Chapter 4: Threads

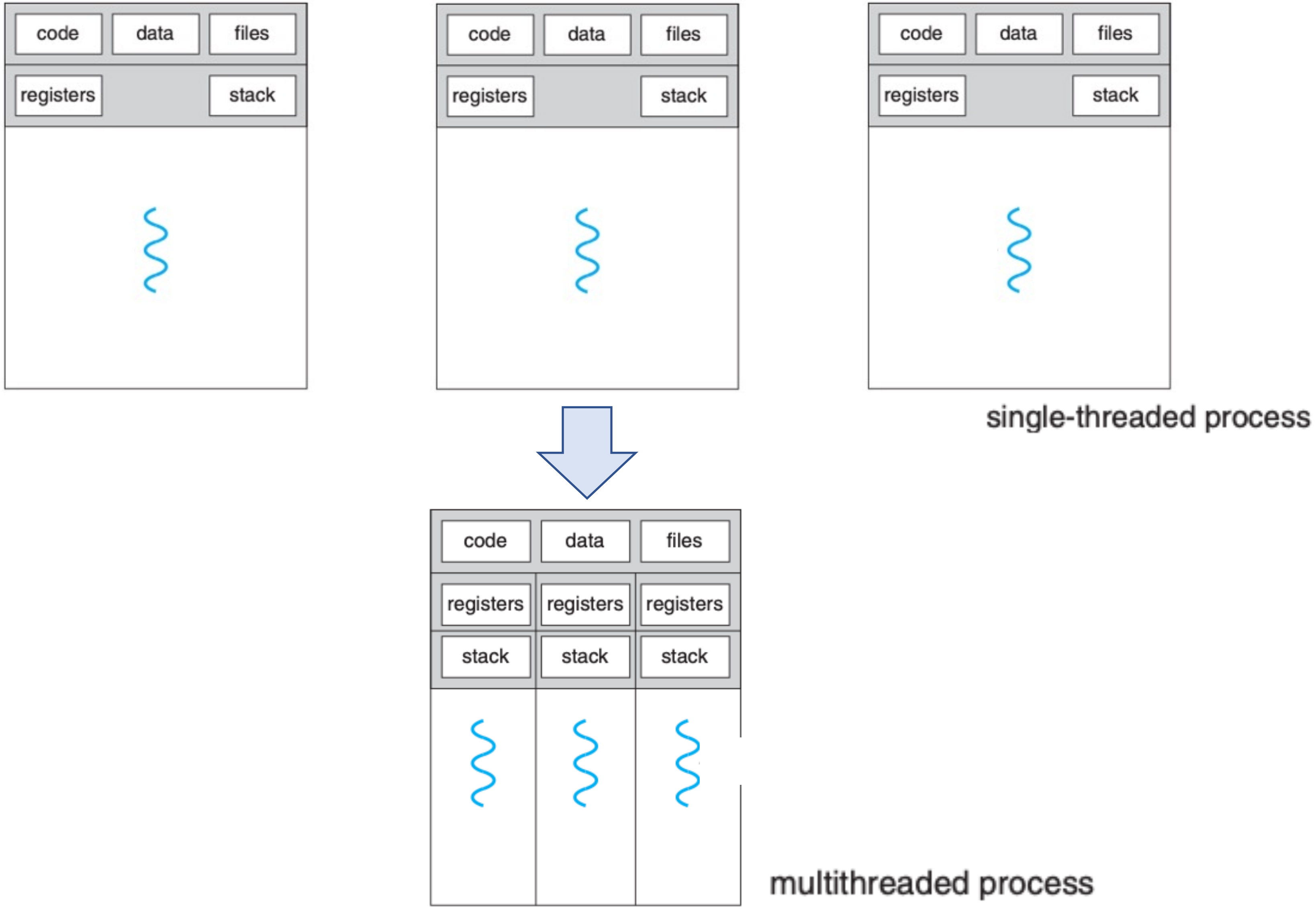
Notion of thread

APIs for the Pthread, Windows, and Java thread libraries

Issues related to multithreaded programming

Contents

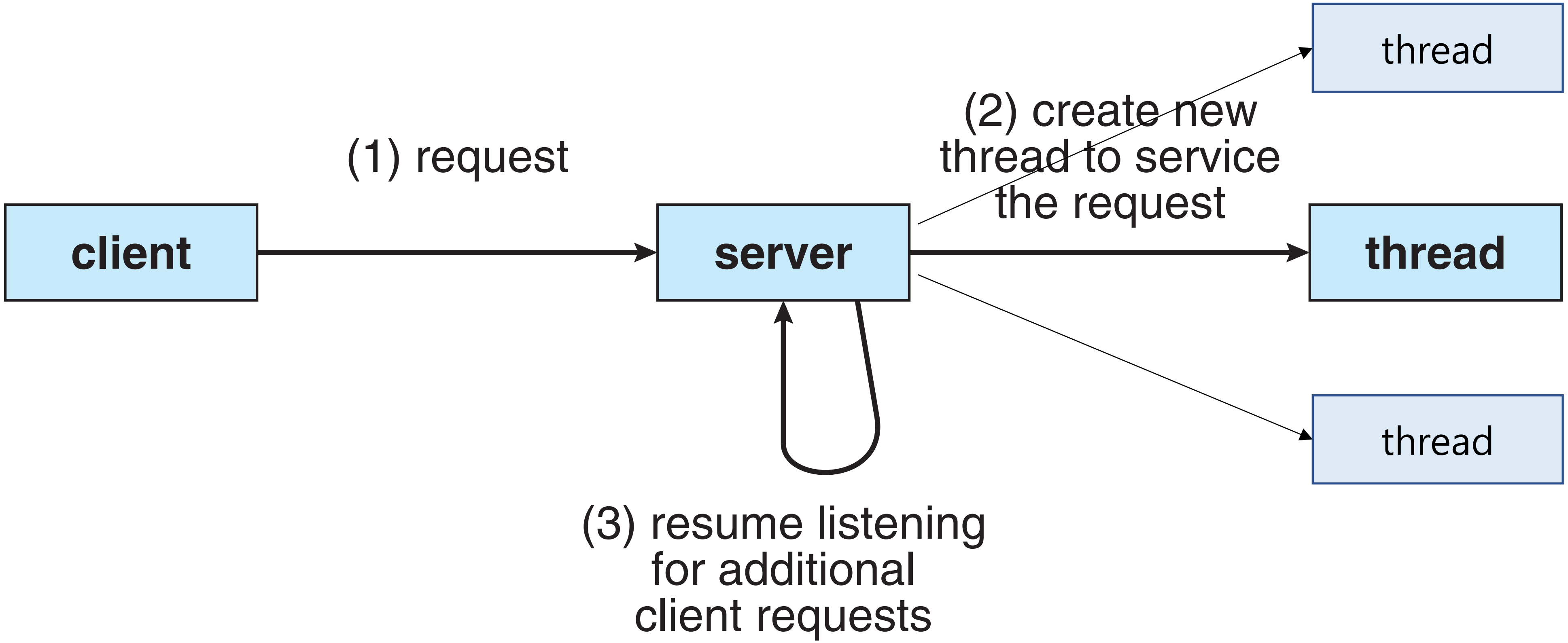
4.1 Overview


code
data
files
registers
stack
code
data
files
registers
stack
code
data
files
registers
stack
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
multithreaded process

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separat e threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weig ht
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture

