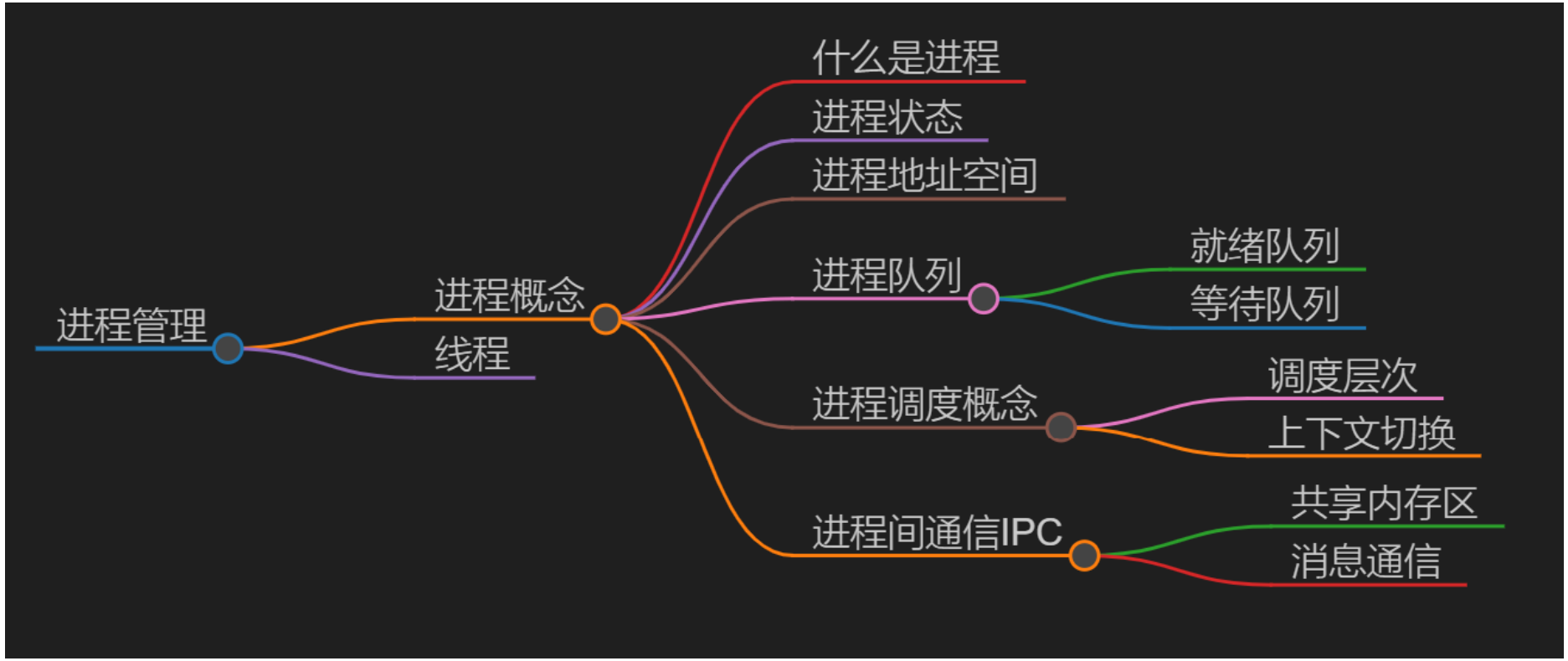
操作系统

L05 线程

胡燕
大连理工大学 软件学院

4.0-前情回顾

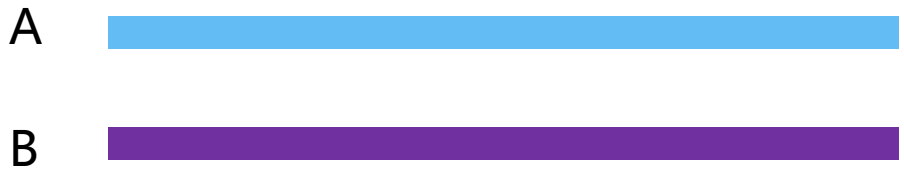
进程

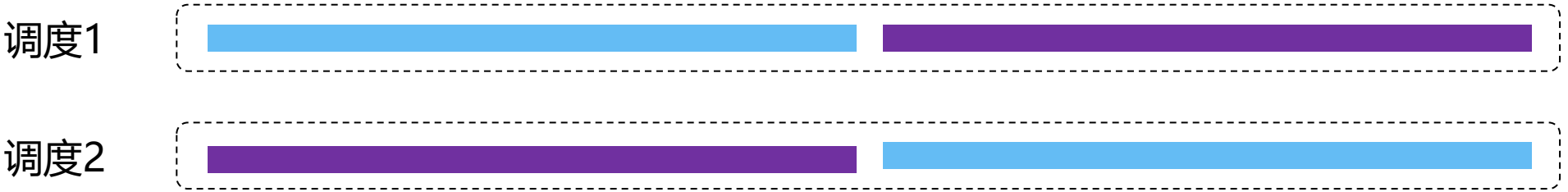

进程管理
进程概念
线程
什么是进程
进程状态
进程地址空间
进程队列
就绪队列
等待队列
进程调度概念
调度层次
上下文切换
进程间通信IPC
共享内存区
消息通信

进程调度：三个层次（　　）、（　　）、（　　）

调度与上下文切换

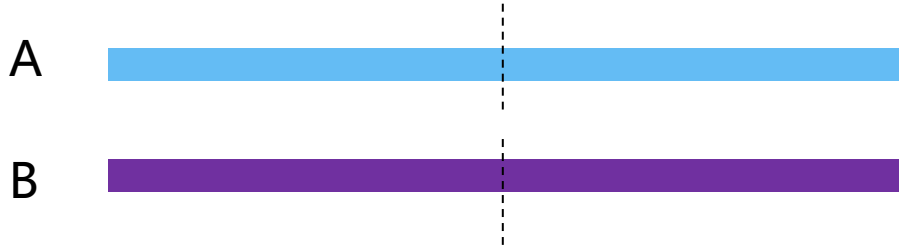
调度的一项关键任务： 处理上下文切换

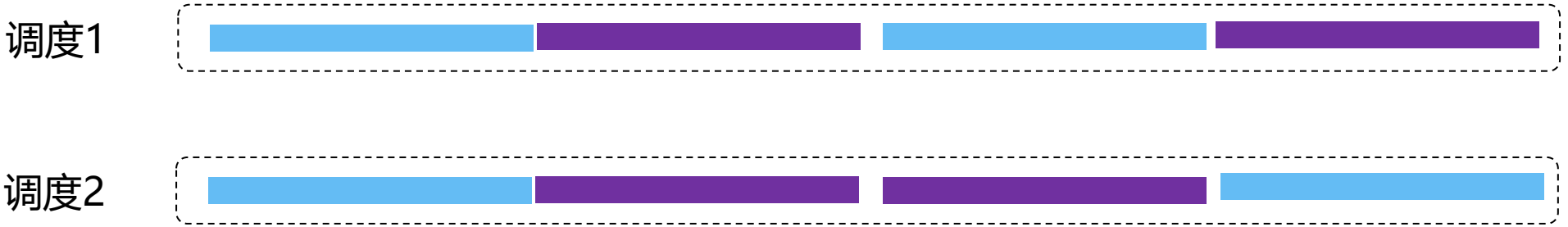




调度与上下文切换

调度的一项关键任务： 处理上下文切换





上下文切换成本

进程上下文切换的成本：

Benchmark	Operating System	Operation	Time per Op
Spawn New Process	NT	spawnl()	12.0 ms
	Linux	fork()/exec()	6.0 ms
Clone Current Process	NT	NONE	N/A
	Linux	fork()	1.0 ms
Spawn New Thread	NT	CreateThread()	0.9 ms
	Linux	pthread_create()	0.3 ms
Switch Current Process (20 runnable processes)	NT	Sleep(0)	0.010 ms
	Linux	sched_yield()	0.019 ms
Switch Current Thread (20 runnable threads)	NT	Sleep(0)	0.006 ms
	Linux	sched_yield()	0.019ms

图片来源：https://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/029/2941/2941f1.jpg

为什么需要上下文切换

既然上下文切换有成本，为什么还要支持它呢?

目标1：提升响应能力（交互）

调度1

此调度对于执行程序B的用户，很长时间后才能得到响应，
缺少良好的交互体验

交互体验略有提升。
若想交互体验更好，可以将任务继续细分为更小的片段，而后通过调度让片段交替执行

为什么需要上下文切换

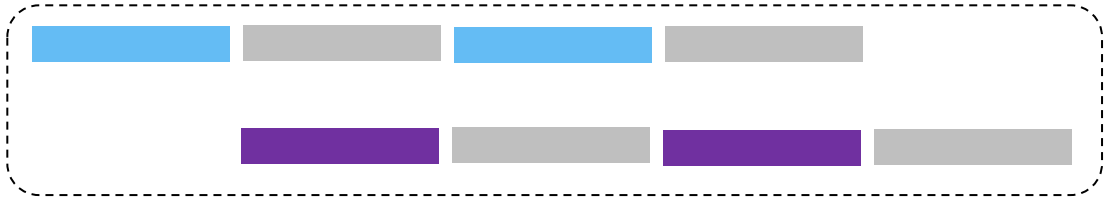
既然上下文切换有成本，为什么还要支持它呢?

目标2：提升效率

调度1

CPU资源浪费严重

调度2



CPU资源利用率大大提升

为什么需要上下文切换

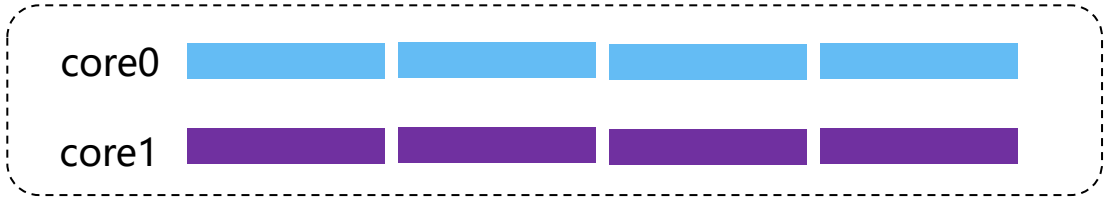
既然上下文切换有成本，为什么还要支持它呢?

目标3：充分利用硬件并行能力

调度1

CPU资源浪费严重

调度2



线程

进程上下文切换的成本：

Operating System	Benchmark	Number of Contexts in Run Queue					
		2	4	8	10	20	40
Linux Before Change	Process Switch	19 us	13 us	13 us	14 us	16 us	27 us
	Thread Switch	16 us	11 us	10 us	10 us	15 us	23 us
Linux After Change	Process Switch	4 us	6 us	11 us	12 us	15 us	28 us
	Thread Switch	3 us	3 us	5 us	7 us	12 us	22 us
NT	Process Switch	10 us	15 us	15 us	17 us	16 us	17 us
	Thread Switch	5 us	8 us	8 us	9 us	10 us	11 us

图片来源：http://euclid.nmu.edu/~rappleto/Research/Papers/Scheduler/understanding.html

