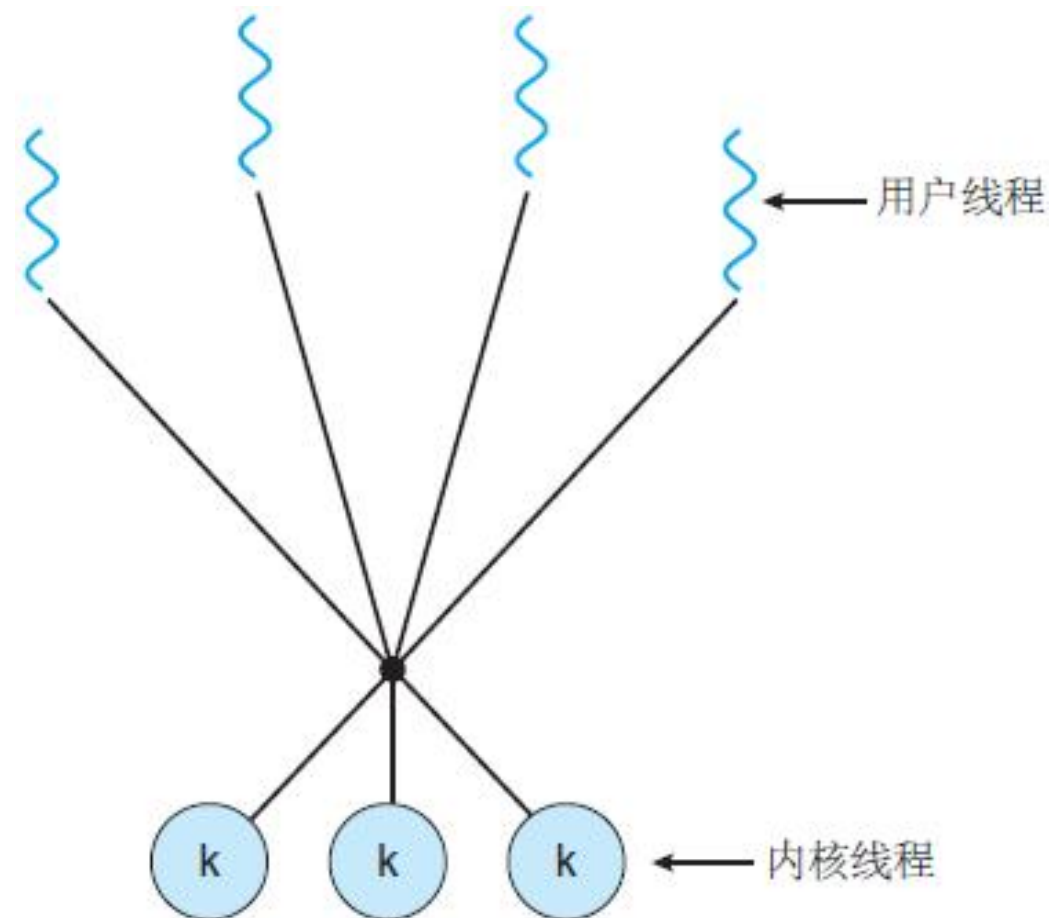
➤ 创建一个用户级线程就要创建一个相应的内核级线程。由于创建内核级线程的开销会影响应用程序的性能，所以这种模型的大多数实现限制了系统支持的线程数量。



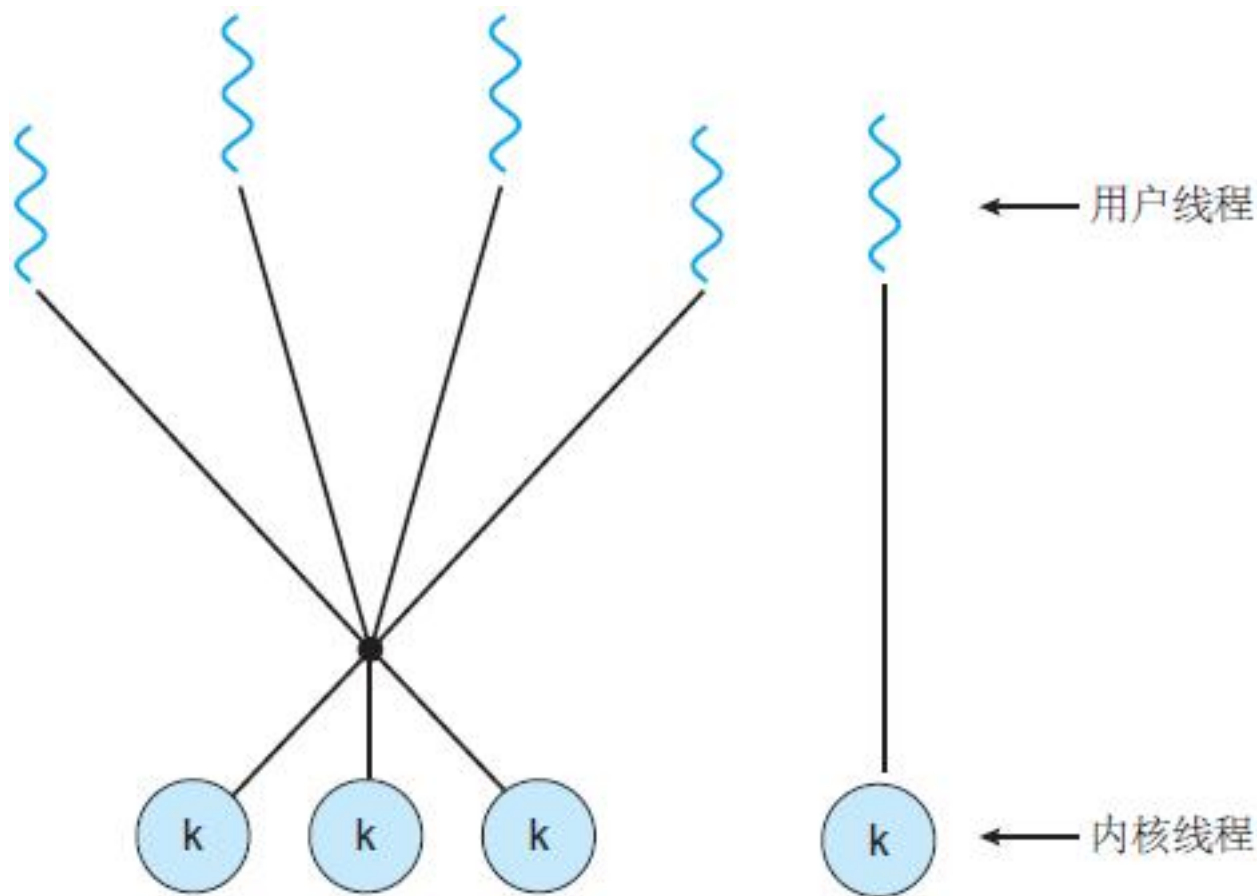
■ 多路复用多个用户级线程到同样数量或更少数量的内核级线程。内核级线程的数量可能与特定应用程序或特定机器有关（应用程序在多处理器上比在单处理器上可能分配到更多数量的线程）。

