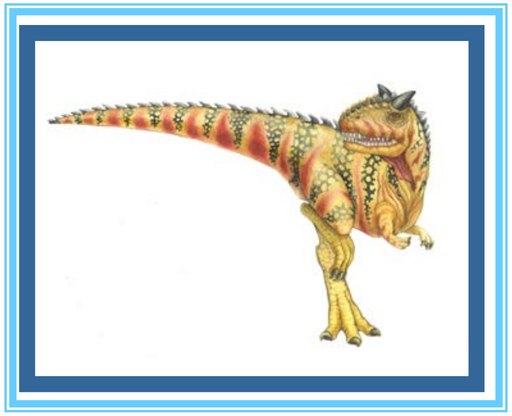
Chapter 4: Multithreaded Programming



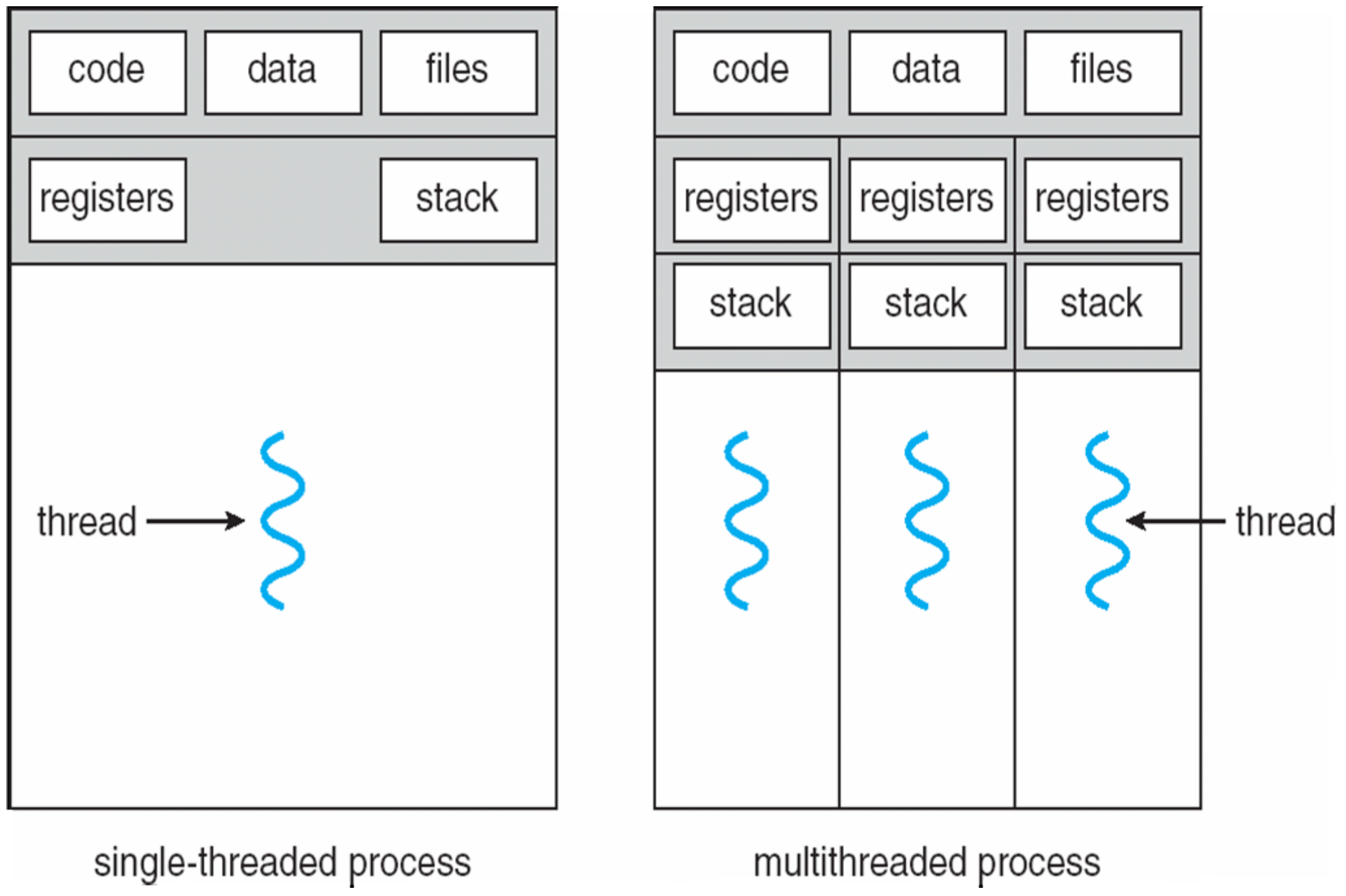
Chapter 4: Multithreaded Programming

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Single and Multithreaded Processes



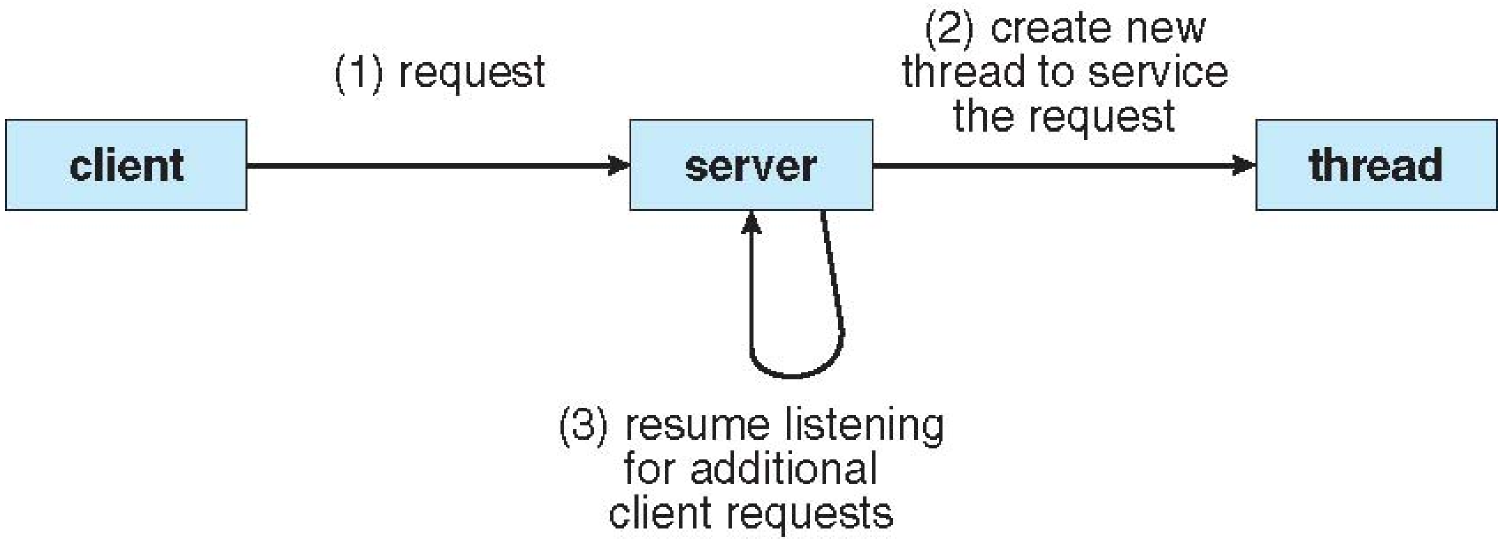
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

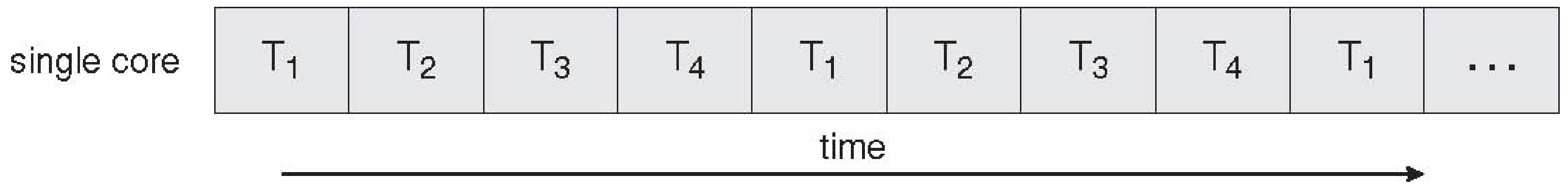
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Multithreaded Server Architecture

