政大資科系

作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

Threads and Concurrency

Chun-Feng Liao

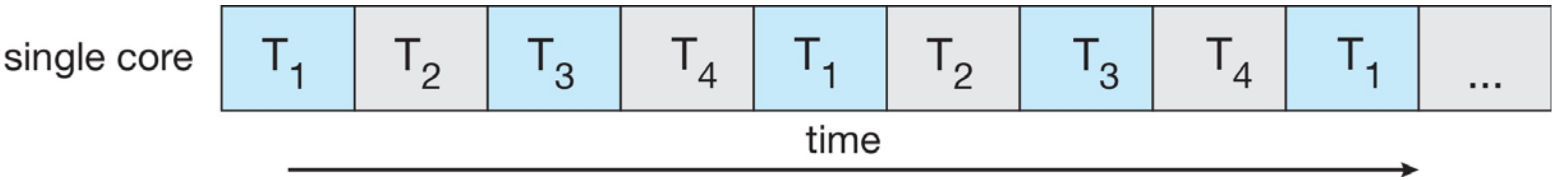
廖峻鋒

Department of Computer Science

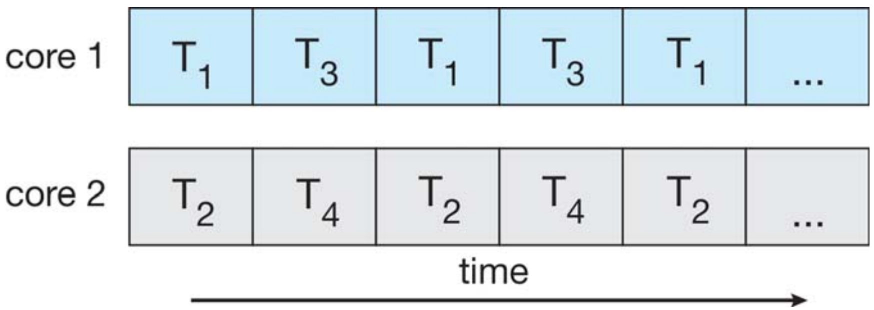
National Chengchi University

Concurrency and Parallelism

- Concurrency : more than one task making progress
  - Single processor / core, scheduler providing concurrency
  - It is possible to realize concurrency without parallelism

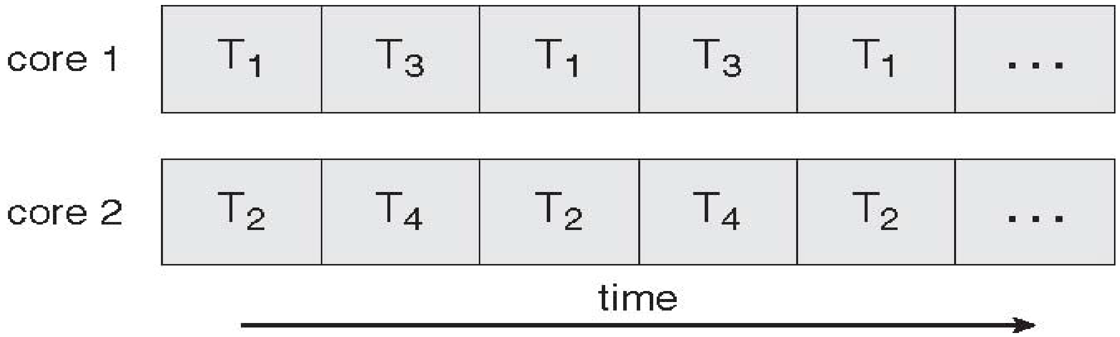


- Parallelism : perform more than one task simultaneously



Multicore Programming

- Multithreaded programming provides efficient parallelism and concurrency using multicore
  - Threads can run in parallel (each on a different core)
  - One core can "run" many (unfinished) threads at a time
- Challenges for…
  - OS designers: design and select scheduling algorithms
  - Application developers: see next few slides…



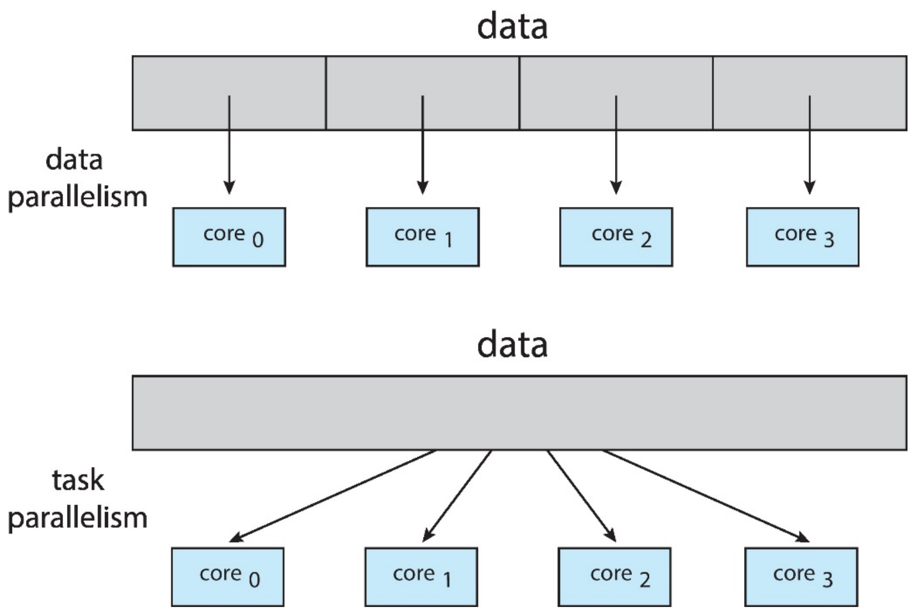
p.163-164

Multicore AP Programming

- Challenges
  - Dividing activities and data
    - Logic and data dependency
    - 找出可平行化的task與data
  - Balance
    - 平均每個core的工作
  - Testing and debugging
    - 平行: 所有threads的動作都會interleave，產生多種排列組合
    - 變得更難測試

Multicore AP Programming

- Data Parallelism (map)
  - Perform the same task on different data
- Task Parallelism
  - Perform different tasks on the same data



