并行计算

（Parallel Computing）

共享内存编程 - Pthreads

学习内容：

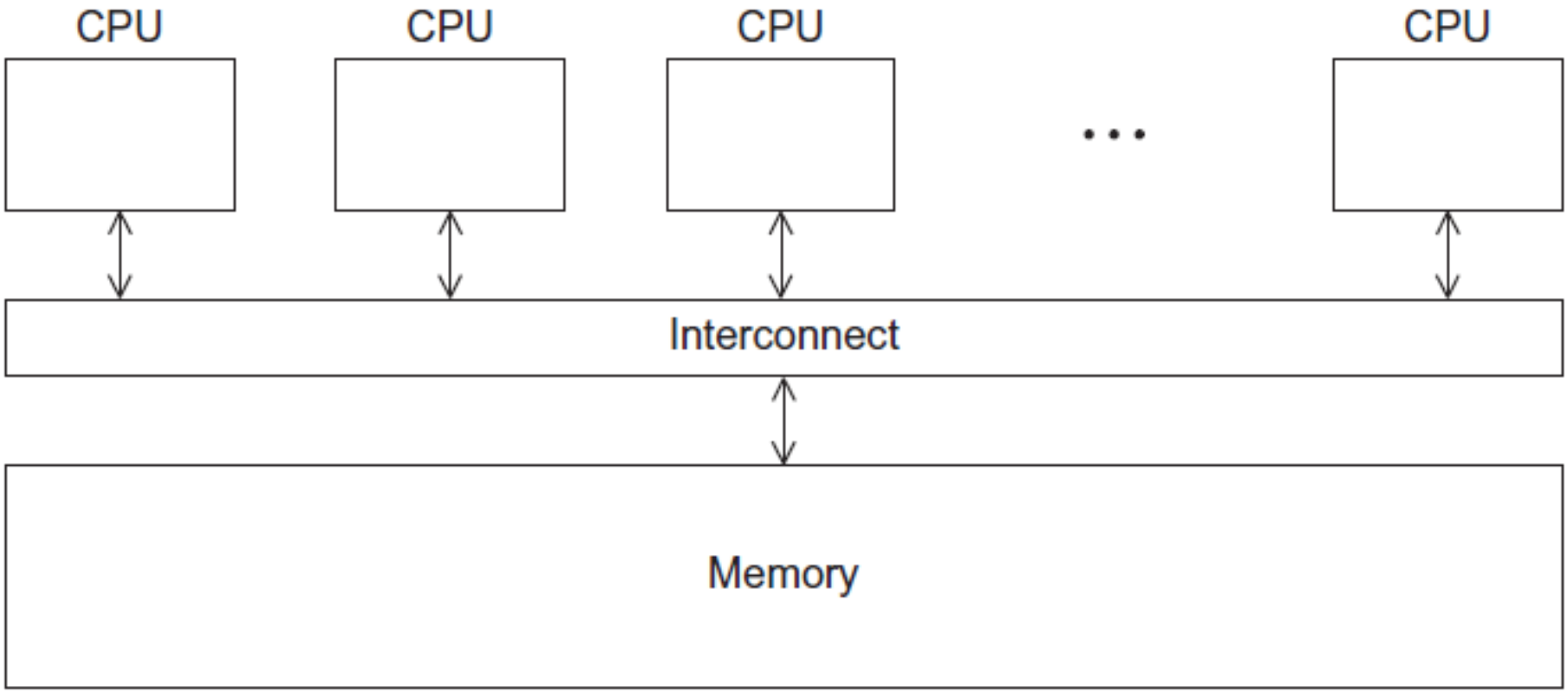
- 进程、线程、Pthreads
- Hello，World
- Pthreads中的矩阵 – 向量乘法
- l临界区（Critical Section）
- 忙 – 等（Busy-Waiting）

共享内存编程 - Pthreads

学习内容：

- 互斥量（Mutex）
- 生产者 – 消费者同步和信号量（Semaphore）
- 屏障（Barrier）和条件变量（Condition Variable）
- 读 – 写锁（Read-Write Lock）
- Cache，Cache一致性，伪共享（False Sharing）
- 线程安全性

共享内存系统



1.进程、线程、Pthreads

- 进程是正在运行（或挂起）程序的实例
  - 栈内存、堆内存
  - 分配给进程的资源的描述符（如：文件）
  - 安全信息（如：进程可以访问的硬件和软件资源）
  - 进程的状态信息（运行状态）
  - 进程的内存块通常是私有的，另一个进程无法直接访问