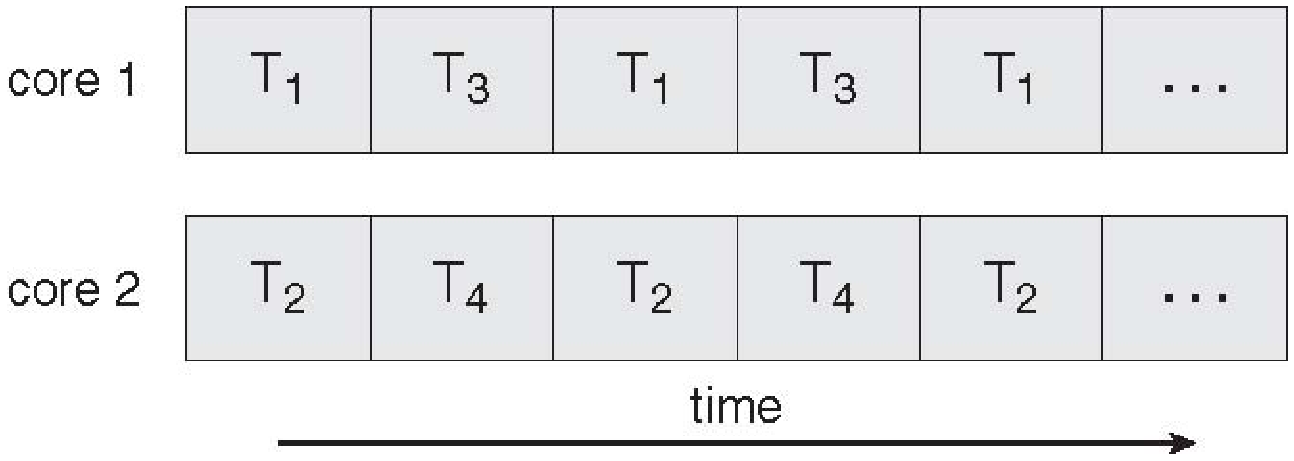

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

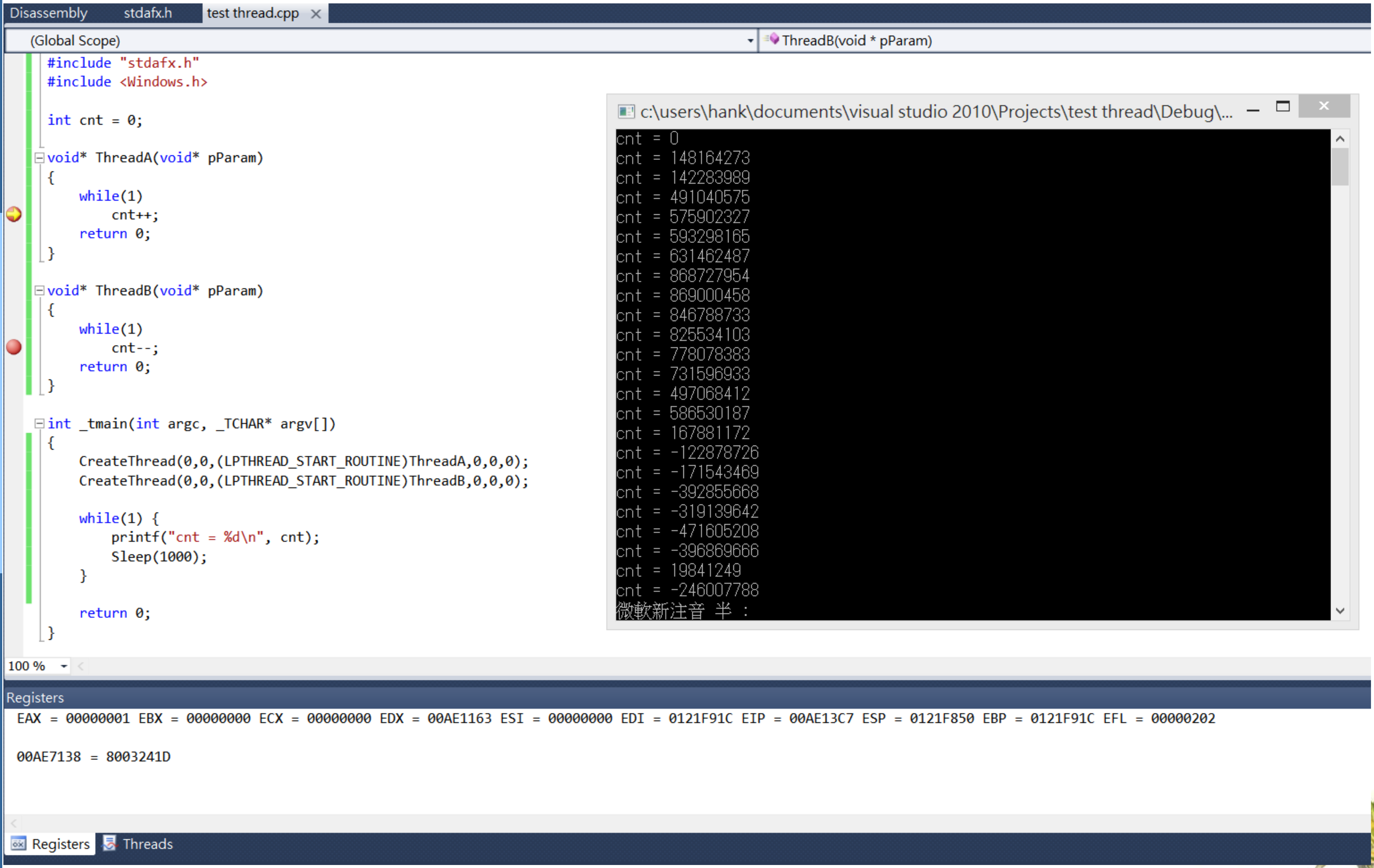
Concurrent Execution on a Single-core System



http://goo.gl/6xhZL

Parallel Execution on a Multicore System





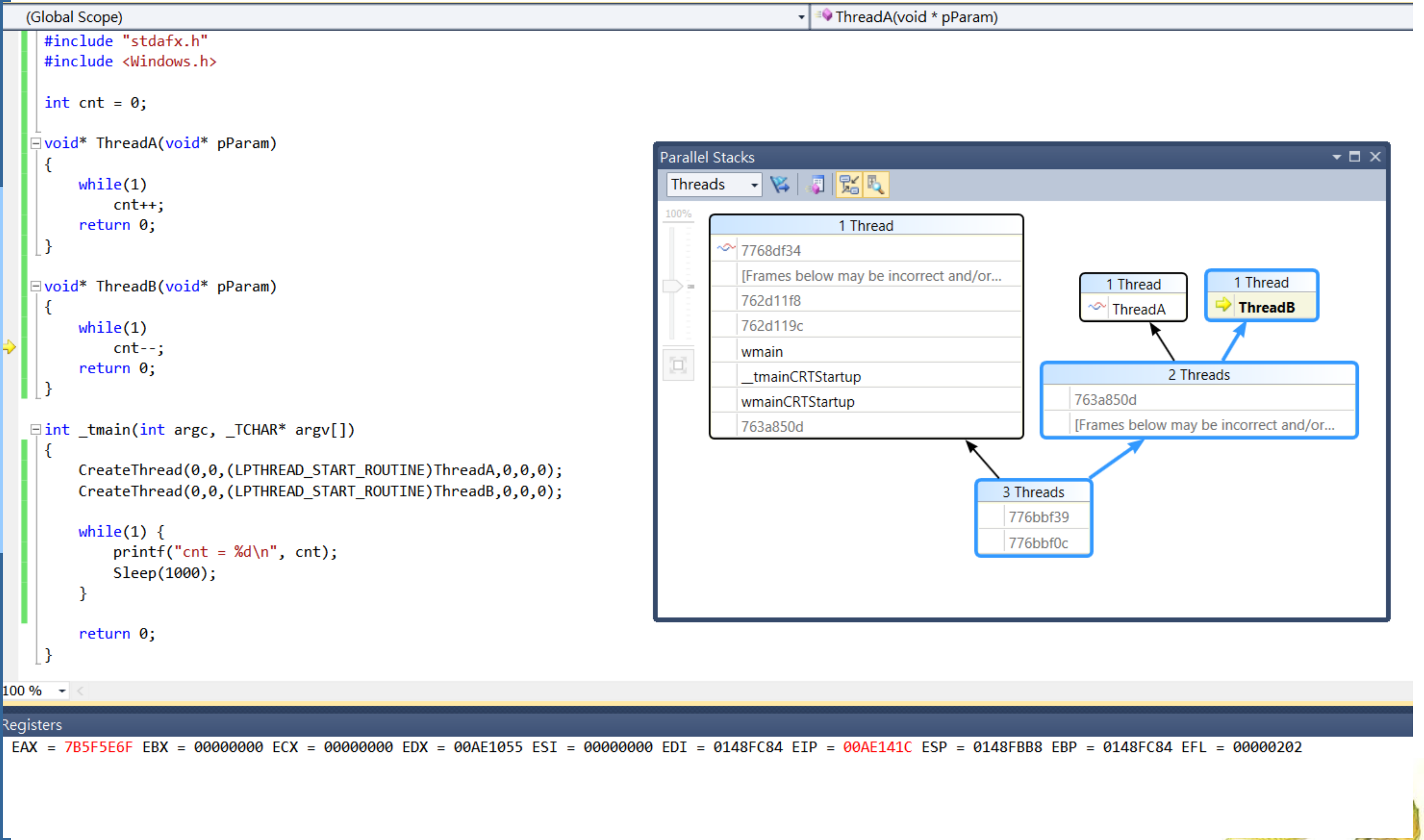
```
#include "stdafx.h"
#include <Windows.h>

int cnt = 0;

void* ThreadA(void* pParam)
{
    while(1)
        cnt++;
    return 0;
}

void* ThreadB(void* pParam)
{
    while(1)
        cnt--;
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadA,0,0,0);
    CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadB,0,0,0);

    while(1) {
        printf("cnt = %d\n", cnt);
        Sleep(1000);
    }

    return 0;
}
```

c:\users\hank\documents\visual studio 2010\Projects\test thread\Debug\...

```
cnt = 0
cnt = 148164273
cnt = 142283989
cnt = 491040575
cnt = 575902327
cnt = 593298165
cnt = 631462487
cnt = 868727954
cnt = 869000458
cnt = 846788733
cnt = 825534103
cnt = 778078383
cnt = 731596933
cnt = 497068412
cnt = 586530187
cnt = 167881172
cnt = -122878726
cnt = -171543469
cnt = -392855668
cnt = -319139642
cnt = -471605208
cnt = -396869666
cnt = 19841249
cnt = -246007788
微軟新注音 半 :
```