Data Parallelism Example

- 使用Map做data parallelism

digits [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

對每個元素加1

newDigits [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

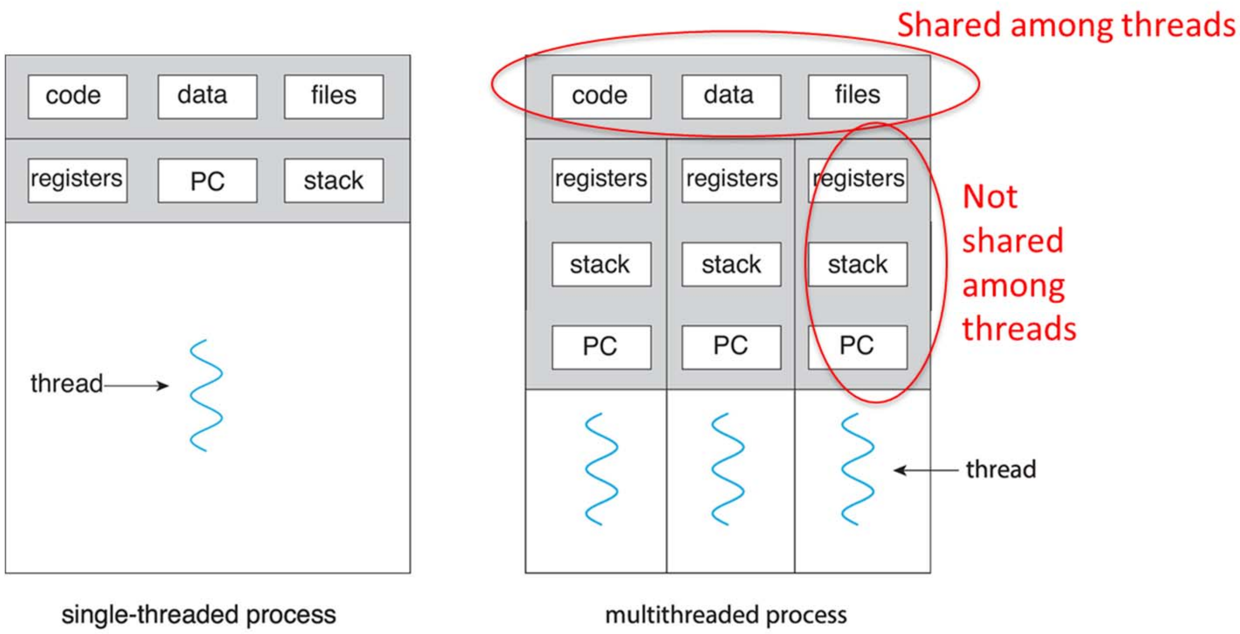
```
let digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
let newDigits =
    digits.map ( function(element) {
        return element+1;
    });
```

Perform the same task on different data

就這個例子來説，那一個elements先被加一不影響最終結果➔可平行化

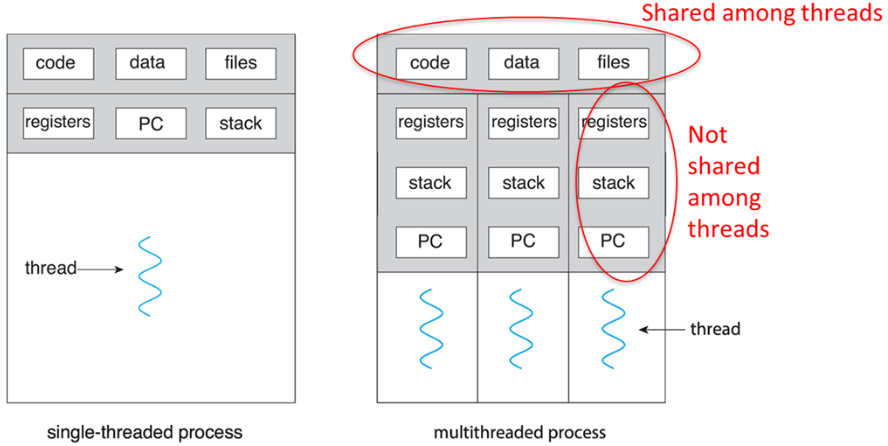
Introduction

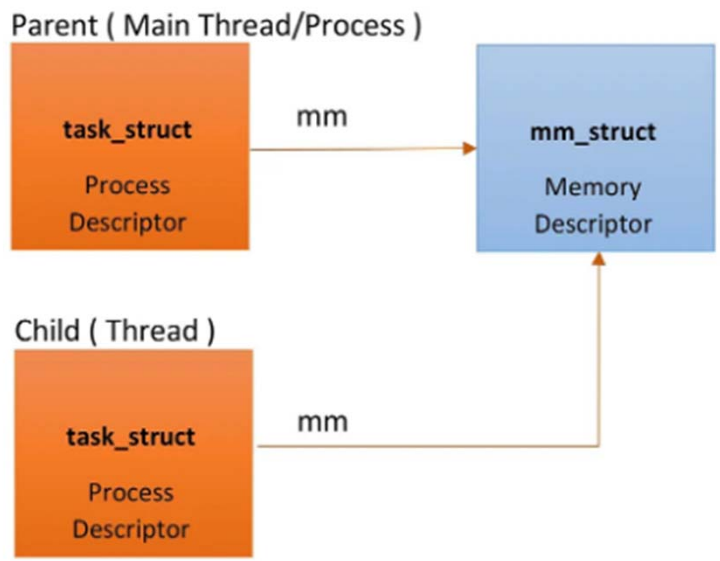
- Thread
  - basic unit of CPU utilization
- Many threads belonging to the same process
  - Share
    - Code, data (static/global variables), and OS resources (e.g. open files and signals)
  - Not share
    - thread ID, program counter, register set, and a stack



Case: Linux Kernel Thread

- Code, Data, Files放在mm_struct
- Registers, PC, stack放在 PCB (task_struct)
  - 每個thread有一個task_struct，但指向同一個mm_Struct





Why Thread?

- Lower creation cost vs. Process

platform	fork()	pthread_create()	speedup
AMD 2.4 GHz Opteron	17.6	1.4	15.6x
IBM 1.5 GHz POWER4	104.5	2.1	49.8x
INTEL 2.4 GHz Xeon	54.9	1.6	34.3x
INTEL 1.4 GHz Itanium2	54.5	2.0	27.3x

- Shared Mem (Thread) vs. Message Passing (IPC) 資料傳輸量

platform	Shared Memory BW (GB/sec)	Message Passing BW (GB/sec)	speedup
AMD 2.4 GHz Opteron	1.2	5.3	4.4x
IBM 1.5 GHz POWER4	2.1	4	1.9x
INTEL 2.4 GHz Xeon	0.3	4.3	14.3x
INTEL 1.4 GHz Itanium2	1.8	6.4	3.6x