1.进程、线程、Pthreads

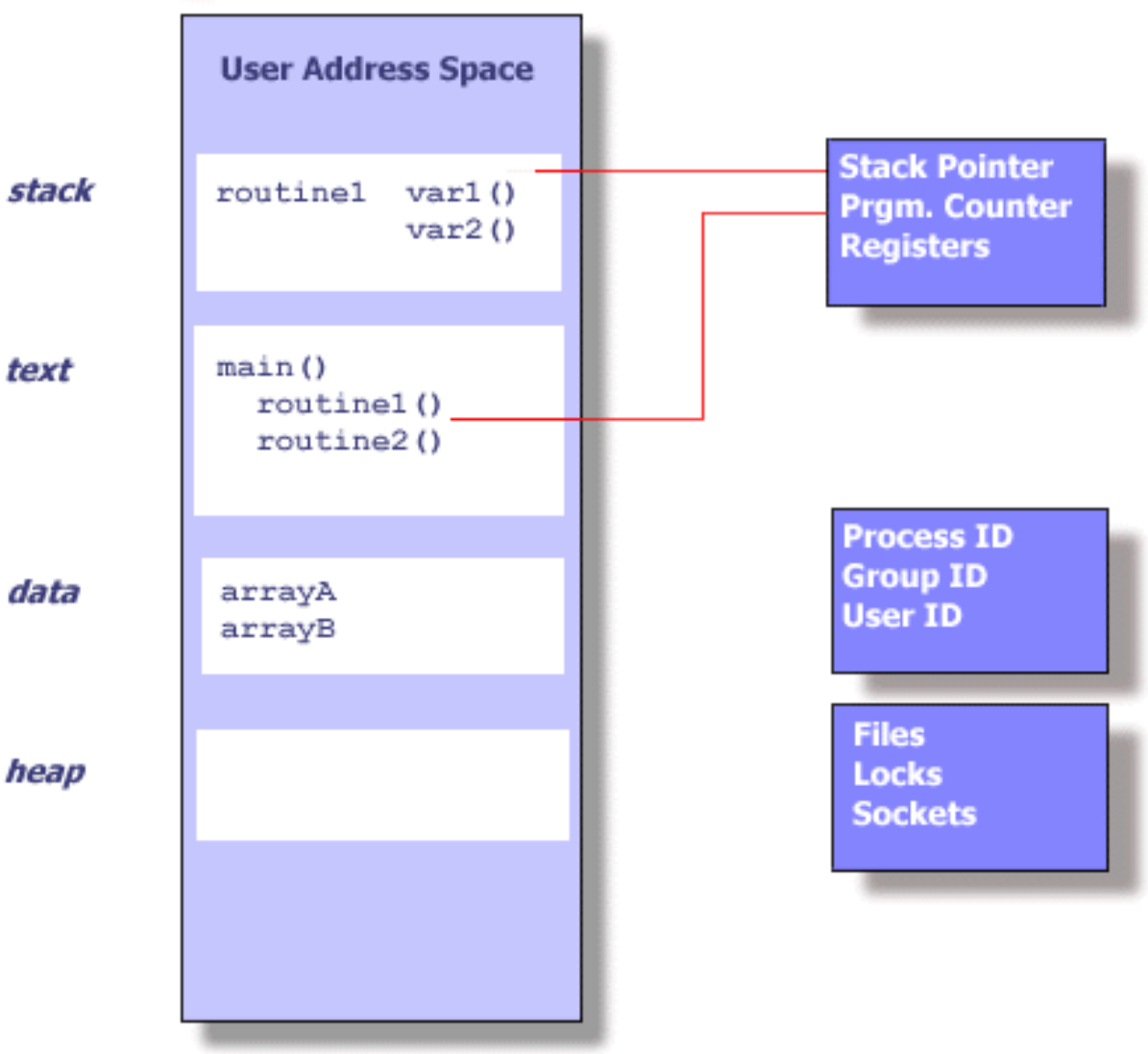
- 什么是线程（Thread）
  - 独立的指令流，操作系统可以对其进行调度
  - 对于软件开发人员来说，线程是独立于主程序（main）运行的“过程”（procedure）
  - 设想一个包含许多过程的主程序（a.out）。如果所有这些过程都能够被操作系统同时和/或独立地调度运行。即：“多线程”（multi-threaded）程序

1.进程、线程、Pthreads

- 什么是线程（Thread）
  - 存在于进程内并使用进程资源
  - 有自己独立的控制流
  - 与其他线程共享进程资源
  - 如其父进程消亡，其跟随消亡
  - 是“轻量级”的，因为大部分开销已经通过创建进程完成