Registers

```
EAX = 00000001 EBX = 00000000 ECX = 00000000 EDX = 00AE1163 ESI = 00000000 EDI = 0121F91C EIP = 00AE13C7 ESP = 0121F850 EBP = 0121F91C EFL = 00000202

00AE7138 = 8003241D
```



(Global Scope) ThreadA(void * pParam)

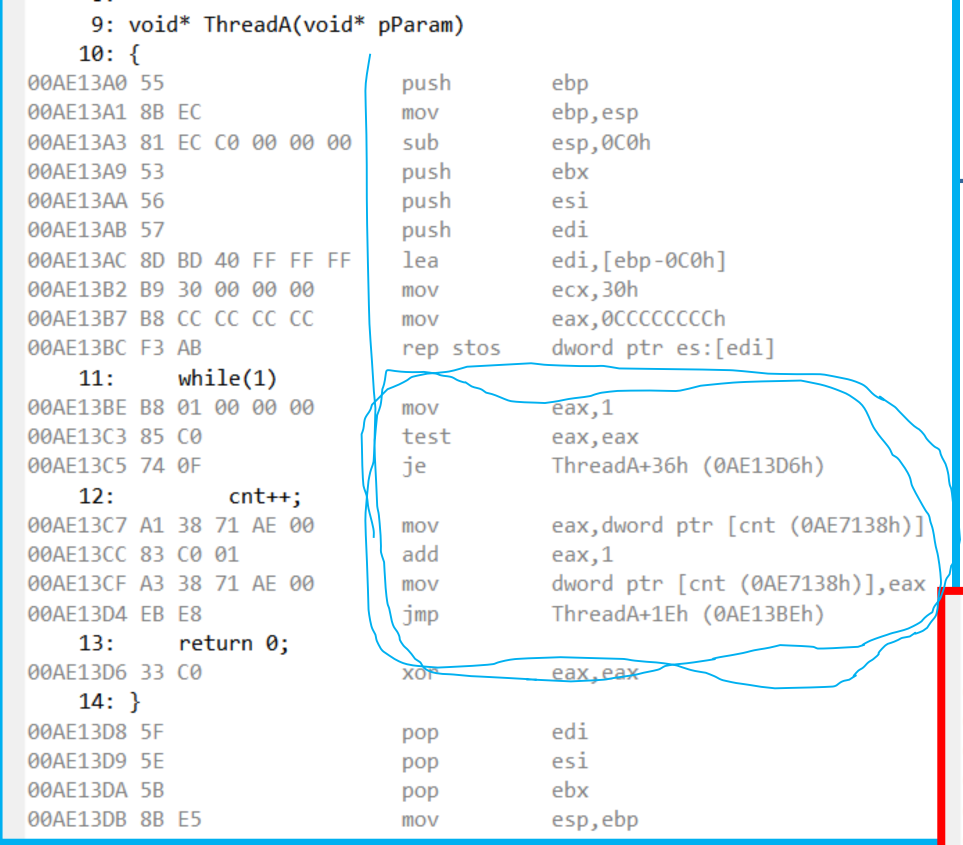
```
#include "stdafx.h"
#include <Windows.h>

int cnt = 0;

void* ThreadA(void* pParam)
{
    while(1)
        cnt++;
    return 0;
}

void* ThreadB(void* pParam)
{
    while(1)
        cnt--;
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadA,0,0,0);
    CreateThread(0,0,(LPTHREAD_START_ROUTINE)ThreadB,0,0,0);

    while(1) {
        printf("cnt = %d\n", cnt);
        Sleep(1000);
    }

    return 0;
}
```

Parallel Stacks

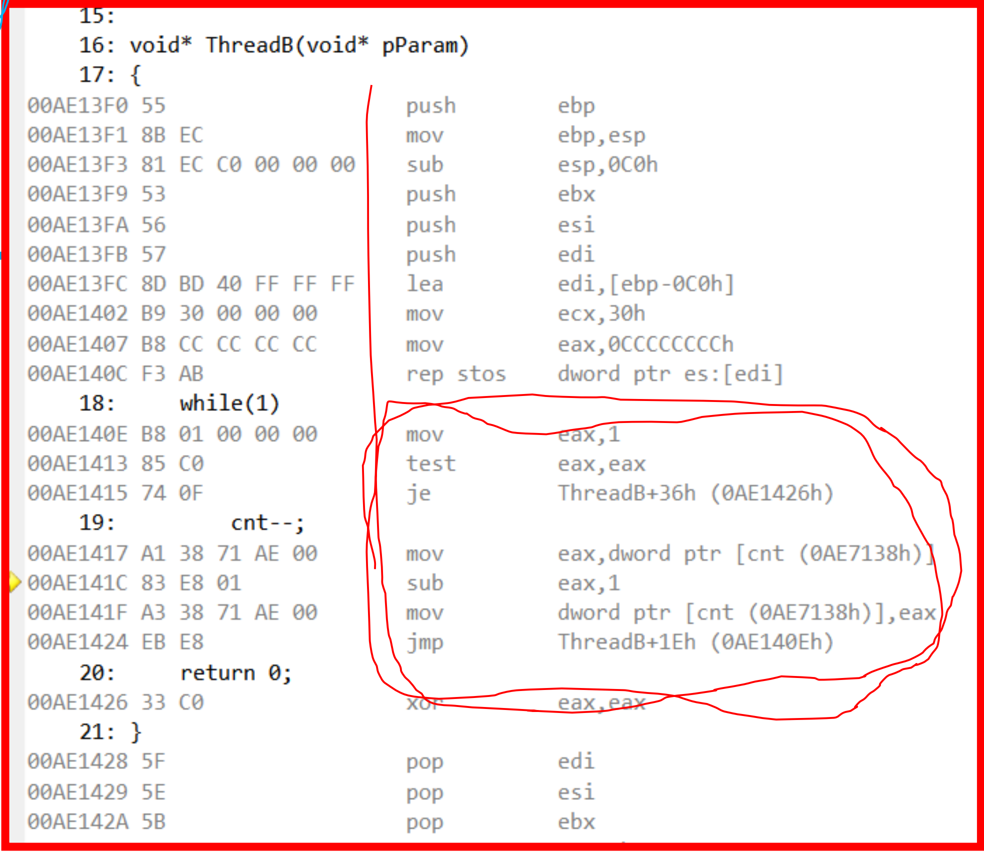
```
1 Thread
7768df34
[Frames below may be incorrect and/or...
762d11f8
762d119c
wmain
__tmainCRTStartup
wmainCRTStartup
763a850d

1 Thread
ThreadA

1 Thread
ThreadB

2 Threads
763a850d
[Frames below may be incorrect and/or...

3 Threads
776bbf39
776bbf0c
```