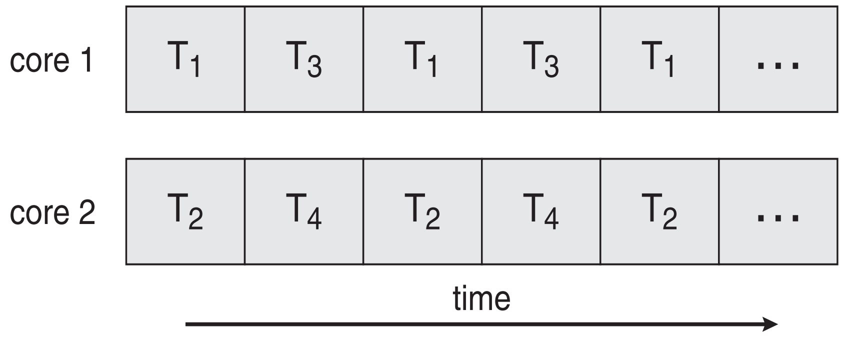


Benefits

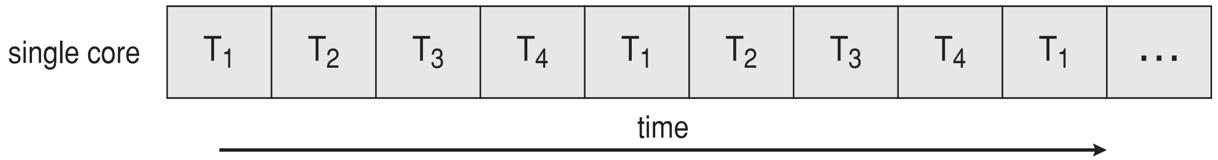
- **Responsiveness –** may allow continued execution if part of p rocess is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easie r than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching l ower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor ar chitectures

4.2 Multicore Programming

- **Multicore** or **multiprocessor** systems
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progre ss
    - Single processor / core, sched uler providing concurrency



Parallelism on a multi-core system



Concurrent execution on single-core system

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion and N processing cores

$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

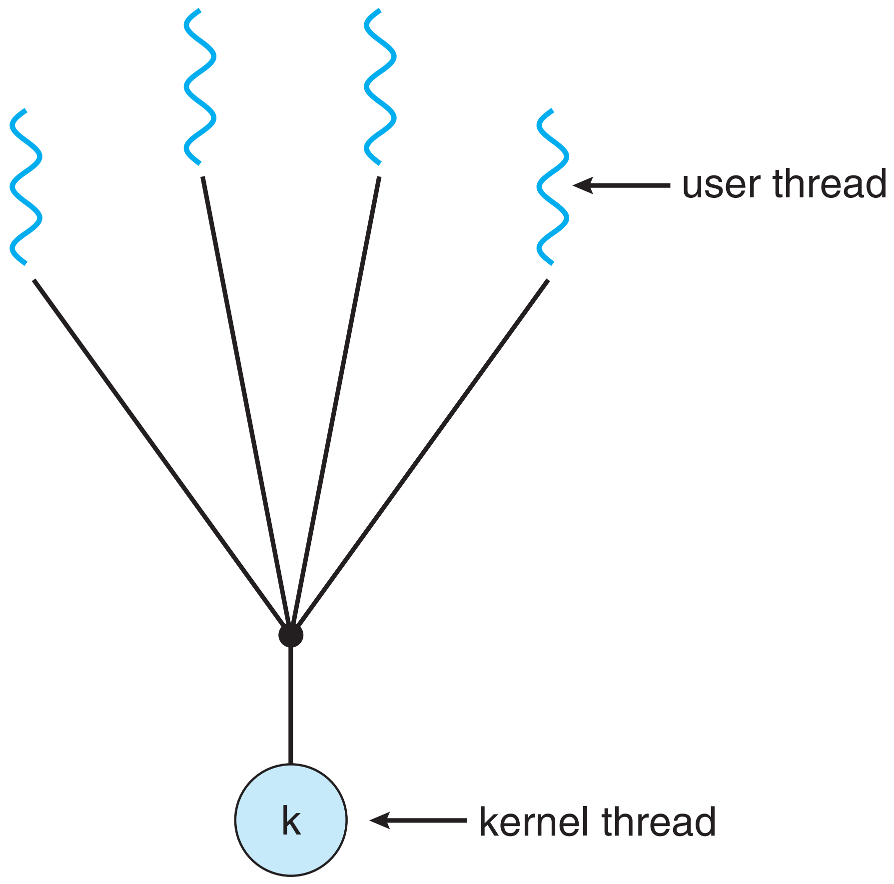
4.3 Multithreading models

- **User threads** - management done by user-level threads library
- Three primary thread libraries: POSIX **Pthreads** / Windows thread / Java threads
- **Kernel threads** - Supported by the Kernel
- Virtually all general purpose operating systems, including: Windows, Linux, Mac OS X, and Solaris