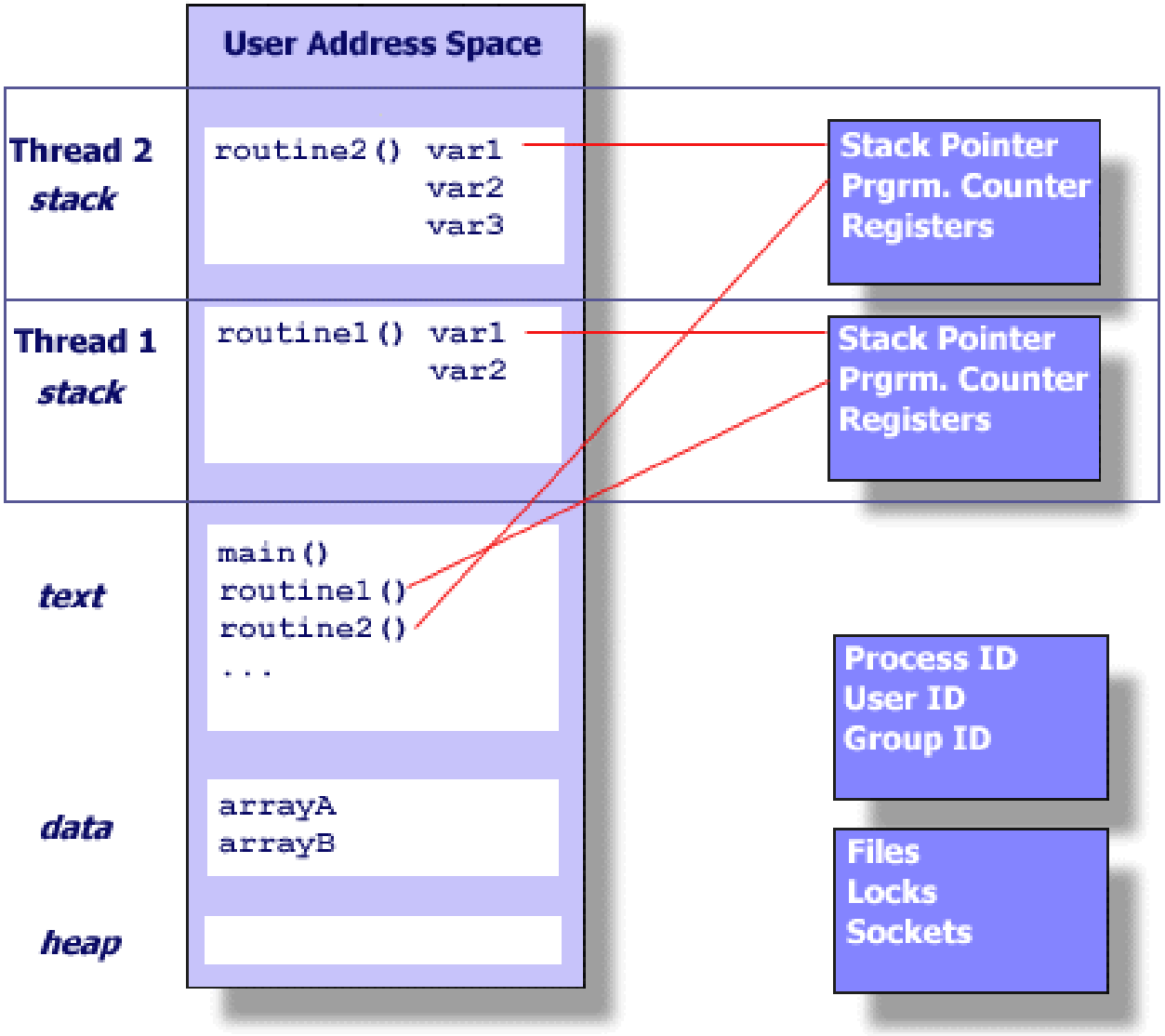
1.进程、线程、Pthreads

- UNIX 进程



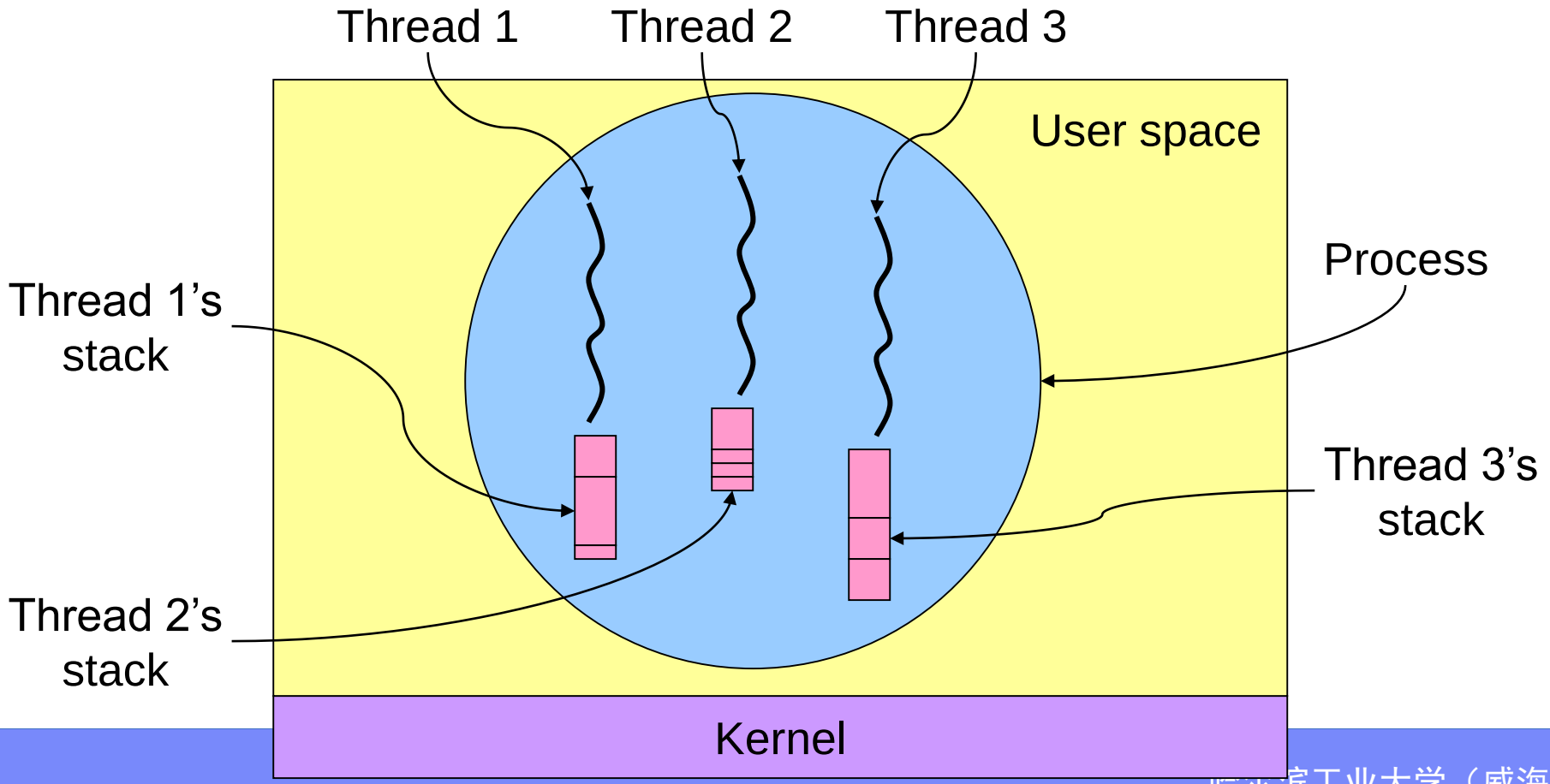
1.进程、线程、Pthreads

- UNIX 进程中的线程



1.进程、线程、Pthreads

- UNIX 进程中的线程