Registers

```
EAX = 7B5F5E6F EBX = 00000000 ECX = 00000000 EDX = 00AE1055 ESI = 00000000 EDI = 0148FC84 EIP = 00AE141C ESP = 0148FBB8 EBP = 0148FC84 EFL = 00000202
```



```
     9: void* ThreadA(void* pParam)
    10: {
00AE13A0 55                  push        ebp
00AE13A1 8B EC               mov         ebp,esp
00AE13A3 81 EC C0 00 00 00   sub         esp,0C0h
00AE13A9 53                  push        ebx
00AE13AA 56                  push        esi
00AE13AB 57                  push        edi
00AE13AC 8D BD 40 FF FF FF   lea         edi,[ebp-0C0h]
00AE13B2 B9 30 00 00 00      mov         ecx,30h
00AE13B7 B8 CC CC CC CC      mov         eax,0CCCCCCCCh
00AE13BC F3 AB               rep stos    dword ptr es:[edi]
    11:     while(1)
00AE13BE B8 01 00 00 00      mov         eax,1
00AE13C3 85 C0               test        eax,eax
00AE13C5 74 0F               je          ThreadA+36h (0AE13D6h)
    12:         cnt++;
00AE13C7 A1 38 71 AE 00      mov         eax,dword ptr [cnt (0AE7138h)]
00AE13CC 83 C0 01            add         eax,1
00AE13CF A3 38 71 AE 00      mov         dword ptr [cnt (0AE7138h)],eax
00AE13D4 EB E8               jmp         ThreadA+1Eh (0AE13BEh)
    13:     return 0;
00AE13D6 33 C0               xor         eax,eax
    14: }
00AE13D8 5F                  pop         edi
00AE13D9 5E                  pop         esi
00AE13DA 5B                  pop         ebx
00AE13DB 8B E5               mov         esp,ebp
```



```
    15:
    16: void* ThreadB(void* pParam)
    17: {
00AE13F0 55                  push        ebp
00AE13F1 8B EC               mov         ebp,esp
00AE13F3 81 EC C0 00 00 00   sub         esp,0C0h
00AE13F9 53                  push        ebx
00AE13FA 56                  push        esi
00AE13FB 57                  push        edi
00AE13FC 8D BD 40 FF FF FF   lea         edi,[ebp-0C0h]
00AE1402 B9 30 00 00 00      mov         ecx,30h
00AE1407 B8 CC CC CC CC      mov         eax,0CCCCCCCCh
00AE140C F3 AB               rep stos    dword ptr es:[edi]
    18:     while(1)
00AE140E B8 01 00 00 00      mov         eax,1
00AE1413 85 C0               test        eax,eax
00AE1415 74 0F               je          ThreadB+36h (0AE1426h)
    19:         cnt--;
00AE1417 A1 38 71 AE 00      mov         eax,dword ptr [cnt (0AE7138h)]
00AE141C 83 E8 01            sub         eax,1
00AE141F A3 38 71 AE 00      mov         dword ptr [cnt (0AE7138h)],eax
00AE1424 EB E8               jmp         ThreadB+1Eh (0AE140Eh)
    20:     return 0;
00AE1426 33 C0               xor         eax,eax
    21: }
00AE1428 5F                  pop         edi
00AE1429 5E                  pop         esi
00AE142A 5B                  pop         ebx
```

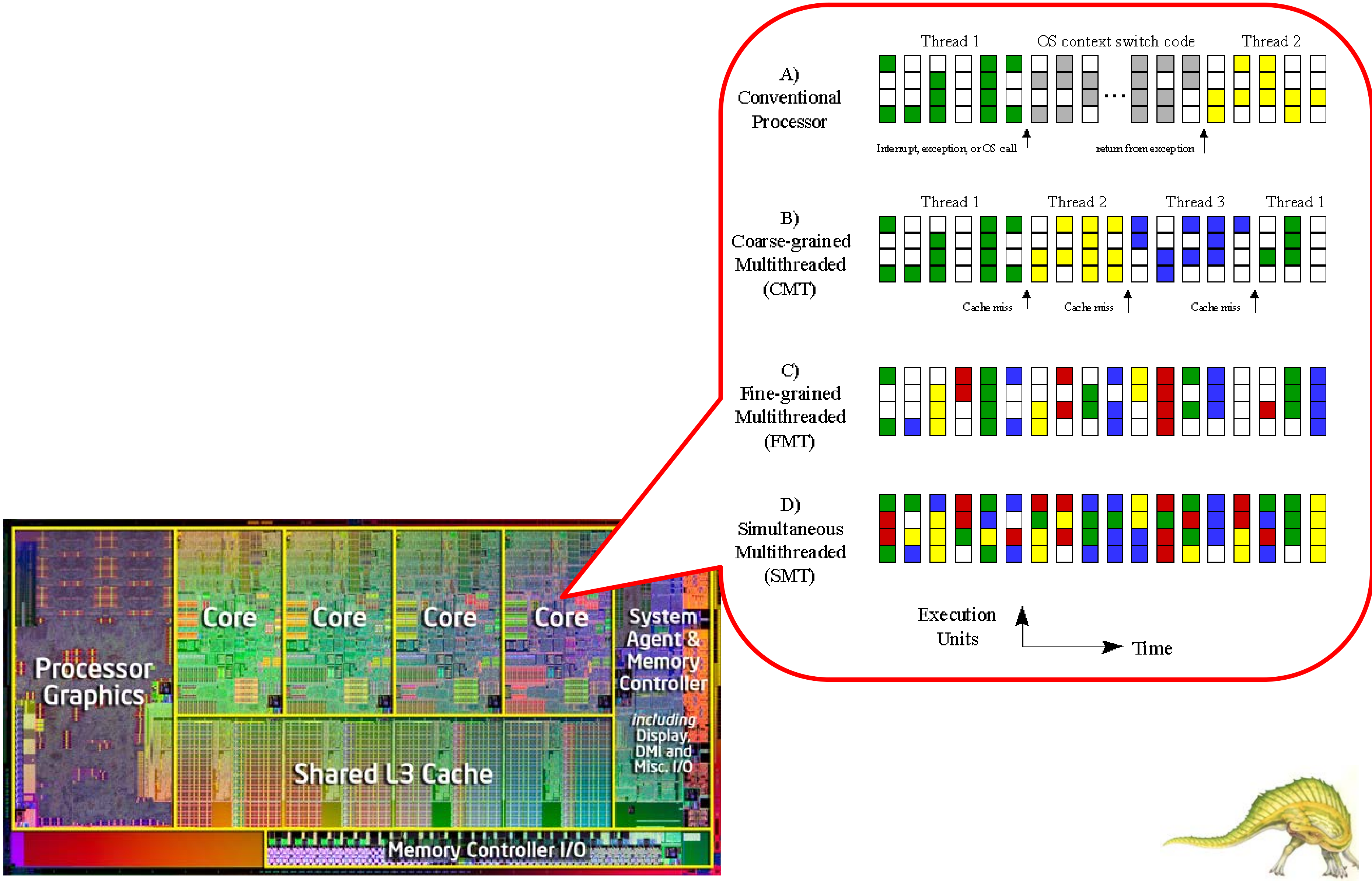
User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Hardware Threads


A) Conventional Processor
Thread 1
OS context switch code
Thread 2
Interrupt, exception, or OS call
return from exception
B) Coarse-grained Multithreaded (CMT)
Thread 1
Thread 2
Thread 3
Thread 1
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)
Execution Units
Time
Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including Display, DMI and Misc. I/O
Shared L3 Cache
Memory Controller I/O

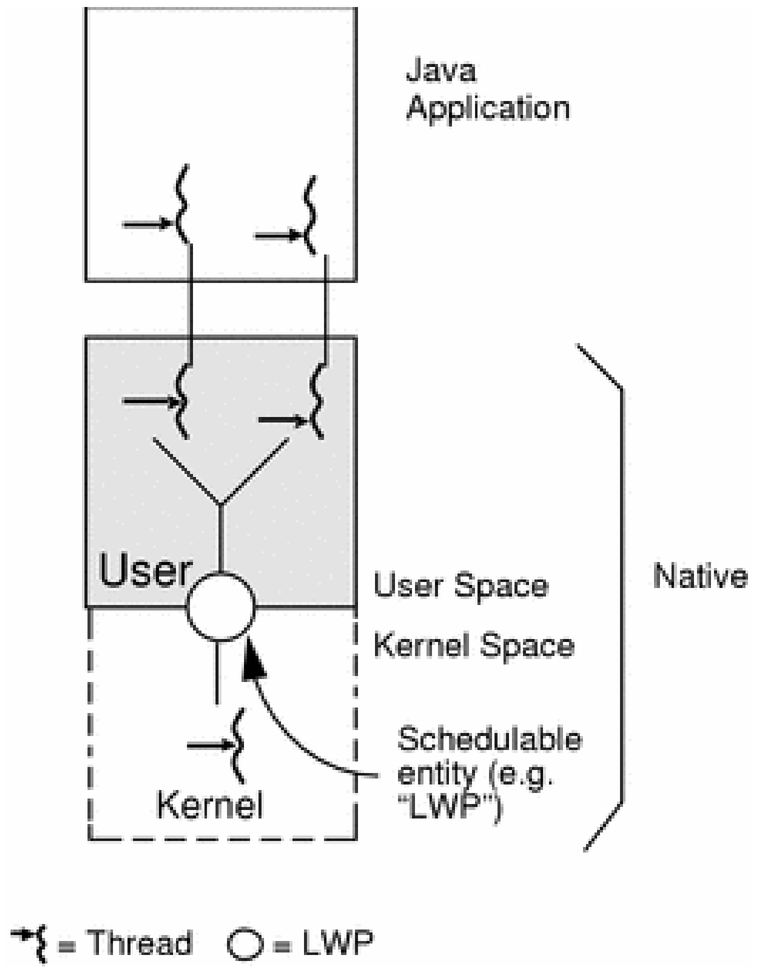
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Many-to-One Model