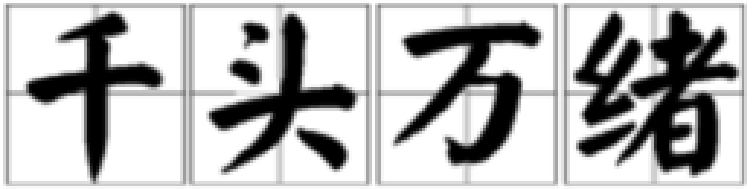
结果：引入线程（Thread）的概念

线程

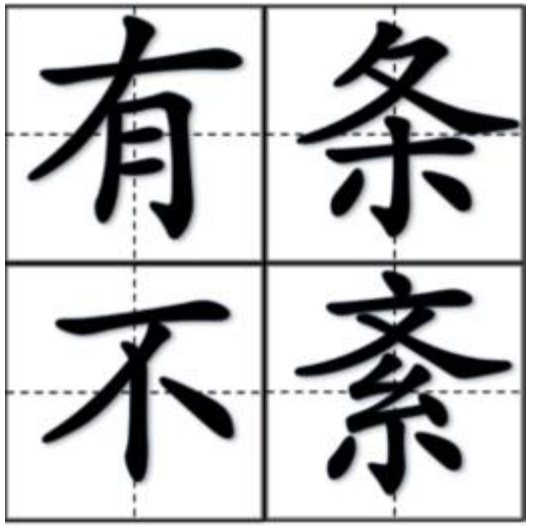
Thread

01

线程的引入

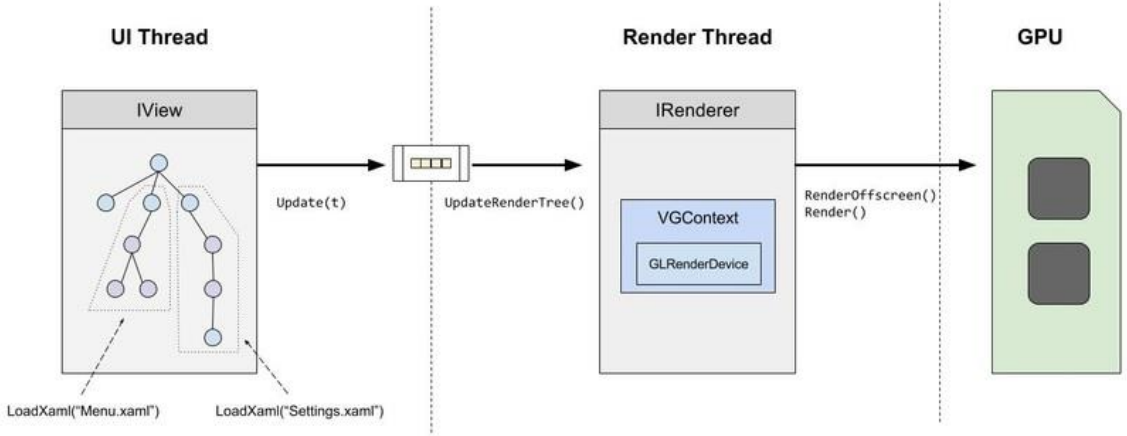


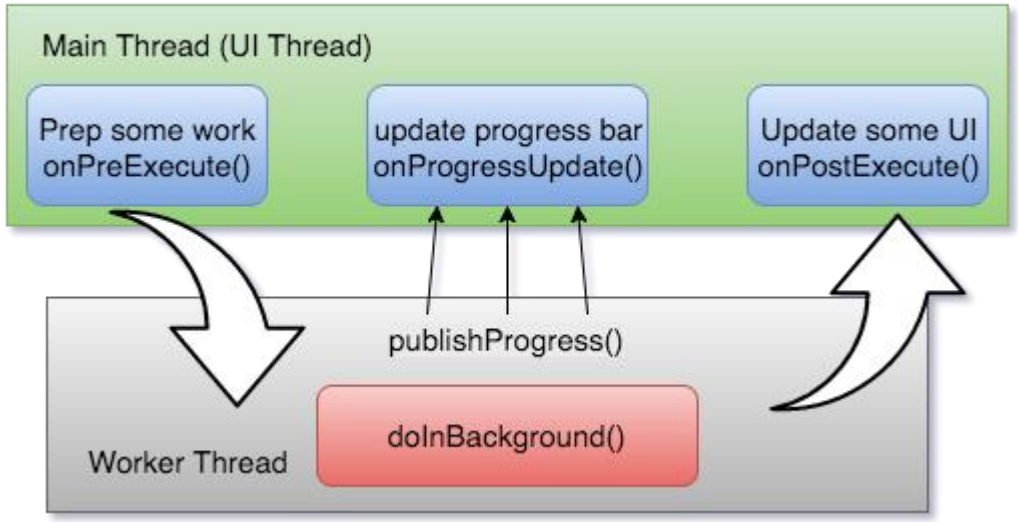
如何做到：



线程的引入

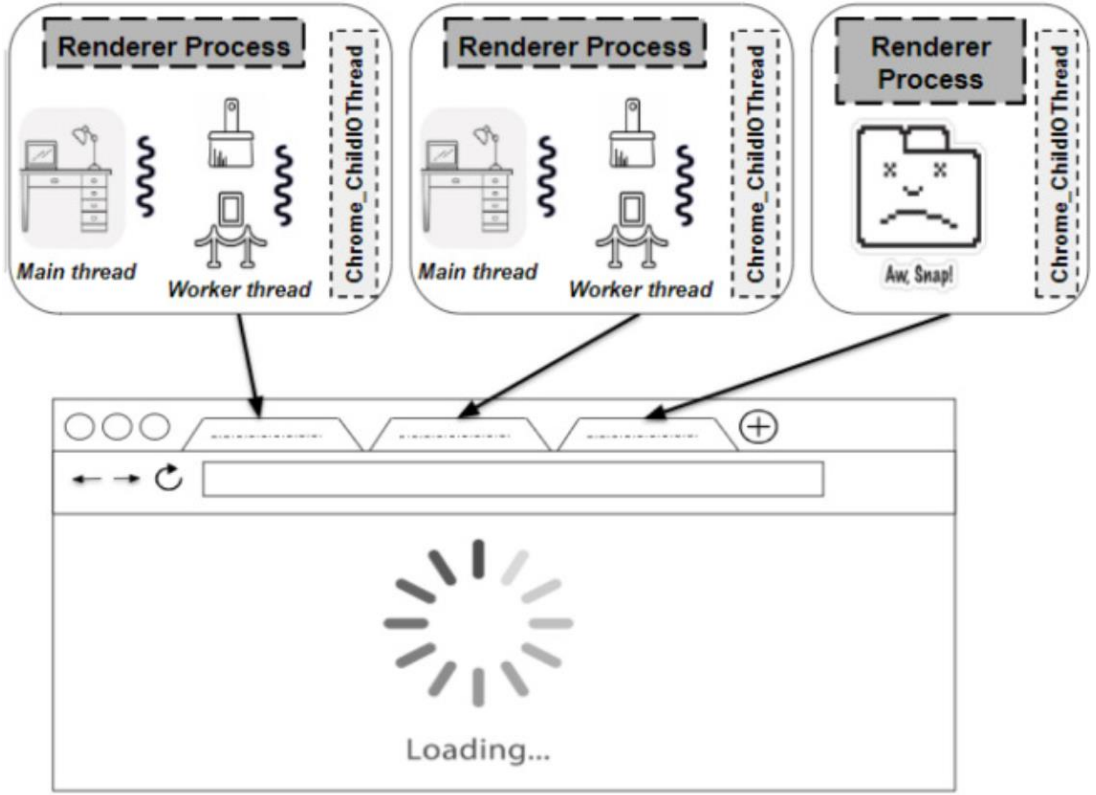
任务示例1：GUI

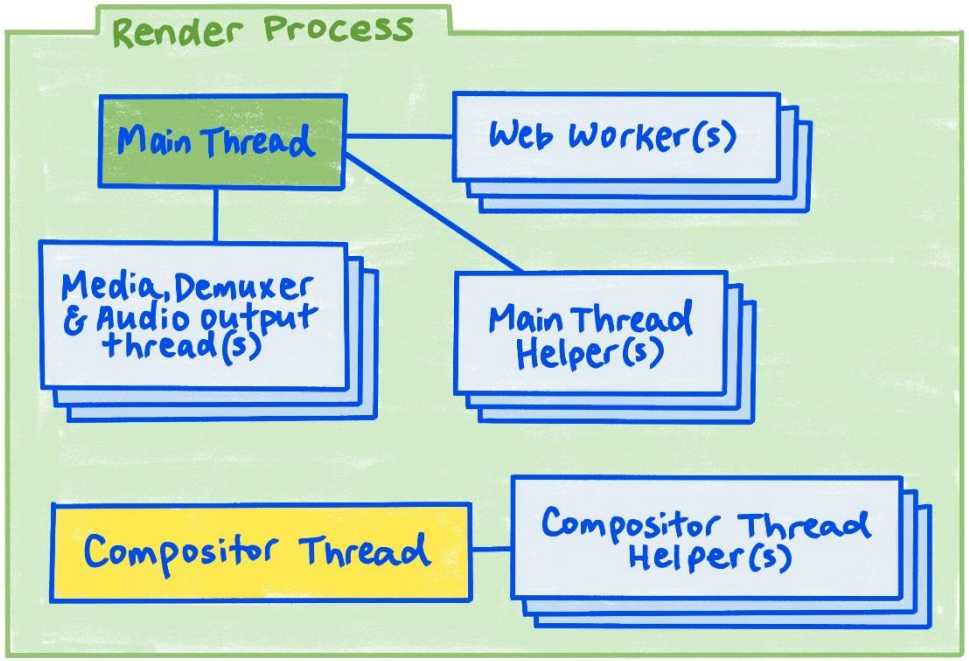




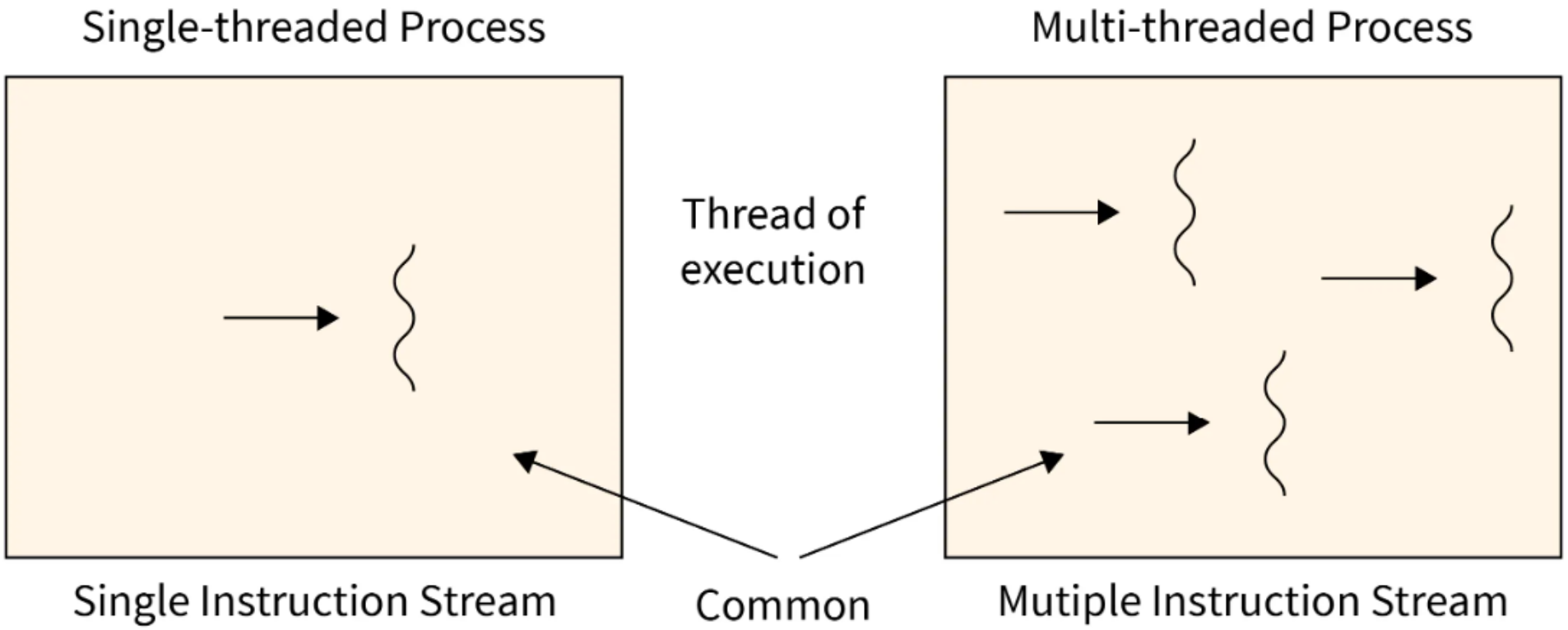
线程的引入

任务示例2：网页渲染



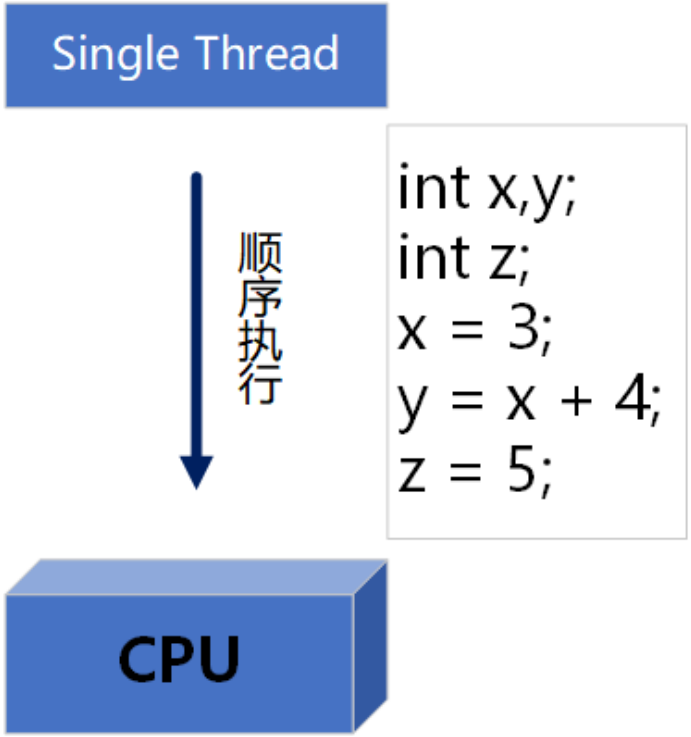


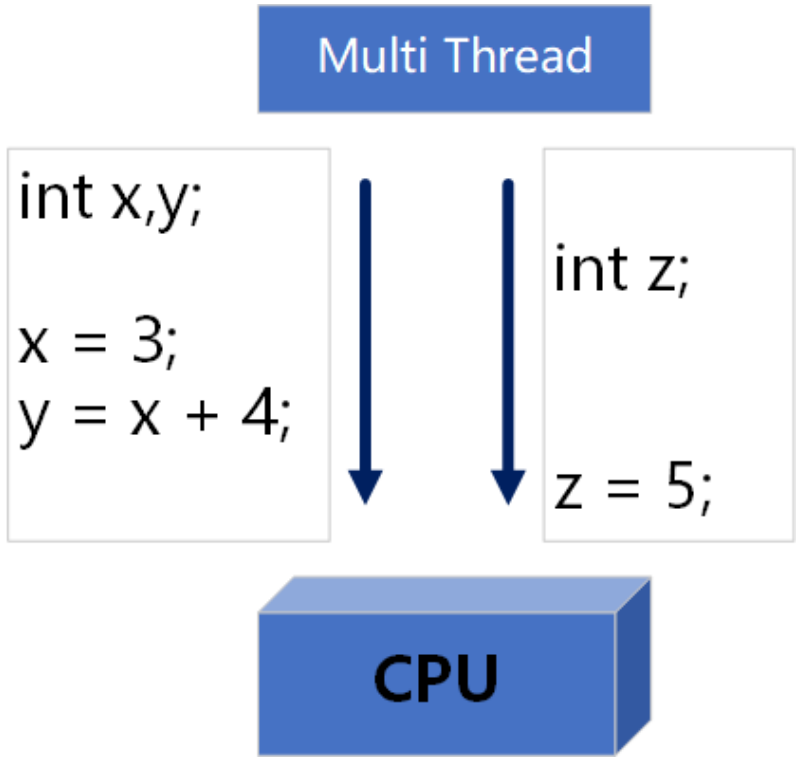
什么是线程



图片来源：https://www.scaler.com/topics/multithreading-in-os/

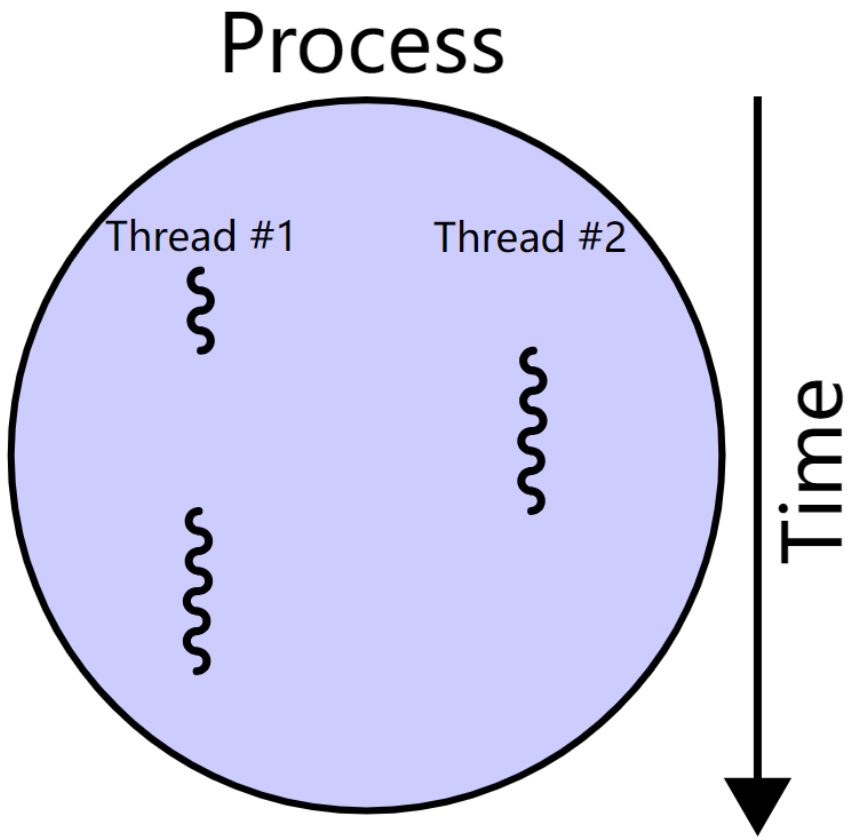
什么是线程





- **线程**是将进程的计算任务进一步细分后得到的更细粒度的计算单位

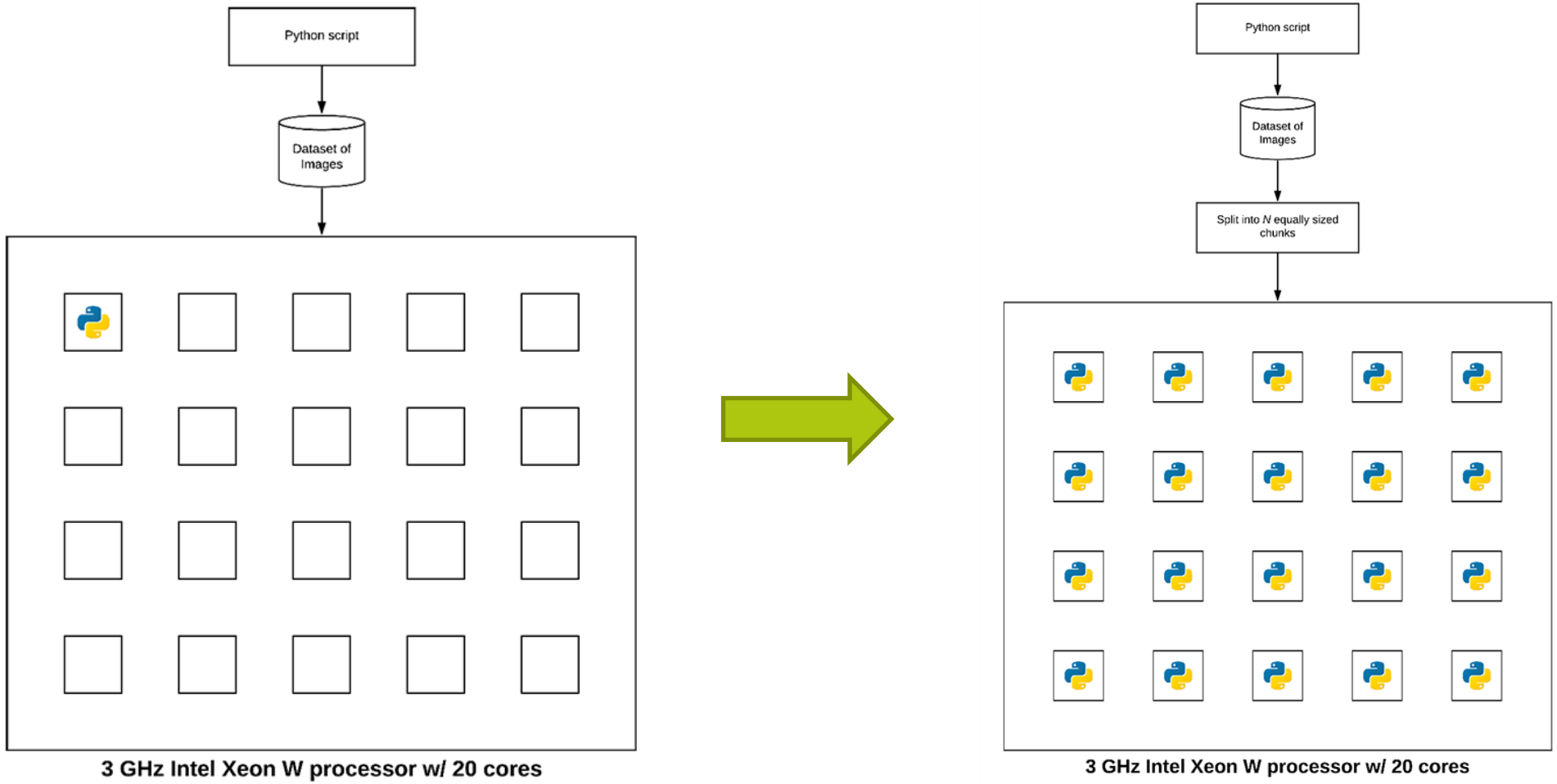
什么是线程



图片来源：https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)

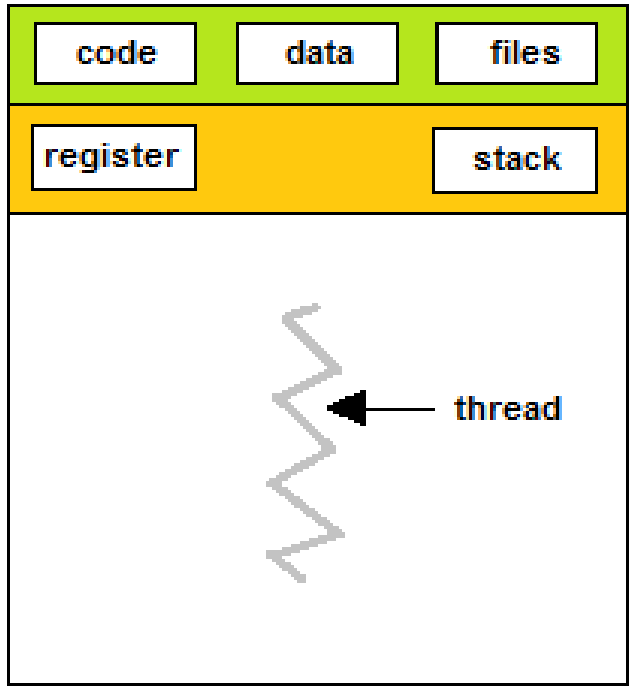
什么是线程

- **引入线程的动机**：带来并发，充分利用多核资源，提升效率

