Chapter 4:  Threads
Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples
Objectives

- **Thread** of a process: basic unit of CPU utilization and is composed of a:
  - Thread ID
  - Program counter: register EIP
  - Register set
  - Stack
  - And *it hares with other threads of the same process*, the
    - Code segment
    - Data segment
    - OS resources: open files, signals, … etc

- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux
### Single and Multithreaded Processes

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**single-threaded process**

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
<th>registers</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stack</th>
<th>stack</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**multithreaded process**
Motivation

- Most modern applications and computers are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
  - For applications performing *multiple similar* tasks
    - It is costly to create a process for each task
  - Ex: web server serving multiple clients requesting the same service
    - Create a separate thread for each service
- Multithreaded programs: can simplify code, increase efficiency.
- Kernels are generally multithreaded. *Interrupt handling, device management, etc*
Multithreaded Server Architecture

1. Request
2. Create new thread to service the request
3. Resume listening for additional client requests
Benefits

- **Responsiveness**
  - may allow continued execution if part (i.e. thread) of a process is time-consuming, especially important for user interfaces. User needs not wait

- **Resource Sharing**
  - threads share resources of process, easier than shared-memory or message-passing. Since programmer need not explicitly code for communication between threads (shared-memory or message-passing codes, in Chap-3)

- **Economy**
  - cheaper than process creation, thread switching has lower overhead than context switching. See figure on Page-4: thread share most of process’s PCB

- **Scalability**
  - process can take advantage of multiprocessor architectures. Threads can run in parallel on different processing cores. See figure on Page-4 again
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers. OS designers must write CPU scheduling algorithms that use multiple cores. App programmers must design multithreaded programs. Challenges include:

  - **Identifying tasks**: What are the separate independent threads?
  - **Balance**: How to divide the work (set of tasks) fairly among CPU cores?
  - **Data splitting**: How to divide the data fairly among CPU cores?
  - **Data dependency**: What do to when there is dependency between threads?
  - **Testing and debugging**: Correct parallel/concurrent/multithreaded program?

- Multithreaded programming provides a mechanism for **more efficient use of multiple cores and improved concurrency**. New software design paradigm?

- **Parallelism** implies a system can perform more than one task simultaneously.

- **Concurrency** supports more than one task making progress.

  - Single processor / core. CPU scheduler providing concurrency.
Concurrency vs. Parallelism

- Concurrent execution on single-core system:

  ![Diagram showing concurrent execution on a single-core system]

  - Time line: T₁ T₂ T₃ T₄ T₁ T₂ T₃ T₄ T₁ ...

- Parallelism on a multi-core system:

  ![Diagram showing parallel execution on multi-core system]

  - Core 1: T₁ T₃ T₁ T₃ T₁ ...
  - Core 2: T₂ T₄ T₂ T₄ T₂ ...

  Time line: T₁ T₂ T₃ T₄ T₁ T₂
Multicore Programming

- **Types of parallelism** (and concurrencies)
  
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - Ex: summing elements of a length-$N$ array: 1-core vs 2-core system
  
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation. Different threads may use same data or distinct data
    - Ex: sorting and summing a length-$N$ array: 1-core vs 2-core system
  
  - **Hybrid Data-and-Task parallelism** in practice –
    - Ex: sorting and summing a length-$N$ array: 1-core, 2-core or 4-core system

- As # of threads grows, so does architectural support for threading
  
  - CPUs have cores as well as *hardware threads*
  
  - Modern Intel CPUs have two hardware threads, i.e. supports 2 threads/core
  
  - Oracle SPARC T4 CPU has 8 cores, with 8 hardware threads per core
User Threads and Kernel Threads

- Support for threads either at user level or kernel level

- **User threads** - management done by user-level threads library
  - Supports thread **programming**: creating and managing program threads
  - Three primary thread libraries: (threads are managed without kernel support)
    - POSIX Pthreads
    - Windows threads
    - Java threads

- **Kernel threads** - Supported by the Kernel.
  - Managed directly by the OS
  - Examples – virtually all general purpose operating systems, including:
    - Windows
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X
Multithreading Models

A relationship must exist between user threads and kernel threads

- Many-to-One
  - Many user-level threads mapped to a single kernel thread

- One-to-One
  - Each user-level thread maps to one kernel thread

- Many-to-Many
  - Allows many user-level threads to be mapped to many kernel threads
Many-to-One

- Many user-level threads mapped to single kernel thread
  - Efficiently managed by the thread library

- One thread blocking causes all to block
  - If the thread makes a blocking system-call