什麼時候適合多執行緒

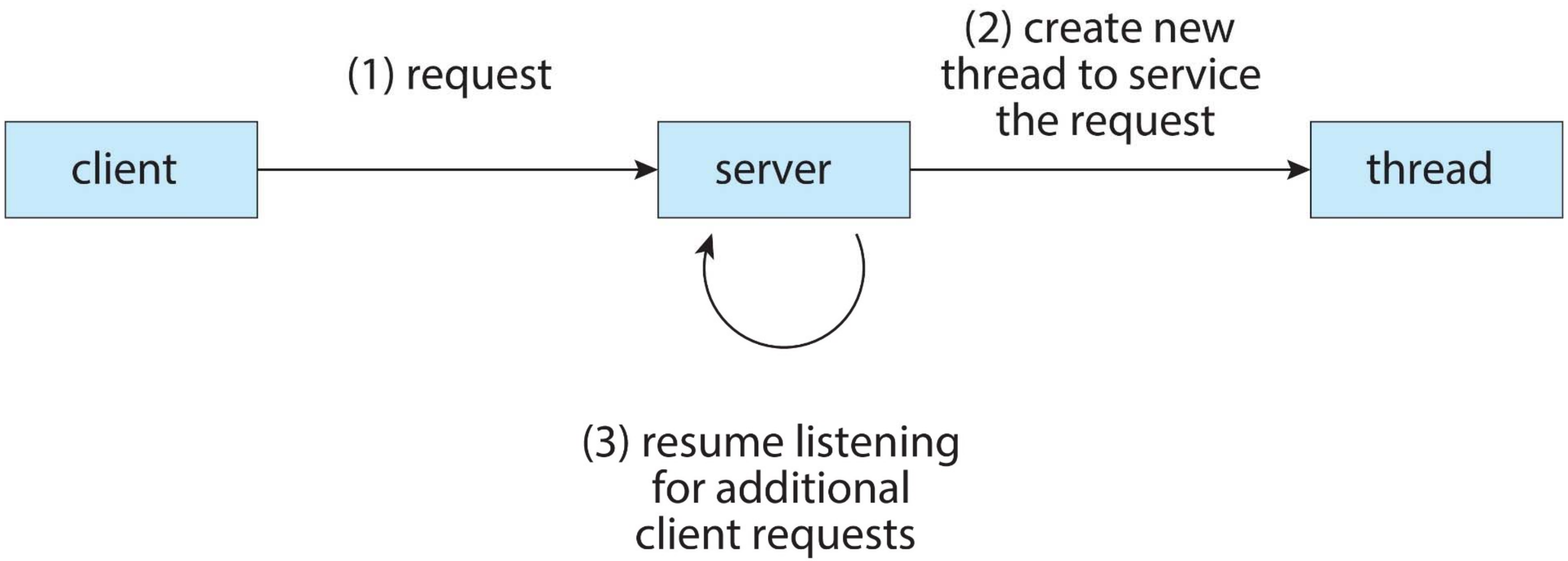
- 為一大堆照片建立縮圖(thumbnails)
  - 為每個照片建立縮圖彼此為獨立工作
- 文書編輯器
  - 一個thread負責顯示畫面
  - 一個負責互動裝置回應(KB/Mouse)
  - 一個負責內文文法檢查

Benefits of Threading

- Responsiveness
  - Allow a program to continue running even if part of it is blocked or is performing a lengthy operation
- Resource sharing
  - 共享data, code與file區段
- Scalability
  - 適用於multi-processor/core架構
- Economy: Allocating memory and resources for process creation is costly
  - Generally, context switching is faster between threads than processes (需要switch的東西比較少)

Example

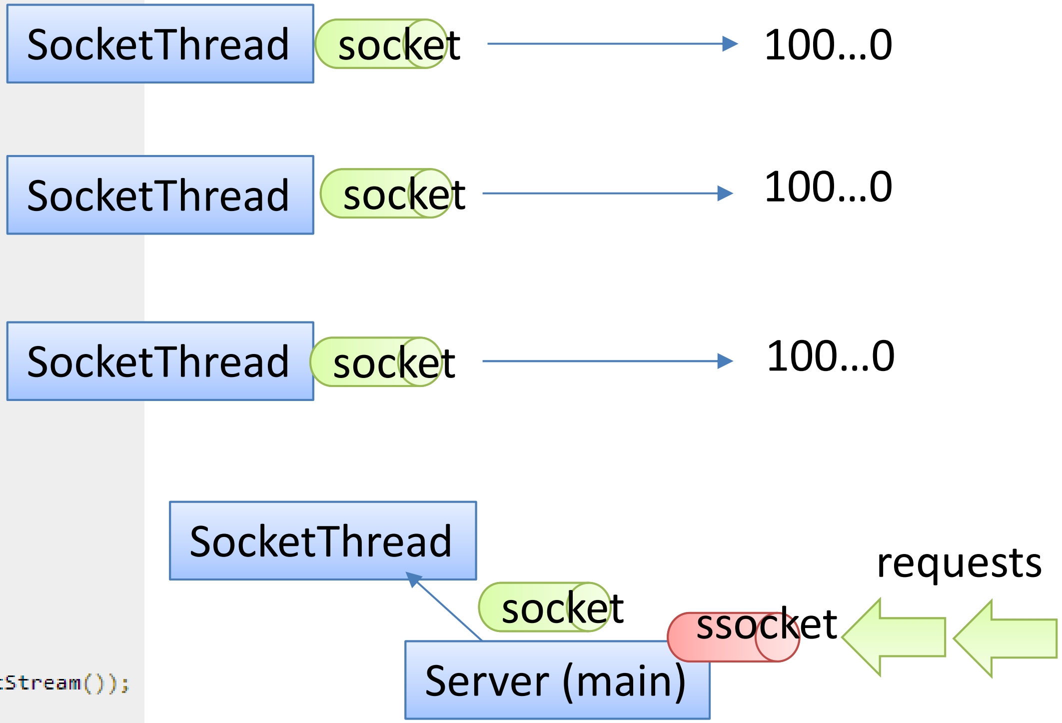
- Example: a network server providing services
  - One request / process: poor performance
  - One request / thread: better performance (less creation time, code and resource sharing)