- 开发人员可以创建任意多的用户级线程，并且相应内核线程能在多处理器系统上并发执行。
- 当一个线程执行阻塞系统调用时，内核可以调度另一个线程来执行。

多对多模型解决了一对一、多对一模型的两个缺点。



- 多对多模型的一种变种仍然多路复用多个用户级线程到同样数量或更少数量的内核级线程，但也允许绑定某个用户级线程到一个内核级线程。这个变种，有时称为双层模型。



■ 线程

- 线程的引入与介绍
- 多线程模型
- Posix 线程

■ Pthreads 约60个函数的标准接口，这些函数可操纵C程序中的线程

➤ Creating threads (创建线程)

✓ pthread_create()

➤ Terminating threads (终止线程)

✓ pthread_cancel()

✓ pthread_exit()

➤ Joining and Detaching threads (回收和分离线程)

✓ pthread_join()

✓ pthread_detach()

■ `int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg)`

- `tid` : 获取创建的新线程的tid
- `attr` (属性) : 设置线程的属性
- `f` : 创建线程后将执行的程序
- `arg`(参数) : 一个可以传递给开始程序的参数。它必须作为空类型的指针强制转换通过引用传递。如果不传递任何参数, 则可以使用NULL
- 返回值: 成功返回0; 失败返回非0值-错误编码

■ 线程属性:

- 默认情况下, 线程在创建时会带有某些属性
- 程序员可以通过线程属性对象来更改其中的一些属性
- `pthread_attr_init` 和 `pthread_attr_destroy` 可以初始化/销毁线程属性对象

Example 1: Creating Threads(创建线程)

- 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

pthread_1.c

```
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 < 100000000; i++) {
10.         result = result + sin(i) * tan(i);
11.     }
12.     printf("Task %d completed with result %e\n", id, result);
13. }
```

Example 1: Creating Threads(创建线程)



```
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.    printf("Main completed\n");
19.    pthread_exit(NULL);
20. }
```

pthread_1.c

Example 1: Creating Threads(创建线程)

```
/* 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 %ld\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);
13.        }
14.    }
15. }
```

Thread ID

Thread attributes (usually NULL)

Thread routine

*Thread arguments (void *p)*

pthread_1.c

Example 1: Creating Threads(创建线程)

```
1. void *serial(int num_tasks) {                                     pthread_1.c
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.*/
```

Example 1: Creating Threads(创建线程)

■ VMware中设置处理器为单核:



Example 1: Creating Threads(创建线程)

■ VMware中单核处理器:

➤ 顺序执行

```
root@ubuntu:/home/liu/Desktop/OS# time ./pthreads_1 serial 4
Task 0 started
Task 0 completed with result 3.135632e+06
Task 1 started
Task 1 completed with result 3.135632e+06
Task 2 started
Task 2 completed with result 3.135632e+06
Task 3 started
Task 3 completed with result 3.135632e+06
Main completed

real    0m2.857s
user    0m2.813s
sys     0m0.004s
```


Example 1: Creating Threads(创建线程)

■ VMware中单核处理器:

➤ 并行执行

```
root@ubuntu:/home/liu/Desktop/OS# time ./pthreads_1 parallel 4
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Main completed
Task 3 started
Task 2 started
Task 1 started
Task 0 started
Task 3 completed with result 3.135632e+06
Task 1 completed with result 3.135632e+06
Task 2 completed with result 3.135632e+06
Task 0 completed with result 3.135632e+06

real    0m2.942s
user    0m2.832s
sys     0m0.004s
```

Example 1: Creating Threads(创建线程)

■ VMware中设置为4核处理器:



Example 1: Creating Threads(创建线程)

■ VMware中设置4核处理器:

➤ 顺序执行

```
root@ubuntu:/home/liu/Desktop/OS# time ./pthreads_1 serial 4
Task 0 started
Task 0 completed with result 3.135632e+06
Task 1 started
Task 1 completed with result 3.135632e+06
Task 2 started
Task 2 completed with result 3.135632e+06
Task 3 started
Task 3 completed with result 3.135632e+06
Main completed

real    0m3.001s
user    0m2.971s
sys     0m0.000s
```

Example 1: Creating Threads(创建线程)

■ VMware中设置4核处理器:

➤ 并行执行

```
root@ubuntu:/home/liu/Desktop/OS# time ./pthreads_1 parallel 4
Creating thread 0
Creating thread 1
Task 0 started
Creating thread 2
Task 1 started
Creating thread 3
Main completed
Task 2 started
Task 3 started
Task 2 completed with result 3.135632e+06
Task 3 completed with result 3.135632e+06
Task 0 completed with result 3.135632e+06
Task 1 completed with result 3.135632e+06

real    0m0.802s
user    0m3.148s
sys     0m0.004s
```