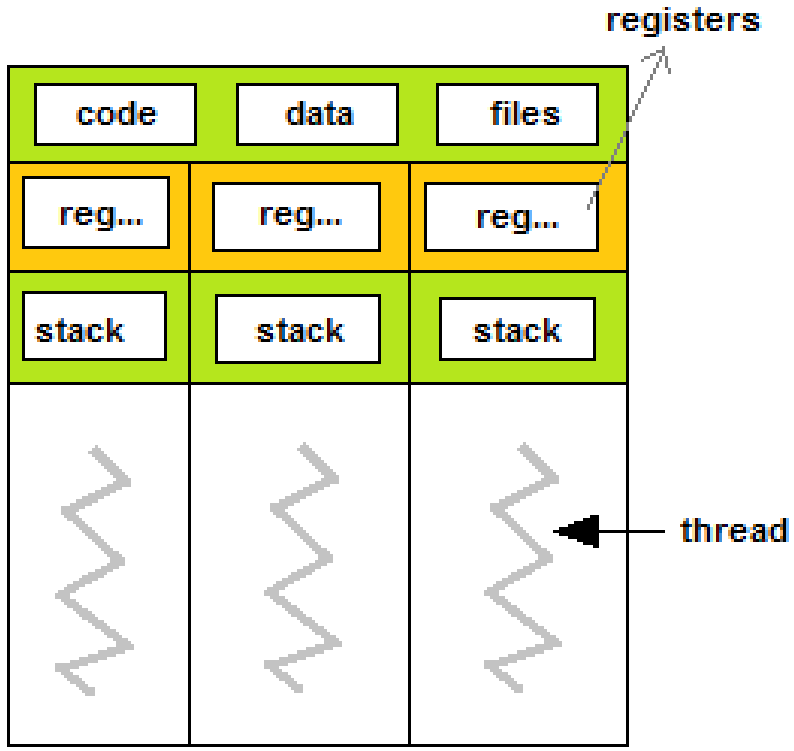


什么是线程

- **线程概念**：一串执行的代码流（通常实现为**线程主函数**）



single-threaded process



multithreaded process

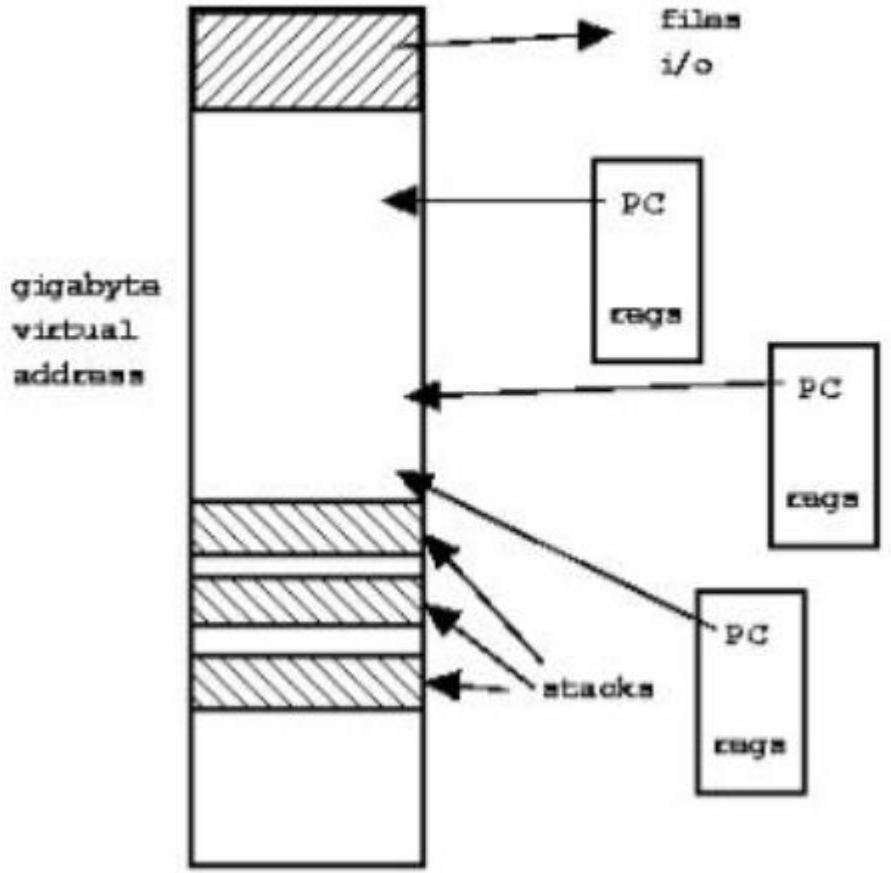
线程：进程中一个单一顺序的控制流。

Win32线程示例

```
#include <windows.h>
#include <iostream>

DWORD WINAPI myThread(LPVOID lpParameter)
{
        unsigned int& myCounter = *((unsigned int*)lpParameter);
        while(myCounter < 0xFFFFFFFF) ++myCounter;
        return 0;
}

int main(int argc, char* argv[])
{
        using namespace std;

        unsigned int myCounter = 0;
        DWORD myThreadID;
        HANDLE myHandle = CreateThread(0, 0, myThread, &myCounter;, 0, &myThreadID;);
        char myChar = ' ';
        while(myChar != 'q') {
                cout << myCounter << endl;
                myChar = getchar();
        }

        CloseHandle(myHandle);
        return 0;
}
```