Request processing的步驟

Multi-threaded Java Server

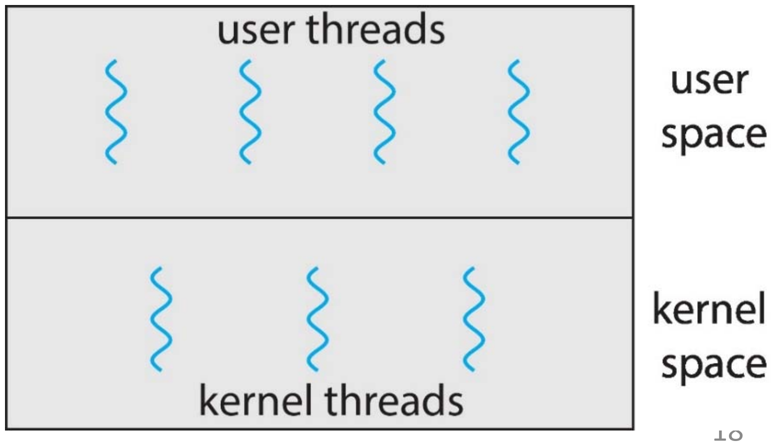
```java
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class MultiThreadServer implements Runnable {
   Socket csocket;
   MultiThreadServer(Socket csocket) {
      this.csocket = csocket;
   }
   public static void main(String args[]) throws Exception {
      ServerSocket ssock = new ServerSocket(1234);
      System.out.println("Listening");

      while (true) {
         Socket sock = ssock.accept();
         System.out.println("Connected");
         new Thread(new MultiThreadServer(sock)).start();
      }
   }
   public void run() {
      try {
         PrintStream pstream = new PrintStream(csocket.getOutputStream());
         for (int i = 100; i >= 0; i--) {
            pstream.println(i + " bottles of beer on the wall");
         }
         pstream.close();
         csocket.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```



p.166, Sec. 4.3

Thread Model

- User threads – thread management done by user- level threads library
  - POSIX Pthreads
  - Java threads
- Kernel threads – supported by the kernel (OS) directly
  - Windows (Win32 thread library)
  - Linux Kernel >2.6 (POSIX Pthreads)
  - Mac OS X
  - iOS
  - Android



POSIX Pthreads 同時支援user和kernel threads

User vs. Kernel Threads

- User threads
  - Thread library provides support for thread creation, scheduling, and deletion (explicit threading)
  - Generally fast to create and manage
  - If the kernel is single-threaded, a user-thread blocks → entire process blocks even if other threads are ready to run
- Kernel threads
  - The kernel performs thread creation, scheduling, etc.
  - Generally slower to create and manage
  - If a thread is blocked, the kernel can schedule another thread for execution

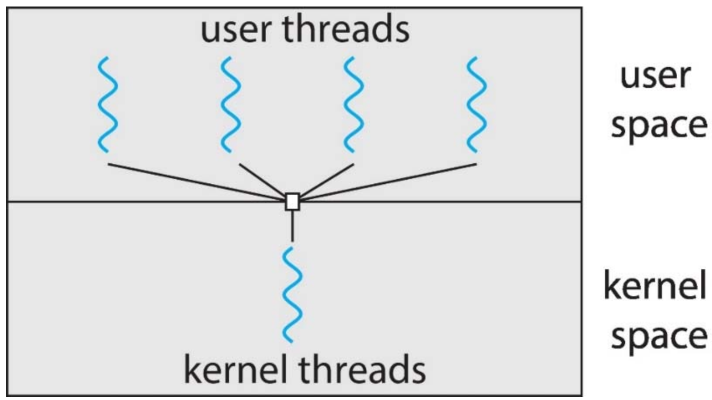
Multithreading Models

P.166, sec. 4.3

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
  - Used on systems that do not support kernel threads
  - Efficient: Thread management is done in user space
- Drawbacks
  - One user thread blocking causes all to block
  - 無法發揮multi-core效益: kernel thread同一時刻只能和一個core互動
- Examples:
  - Solaris Green Threads
  - Few systems currently use this model!
  - Initial Java, Python GIL, Node.js 12.13.0之前



Python without GIL

python SOFTWARE FOUNDATION

News from the Python Software Foundation

Wednesday, May 11, 2022

The 2022 Python Language Summit: Python without the GIL

If you peruse the archives of language-summit blogs, you'll find that one theme comes up again and again: the dream of Python without the GIL. Continuing this venerable tradition, Sam Gross kicked off the 2022 Language Summit by giving the attendees an update on nogil, a project that took the Python community by storm when it was first announced in October 2021.