- `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4 */
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.}
```

pass_arg2.c

所有线程共用变量t

```
liu@ubuntu:~/Desktop/OS$ ./pass_arg2
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Hello from thread 4
Hello from thread 4
Hello from thread 4
Hello from thread 4
```



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.}
```

pass_arg1.c

向每个线程单独传递一个变量

```
liu@ubuntu:~/Desktop/OS$ ./pass_arg1
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

■ `void pthread_exit(void *thread_return)`

■ 线程终止:

- 当比他更高一级的进程返回时线程**隐式终止**;
- 当调用 `pthread_exit` 时线程**显式终止**;
- 如果主线程调用了 `pthread_exit`, 会等待所有其他线程终止后**终止主线程**, 整个进程将 `thread_return` 作为返回值;
- 当调用 `exit` 时, **整个进程都会被终止**。

■ `int pthread_cancel(pthread_t tid)`

- 调用 `pthread_cancel` 并**不等于线程终止**, 它只提出请求。线程在取消请求发出后会继续运行;
- 直到到达某个**取消点**。取消点是线程检查是否被取消并按照请求进行动作的一个位置。

Pthread Creation and Termination



```
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.     long taskids[NUM_THREADS];
12.     int rc;
13.     long t;
14.     for (t=0; t<NUM_THREADS; t++) {
15.         printf("In main: creating thread %ld\n", t);
16.         rc = pthread_create(&threads[t], NULL, PrintHello, taskids+t);
17.         if (rc) {
18.             printf("ERROR: return code from pthread_create() is %d\n", rc);
19.             exit(-1);
20.         }
21.     }
22.     /* Last thing that main() should do */
23.     pthread_exit(NULL); 回收主线程
24. }
```

terminate_pthread.c

第一次运行

```
liu@ubuntu:~/Desktop/OS$ ./terminate_pthread
```

```
In main: creating thread 0  
In main: creating thread 1  
Hello World! It's me, thread #0!  
In main: creating thread 2  
Hello World! It's me, thread #1!  
In main: creating thread 3  
Hello World! It's me, thread #2!  
In main: creating thread 4  
Hello World! It's me, thread #3!  
Hello World! It's me, thread #4!
```

第二次运行

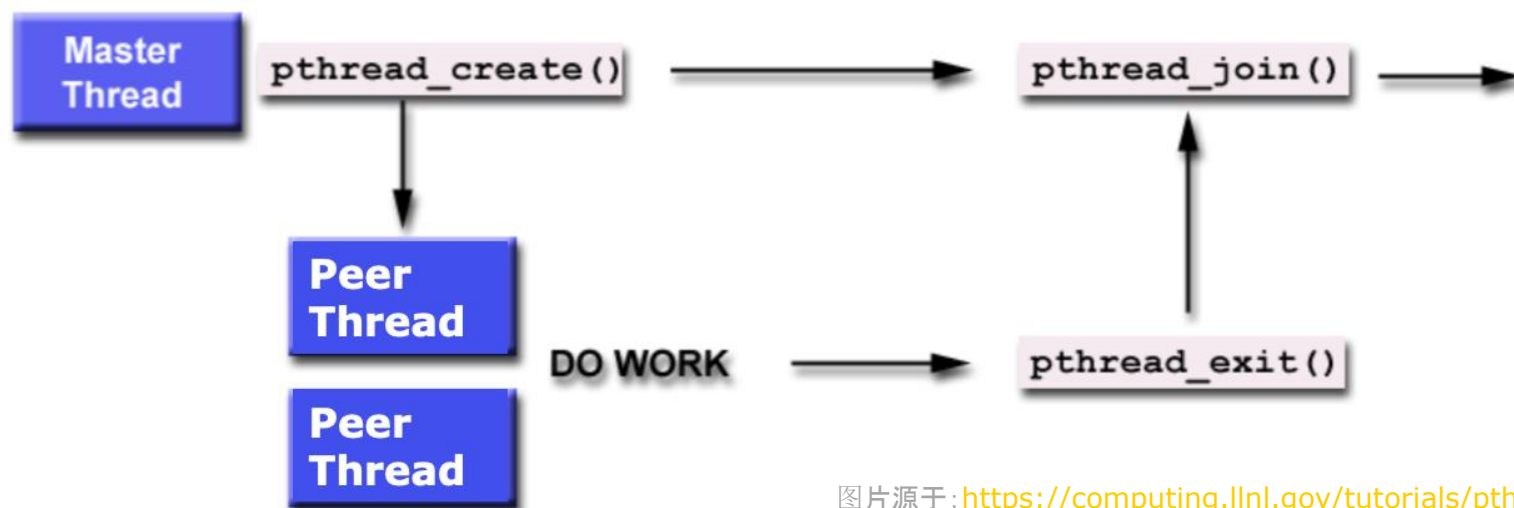
```
liu@ubuntu:~/Desktop/OS$ ./terminate_pthread
```

```
In main: creating thread 0  
In main: creating thread 1  
In main: creating thread 2  
Hello World! It's me, thread #0!  
Hello World! It's me, thread #2!  
Hello World! It's me, thread #1!  
In main: creating thread 3  
In main: creating thread 4  
Hello World! It's me, thread #3!  
Hello World! It's me, thread #4!
```


Joining (结合) and Detaching (分离) Threads

■ `int pthread_join(pthread_t tid, void **thread_return)`

- 连接是线程之间同步的一种方法
- `pthread_join` 阻塞线程直到tid线程终止
- 与 linux 中的 `wait` 函数不同，`pthread_join` 函数可以只等候特定的线程终止
- 一个线程只能响应一个 `pthread_join()` 请求。对同一个线程尝试多个 `join` 操作会发生逻辑错误。



■ `int pthread_detach(pthread_t tid)`

- `pthread_detach` 函数将线程 `tid` 分离出来
- 默认情况下，线程创建时为可连接的(`joinable`)
- 每个可连接线程应由另一个线程显式获取，或通过调用`pthread_detach`函数来分离。

■ 要将线程显式地创建为可连接或可分离的线程，请使用`pthread_create()` 例程中的`attr`参数。典型的步骤过程是：

- 声明`pthread_attr_t`数据类型的`pthread`属性变量
- 用`pthread_attr_init()` 初始化属性变量
- 使用`pthread_attr_setdetachstate()` 设置属性分离状态
- 完成后，释放属性通过`pthread_attr_destroy()` 使用的库资源

创建时向线程传递参数、join操作



sharing.c