- Multithreading models
  - Many-to-One
  - One-to-One
  - Many-to-Many

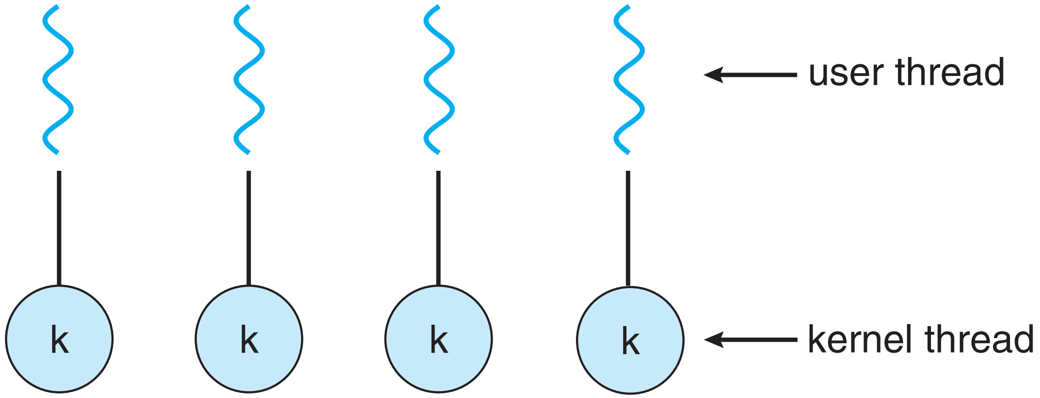
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



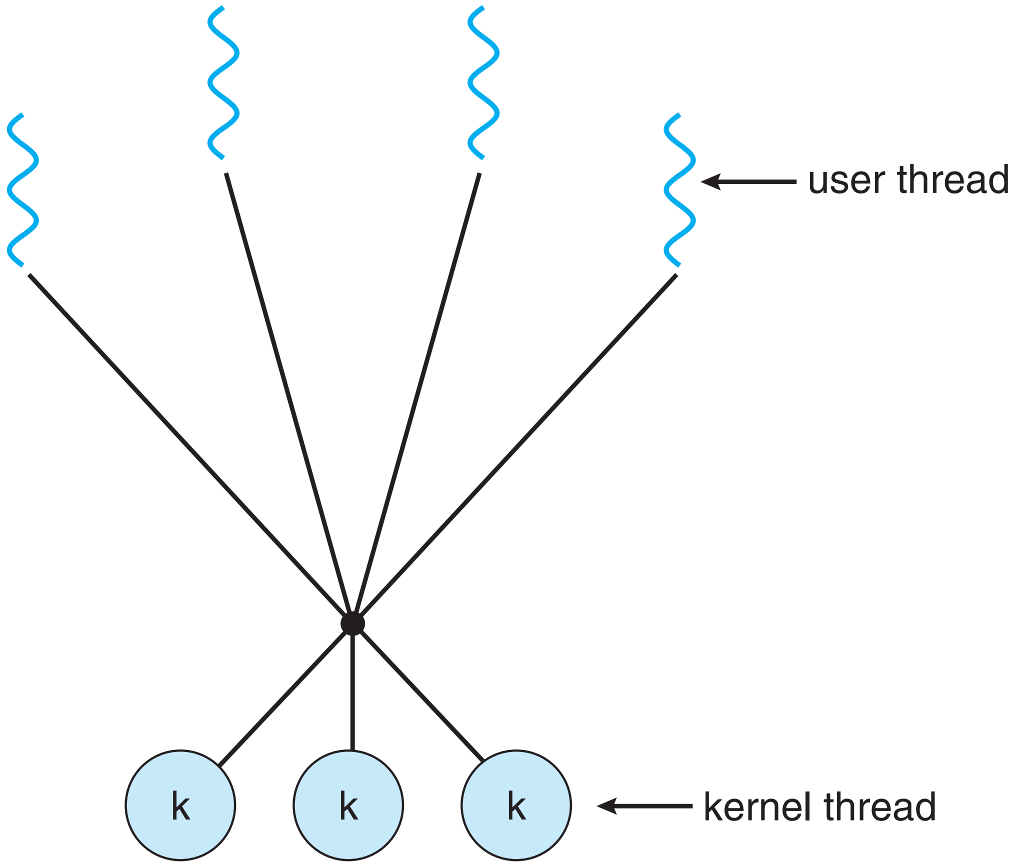
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