Python without GIL

Python moves to remove the GIL and boost concurrency

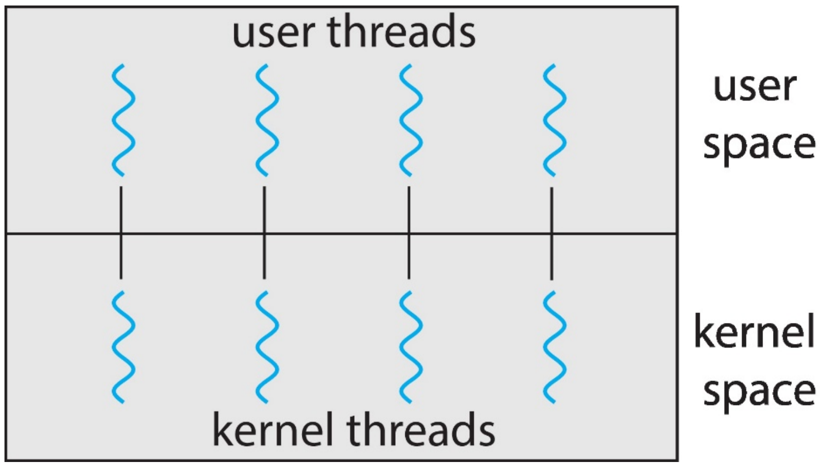
Formal plans for a Python that supports true parallelism are finally on the table. Here's how a GIL-free Python will finally come together.

By Serdar Yegulalp

Senior Writer, InfoWorld | AUG 4, 2023 1:00 PM PDT

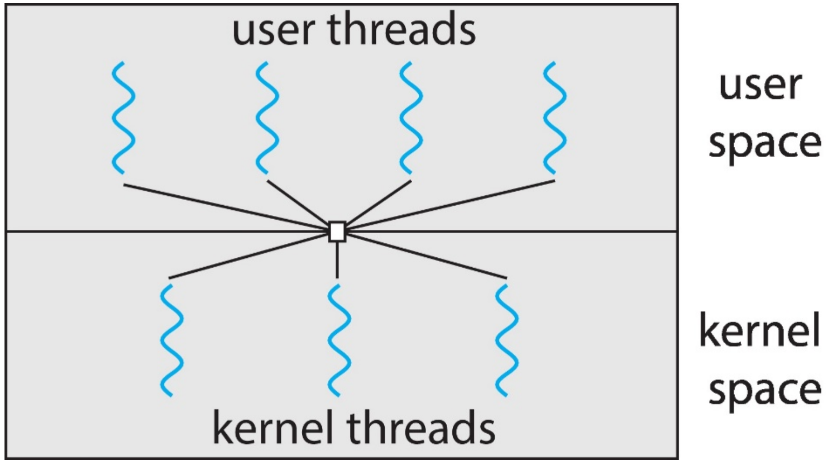
One-to-One

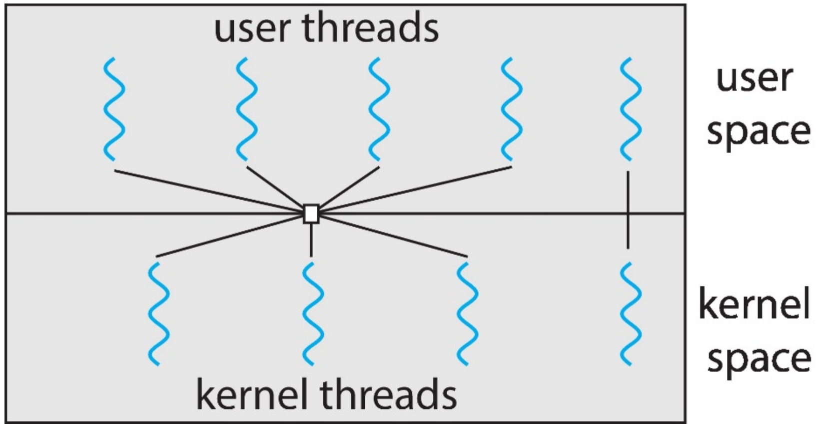
- Each user-level thread maps to kernel thread
  - More concurrency than many-to-one
  - Switch to another thread when current thread blocks
- Drawback
  - Slower creation: creating a user-level thread creates a kernel thread (可以用pooling技巧解決)
  - Performance: large number of kernel threads may burden the performance of a system
- Examples
  - Windows XP/NT/2000
  - Linux
  - Solaris 9



Many-to-Many Model

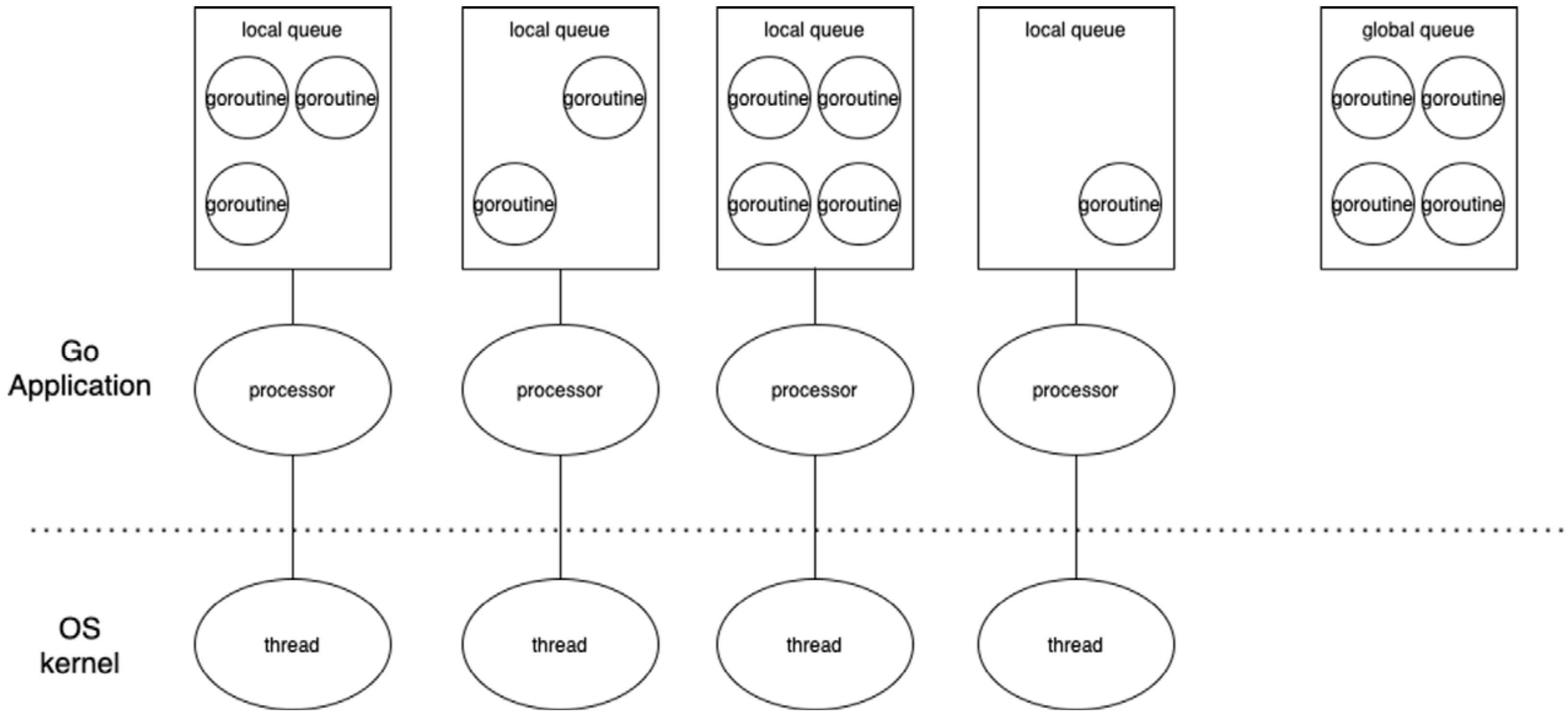
- Allows many user level threads to be mapped to many kernel threads
  - Allows the OS to determine the number of kernel threads
  - When a thread blocks, another thread can run
- Drawback
  - Difficult to implement
- The number of cores of CPU is increasing → M to M的必要性降低
- M to M is used by Implicit Threading
  - Java Executor
  - Node.js 12.13.0之後 (with Worker Thread)
  - Goroutine



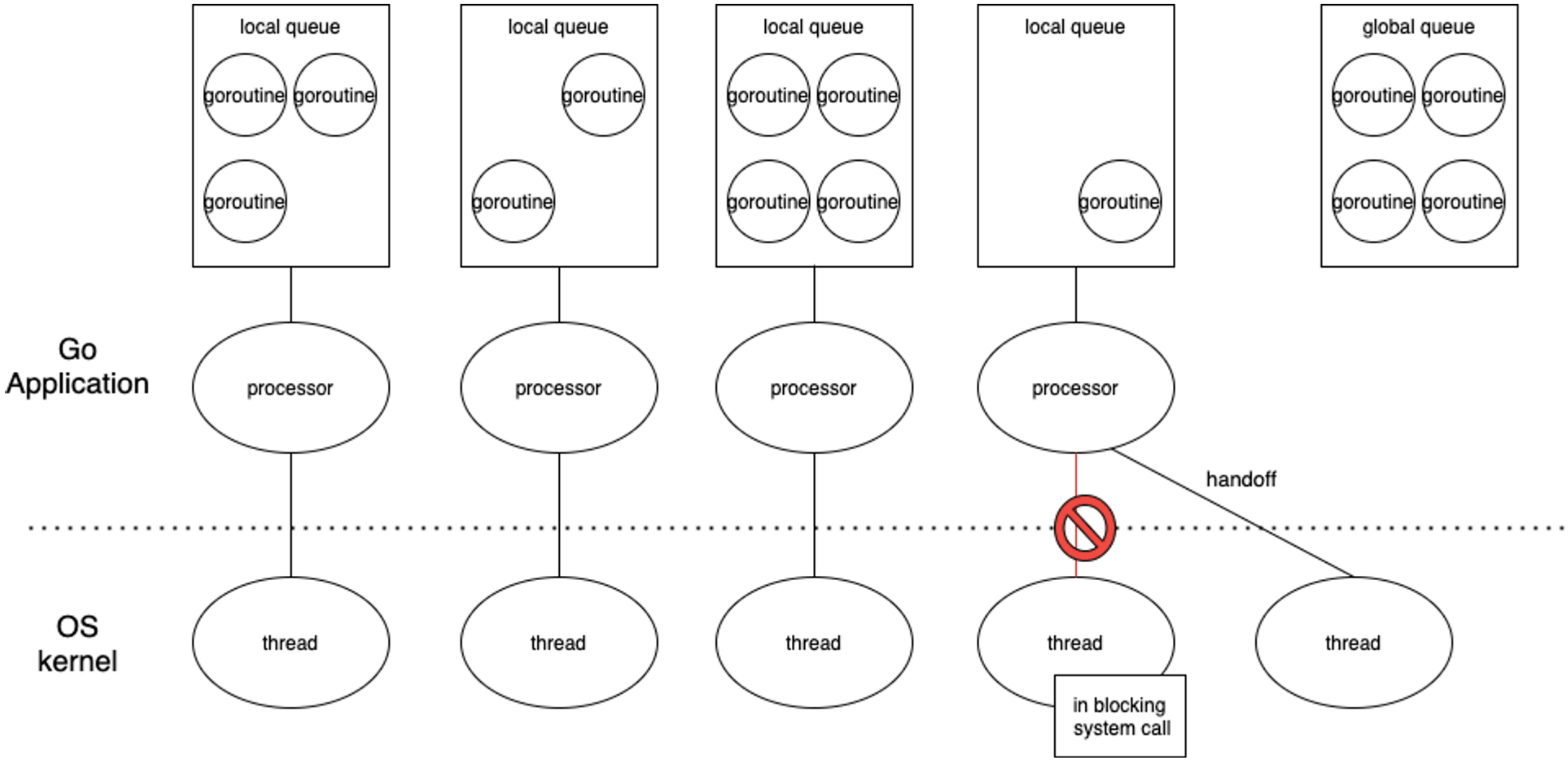


2-level model: also permits one-to-one

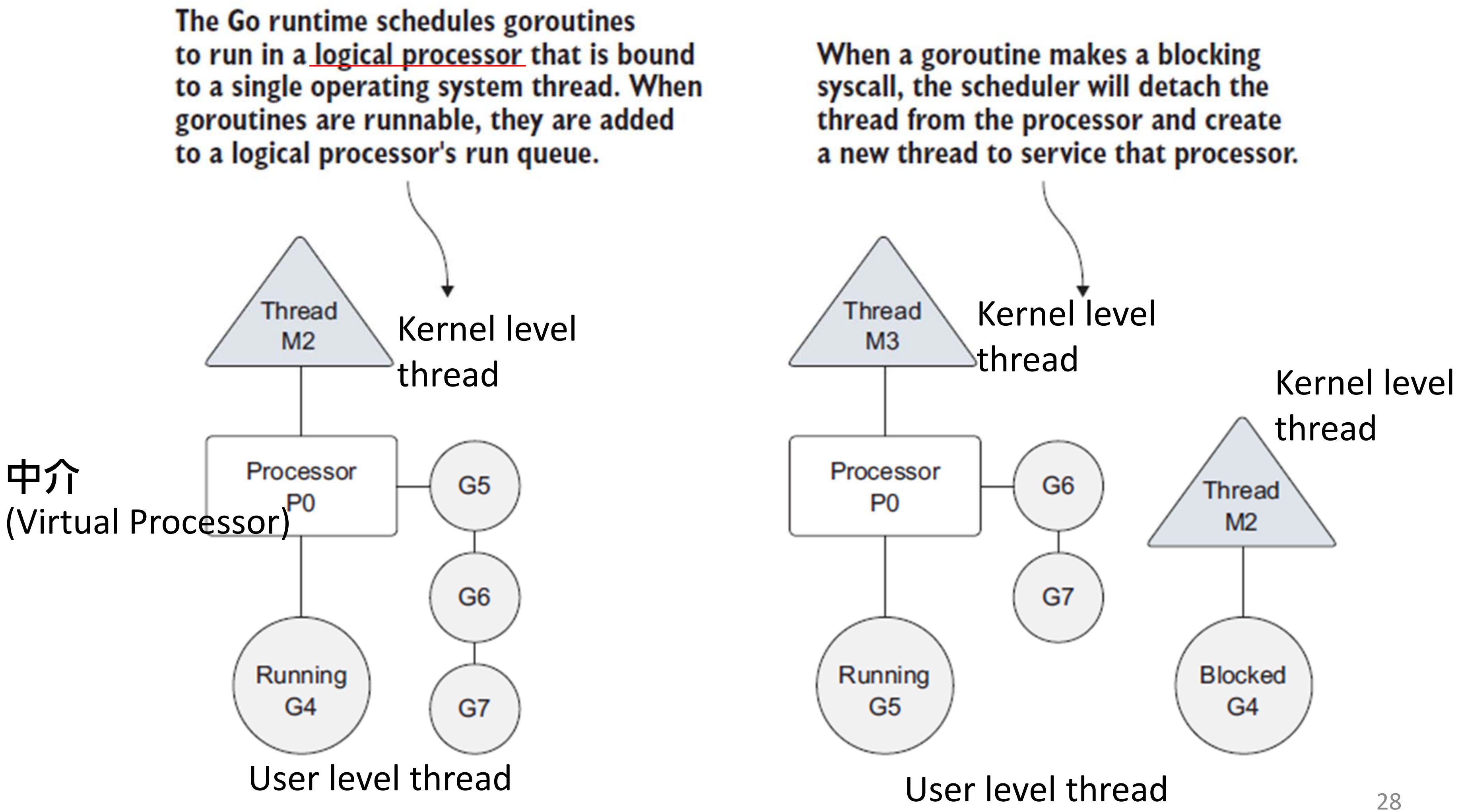
Goroutine


local queue
goroutine
goroutine
goroutine
local queue
goroutine
goroutine
local queue
goroutine
goroutine
goroutine
goroutine
local queue
goroutine
global queue
goroutine
goroutine
goroutine
goroutine
Go
Application
processor
processor
processor
processor
OS
kernel
thread
thread
thread
thread

Goroutine


local queue
goroutine
goroutine
goroutine
local queue
goroutine
goroutine
local queue
goroutine
goroutine
goroutine
goroutine
local queue
goroutine
global queue
goroutine
goroutine
goroutine
goroutine
Go
Application
processor
processor
processor
processor
handoff
OS
kernel
thread
thread
thread
thread
thread
in blocking
system call

Goroutine


The Go runtime schedules goroutines to run in a logical processor that is bound to a single operating system thread. When goroutines are runnable, they are added to a logical processor's run queue.
Thread M2
Kernel level thread
中介
(Virtual Processor)
Processor P0
G5
G6
G7
Running G4
User level thread
When a goroutine makes a blocking syscall, the scheduler will detach the thread from the processor and create a new thread to service that processor.
Thread M3