```
10  #define NUM_THREADS 5
11
12  pthread_t threads[NUM_THREADS];
13  void *Hello(void *vargp);
14  char **ptr; /* Global variable */ //line:conc:sharing:ptrdec
15  int main()
16  {
17      int i;
18      char *msgs[NUM_THREADS] = {
19          "Hello from A",
20          "Hello from B",
21          "Hello from C",
22          "Hello from D",
23          "Hello from E"};
24      ptr = msgs;
25      int taskids[NUM_THREADS];
26      for (i = 0; i < NUM_THREADS; i++)
27      {
28          taskids[i] = i;
29          pthread_create(&threads[i], NULL, Hello, &taskids[i]);
30          printf("creat No. %d thread, tid=%d\n", i, threads[i]);
31          //pthread_join(threads[i], NULL);
32      }
33      pthread_exit(NULL);
34  }
```

```
36  void *Hello(void *vargp)
37  {
38      int myid = *((int *)vargp);
39      static int cnt = 0; //line:conc:sharing:cntdec
40      //line:conc:sharing:stack
41      //printf("tid=[%d]: %s (cnt=%d)\n", threads[myid], ptr[myid], ++cnt);
42      printf("Hello from No. %d thread, tid=%d. count=%d.\n", myid, threads[myid], ++cnt);
43      printf("%s\n", ptr[myid]);
44      pthread_exit(NULL);
45      return NULL;
46  } /* $end sharing */
```

创建时向线程传递参数、join操作



```
zhangshuyu — Solarized Dark xterm-256
argv[0] = '/Users/zhangshuyu/jupyter_notebook/thread'
creat No.0 thread,tid=263811072
creat No.1 thread,tid=264347648
creat No.2 thread,tid=264884224
creat No.3 thread,tid=265420800
creat No.4 thread,tid=265957376
Hello from No.0 thread,tid=263811072. count= 1.
Hello from A
Hello from No.1 thread,tid=264347648. count= 2.
Hello from B
Hello from No.2 thread,tid=264884224. count= 3.
Hello from C
Hello from No.4 thread,tid=265957376. count= 5.
Hello from E
Hello from No.3 thread,tid=265420800. count= 4.
Hello from D
Process exited with status 0
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

```
zhangshuyu — Solarized Dark xterm-256color
argv[0] = '/Users/zhangshuyu/jupyter_notebook/thread'
creat No.0 thread,tid=65306624
creat No.1 thread,tid=65843200
creat No.2 thread,tid=66379776
Hello from No.0 thread,tid=65306624. count= 1.
Hello from A
Hello from No.1 thread,tid=65843200. count= 2.
Hello from No.2 thread,tid=66379776. count= 3.
Hello from C
Hello from B
creat No.3 thread,tid=66916352
Hello from No.3 thread,tid=66916352. count= 4.
Hello from D
creat No.4 thread,tid=67452928
Hello from No.4 thread,tid=67452928. count= 5.
Hello from E
Process exited with status 0
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```

```
zhangshuyu — Solarized Dark xterm-256color
argv[0] = '/Users/zhangshuyu/jupyter_notebook/thread'
creat No.0 thread,tid=22339584
creat No.1 thread,tid=22876160
creat No.2 thread,tid=23412736
Hello from No.0 thread,tid=22339584. count= 1.
Hello from A
Hello from No.1 thread,tid=22876160. count= 2.
Hello from No.3 thread,tid=23949312. count= 4.
Hello from D
creat No.3 thread,tid=23949312
creat No.4 thread,tid=24485888
Hello from B
Hello from No.4 thread,tid=24485888. count= 5.
Hello from E
Hello from No.2 thread,tid=23412736. count= 3.
Hello from C
Process exited with status 0
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```


创建时向线程传递参数、join操作



```
zhangshuyu — Solarized Dark xterm-256color — 80x24
creat No.0 thread,tid=221229056
Hello from No.0 thread,tid=221229056. count= 1.
Hello from A
creat No.1 thread,tid=221229056
Hello from No.1 thread,tid=221229056. count= 2.
Hello from B
creat No.2 thread,tid=221229056
Hello from No.2 thread,tid=221229056. count= 3.
Hello from C
creat No.3 thread,tid=221229056
Hello from No.3 thread,tid=221229056. count= 4.
Hello from D
creat No.4 thread,tid=221229056
Hello from No.4 thread,tid=221229056. count= 5.
Hello from E
Process exited with status 0
logout
Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[进程已完成]
```