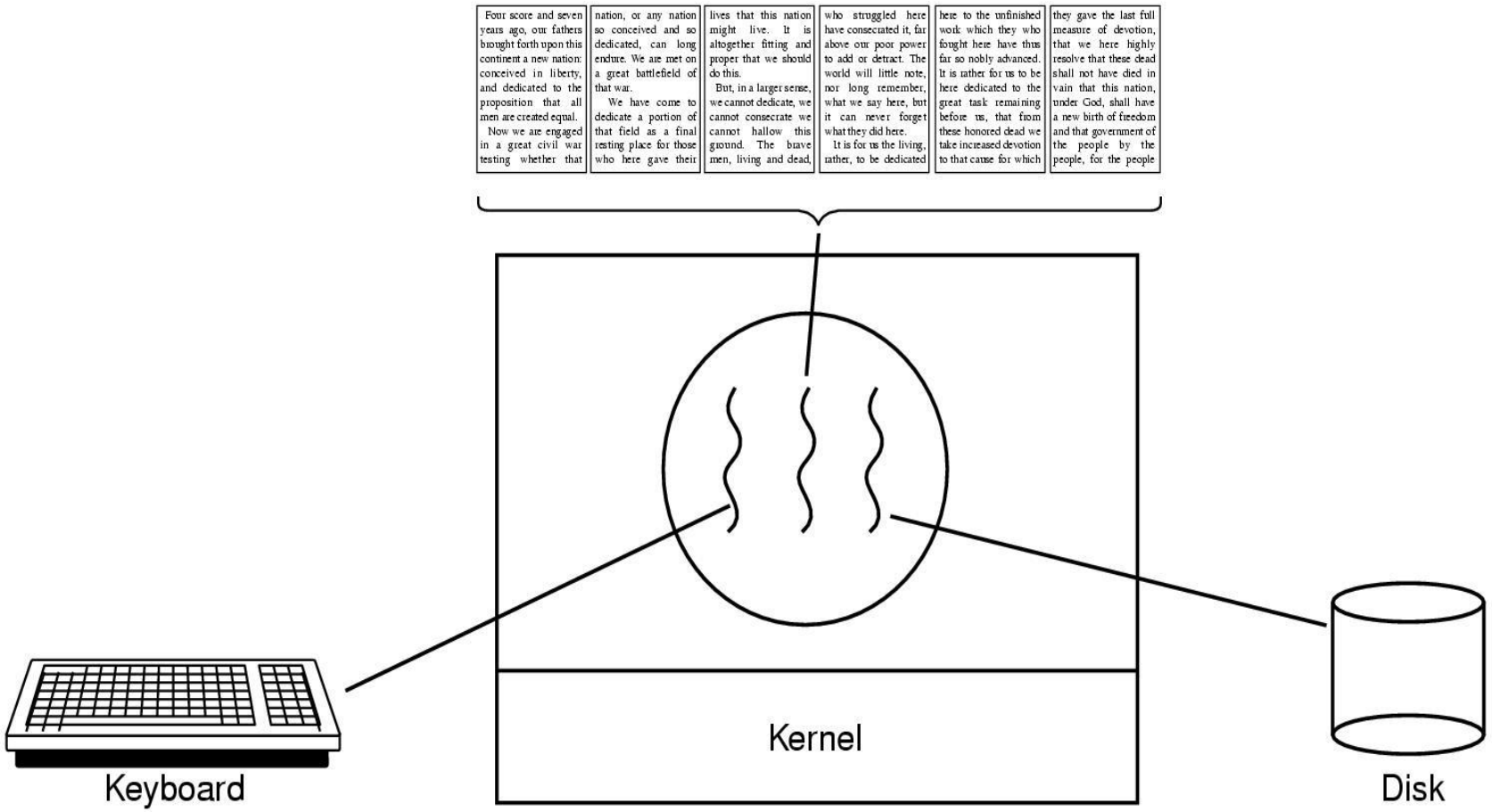


1.进程、线程、Pthreads

- 为什么使用线程
  - 性能：充分利用多处理器
  - 自然的程序结构：
    - 表示逻辑上并发的任务
    - 更新屏幕、获取数据、接收用户输入
  - 响应性：
    - 拆分命令，生成线程在后台工作
  - 避免 I/O 设备长时间的延迟
    - 等待时做有用的工作