什么是线程

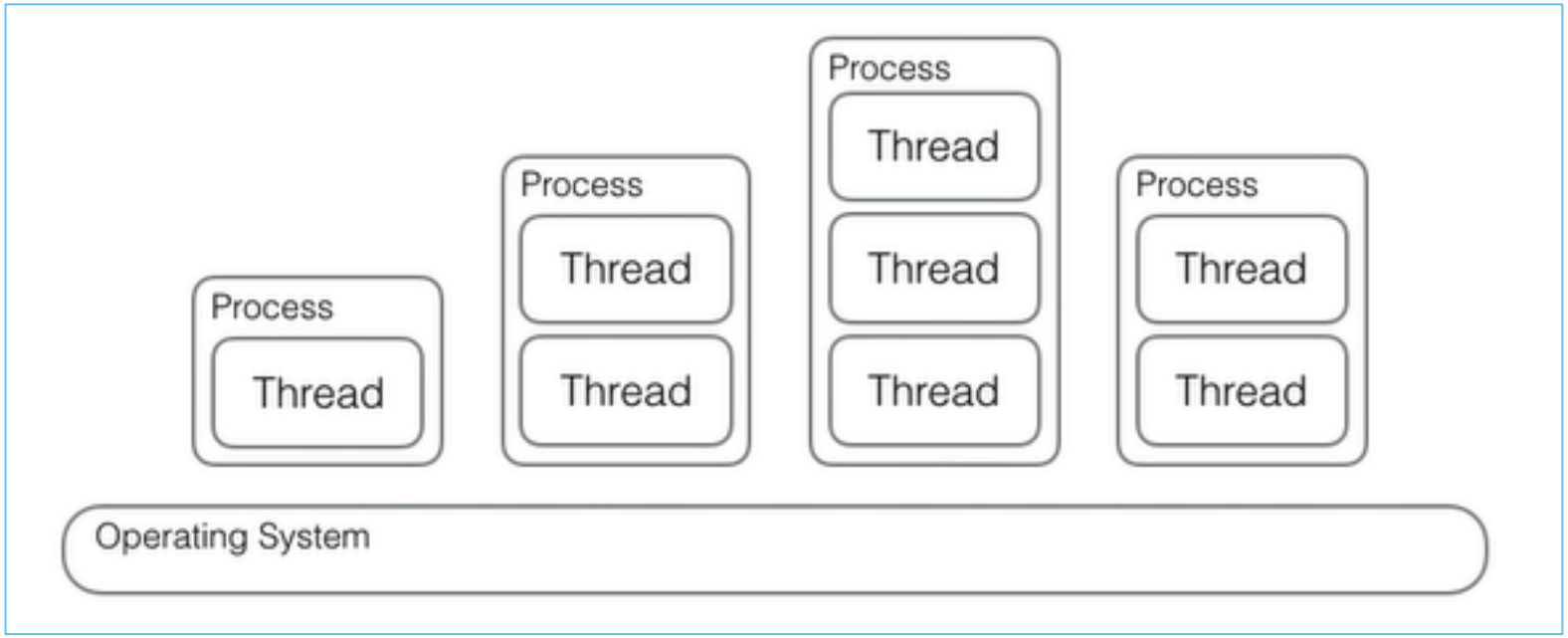
- **线程之间：Share address space, files, …**



Each thread has its own stack, PC, registers。

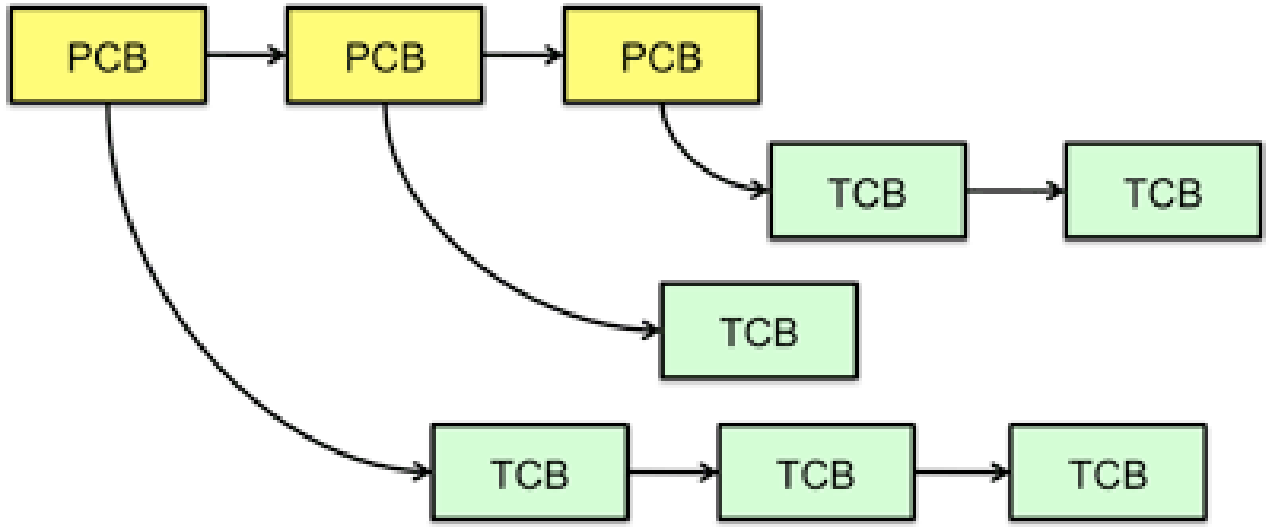
什么是线程

- **支持线程后的操作系统多任务模式：**
  - **单个进程可以支持多个线程**



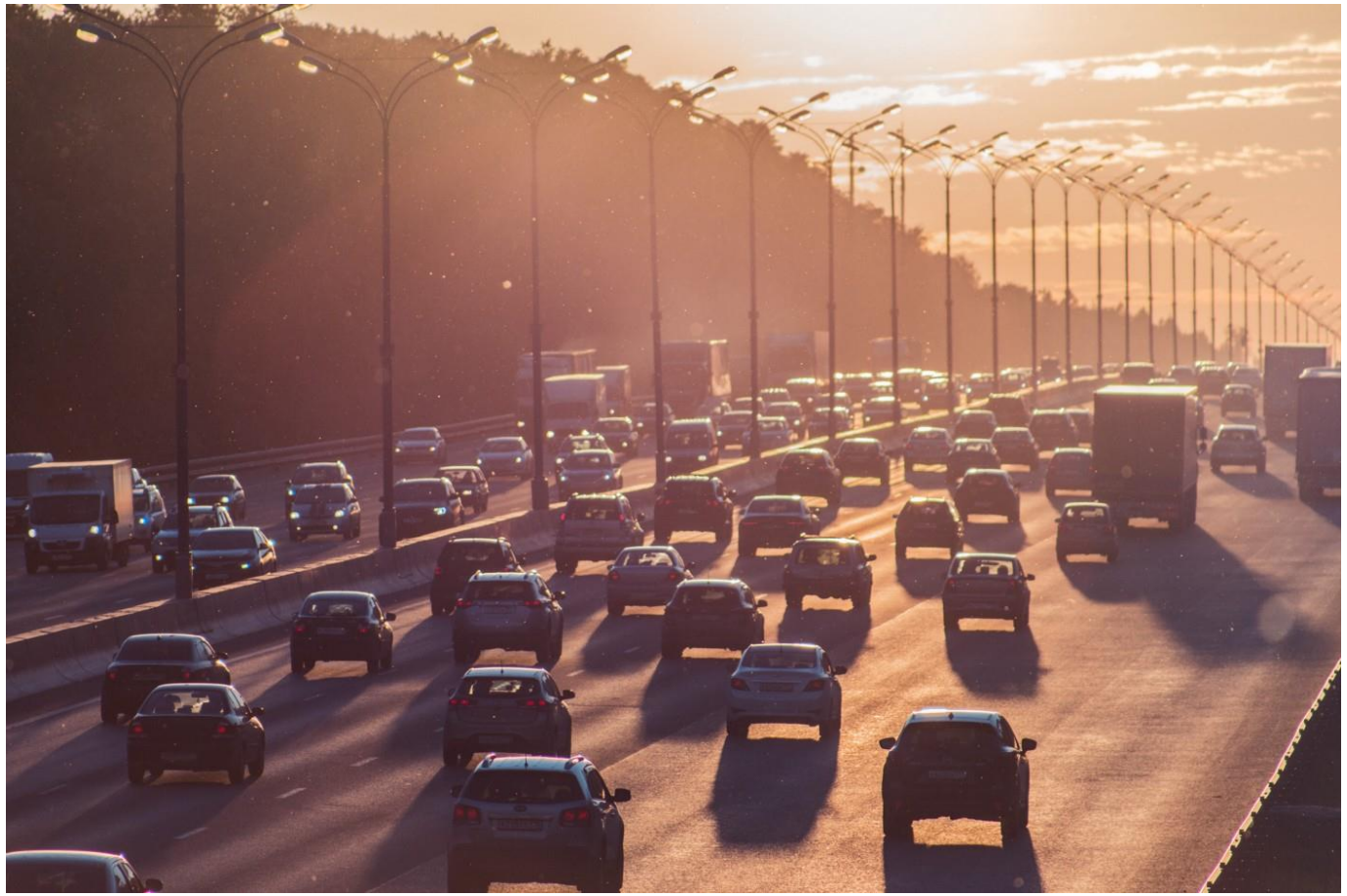
什么是线程

- **线程管理数据结构**：线程控制块（Thread Control Block, TCB）
  - 支持线程的操作系统中，通常提供TCB对线程进行管理



线程的意义

典型生活场景：Traffic



线程的意义

分析计算中的并行性（示例：data parallelism）

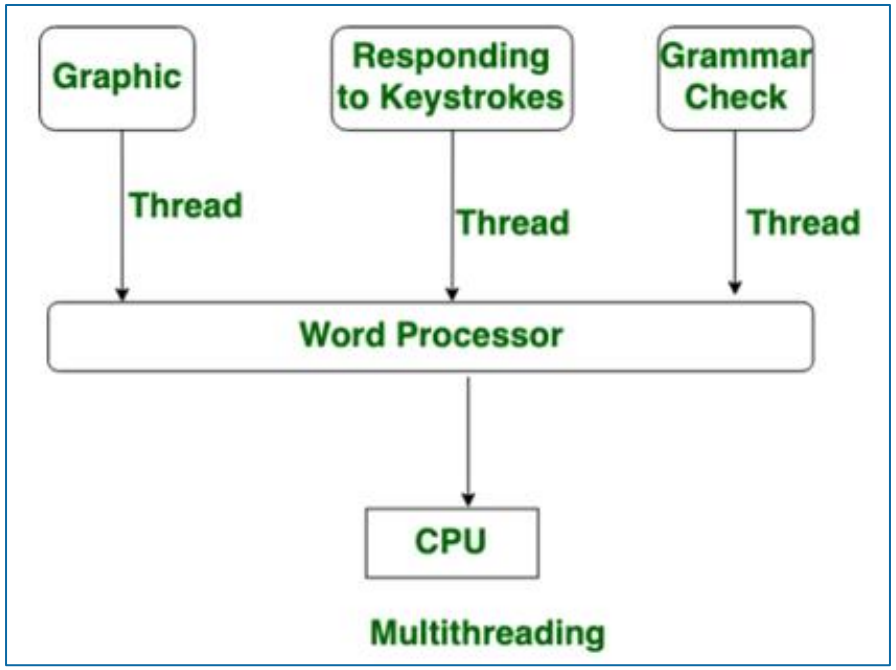
$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