创建时向线程传递参数、join操作



```
10  #define NUM_THREADS 5
11
12  pthread_t threads[NUM_THREADS];
13  void *Hello(void *vargp);
14  char **ptr; /* Global variable */ //line:conc:sharing:ptrdec
15  int main()
16  {
17      int i;
18      char *msgs[NUM_THREADS] = {
19          "Hello from A",
20          "Hello from B",
21          "Hello from C",
22          "Hello from D",
23          "Hello from E"};
24      ptr = msgs;
25      int taskids[NUM_THREADS];
26      for (i = 0; i < NUM_THREADS; i++)
27      {
28          taskids[i] = i;
29          pthread_create(&threads[i], NULL, Hello, &taskids[i]);
30          printf("creat No.%d thread,tid=%d\n", i, threads[i]);
31          //pthread_join(threads[i], NULL);
32      }
33      pthread_exit(NULL);
34  }
```

sharing.c

```
36  void *Hello(void *vargp)
37  {
38      int myid = *((int *)vargp);
39      static int cnt = 0; //line:conc:sharing:cntdec
40      //line:conc:sharing:stack
41      //printf("tid=[%d]: %s (cnt=%d)\n", threads[myid], ptr[myid], ++cnt);
42      printf("Hello from No.%d thread,tid=%d count=%d\n", myid, threads[myid], ++cnt);
43      printf("%s\n", ptr[myid]);
44      pthread_exit(NULL);
45      return NULL;
46  } /* $end sharing */
```

向线程传递参数

变量实例	被主线程引用？	被对等线程 P0 引用？	被对等线程 P1 引用？	说明
ptr	✓	✓	✓	由 主线程 写入并由对等线程读取的全局变量。
cnt	✗	✓	✓	一个 静态变量 ，在内存中只有一个实例，由两个对等线程读取和写入。
i.m	✓	✗	✗	存储在主线程堆栈中的 局部自动变量
msgs.m	✓	✓	✓	一个 局部自动变量 ，存储在主线程的堆栈上，两个对等线程通过ptr间接引用。
myid.p0	✗	✓	✗	局部自动变量的实例位于对等线程0的堆栈上
myid.p1	✗	✗	✓	局部自动变量的实例位于对等线程1的堆栈上

■ 1. 下列关于线程的叙述中，正确的是（）

- A. 线程包含CPU现场，可以独立执行程序
- B. 每个线程有自己独立的地址空间
- C. 进程只能包含一个线程
- D. 线程之间的通信必须使用系统调用函数

答案:A

➤ 解析:

- A 线程是CPU调度的基本单位，可以独立执行程序
- B 线程没有自己独立的地址空间，线程共享其所属进程的地址空间
- C 进程可以拥有多个线程
- D 与所属同一进程下的不同线程可以通过它们共享的存储空间进行通信

■ 2.下面的叙述中，正确的是（）

- A.引入线程后，CPU只能在线程间切换
- B.引入线程后，CPU仍在进程间切换
- C.线程的切换，不会引起进程的切换
- D.线程的切换，可能引起进程的切换

答案:D

➤ 解析:

在同一进程中，线程的切换不会引起进程的切换。当从一个进程中的线程切换到另一个进程中的线程时，才会引起进程的切换。

■ 3.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，可以脱离进程独立运行
- B.引入线程可以提高程序并发执行的程度，可进一步提高系统效率
- C.线程的引入增加了程序执行时的时空开销
- D.一个进程一定包含多个线程

答案:B

➤ 解析:

- A 线程是进程内一个相对独立的执行单元，但不能脱离进程单独运行，只能在进程中运行。
- C 引入线程是为了减少程序执行时的时空开销。
- D 一个线程可包含一个或者多个线程。

■ 4.下面关于进程和线程的叙述中，正确的是（）

- A.不管系统是否支持线程，进程都是资源分配的基本单位
- B.线程是资源分配的基本单位，进程是调度的基本单位
- C.系统级线程和用户级线程的切换都需要内核的支持
- D.同一进程中的各个线程拥有各自不同的地址空间

答案:A

➤ 解析:

A B: 引入线程后，进程依然是资源分配的基本单位，线程是调度的基本单位

C 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，无须内核的干预，内核意识不到线程的存在

D 同一进程中的各个线程共享进程的地址空间

- Berger E D, McKinley K S, Blumofe R D, et al. Hoard: A scalable memory allocator for multithreaded applications[C]. ASPLOS, 2000;
- Ranger C, Raghuraman R, Penmetsa A, et al. Evaluating mapreduce for multi-core and multiprocessor systems[C]//2007 IEEE 13th International Symposium on High Performance Computer Architecture. 2007: 13-24.

- Hoard is a fast, scalable, and memory-efficient memory allocator that can speed up your applications. It's much faster than built-in system allocators: as much as 2.5x faster than Linux, 3x faster than Windows, and 7x faster than Mac.
- Hoard has been licensed by numerous companies to improve their application performance, including British Telecom, Cisco, Crédit Suisse, Reuters, Royal Bank of Canada, SAP, and Tata.

Pricing Information:

Source code license (exclusive of re-distribution rights)			
per user	5-user license	10-user license	Unlimited
\$1,000	\$4,000	\$6,000	\$20,000