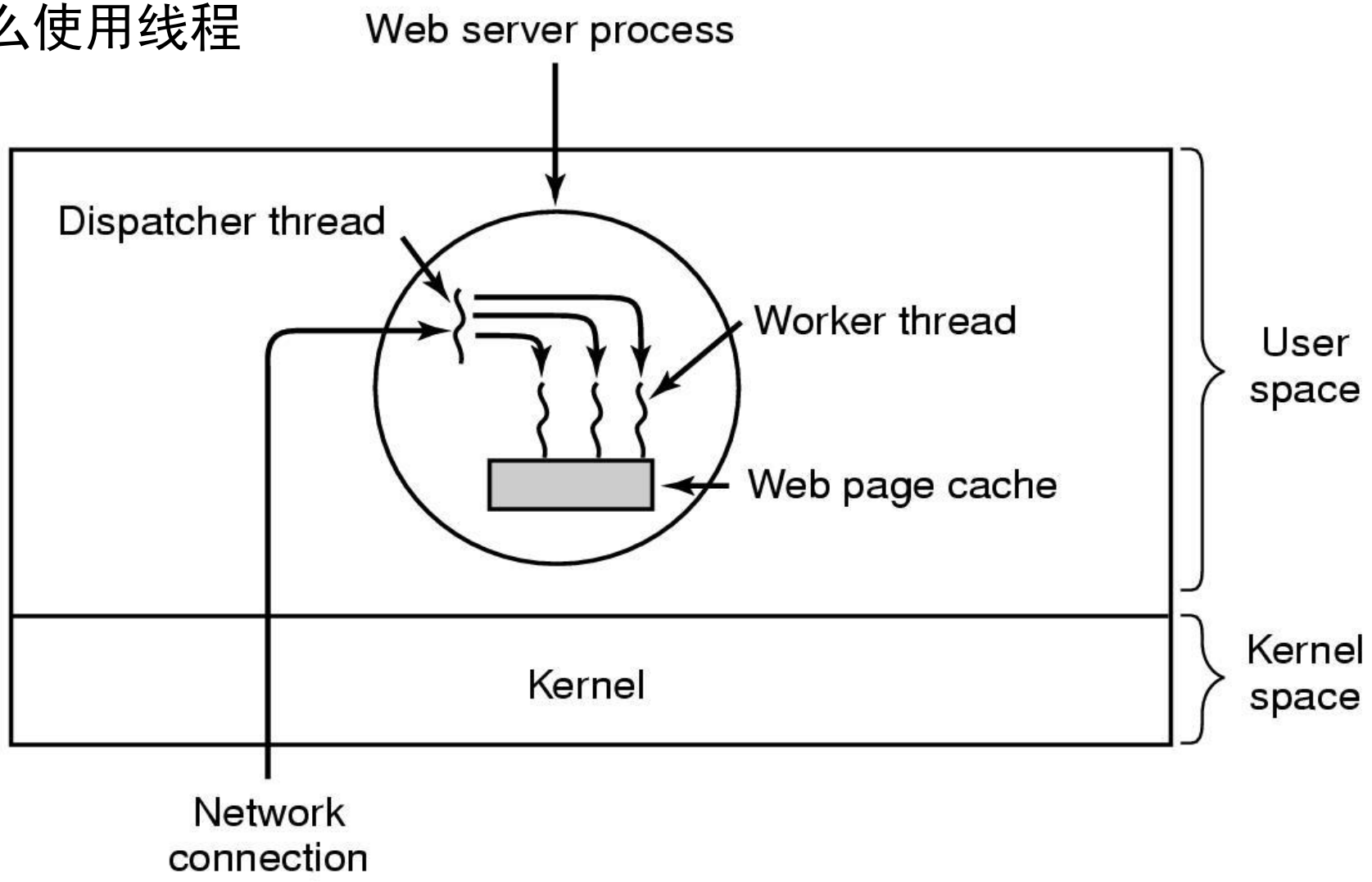
1.进程、线程、Pthreads

- 为什么使用线程



1.进程、线程、Pthreads

- 为什么使用线程

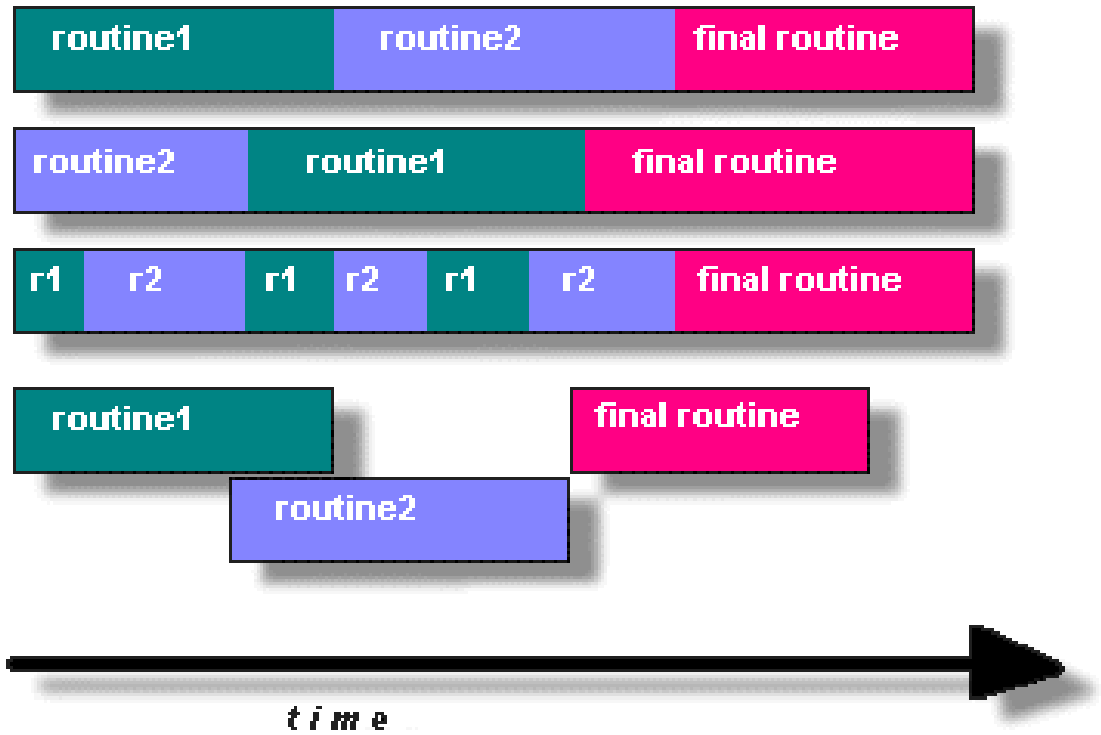


1.进程、线程、Pthreads

- 什么是 Pthreads
  - 历史上，硬件供应商已经实现了他们自己的线程专有版本。这些实现之间有很大的不同，使得程序员很难开发可移植的线程应用程序
  - 为了充分利用线程提供的功能，需要一个标准化的编程接口
    - 对于 UNIX 系统，IEEE POSIX 1003.1c 标准（1995）
    - 遵循本标准的实现称为 POSIX 线程或 Pthreads
  - 定义为一组C语言编程类型和过程调用（由 pthread.h 和线程库实现）
  - 仅适用于POSIX系统中，如：Linux，Mac OS X，Solaris，HPUX等

1.进程、线程、Pthreads

- Pthreads 编程



1.进程、线程、Pthreads

- 几种常见的多线程编程模型
  - Manager/worker：manager 线程将工作分配给其他线程，即 worker 线程。通常， manager 处理所有输入，并将工作分配给其他任务。 Manager/worker 模型有两种常见形式： static worker pool and dynamic worker pool
  - Pipeline：一个任务被分解成一系列子操作，每个子操作由不同的线程按顺序（但同时）处理。如：汽车装配线
  - Peer：类似于manager/worker模型，主线程创建其他线程之后，参与工作

2. Hello，World

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

declares the various Pthreads functions, constants, types, etc.

2. Hello，World

```
    for (thread = 0; thread < thread_count; thread++)
       pthread_create(&thread_handles[thread], NULL,
           Hello, (void*) thread);

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread_count; thread++)
       pthread_join(thread_handles[thread], NULL);

    free(thread_handles);
    return 0;
}   /* main */
```

2. Hello，World

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

2. Hello，World

- strtol：将字符串转换为 long int
  - number_p：输入字符串
  - end_p：如果不为NULL，指向未被识别的第一个字符
  - base：进制

```
long strtol(
        const char*   number_p   /* in  */,
        char**        end_p      /* out */,
        int           base       /* in  */);
```