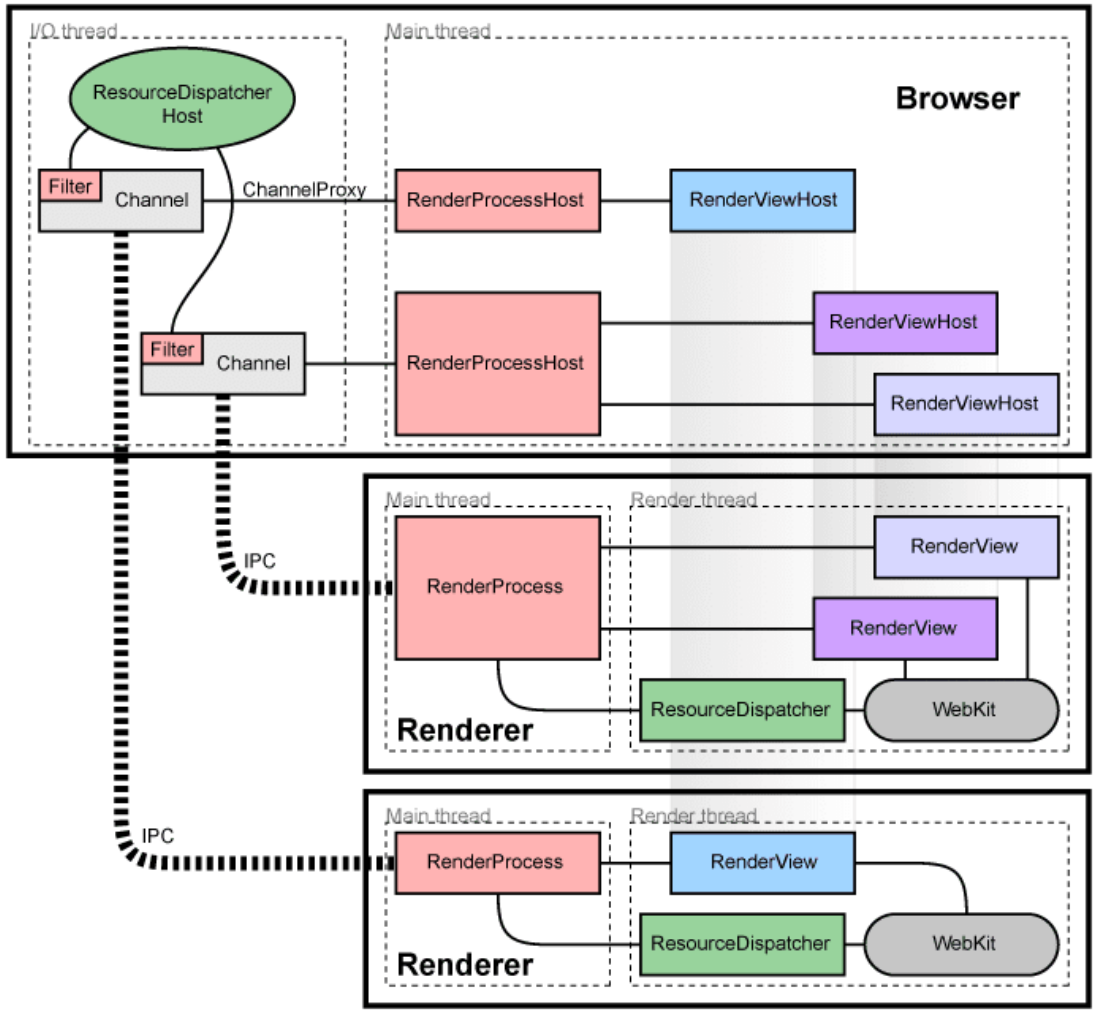
```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

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

线程的意义

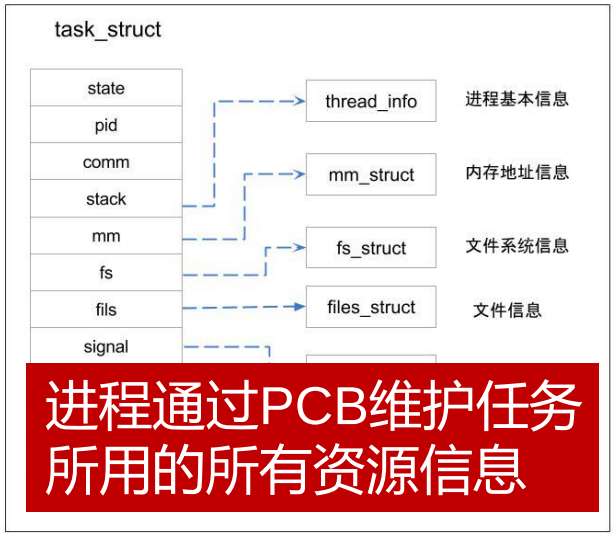
分析计算中的并行性（示例：instruction parallelism）



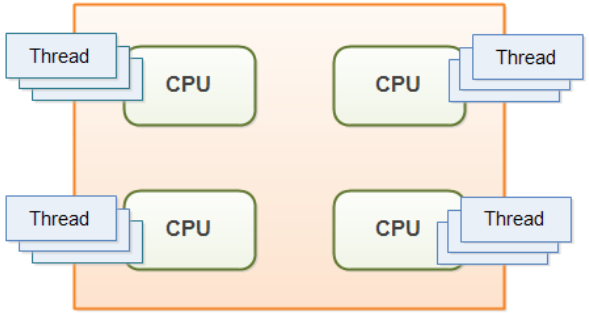


线程 v.s. 进程

- **线程与进程之间的联系**
  - 进程：拥有资源
  - 线程：使用所隶属进程的资源进行计算



进程通过PCB维护任务所用的所有资源信息



线程利用CPU资源完成计算任务

MultiTasking v.s. MultiThreading

线程 v.s. 进程

- **线程与进程之间的联系**
  - 进程：拥有资源
  - 线程：使用所隶属进程的资源进行计算

进程是舞台，线程是演员
进程是平台，线程是人才

引入线程的意义

- **引入线程可以带来的好处**
  - 提升系统性能
  - 节约资源
  - 简化编程模型（可以通过共享变量实现不同线程间的通信）

- **注意：线程的负面因素**
  - 多线程编程模型的复杂性
  - 因共享资源可能引发的各种问题，如数据一致性、并发访问冲突等
  - 线程过多也会导致系统的负荷增加，进而影响系统的稳定性和性能

线程模型

Threading Model

02

线程分类

用户级线程vs内核级线程

用户级线程

- 在用户态以线程库的形式实现
- 对用户级线程的操作通过调用用户态线程库API进行

内核级线程

- 在内核态实现，OS内核直接管理
- 线程的创建由系统调用完成

用户级线程

优势：

- 线程的调度不需要内核直接参与，控制简单。
- 没有内核线程机制支持，也可以实现用户级线程

缺点：

- 一个用户级线程的阻塞将会引起整个进程的阻塞

内核级线程

优势：

- 是内核中参与竞争CPU资源的基本调度单位。
- 内核级线程所耗费的资源比进程小，切换效率比进程切换高

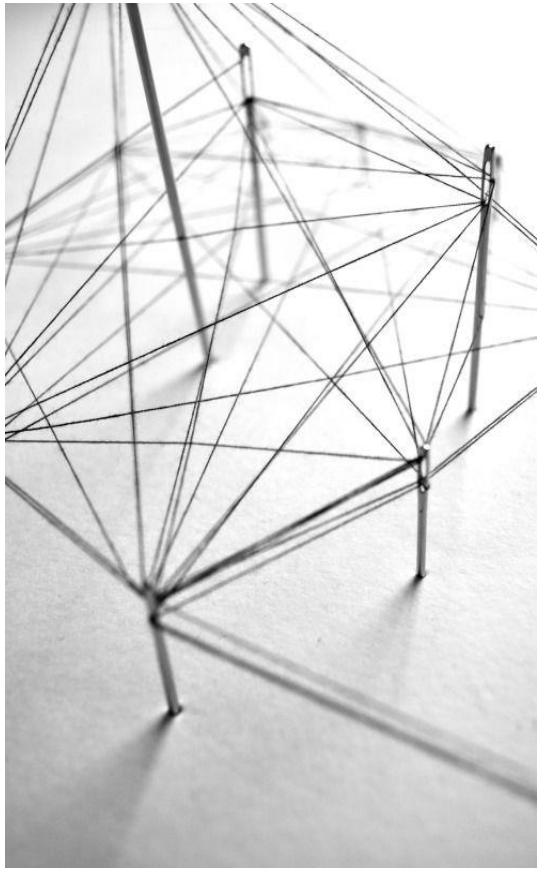
缺点：

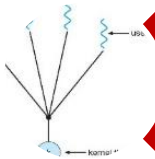
- 比用户级线程还是更重量级，切换效率不如用户级线程

用户级线程与内核级线程

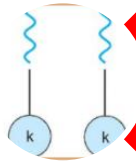
- 两者之间的差别在于性能。
- 内核级线程能够被内核感知，而用户级线程不能
- 用户级线程的创建、撤销和调度不需要内核的支持，由用户级线程库完成
- 用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断

线程模型

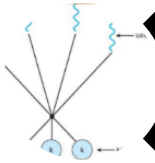




M:1



1:1



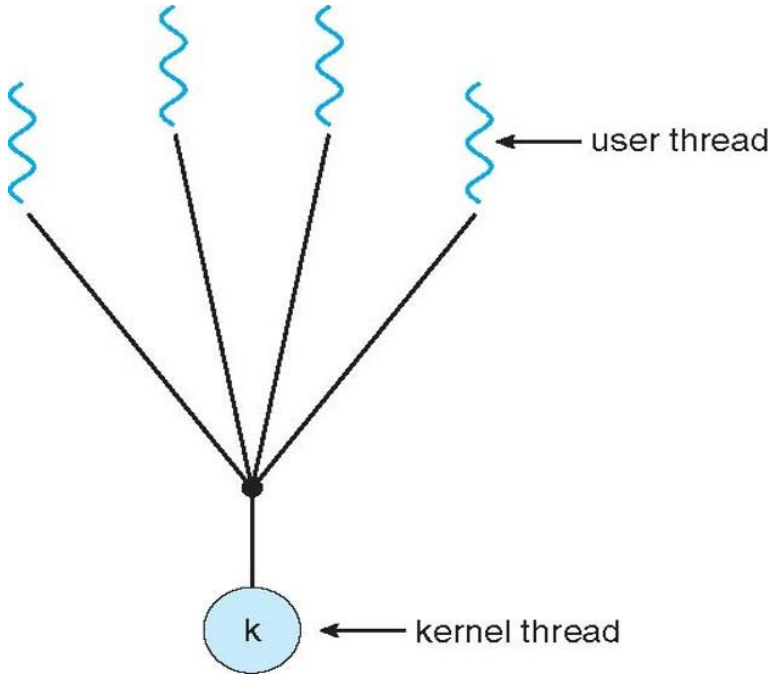
M:N

用户级线程 – 内核级线程

线程模型

线程模型1（M:1模型）

- 多个用户级线程绑定到一个内核级线程(M:1)



M:1线程模型代表：

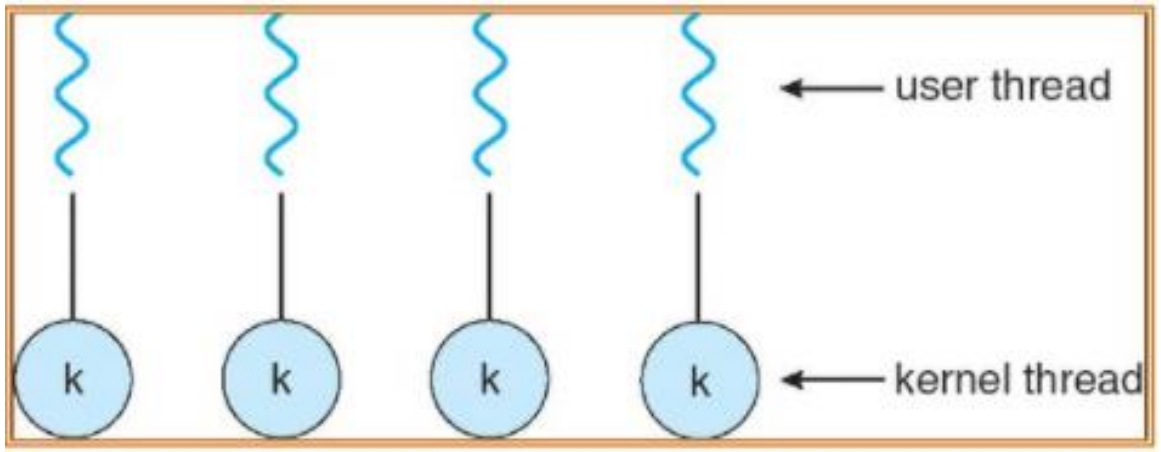
GNU Portable Threads

问题：
如果一个用户级线程引起阻塞，会导致整个进程中所有的线程阻塞

线程模型

线程模型2（1:1模型）

- 将1个用户级线程绑定到1个内核级线程(1:1)



1:1线程模型代表：

NPTL（by redhat）

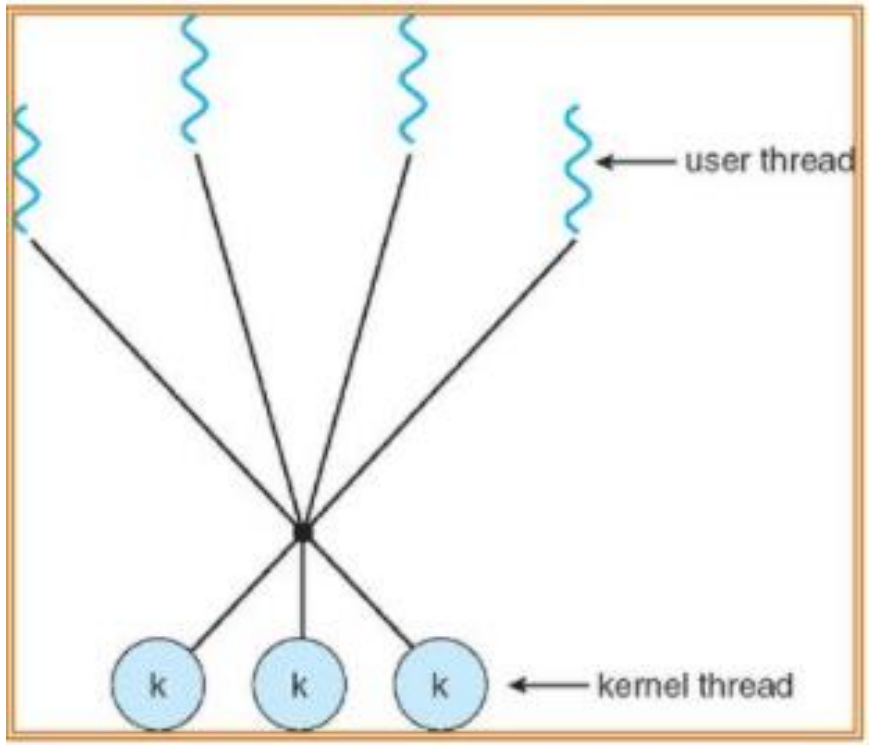
redhat

问题：
需要内核级线程的数量多，消耗更多内核资源

线程模型

线程模型3（M:N模型）

- 将多个用户级线程绑定到多个内核级线程(M:N)



M:N线程模型代表：

Solaris threads

SOLARIS™ 1992 - 2005

solaris™ 2005 - 2010

ORACLE SOLARIS 2010 - now

问题：
管理复杂，影响效率

线程模型示例

- 混合模型

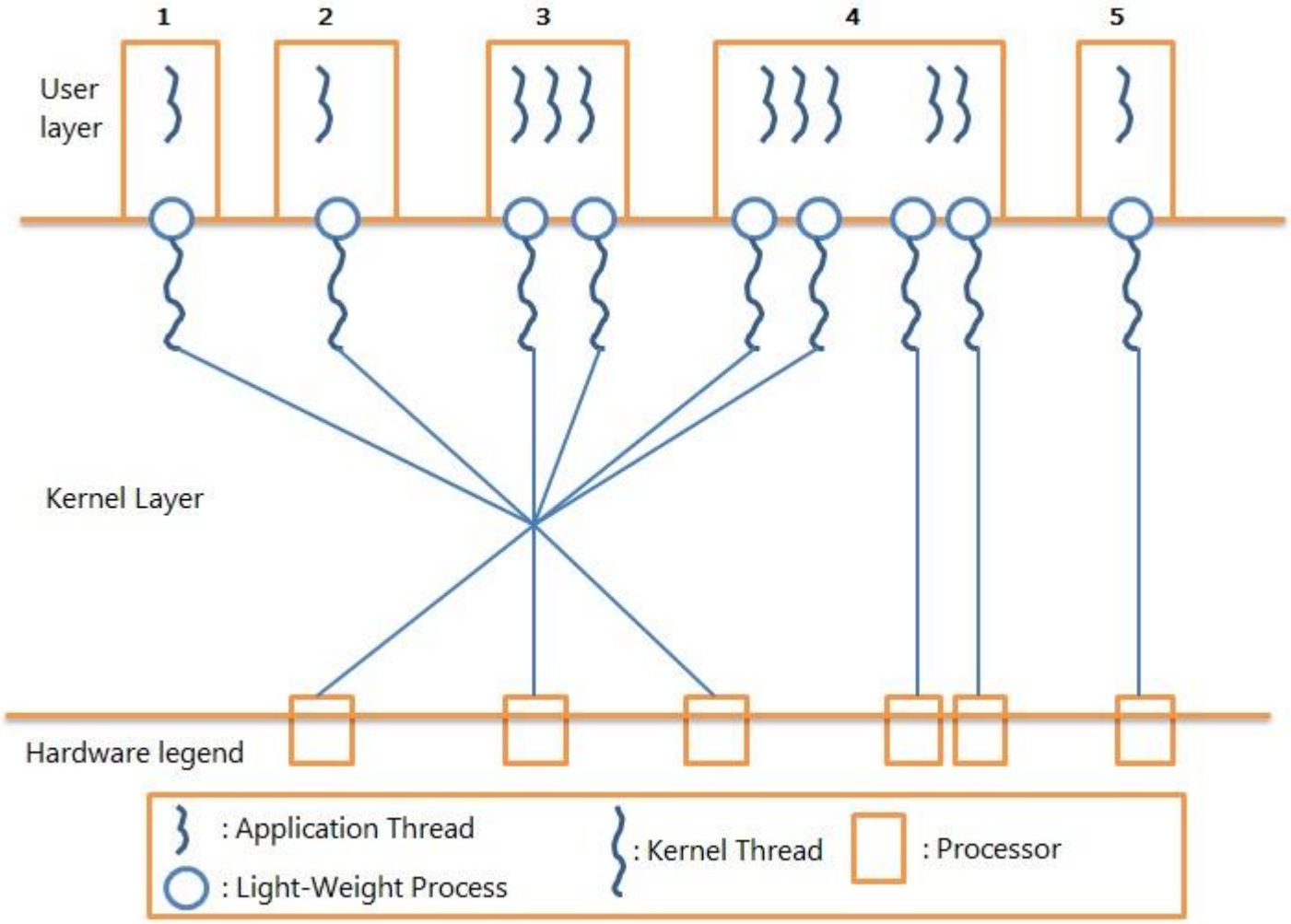


Fig: Solaris Multi-threaded Architecture

Pthreads编程

Pthreads programming

03

Linux Pthreads库

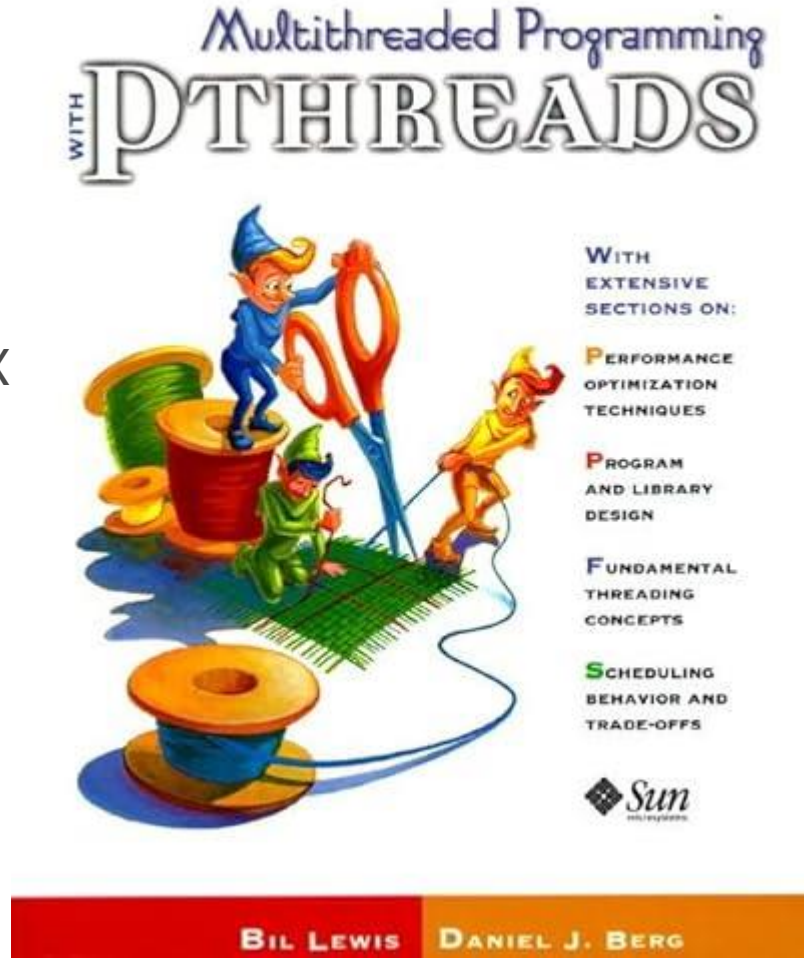
- 线程的历史

1993 年 5 月，Sun Solaris 2.2 推出了 UI 线程，以及一个名为*libthread*的线程库

与之竞争的 Unix 供应商很快就提出了他们自己的专有多线程解决方案(使用暴露 API 的运行时库)–带 DECthread 的数字解决方案(后来被 Compaq Tru64 Unix 和后来的 HP-UX 吸收)、带 AIX 的 IBM、带 IRIX 的 Silicon Graphics 等等

1995 年，IEEE 成立了一个单独的 POSIX 解决方案委员会-IEEE 1003.1c-**POSIX 线程**和(**pthreads**)委员会，以发展多线程 API 的标准化解决方案

1996 年，Xavier Leroy 就着手构建了 Linux 的第一个 pthread 实现–LinuxThreads



https://m.media-amazon.com/images/I/51ihhiIN0jL._SY522_.jpg

- NGPT vs. NPTL

早期解决问题的努力被称为**下一代 POSIX 线程(NGPT)**。

大约在同一时间，RedHat 也投入了一个团队来从事这方面的工作；他们把这个项目称为**Native POSIX 线程库(NPTL)**。

NPTL秉承开源文化的最好传统，NGPT 开发人员与他们在 NPTL 的同行一起工作，并开始将 NGPT 的最佳功能合并到 NPTL 中。 NGPT 开发在 2003 年的某个时候就被放弃了；到那时，Linux 上 pthread 的现实实现–一直保留到今天–是 NPTL。

可以使用以下代码(在 Fedora 28 系统上)查找线程实现(从 glibc 2.3.2 开始):