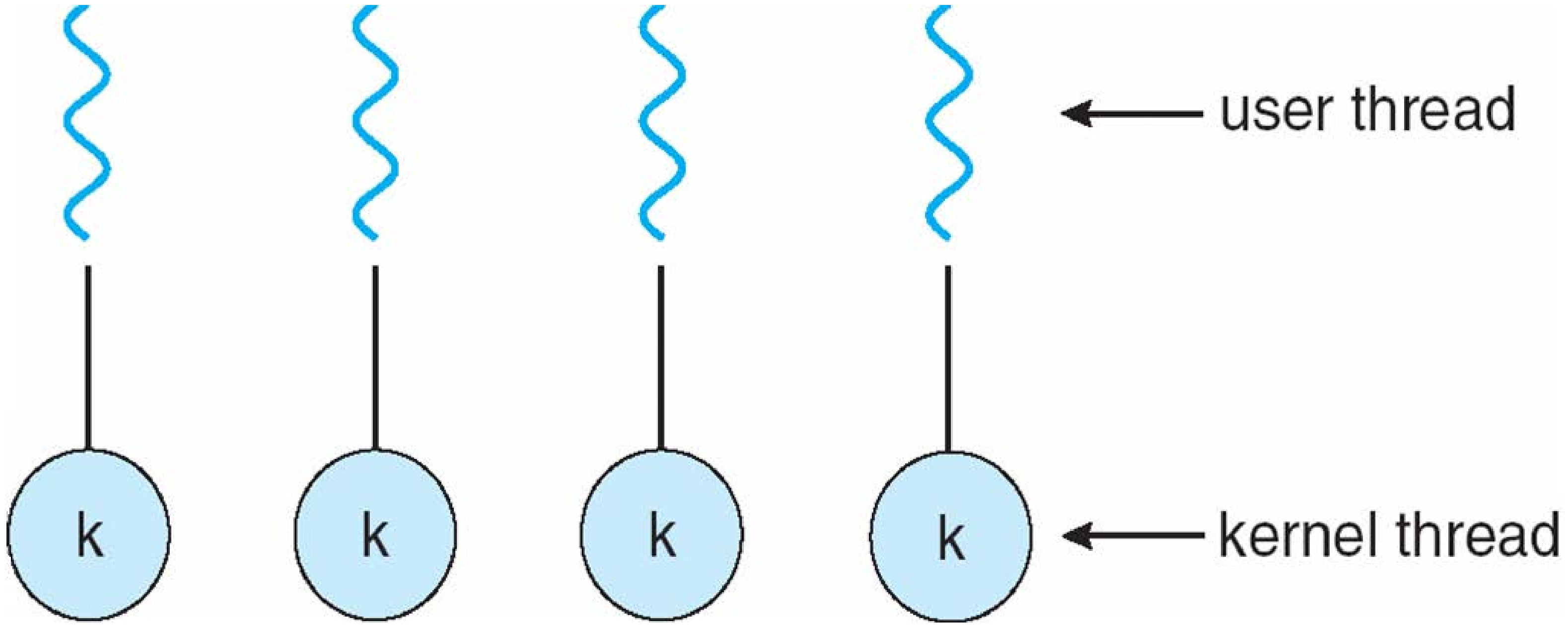
Many-to-one

- In this model, the library maps all threads to a single lightweight process
- Advantages:
  - totally portable
  - easy to do with few systems dependencies
- Disadvantages:
  - cannot take advantage of parallelism
  - may have to block for synchronous I/O
  - there is a clever technique for avoiding it
- Mainly used in language systems, portable libraries

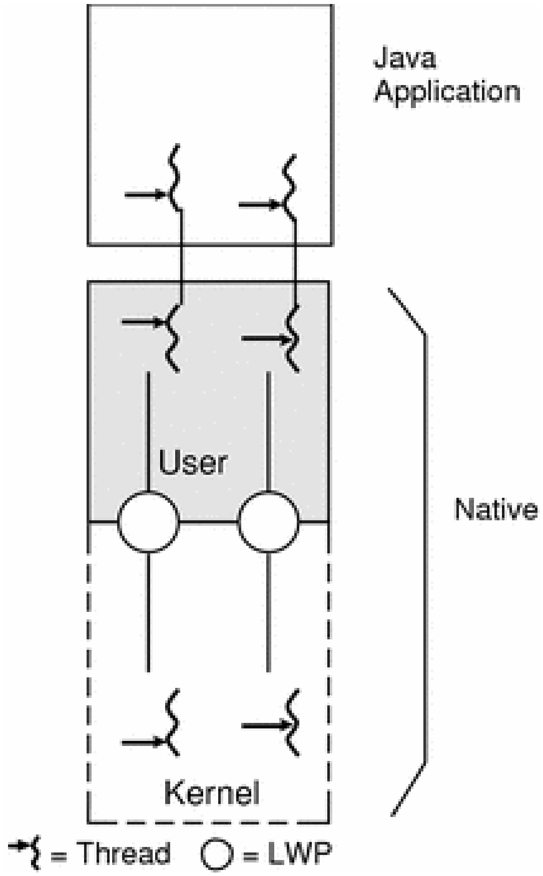
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

One-to-one Model



One-to-one model



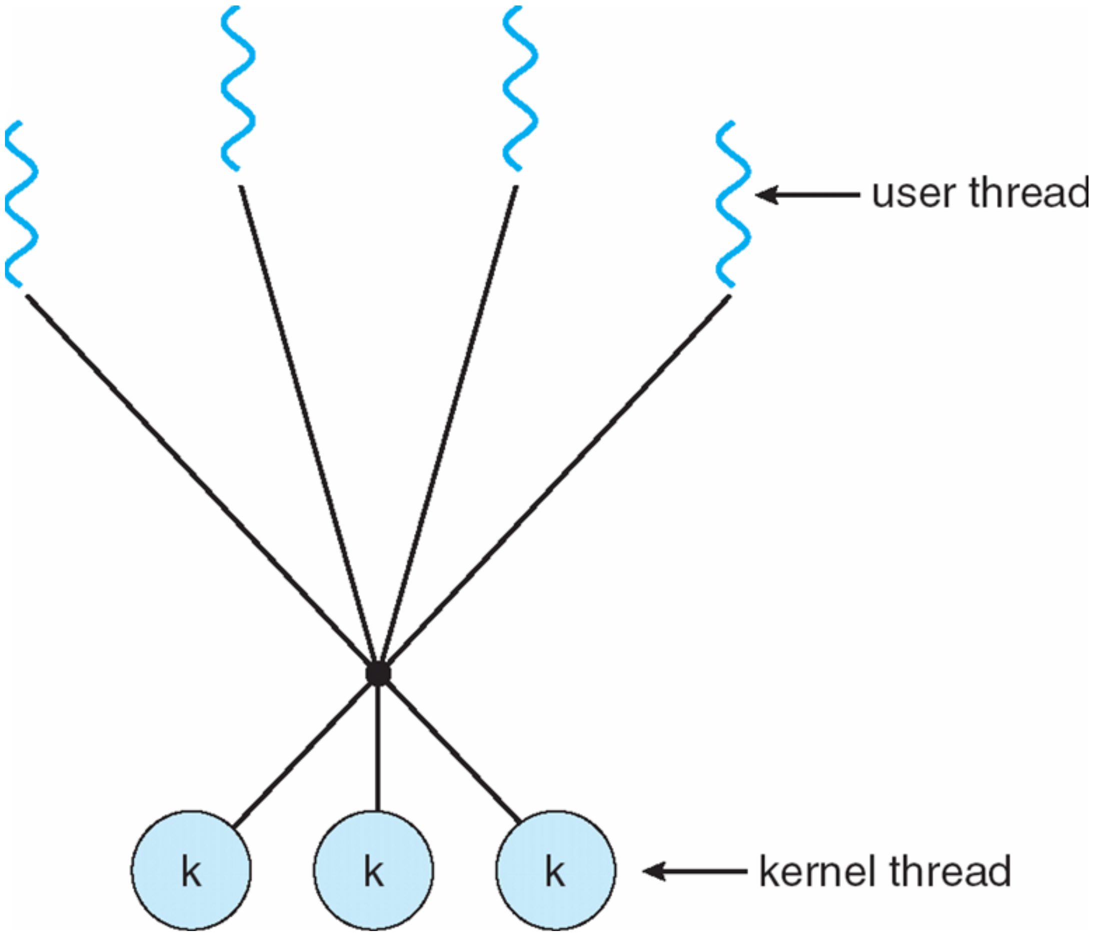
One-to-one

- In this model, the library maps each thread to a different lightweight process
- Advantages:
  - can exploit parallelism, blocking system calls
- Disadvantages:
  - thread creation involves LWP creation
  - each thread takes up kernel resources
  - limiting the number of total threads
- Used in LinuxThreads and other systems where LWP creation is not too expensive