2. Hello， World

- 编译 Pthread 程序

```
gcc -g -Wall -o pth_hello pth_hello . c -lpthread
```

2. Hello，World

●执行 Pthread 程序

. /pth_hello <number of threads>

. /pth_hello 1

Hello from the main thread

Hello from thread 0 of 1

. /pth_hello 4

Hello from the main thread

Hello from thread 0 of 4

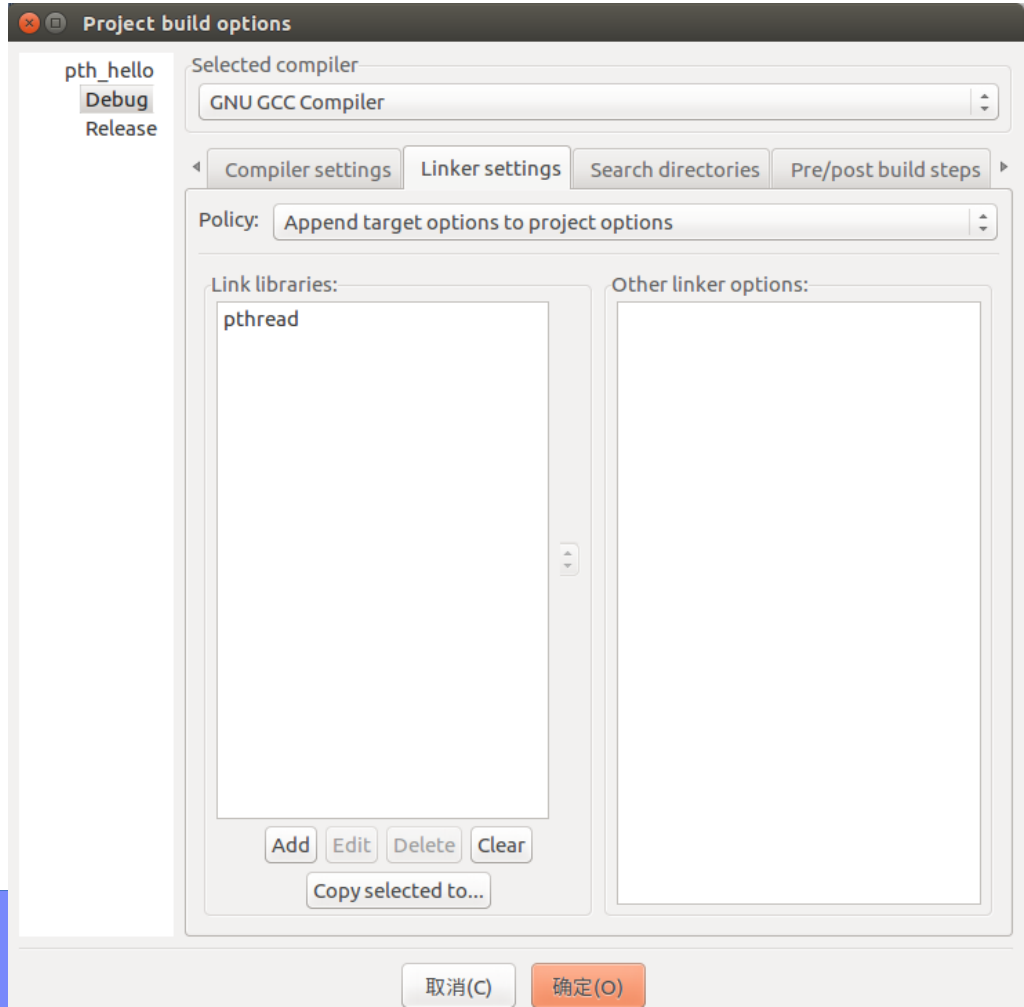
Hello from thread 1 of 4

Hello from thread 2 of 4

Hello from thread 3 of 4

2. Hello， World

- Linux/Ubuntu下配置Code::blocks



2. Hello， World

- Visual Studio下配置Pthreads

https://blog.csdn.net/user11223344abc/article/details/80536280

- Windows Code::blocks 配置Pthreads

https://www.cnblogs.com/lca1826/p/6606350.html

2. Hello，World

- 启动线程
  - 在 MPI 中，进程通常由脚本启动
  - 在 Pthread 中，线程通过程序启动

2. Hello， World

- 启动线程：pthread_t objects
  - 不透明（Opaque）对象，实际存储的数据与系统相关
  - 数据成员不能直接通过用户代码访问
  - Pthread 标准保证 pthread_t 存储区分进程的足够信息

```
thread_handles = malloc(thread_count * sizeof(pthread_t));
```

2. Hello， World

- 启动线程

```
int pthread_create (
    pthread_t*  thread_p /* out */ ,
    const pthread_attr_t*  attr_p /* in */ ,
    void*  (*start_routine ) ( void ) /* in */ ,
    void*  arg_p /* in */ ) ;
```

We won't be using, so we just pass NULL.

Allocate before calling.

2. Hello， World

- 启动线程

```
int pthread_create (
      pthread_t*  thread_p /* out */ ,
      const pthread_attr_t*  attr_p /* in */ ,
      void*  (*start_routine ) ( void ) /* in */ ,
      void*  arg_p /* in */ ) ;
```

Pointer to the argument that should be passed to the function *start_routine*.

The function that the thread is to run.

2. Hello，World

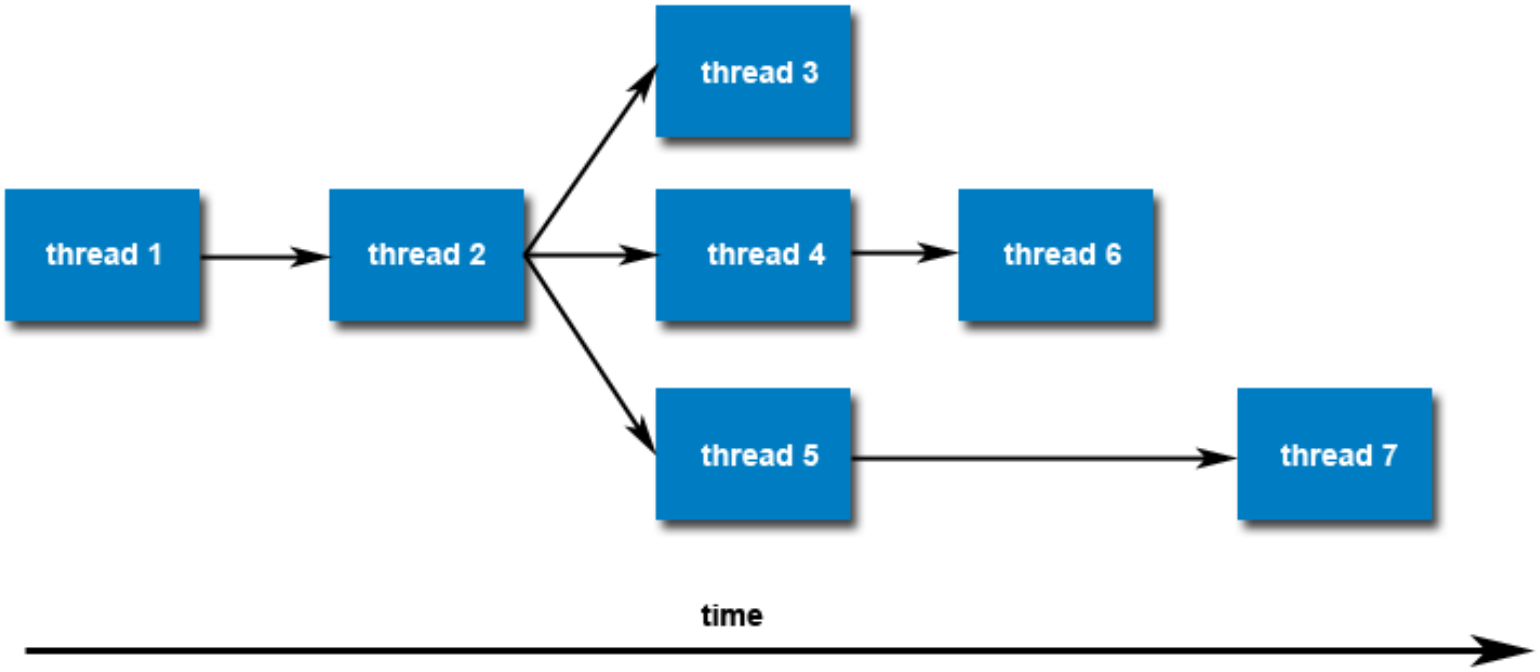
- 启动线程：由 pthread_create 启动的函数
  - 原型：**void* thread_function (void* args_p) ;**
  - 在 C 中 void* 可以转换为任意指针类型
  - 所以 args_p 可以指向包含一个或多个值的列表
  - 类似的，thread_function 的返回值也可指向一段地址