```
$ getconf GNU_LIBPTHREAD_VERSION
NPTL 2.27
```

线程创建

- pthread_create()

```
int pthread_create(pthread_t *threadID,
                   const  pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

Input *Parameters:*

- `attr` = contains the **attributes (properties)** of the ***new* thread**
  - **Thread attributes** are **discussed *later***
- `start_routine` = **name** of the **function** that the ***new* thread** will **execute**
  - This **function** has **one parameter** of the **type** `(void *)`
  - The **return value type** of this **function** is `void *`

  I.e.:

```
void *  start_routine( void * arg )
{
    ....
}
```

- `arg` = the **argument** that will be **passed** to the `start_routine` function when it **executes**

Output parameters

- `threadID` = the **identifier** of the ***new* thread**
  - We will use the `threadID` to **wait** for the **thread** to **finish**

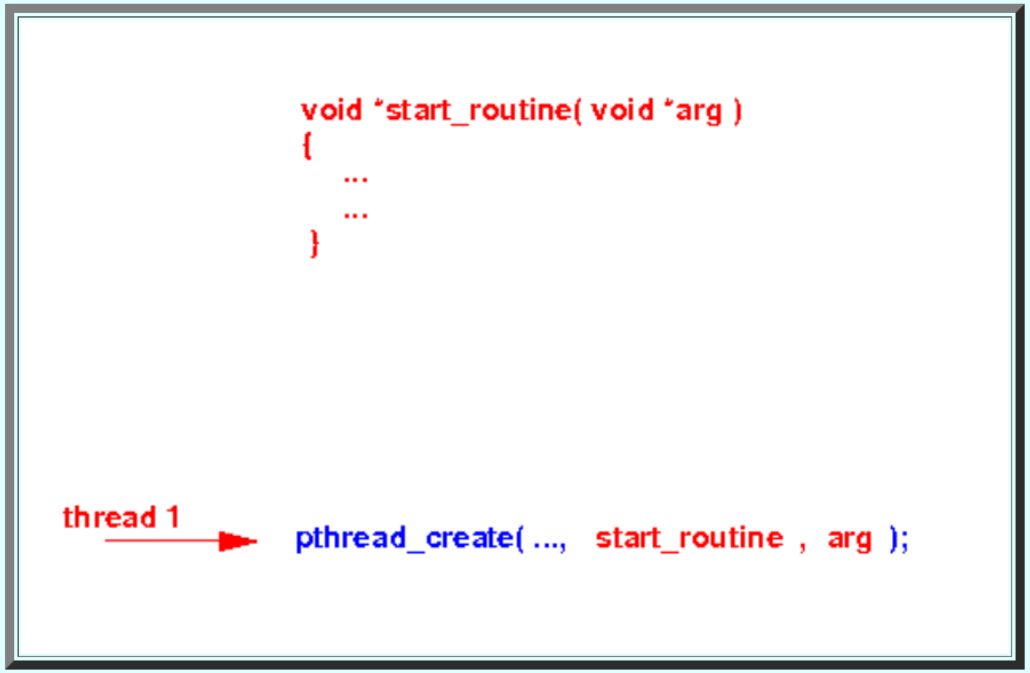
Return value:

- Returns **0** if **thread** is **created successfully**
- **Otherwise**, returns an **error code**

线程创建

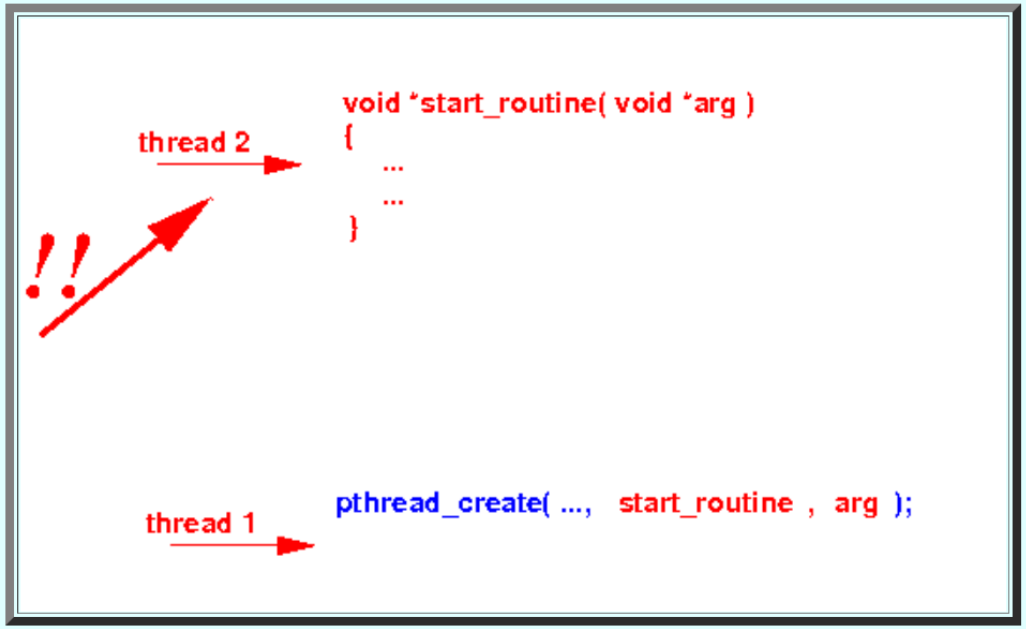
- 在程序中调用pthread_create()的效果

- **Before** the `pthread_create( )` call:



Thread 1 is **executing** the `pthread_create( )` call

- **After** the `pthread_create( )` call:



Thread 1 will **continue execution** the **statements after** the `pthread_create( )` call
A ***new*** **thread** Thread 2 will **start** its **execution** with the **function** `start_routine`

Pthreads Hello World

- Hello示例

```
#include <pthread.h>

/*====================================
   Thread prints "Hello World"
   ==================================== */
void *worker(void *arg)
{
   printf ("Hello World !\n  “);

  return(NULL);     /* Thread exits (dies) */
}
```