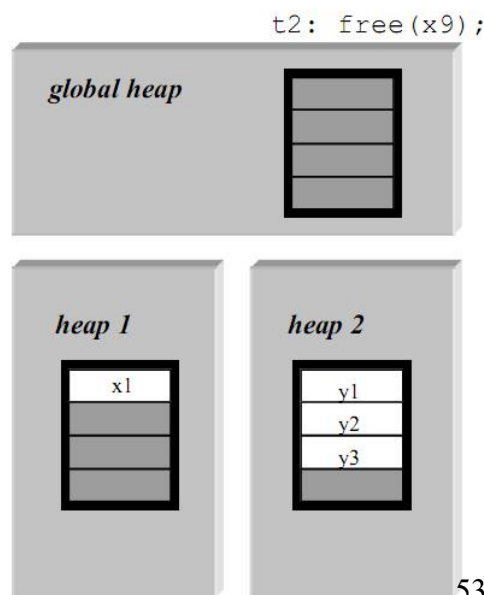
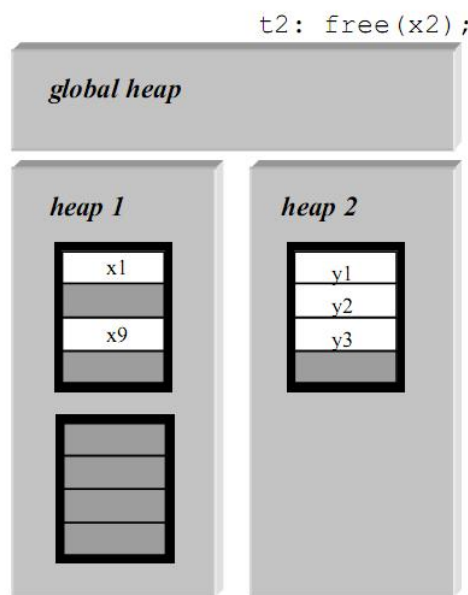
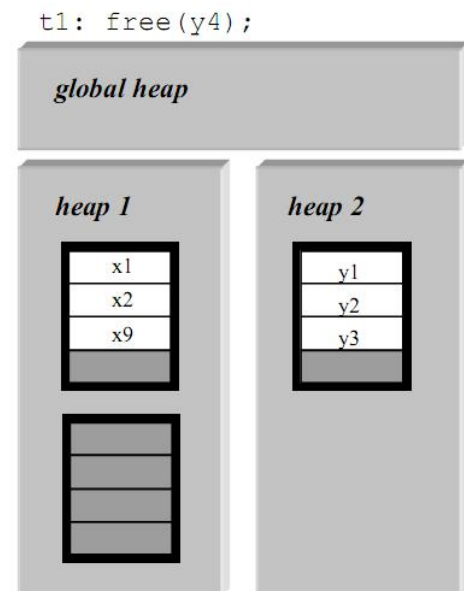
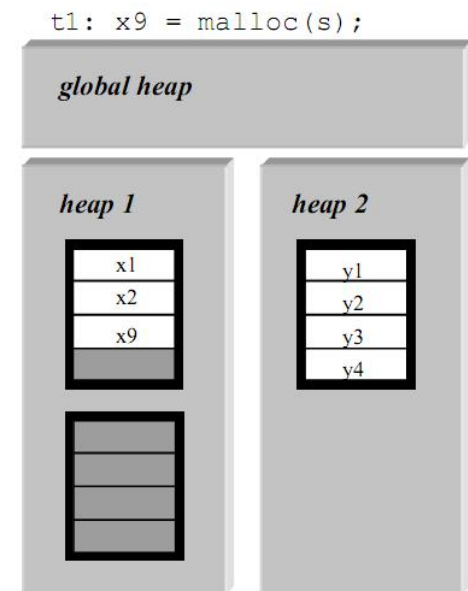
Object code license (exclusive of re-distribution rights)			
per user	5-user license	10-user license	Unlimited
\$500	\$1,500	\$3,000	\$10,000

Re-distribution rights (incremental to base rates)				
<10	<100	<1,000	<10,000	Unlimited
\$500	\$2,500	\$5,000	\$10,000	\$20,000

Hoard解决方案



- 解决方案：超级块&多堆
- Superblock（8KB）：线程内存预分配的基本单元
- Heap：线程分配单元
- →减少频繁的malloc和free操作
- superblock多个block组成，为了充分利用局部性采用LIFO原则
- 每个superblock大小相同，以superblock为单位申请内存
- 如果堆足够空，那么将移出一部分superblock到global heap
- →回收利用给其他线程使用



- Phoenix is a shared-memory implementation of Google's MapReduce model for data-intensive processing tasks. Phoenix can be used to program multi-core chips as well as shared-memory multiprocessors (SMPs and ccNUMAs).
- Phoenix was developed as a class project for the EE382a course at Stanford. The paper on Phoenix won the best paper award in the HPCA'07 conference.

Evaluating mapreduce for multi-core and multiprocessor systems

C Ranger, R Raghuraman, A Penmetsa... - 2007 IEEE 13th ..., 2007 - ieeexplore.ieee.org

This paper evaluates the suitability of the MapReduce model for multi-core and multiprocessor systems. MapReduce was created by Google for application development on data-centers with thousands of servers. It allows programmers to write functional-style code that is ...

☆ 被引用次数: 1295 相关文章

不要害怕，勇于实践。



Course Description

In parallel computing, the current research emphasis is on multi-processor architectures. GPUs, which have demonstrated significant potential in different applications and Tensor Processing Unit (TPU), are leveraging these architectures. The problem in using these alternatives is the scalability of these architectures, and then focuses on the Single Instruction Multiple Data (SIMD) programming model. This course explores SIMD programming by means of a simulator. This gives the students an exciting first-hand experience of the algorithms to solve known problems when making them SIMD. The dataflow (or Multiple Instruction-Single Data, MISD) machine (www.maxeler.com) and include step-by-step hands-on tutorial platform. Recently, Maxeler technology has become available to