Many-to-Many Model

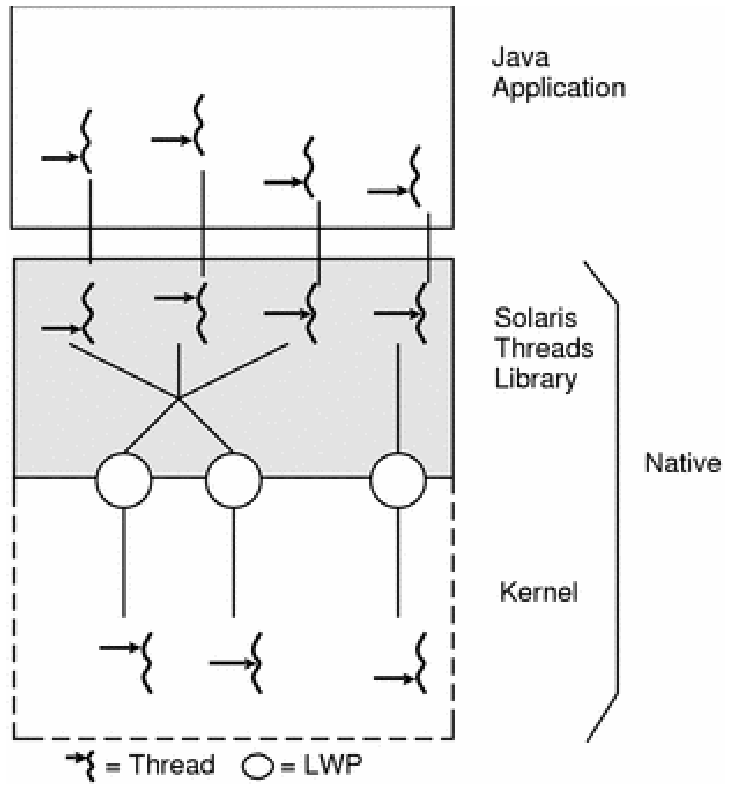
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



Two level

- In this model, the library has two kinds of threads: *bound* and *unbound*
  - bound threads are mapped each to a single lightweight process
  - unbound threads *may* be mapped to the same LWP
- Probably the best of both worlds
- Used in the Solaris implementation of Pthreads (and several other Unix implementations)



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

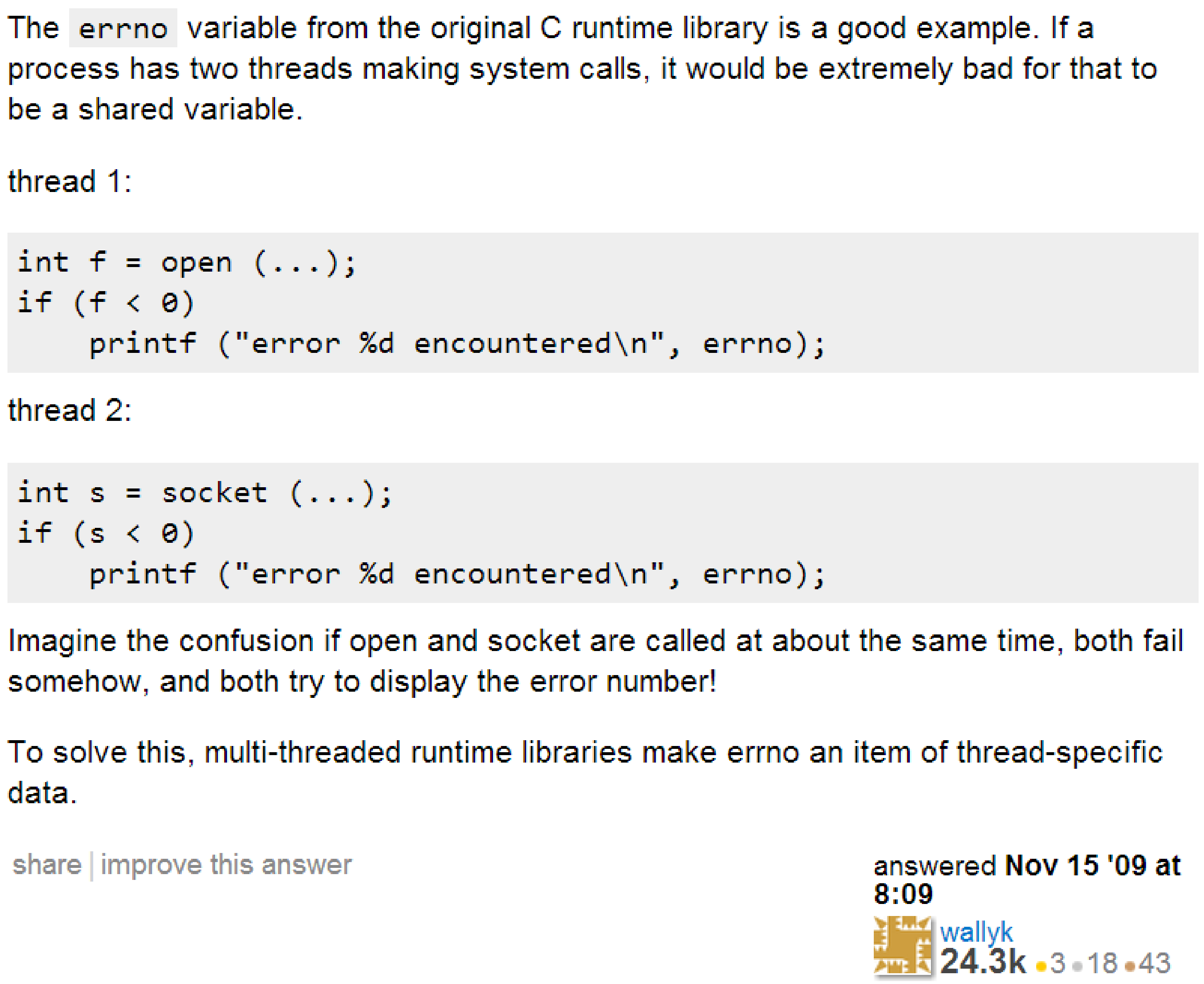
Thread Pools

- 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
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Thread Specific Data



The `errno` variable from the original C runtime library is a good example. If a process has two threads making system calls, it would be extremely bad for that to be a shared variable.

thread 1:

```
int f = open (...);
if (f < 0)
    printf ("error %d encountered\n", errno);
```

thread 2:

```
int s = socket (...);
if (s < 0)
    printf ("error %d encountered\n", errno);
```

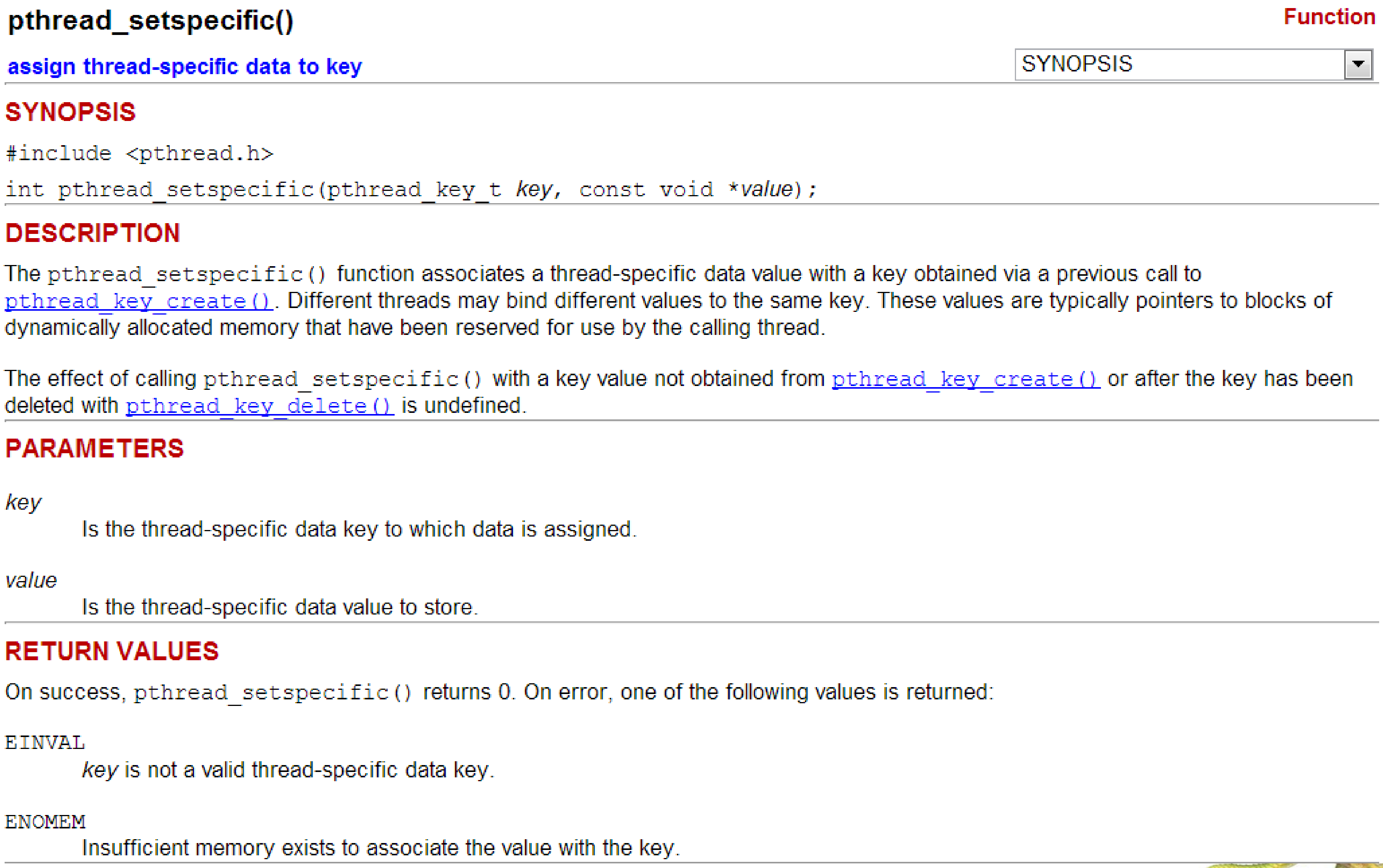
Imagine the confusion if open and socket are called at about the same time, both fail somehow, and both try to display the error number!