2. Hello，World

- 运行线程
  - 运行 main 函数的线程称为主线程
  - 在 Pthread 中，程序员无法直接控制线程在哪里运行
  - pthread_create 中没有参数设置线程在哪个 core 中运行
  - 线程的配置由操作系统控制

2. Hello，World

- 一旦创建，线程之间是对等的，线程可以创建其他线程



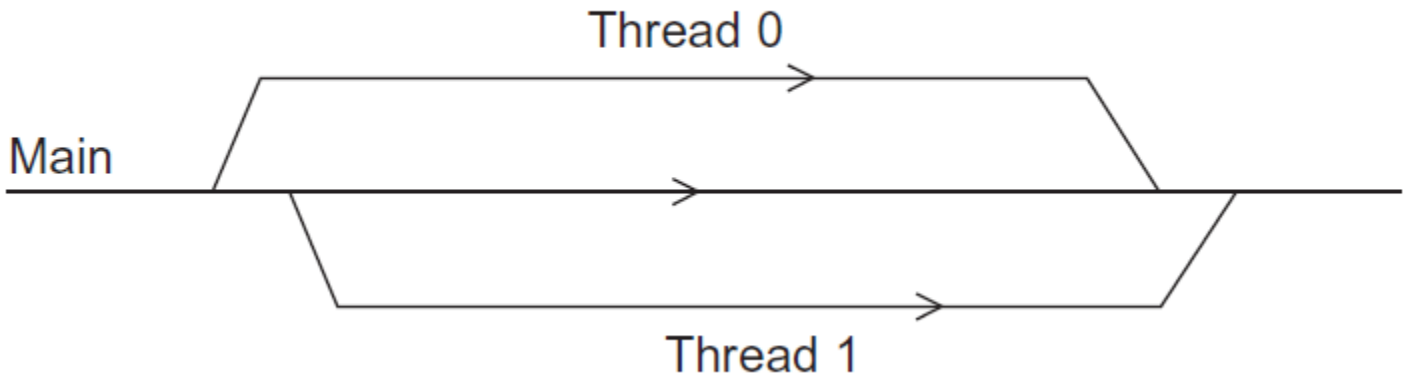
2. Hello， World

- 停止线程
  - 为每个线程调用 pthread_join 函数
  - pthread_join 将等待与 pthread_t 对象相关的进程完成其操作
  - ret_val_p 可以用来接收线程返回的值

```
int pthread_join(
      pthread_t thread      /* in */,
      void**      ret_val_p  /* out */);
```

2. Hello，World

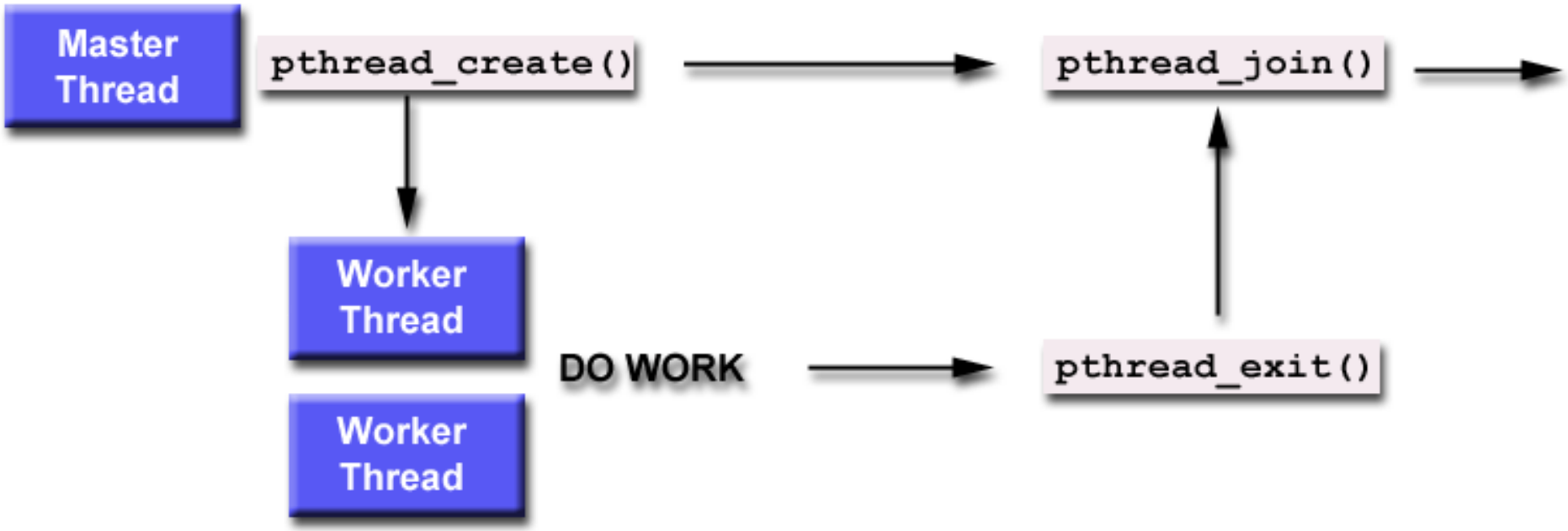
●停止线程



Main thread forks and joins two threads.