Kernel level thread
Processor P0
G6
G7
Running G5
Kernel level thread
Thread M2
Blocked G4
User level thread

P.168, sec. 4.4

Thread Libraries

- Pthreads
  - A POSIX standard (IEEE 1003.1c) API
  - Common in UNIX operating systems (Linux & Mac OS X)
- Windows Threads
  - Similar to POSIX Pthreads
- Java Threads
  - Managed by the JVM
  - Implemented using the threads model provided by underlying OS

Pthreads

- pthread_create(thread, attr, runner, arg)
  - thread: An unique identifier (token) for the new thread
  - attr: It is used to set thread attributes. NULL for the default values
  - runner: The routine that the thread will execute once it is created
  - arg: A single argument that may be passed to routine

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

Sum是全域變數

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

atoi→將字串轉為數字

g++ -fpermissive –pthread thrd-posix.c –o thrd-posix

Win32 Threads

Unsigned 32-bit integer

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Sum是全域變數

從1加到幾?

什麼是*(DWORD*) → 轉型後取值

https://stackoverflow.com/questions/20219188/whats-the-difference-between-dword-dword-and-dword

Win32 Threads

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Sum是全域變數

為什麼結果要用全域變數來表示? 因為thread很難有回傳值(main不知thread何時結束)

Java Threads

- Thread is created by
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

  - Standard practice is to implement Runnable interface
    - Why not extends Thread?

Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Supporting lib/frameworks
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages
  - Usually slightly faster to service a request with an existing thread than create a new thread (standby in the pool)
  - Allows the number of threads in the application(s) to be bound to the size of the pool (to prevent funnel effects)
- # of threads in a pool
  - # of CPUs, expected # of requests, amount of physical memory

Java Thread Pools

- Three factory methods for creating thread pools in Executors class:
  - `static ExecutorService newSingleThreadExecutor()`
  - `static ExecutorService newFixedThreadPool(int size)`
  - `static ExecutorService newCachedThreadPool()`

creates new threads as needed

Java Thread Pools

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
public static void main(String[] args) {
   int numTasks = Integer.parseInt(args[0].trim());

   /* Create the thread pool */
   ExecutorService pool = Executors.newCachedThreadPool();

   /* Run each task using a thread in the pool */
   for (int i = 0; i < numTasks; i++)
      pool.execute(new Task());

   /* Shut down the pool once all threads have completed */
   pool.shutdown();
}
```