Pthreads Hello World

- Hello示例

```
#include <pthread.h>

/* ====================================
   Thread prints "Hello World"
   ==================================== */
void *worker(void *arg)
{
   printf ("Hello World !\n“);

  return(NULL);     /* Thread exits (dies) */
}
```

```
/* ========================================
   MAIN: create a thread and wait for it to finish
   ======================================== */
 int main(int argc, char *argv[])
 {
   pthread_t  tid;

   if ( pthread_create(&tid, NULL, worker, NULL) )
   {
      printf( "Cannot create thread\n“);
      exit(1);
   }

   printf( "Main waits for thread to finish....\n“) ;

   pthread_join(tid, NULL);

   exit(0);
 }
```

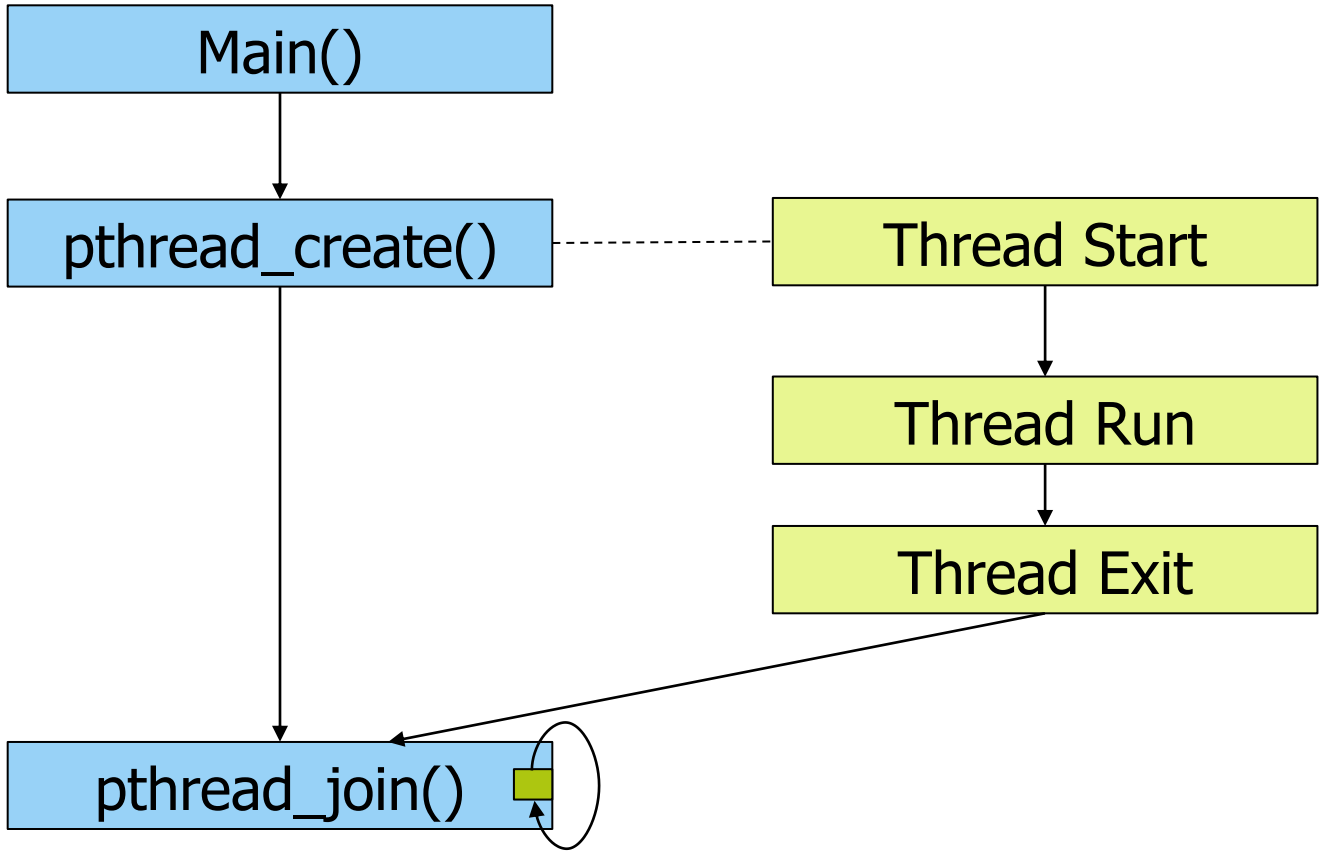
线程控制：pthread_join

- pthread_join()

```
NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);

       Compile and link with -pthread.
```

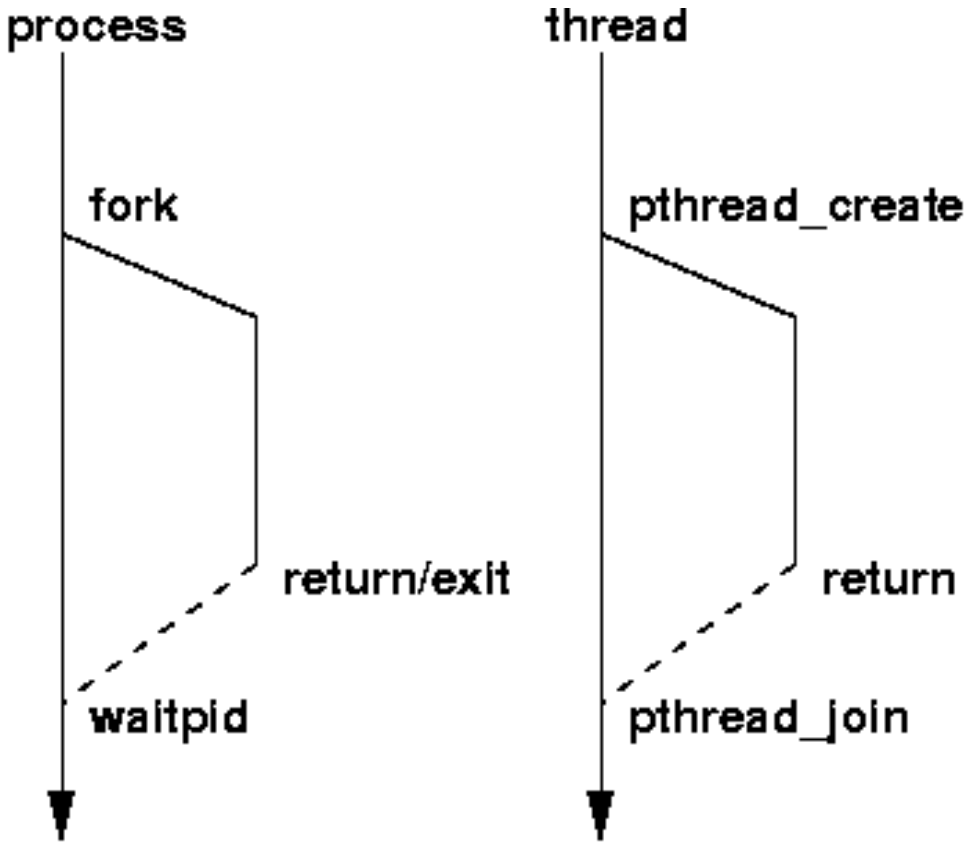
线程控制：pthread_join

- pthread_join()



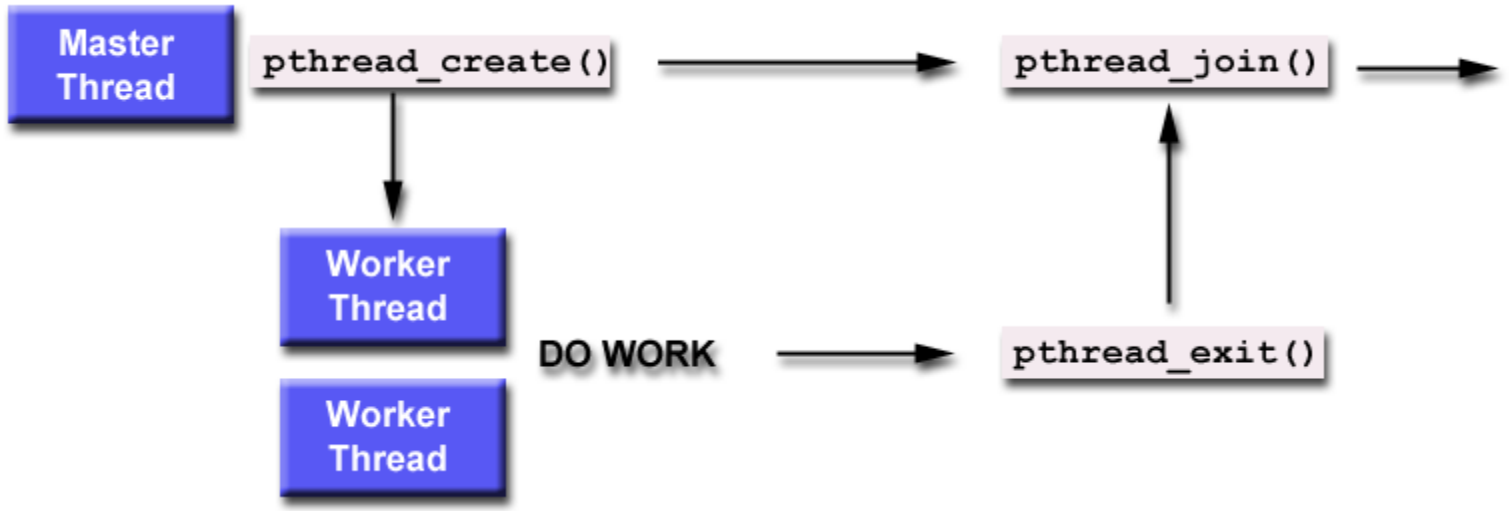
线程控制：pthread_join

- pthread_join()



Linux Pthreads库

- Linux Pthread接口：pthread_join



Picture from https://computing.llnl.gov/tutorials/pthreads/

- Linux Pthread程序编译

1. `#include <pthread.h>` in main file

2. Compile source with -lpthread or -pthread

```
Intro to OS ~ ==> gcc -o main main.c -lpthread
Intro to OS ~ ==> gcc -o main main.c -pthread
```

3. Check return values of common functions

Summary

小结：

- 线程概念
- 线程模型
- Linux线程

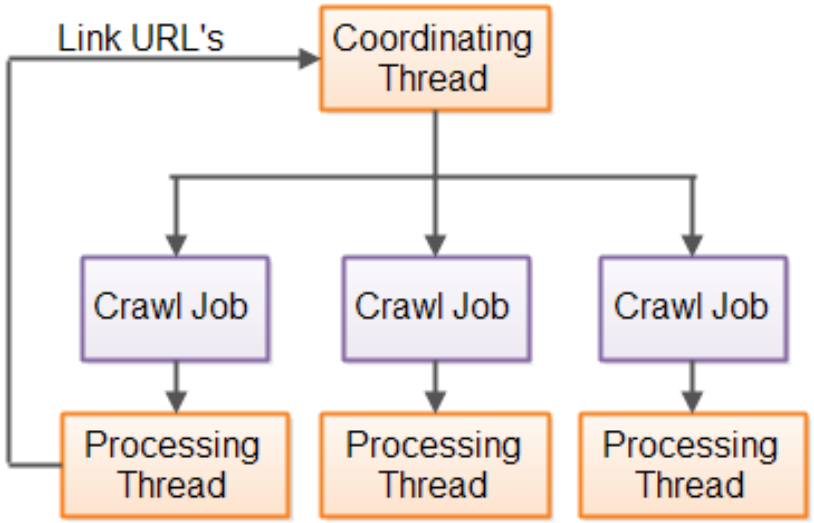
- Linux Pthread：示例

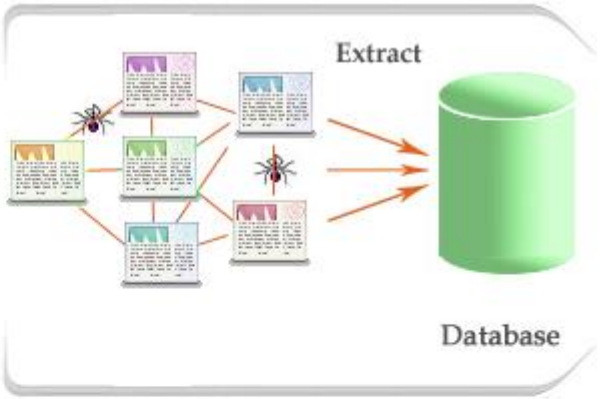
https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

- Thread Basics
- Thread Creation and Termination
- Thread Synchronization
- Thread Scheduling
- Thread pitfalls
- Thread Debugging