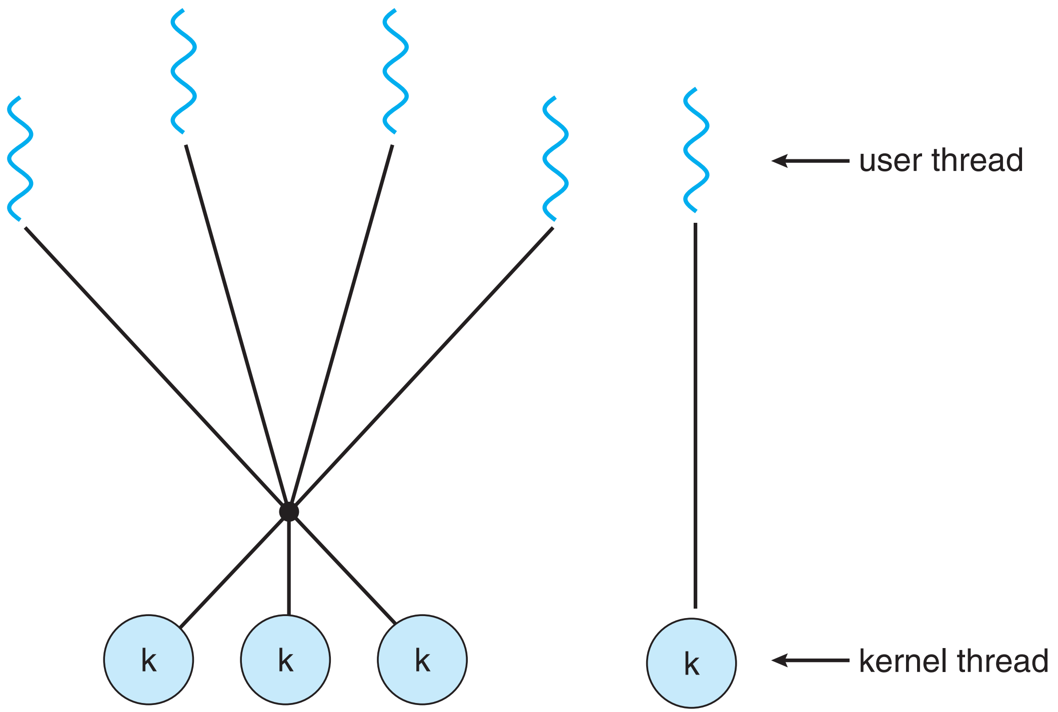
Many-to-Many Model

- Allows many user level threa ds to be mapped to many ke rnel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9 / W indows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **b ound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



4.4 Thread Libraries

- **Thread library** provides programmer with API for creating an d managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- POSIX Pthreads / Windows Threads / Java Thread

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and s ynchronization
- ***Specification***, not ***implementation.*** API specifies behavior of the thread library, implementation is up to development of th e library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

```
   /* get the default attributes */
   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);
}

/* The thread will begin control 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);
}
```

Windows Multithreaded C Program

(1)
```c
#include <windows.h>
#include <stdio.h>
```
(2)
```c
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
```
(3)
```c
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}
```
(4)
```c
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -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 */

   if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

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

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

4.5 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
- Three methods explored
  - Thread Pools
  - 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 ne w thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different s trategies for running task
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
   /*
    * this function runs as a separate thread.
    */
}
```

OpenMP

- Set of compiler directives and an API for C, C++, F ORTRAN
- Provides support for parallel programming in share d-memory environments. dentifies **parallel regions** – blocks of code that can run in parallel

- Create as many threads as there are cores

```
#pragma omp parallel
```

- Run for loop in parallel

```
#pragma omp parallel for for(i=0;i<N;i++
  ) {
    c[i] = a[i] + b[i];
}
```

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems. Extensions to C, C++ languages, API, and run-time library
- Block is in "^{ }" - **`^{ printf("I am a block"); }`**
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

- Allows identification of parallel sections
- Manages most of the details of threading

Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process , called **main queue**
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be re moved at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

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