觀察:
1. task比thread多，那些 tasks分配到那些thread是由JVM來控制
2. 從這裡也可以看出M to M thread model 的樣態: Task是user level thread; 印出的tid是kernel level thread
3. 在VM的linux中執行明顯地thread數少很多

Callable: 有回傳值的thread

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```

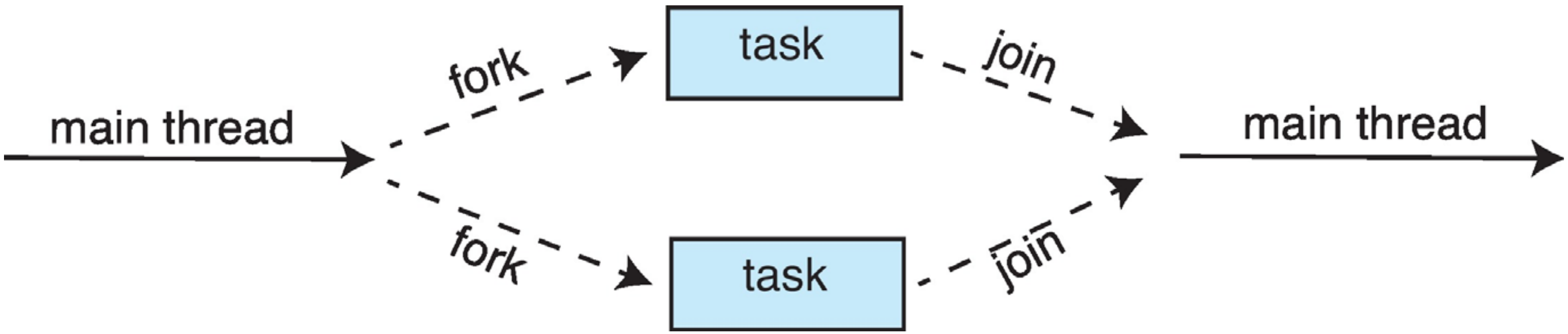
Callable: 有回傳值的thread

```
public class Driver
{
  public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

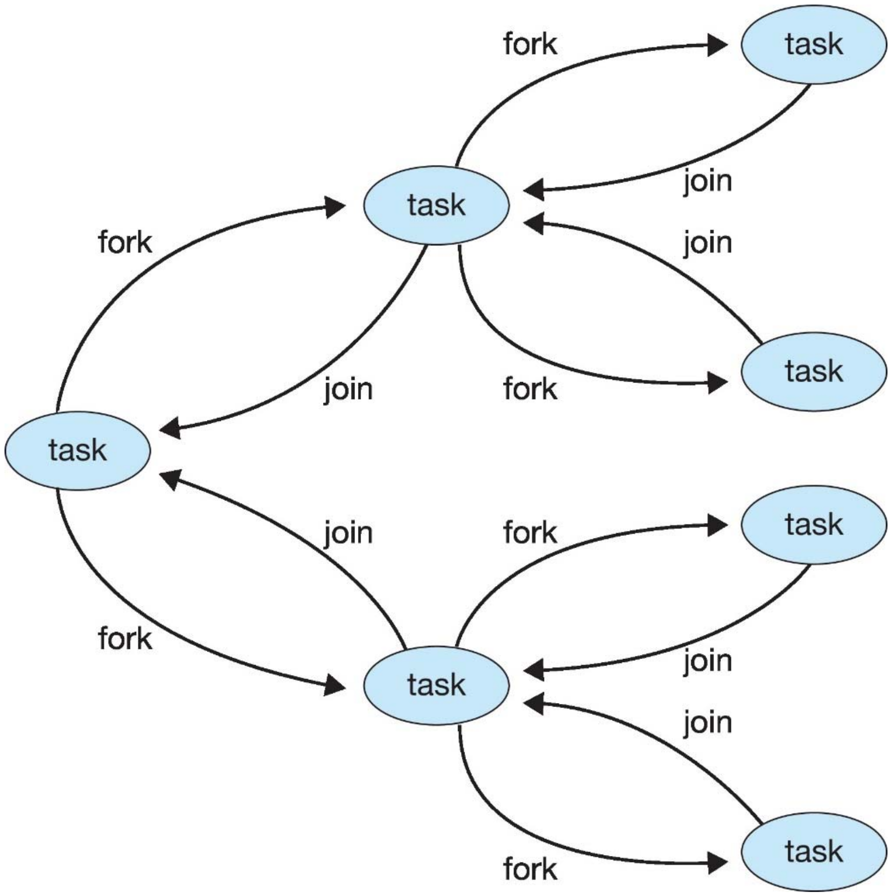
Wait until completion

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



Fork-Join Parallelism



Fork-Join Parallelism

- General algorithm for fork-join strategy:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Fork-Join Parallelism in Java

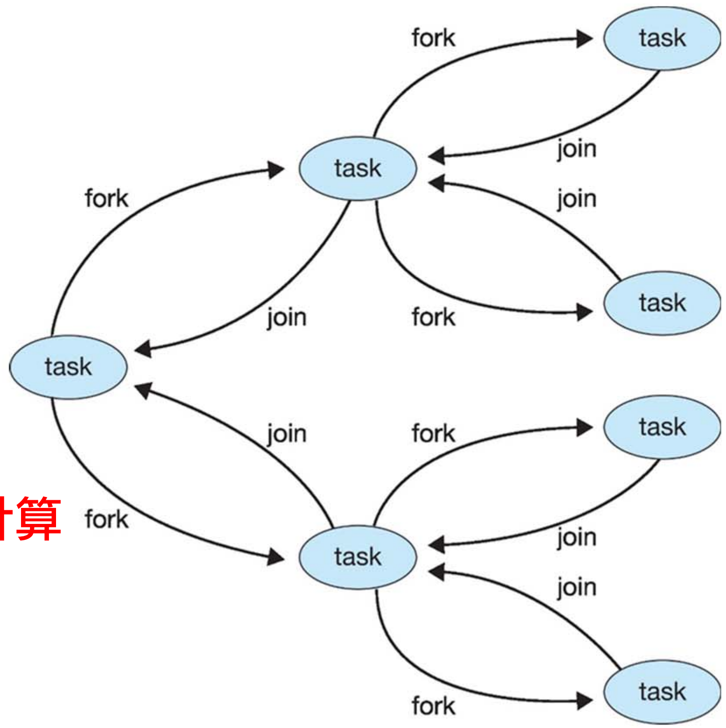
```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

必須是ForJoinTask的子類

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
  static final int THRESHOLD = 1000;

  private int begin;
  private int end;
  private int[] array;

  public SumTask(int begin, int end, int[] array) {
    this.begin = begin;
    this.end = end;
    this.array = array;
  }

  protected Integer compute() {
    if (end - begin < THRESHOLD) {
      int sum = 0;
      for (int i = begin; i <= end; i++)
        sum += array[i];
      return sum;
    }
    else {
      int mid = (begin + end) / 2;