To solve this, multi-threaded runtime libraries make errno an item of thread-specific data.

share | improve this answer

answered **Nov 15 '09 at 8:09**

wallyk
24.3k ●3 ●18 ●43



pthread_setspecific()

Function

assign thread-specific data to key

SYNOPSIS

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

The `pthread_setspecific()` function associates a thread-specific data value with a key obtained via a previous call to `pthread_key_create()`. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling `pthread_setspecific()` with a key value not obtained from `pthread_key_create()` or after the key has been deleted with `pthread_key_delete()` is undefined.

PARAMETERS

key
Is the thread-specific data key to which data is assigned.

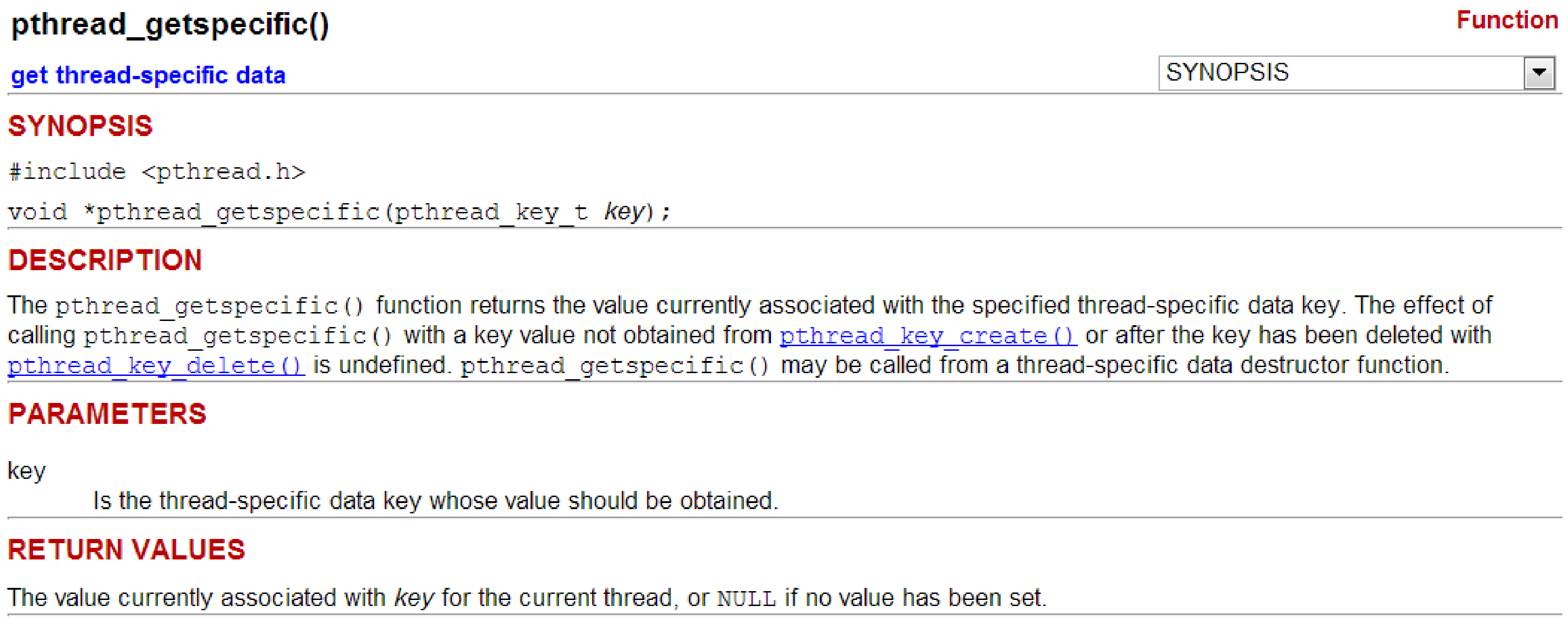
value
Is the thread-specific data value to store.

RETURN VALUES

On success, `pthread_setspecific()` returns 0. On error, one of the following values is returned:

`EINVAL`
key is not a valid thread-specific data key.

`ENOMEM`
Insufficient memory exists to associate the value with the key.



pthread_getspecific() Function

get thread-specific data

SYNOPSIS

SYNOPSIS

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

DESCRIPTION

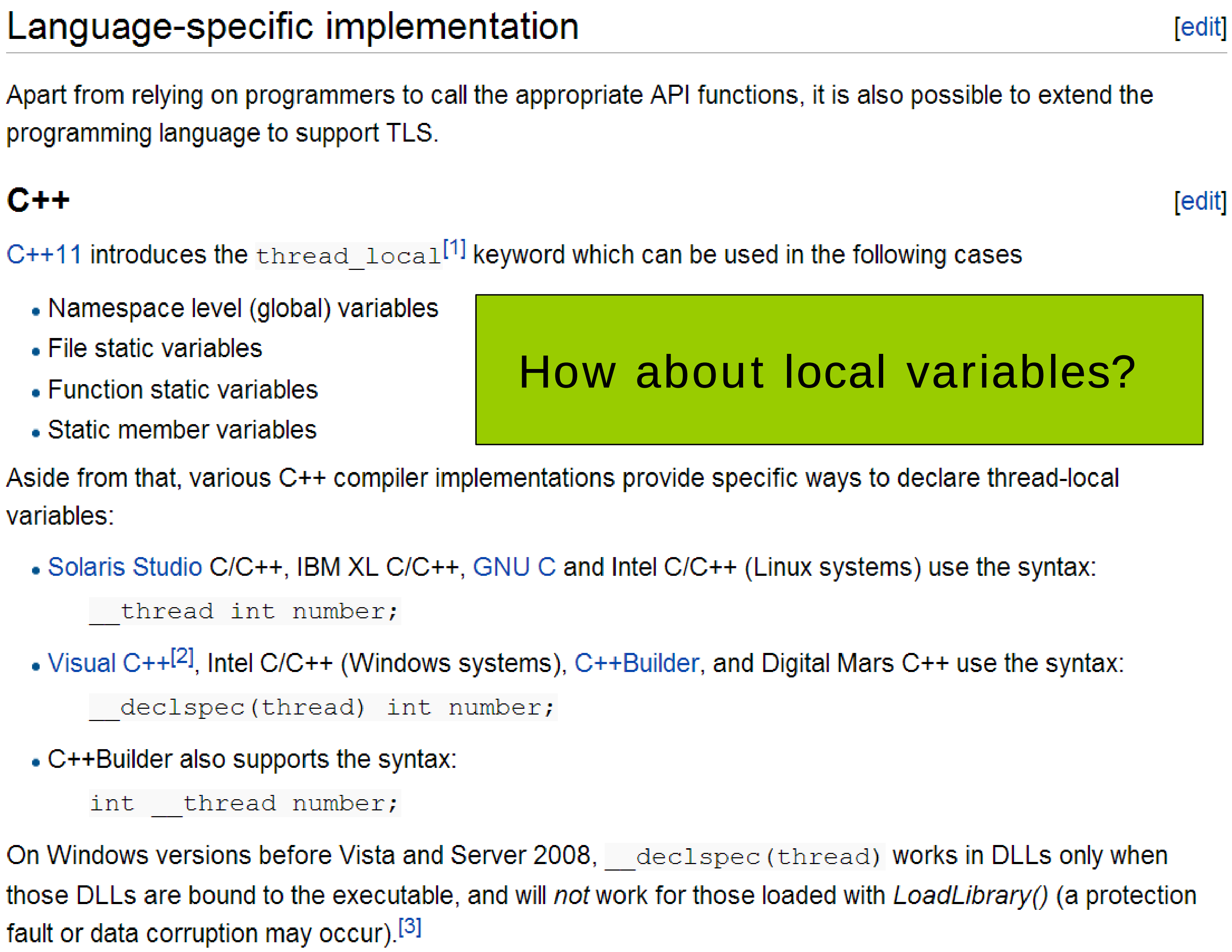
The `pthread_getspecific()` function returns the value currently associated with the specified thread-specific data key. The effect of calling `pthread_getspecific()` with a key value not obtained from `pthread_key_create()` or after the key has been deleted with `pthread_key_delete()` is undefined. `pthread_getspecific()` may be called from a thread-specific data destructor function.