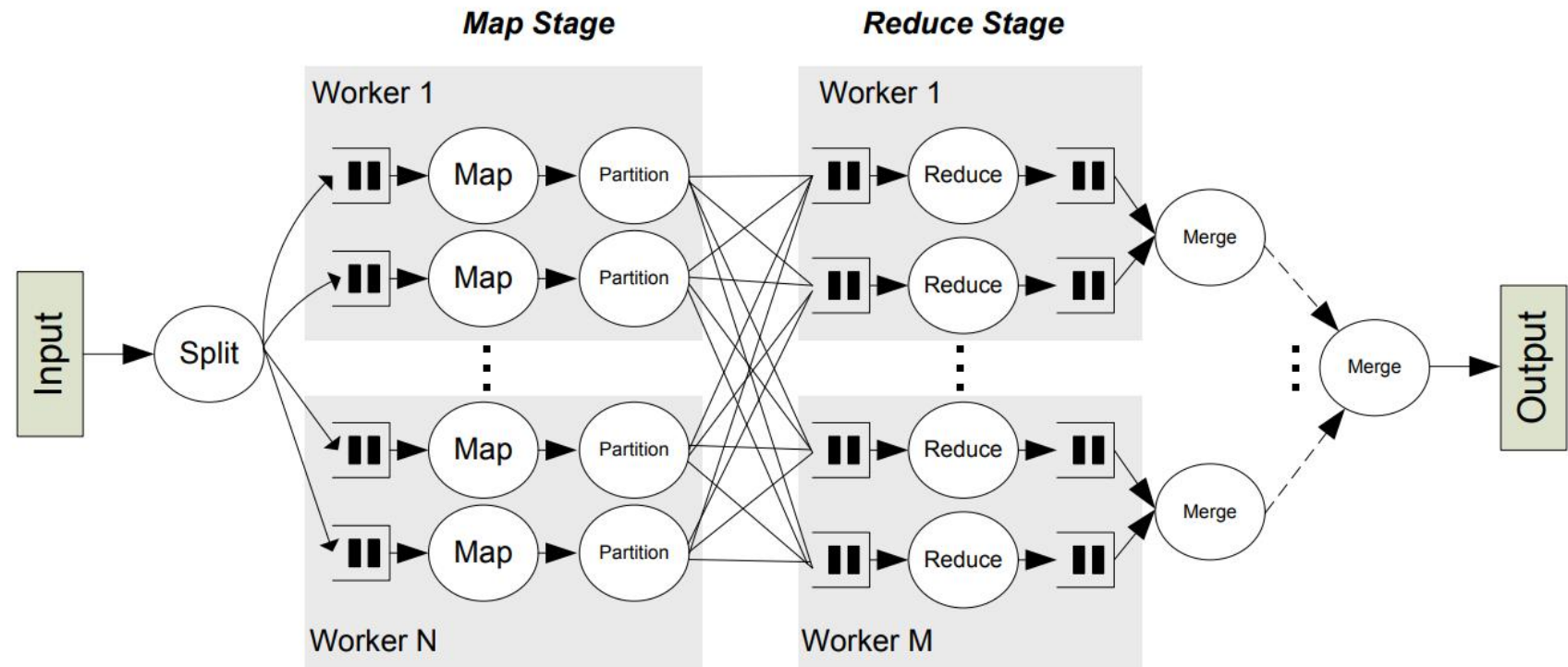
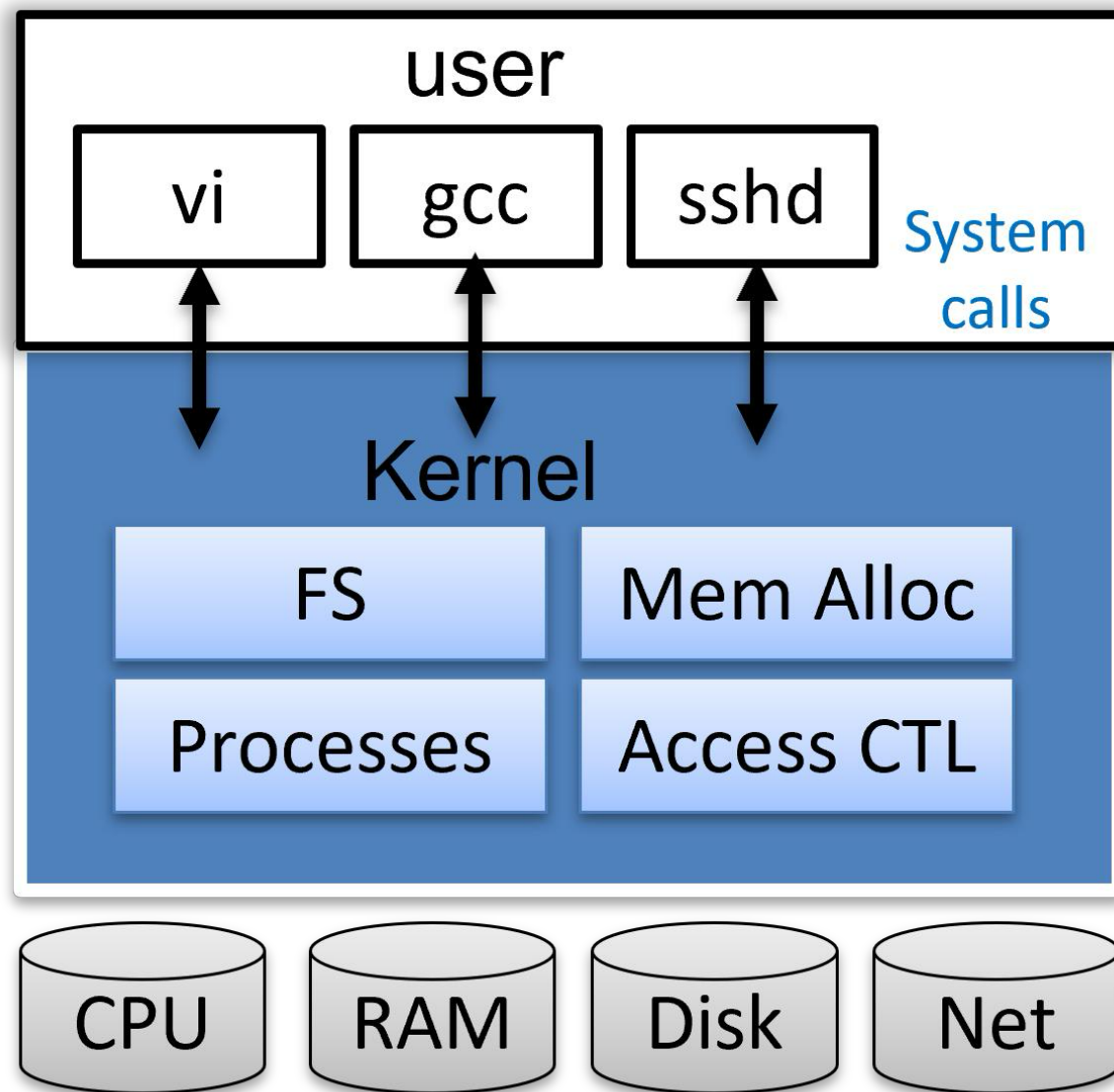