      SumTask leftTask = new SumTask(begin, mid, array);
      SumTask rightTask = new SumTask(mid + 1, end, array);

      leftTask.fork();
      rightTask.fork();

      return rightTask.join() + leftTask.join();
    }
  }
}
```

工作分割夠細了，開始計算

繼續工作分割



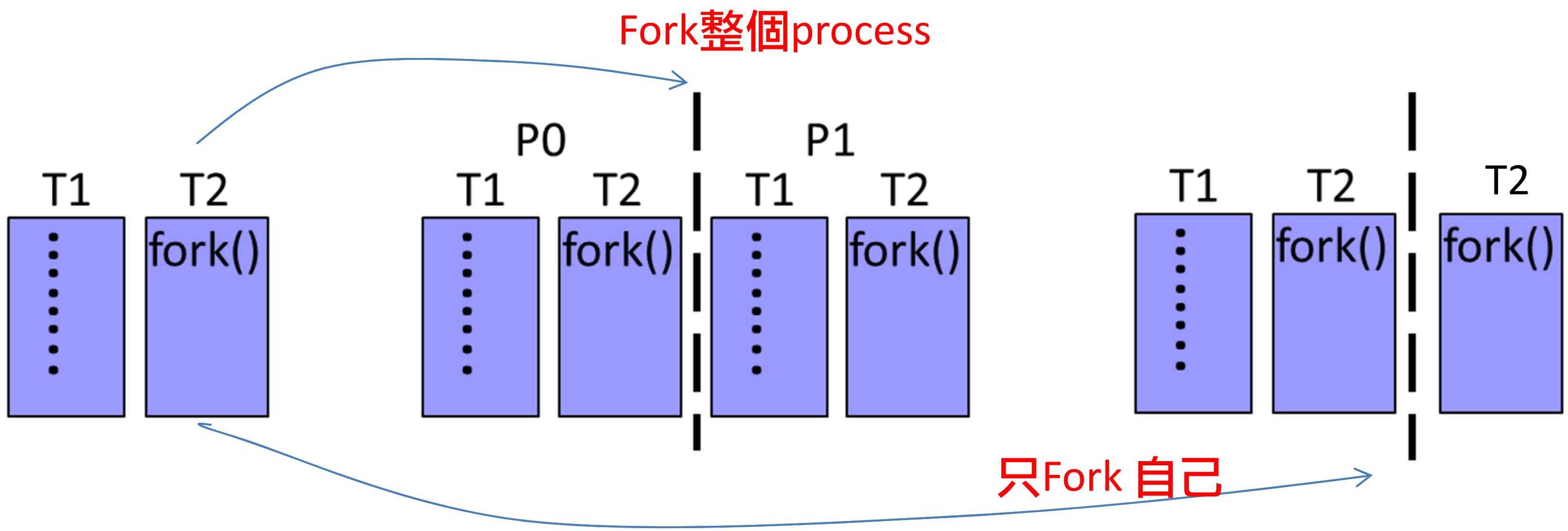
P.188, sec. 4.6

Threading Issues

- Semantics of fork() and exec() system calls
- Signal handling
  - Synchronous and asynchronous
- Cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage

Semantics of fork() and exec()

- Does **fork() in a thread** duplicate only the calling thread or process?
  - Some UNIX system support two versions of fork()
- **exec()** works the same: replace the entire process
  - If exec() is called immediately after forking, then duplicating all threads is unnecessary (all threads will be destroyed after the call)



Signal Handling

- Signals (synchronous or asynchronous) are used in UNIX systems to notify a process that an event has occurred
  - Synchronous: code 執行造成的錯誤, sent to the causing process
    - 造成的當下就送出
    - Ex: illegal memory access, 1/0 (div by zero)
  - Asynchronous: generated external to the process
    - Ex: <control-C>, timeout
- A **signal handler** is used to process signals
  - Default or user-defined

Signal Handling

- Multi-threaded signal delivery
  - Deliver the signal to the thread to which the signal applies
    - Synchronous signals
  - Deliver the signal to every thread in the process
    - Key-pressing
  - Deliver the signal to certain threads in the process
    - pthread_kill
    - Windows APC (Asynchronous procedure calls)
  - Assign a specific thread to receive all signals for the process
- Standard function to send signals

```
kill(pid_t pid, int signal) // to process
pthread_kill(pthread t_tid, int signal) // to thread
```

Thread Cancellation

- What happen if a thread is terminated before completion?
  - E.g, terminate web page loading
- Approaches:
  - Target thread: the thread that is to be cancelled
  - Asynchronous cancellation
    - One thread terminates the target thread immediately
    - May not free a system-wide resource
  - Deferred cancellation (default)
    - The target thread periodically checks whether it should be terminated, allowing it an opportunity to terminate itself in an orderly fashion (canceled safely)
    - Check at cancellation points
      - pthread_testcancel()
      - Java interrupt

Thread Cancellation

- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

- Pthread cancellation
  - Only <u>requests</u> cancellation
  - Actual cancellation depends on the <u>mode</u> of target thread

Thread Cancellation (Cont.)

- Cancellation modes:

是否允許取消?

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

不允許取消

Default (中斷點取消)

立即取消

Not recommended in Pthread

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is “deferred”
  - Cancellation only occurs when thread reaches **cancellation point**

```
while (1) {
   /* do some work for awhile */

    . . .

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

Thread terminates when there is a cancel signal

Thread Cancellation in Java

- Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

Try to cancel the thread

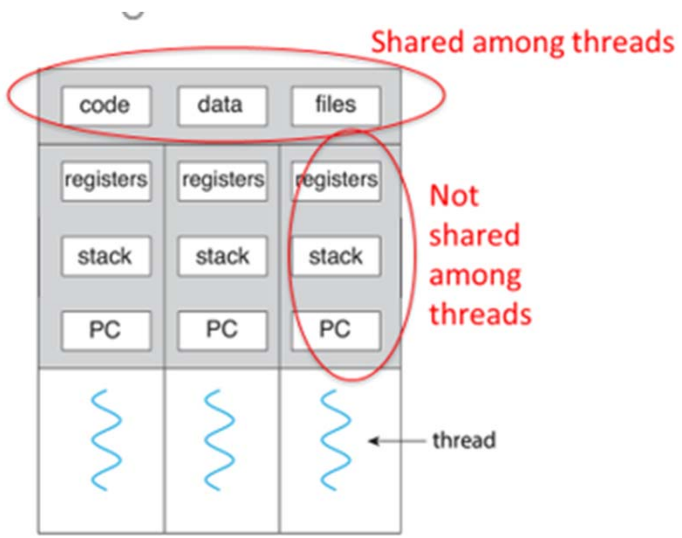
A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

Handle the cancellation

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
  - 用途: 想要同一個thread中的多個函式，共享一份變數
  - 在multithread環境下global/static variable是所有threads共享
- Comparison
  - Local variable
    - Visible only during single function invocation
  - TLS
    - Similar to static variables, but unique to each thread
    - visible across function invocations



課後閱讀

- P.189 APC
- P.176 為什麼除了Thread之外，還需要Callable/Executor機制?
- P.177 JVM用在Windows上時，是採用什麼model
- P.177 Implicit Threading實作時常採用什麼Model?
- P.183 OpenMP
- P.185 Grand Central Dispatch
- P.195 Windows中的thread local storage存在ETHREAD, KTHREAD 還是TEB?

Q & A