2. Hello， World

●停止线程



2. Hello，World

- 向线程传递参数
  - pthread_create 允许向 start_routine 传递一个参数

```
int pthread_create (
    pthread_t*  thread_p /* out */ ,
    const pthread_attr_t*  attr_p /* in */ ,
    void*  (*start_routine ) ( void ) /* in */ ,
    void*  arg_p /* in */ ) ;
```

2. Hello， World

- 向线程传递参数：例子1

```
long taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
    taskids[t] = t;
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) taskids[t]);
    ...
}
```

2. Hello，World

- 向线程传递参数：例子2

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}
```

2. Hello，World

- 向线程传递参数：例子2

```
int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
   ...
}
```

2. Hello， World

- 向线程传递参数：例子3？

```
int rc;
long t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %ld\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
   ...
}
```

3. Pthreads中的矩阵 – 向量乘法

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

$=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

3. Pthreads中的矩阵 – 向量乘法

- 串行伪代码

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

3. Pthreads中的矩阵 – 向量乘法

- 假设 m = n = 6, 使用 3 个线程

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

3. Pthreads中的矩阵 – 向量乘法

- 假设 m = n = 6, 使用 3 个线程
  - x 为共享变量
  - 将 A 和 y 也设置为共享变量
  - m 和 n 可以被线程数 t 整除
    - 线程 q 需要计算的 y 值：

$$\text{first component: } q \times \frac{m}{t}$$

$$\text{last component: } (q+1) \times \frac{m}{t} - 1$$

3. Pthreads中的矩阵 – 向量乘法

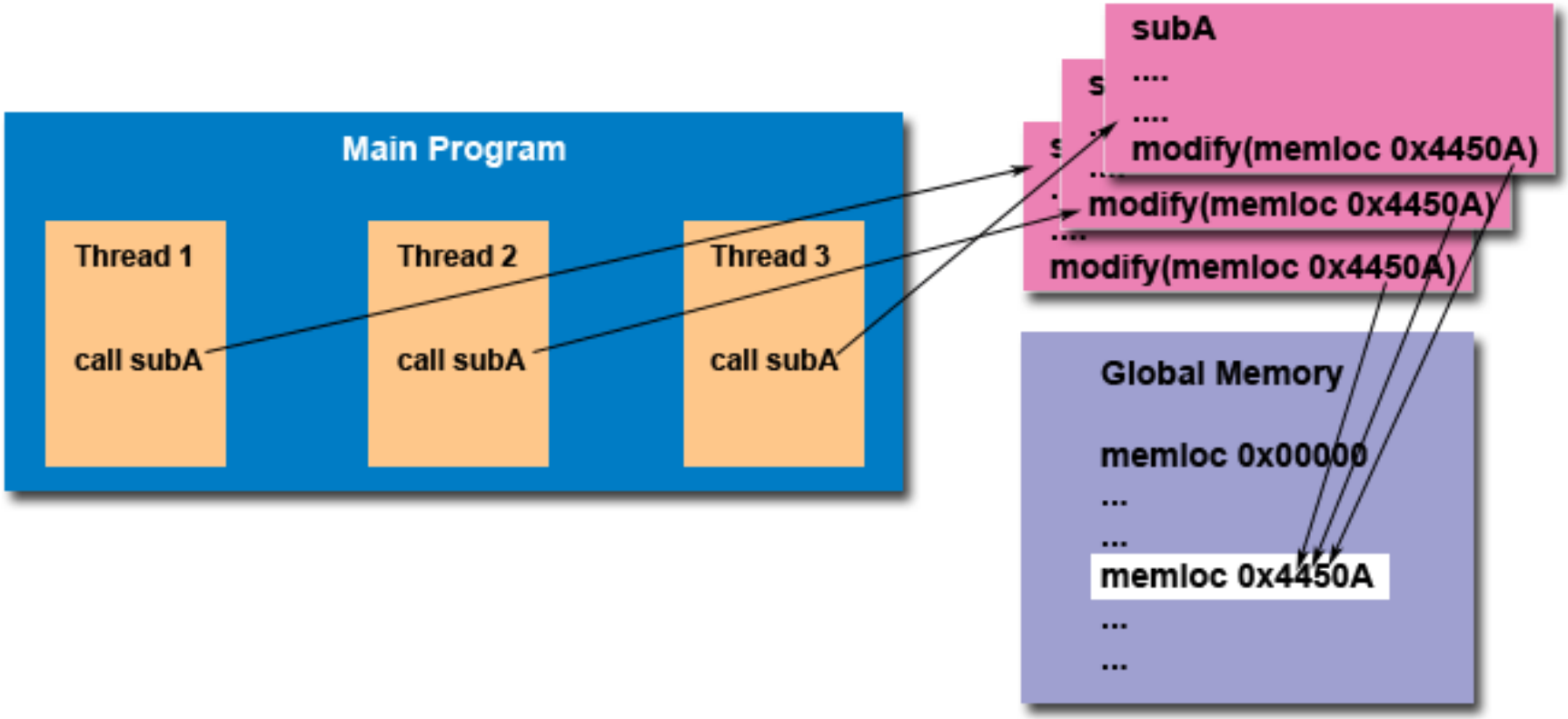
```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
} /* Pth_mat_vect */
```

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？



4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

4. 临界区（Critical Sections）

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)  /* my_first_i is even */
10       factor = 1.0;
11    else  /* my_first_i is odd */
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       sum += factor/(2*i+1);
16    }
17
18    return NULL;
19 }  /* Thread_sum */
```

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

有什么问题？

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

x = x + y;

通常并不是由一条机器指令完成的

1. x 和 y 从内存->寄存器
2. 累加
3. 结果从寄存器->内存

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

```
y = Compute(my rank);
x = x + y;
```

假设线程0计算，y = 1；线程1计算，y = 2，则正确结果应为：x = 3

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 估算圆周率

```
y = Compute(my rank);

x = x + y;
```

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

4. 临界区（Critical Sections）

- 当多个线程试图更新同一块内存区域，结果如何？
  - 多个线程试图更新共享资源（如：共享变量），结果无法预测
  - 竞争条件（race condition）
  - 在一个线程对共享资源更新结束后（x = x + y），才允许其他线程更新
  - 更新共享资源的代码块称为临界区（Critical Sections）（ x = x + y ）