线程应用场景

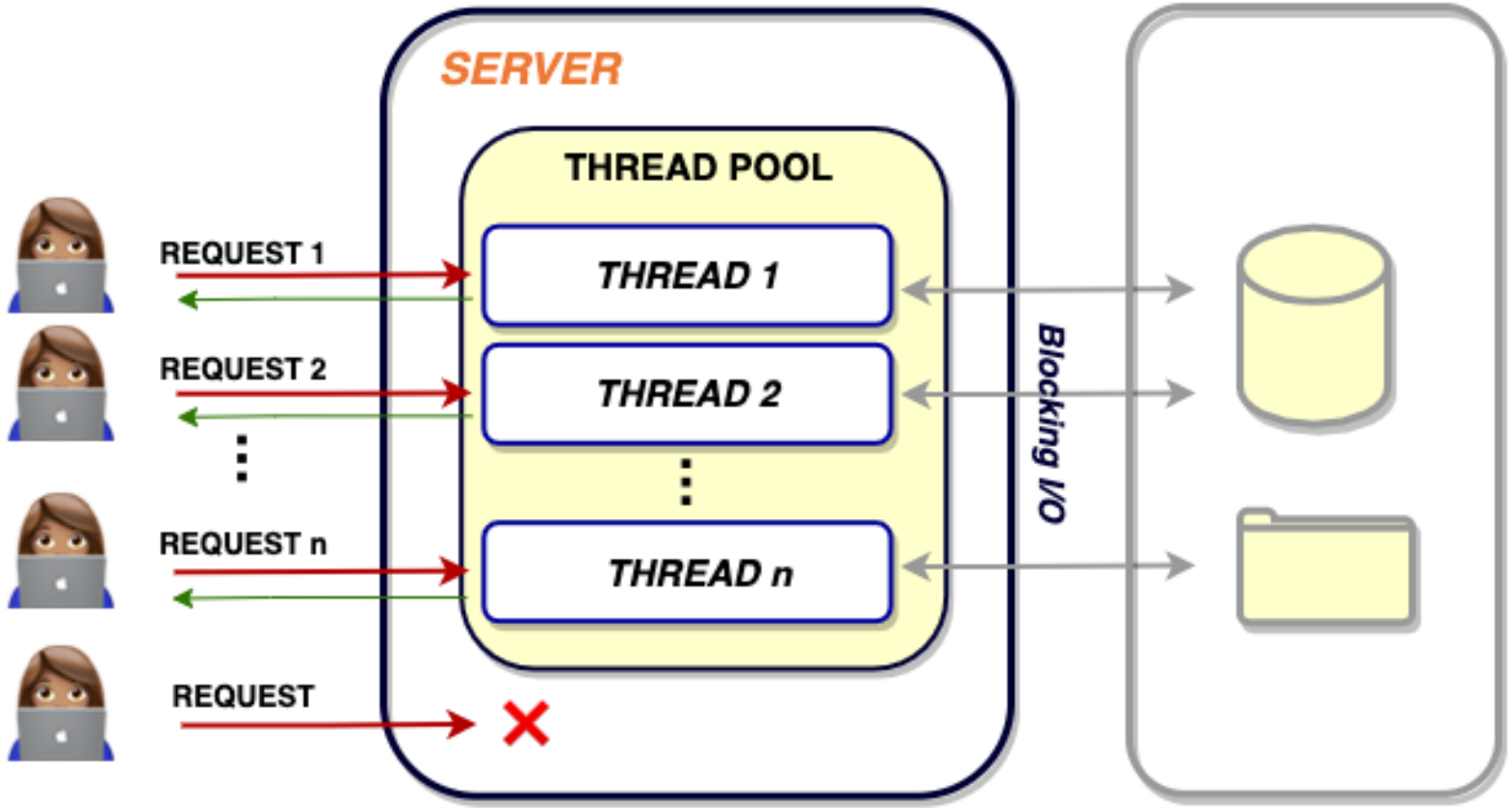
典型应用场景1：Web Crawler





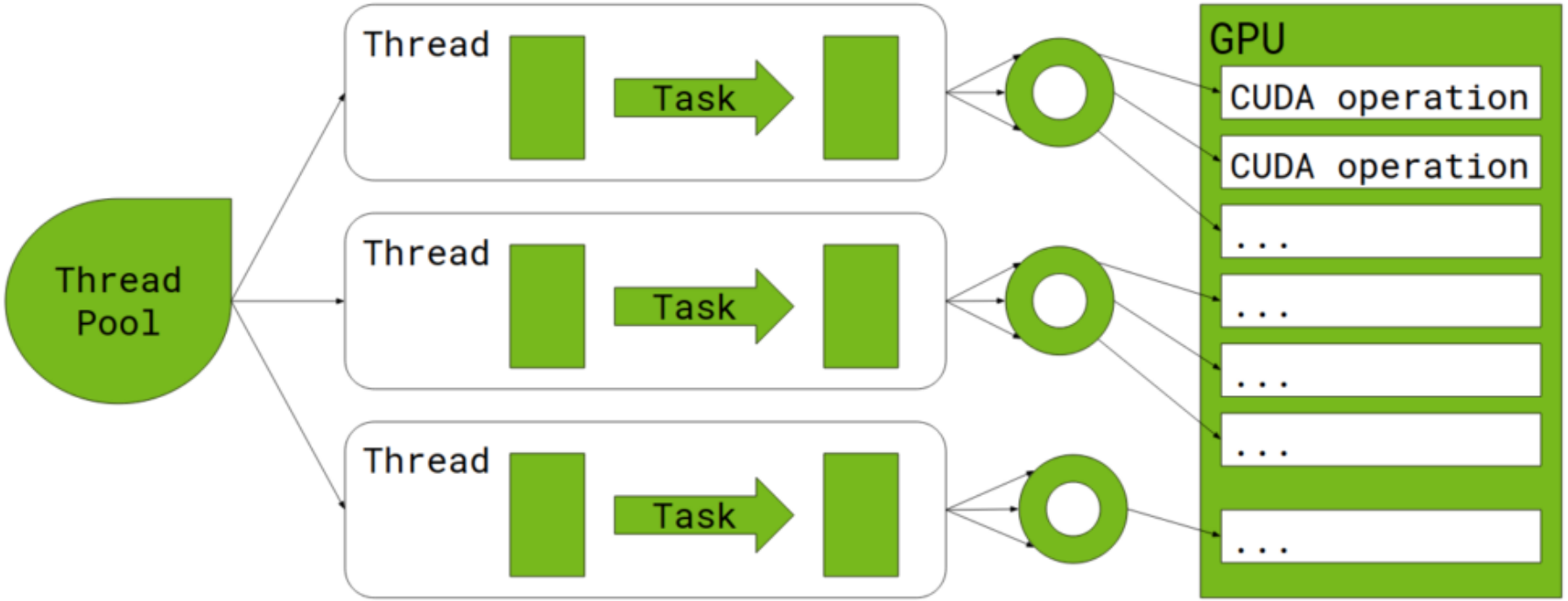
线程应用场景

典型应用场景2：Web Server



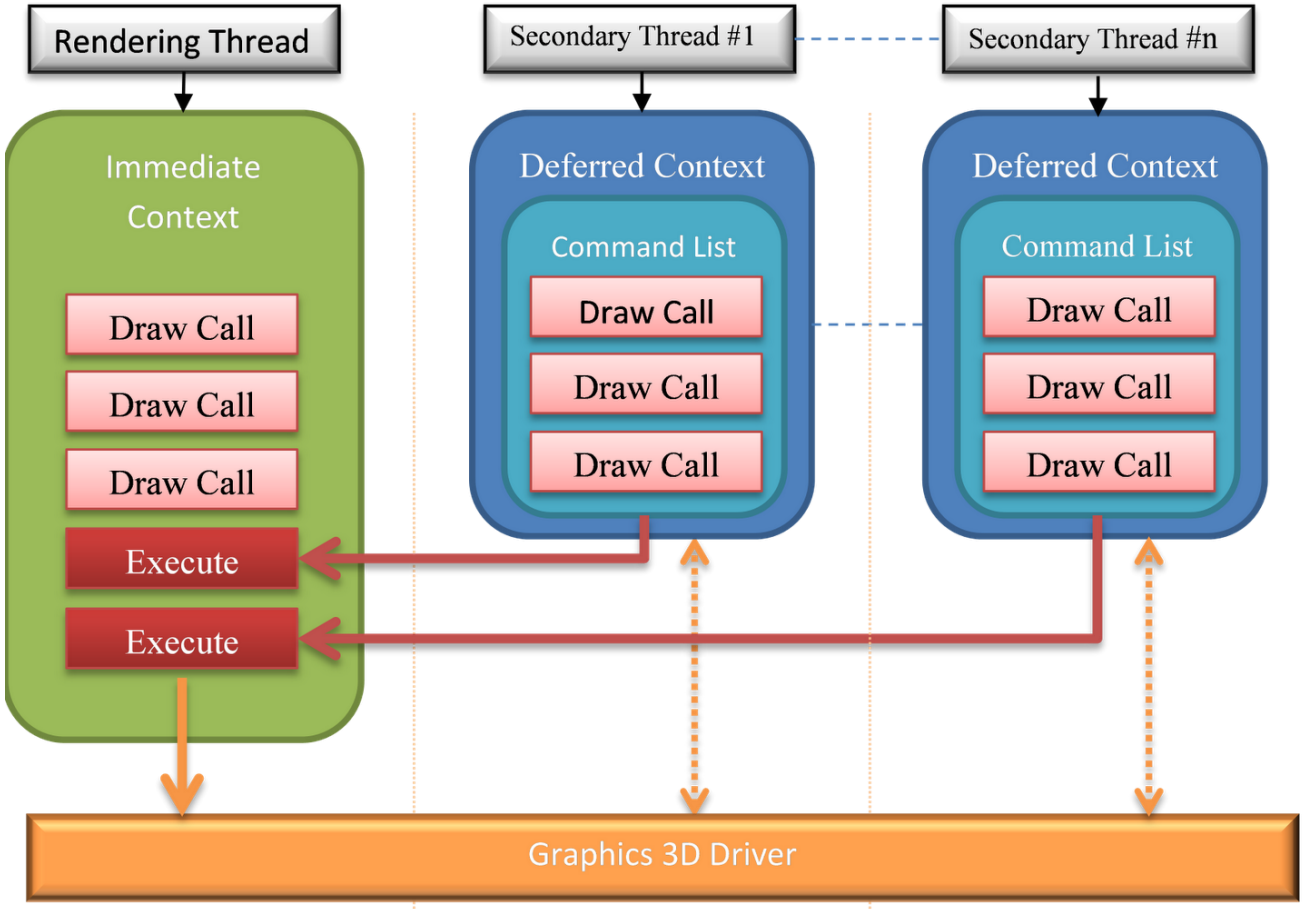
线程应用场景

典型应用场景3：Big data (data transformation)



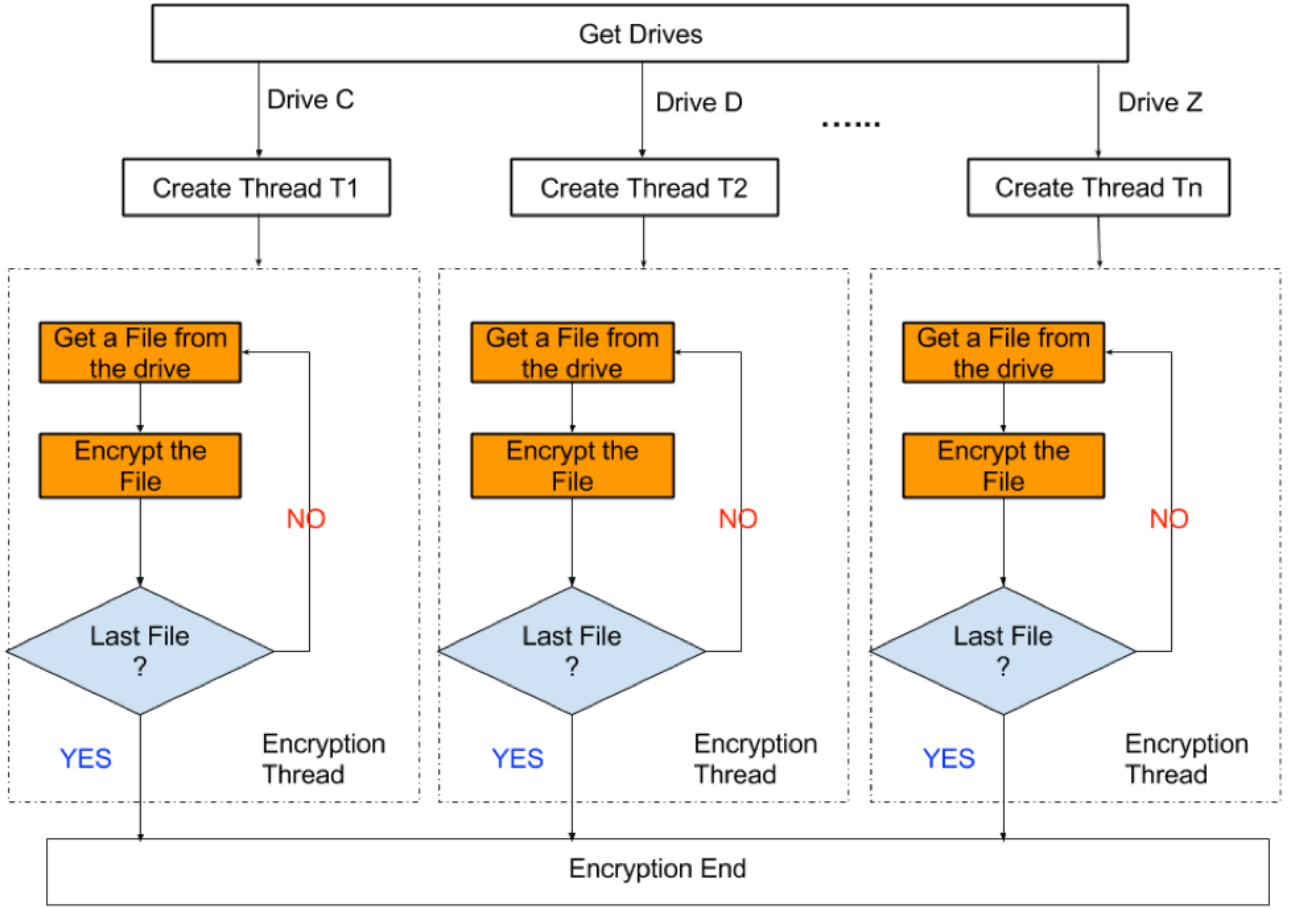
线程应用场景

典型应用场景4：Graphics Rendering



线程应用场景

典型应用场景5：Disk Encryption



谢 谢!
Thank you!