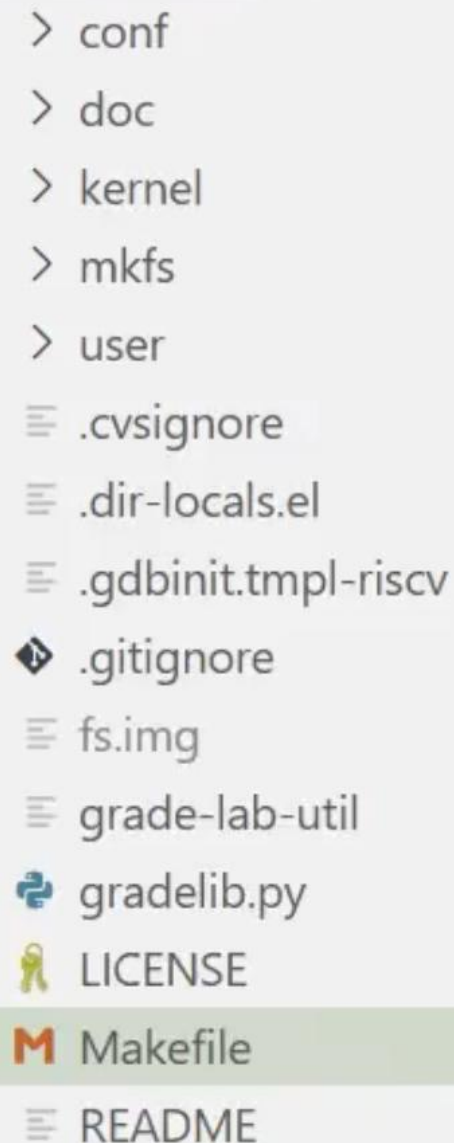
Figure 1. The basic data flow for the Phoenix runtime.

XV6架构简介

- XV6采用**宏内核结构**，它包含操作系统一些最基本的要素，包括**系统调用、进程调度、内存管理、中断处理和文件系统**等
- XV6区分**内核态**和**用户态**
 - 应用程序的进程运行在用户态（又称非特权模式，用户模式），无法直接访问系统硬件和操作系统中的系统数据，
 - 操作系统的系统调用运行在**内核态**（又称特权模式，内核模式），可以访问系统硬件和核心数据。



- **conf:** 配置文件
- **doc:** 文档文件
- **kernel:** 内核文件
- **mkfs:** 创建文件系统
- **user:** 用户程序文件，通过调用kernel中系统服务实现功能
(在这里完成大部分实验内容)
- **Makefile:** 编译链接整个项目



```
> conf
> doc
> kernel
> mkfs
> user
≡ .cvsignore
≡ .dir-locals.el
≡ .gdbinit.tmpl-riscv
📁 .gitignore
≡ fs.img
≡ grade-lab-util
🔗 gradelib.py
📄 LICENSE
M Makefile
≡ README
```

- 系统调用被封装为函数以向用户提供接口，用户程序可以通过函数调用的方式请求操作系统的服务，常见的系统调用接口定义如下：（系统调用接口头文件user/user.h）

调用接口名	描述
int sleep(int n)	Pause for n clock ticks.
int open(char* filename, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int read(int fd, void* buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int write(int fd, void* buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int close(int fd)	Release open file fd.

- QEMU是一套处理器模拟软件，用于在电脑（X86）上模拟RISC-V架构的CPU。对于每一个CPU核，QEMU都会运行取指、译码、执行的循环。QEMU主循环里也需要有寄存器文件，用于维护寄存器的状态。QEMU主循环里也需要有寄存器文件，用于维护寄存器的状态。
- 为什么不能用物理机运行xv6？
 - 实验中的xv6是RISC-V移植版，而目前主流物理机是x86-64架构
 - 操作系统需要操作特权态环境（特权寄存器、页表等）
- xv6是可以运行在RISC-V开发板上的（如k210）相关工具链：
 - Linux发行版：由Linux内核、GNU工具、附加软件和软件包管理器组成的操作系统
 - RISC-V工具链：包括一系列交叉编译的工具：gcc, binutils, glibc等

- Makefile 文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建哪些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。
- 一个 Makefile 中可以定义多个目标。当没有指明具体的目标是什么时，那么 make 以 Makefile 文件中定义的第一个目标作为这次运行的目标。

```
all:
    @echo "Hello world"
test:
    @echo "test game"
```

```
paul@paul-PC-MAC:~/cpp/make_train$ make
Hello world
paul@paul-PC-MAC:~/cpp/make_train$ make all
Hello world
paul@paul-PC-MAC:~/cpp/make_train$ make test
test game
```

```
all: test
    @echo "Hello world"
test:
    @echo "test game"
```

```
paul@paul-PC-MAC:~/cpp/make_train$ make
test game
Hello world
```

foo.c

```
#include <stdio.h>

void foo()
{
    printf("this is foo() !\n");
}
```

main.c

```
extern void foo();

int main()
{
    foo();
    return 0;
}
```

makefile

```
all: main.o foo.o
    gcc -o simple main.o foo.o
main.o: main.c
    gcc -o main.o -c main.c
foo.o: foo.c
    gcc -o foo.o -c foo.c
clean:
    rm simple main.o foo.o
```

```
paul@paul-PC-MAC:02$ make
gcc -o main.o -c main.c
gcc -o foo.o -c foo.c
gcc -o simple main.o foo.o
paul@paul-PC-MAC:02$ ./simple
this is foo() !
paul@paul-PC-MAC:02$ make
gcc -o simple main.o foo.o
paul@paul-PC-MAC:02$ ./simple
this is foo() !
```

第一次编译

第二次编译

Hope you enjoyed the OS course!