PARAMETERS

key
    Is the thread-specific data key whose value should be obtained.

RETURN VALUES

The value currently associated with *key* for the current thread, or `NULL` if no value has been set.



Language-specific implementation

[edit]

Apart from relying on programmers to call the appropriate API functions, it is also possible to extend the programming language to support TLS.

C++

[edit]

C++11 introduces the `thread_local`[1] keyword which can be used in the following cases

- Namespace level (global) variables
- File static variables
- Function static variables
- Static member variables

How about local variables?

Aside from that, various C++ compiler implementations provide specific ways to declare thread-local variables:

- Solaris Studio C/C++, IBM XL C/C++, GNU C and Intel C/C++ (Linux systems) use the syntax:

```
__thread int number;
```

- Visual C++[2], Intel C/C++ (Windows systems), C++Builder, and Digital Mars C++ use the syntax:

```
__declspec(thread) int number;
```

- C++Builder also supports the syntax:

```
int __thread number;
```

On Windows versions before Vista and Server 2008, `__declspec(thread)` works in DLLs only when those DLLs are bound to the executable, and will *not* work for those loaded with *LoadLibrary()* (a protection fault or data corruption may occur).[3]

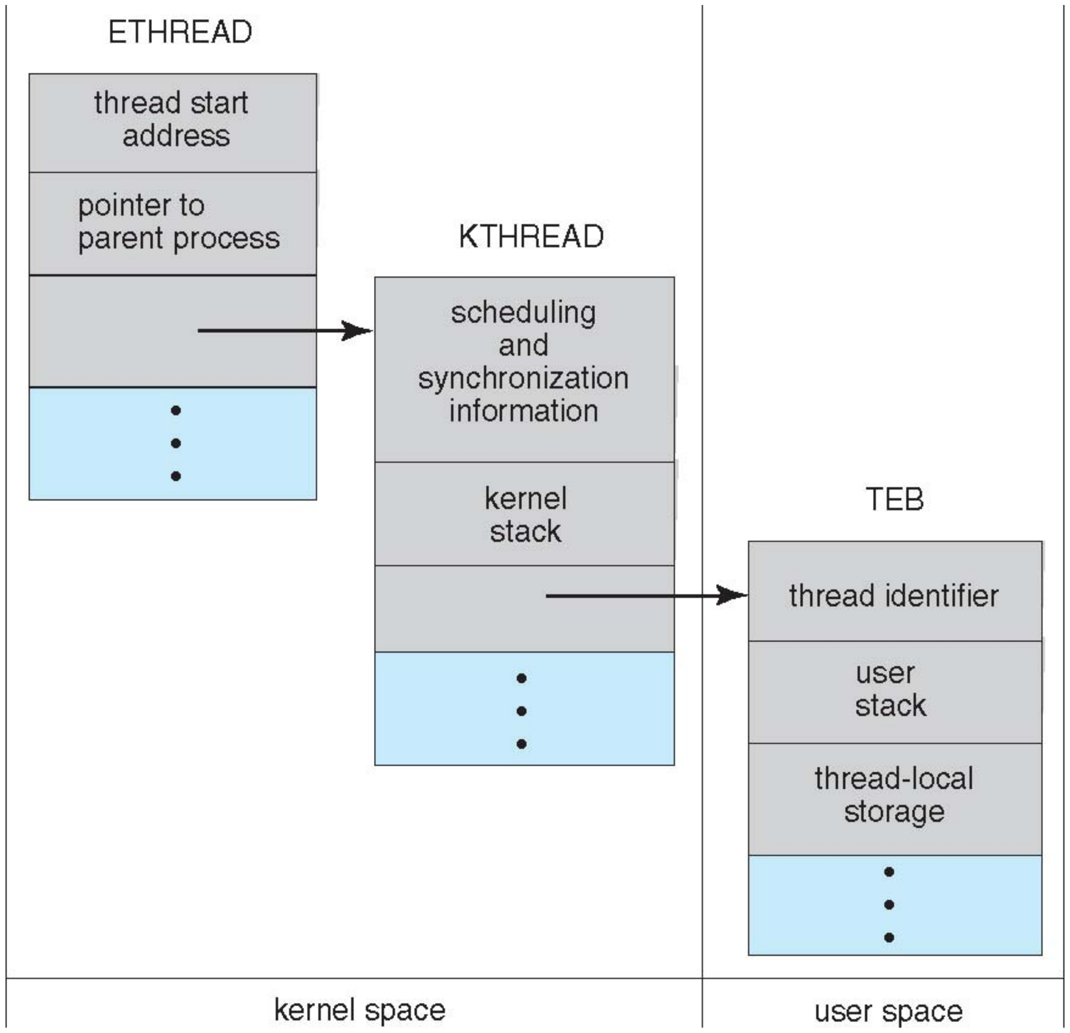
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

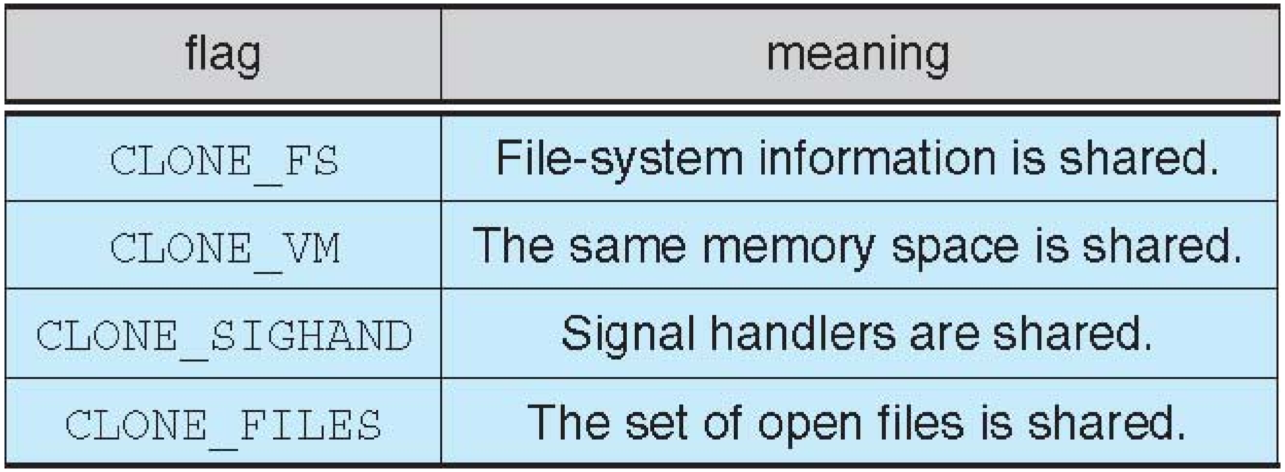
Operating System Examples

- Windows XP Threads
- Linux Thread

Windows XP Threads



Linux Threads



flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

High-Level Program Structure Ideas

- Boss/workers model
- Pipeline model
- Up-calls
- Keeping shared information consistent using version stamps

Thread Design Patterns

Common ways of structuring programs using threads

- Boss/workers model
  - boss gets assignments, dispatches tasks to workers
  - variants (thread pool, single thread per connection…)
- Pipeline model
  - do some work, pass partial result to next thread
- Up-calls
  - fast control flow transfer for layered systems
- Version stamps
  - technique for keeping information consistent

Boss/Workers

Boss:

```
forever {
    get a request
    switch(request)
      case X: Fork (taskX)
      case Y: Fork (taskY)
    …
}
```

Worker:

```
taskX();
```

- Advantage: simplicity
- Disadvantage: bound on number of workers, overheard of threads creation, contention if requests have interdependencies
- Variants: fixed thread pool (aka *workpile*, *workqueue*), producer/consumer relationship, workers determine what needs to be performed…

Pipeline

- Each thread completes portion of a task, and passes results
- like an assembly line or a processor pipeline
- Advantages: trivial synchronization, simplicity
- Disadvantages: limits degree of parallelism, throughput driven by slowest stage, handtuning needed

Up-calls

- Layered applications, e.g. network protocol stacks have top-down and bottom-up flows
- Up-calls is a technique in which you structure layers so that they can expect calls from below
- Thread pool of specialized threads in each layer
  - essentially an up-call pipeline per connection
- Advantages: best when used with fast, synchronous control flow transfer mechanisms or program structuring tool
- Disadvantages: programming becomes more complicated, synchronization required for top-down

Version Stamps

- (Not a programming structure idea but useful technique for any kind of distributed environment)
- Maintain “version number” for shared data
  - keep local cached copy of data
  - check versions to determine if changed

End of Chapter 4