- Multiple threads are unable to run in parallel on multicore system because only one thread can be in kernel at a time
  - Does not benefit from multiple cores

- Few systems currently use this model

- Examples of many-to-one models:
  - Solaris Green Threads (adopted in early versions of Java thread library)
  - GNU Portable Threads
One-to-One

- Each user-level thread maps to one kernel thread

Problem: creating a user thread requires creating the corresponding kernel thread
  - Thread creations burden the performance of an application; an overhead

- Provides more concurrency than many-to-one model in case a thread has blocked, and allows multiple threads to run in parallel on multiple CPU systems

- Number of threads per process sometimes restricted due to overhead

- Examples of one-to-one models
  - Windows
  - Linux
  - Solaris 9 and later
Many-to-Many Model

- Allows $m$ user-level threads to be mapped to $n$ kernel threads; $n \leq m$

- User can create as many user threads as wished

- Allows the operating system to create a sufficient number of kernel threads to be allocated to applications

- Does not have the problems of other models

- Solaris prior to version 9

- Windows with the ThreadFiber package
Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to a kernel thread

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Thread library is entirely in user space with no kernel support
    - Codes and data structures for thread library are available to the user
    - Thread library functions are *not* system-calls
  - Kernel-level thread library is supported directly by the OS
    - Codes and data structures for thread library are *not* available to the user
    - Thread library functions are system-calls to the kernel
POSIX Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- Threads extensions of POSIX may be provided either as user-level or kernel-level

- **Specification**, not **implementation**

- API specifies behavior of the thread library, implementation is up to development of the library. *OS designers implement Pthreads specification in any way they wish*

- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
Pthreads Example

```c
#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;
    }
```
Pthreads Example

/* 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);
}
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
Windows Multithreaded C Program

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

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

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;
    }
```
Windows  Multithreaded C Program (Cont.)

/* 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);
}
}
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:
  
  ```java
  public interface Runnable
  {
    public abstract void run();
  }
  ```

  - Extending Thread class
  - Implementing the Runnable interface
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getS Um());
                } catch (InterruptedException ie) { }
            }
        } else {
            System.err.println("Usage: Summation <integer value>");
        }
    }
}
Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
  - Debugging an application containing 1000s of threads?

- **Solution = Implicit Threading**: Let compilers and runtime libraries create and manage threads rather than programmers and applications developers

- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package
Thread Pools

- Create a number of threads at process startup in a pool where they await work

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread. A thread returns to pool once it completes servicing a request.
  - Allows the number of threads in the application(s) to be bound to the size of the pool. Limits the number of threads that exist at any one point.
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically or after a time delay.

- Windows thread pool API supports thread pools:

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

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```c
#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 */

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

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

- **Block** specified by “^{ }” - ^{ printf("I am a block"); } 
  - Block = self-contained unit of work identified by the programmer as above

- Blocks placed in a dispatch queue
  - Assigned to available thread in thread pool when removed from queue
Grand Central Dispatch

- GCD schedules by placing them blocks on dispatch queue. Two types of 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 removed at a time
    - Three system wide concurrent queues with priorities: low, default, high

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

dispatch_async(queue, ^{ printf("I am a block."); });
```
Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7

- Implements the one-to-one mapping, kernel-level
  - App run as separate processes, each process may contain many threads

Each thread contains

- A thread id
- Register set representing state of processor
- Separate user and kernel stacks; for threads running in user or kernel mode
- Private data storage area used by run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the context of the thread
Windows Threads

The primary data structures of a thread include:

- ETHREAD (executive thread block) – includes: pointer to process to which thread belongs, and address of routine in which the thread starts control, and pointer to KTHREAD; exists in kernel space only

- KTHREAD (kernel thread block) – includes: scheduling and synchronization info, kernel-mode stack, pointer to TEB; exists in kernel space only

- TEB (thread environment block) – includes: thread id, user-mode stack, thread-local storage; exists in user space only
Windows Threads Data Structures

ETHREAD

- thread start address
- pointer to parent process

KTHREAD

- scheduling and synchronization information
- kernel stack

TEB

- thread identifier
- user stack
- thread-local storage

Kernel space

User space
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)
- Flags control behavior: determine what and how much to share

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>

`struct task_struct` points to process data structures (shared or unique)
End of Chapter 4
Amdahl’s Law

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

\[
speedup \leq \frac{1}{S + \frac{(1-S)}{N}}
\]

- 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?
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
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork

- `exec()` usually works as normal – replace the running process including all threads
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 by one of two signal handlers:
     1. default
     2. user-defined

- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process
Signal Handling (Cont.)

Where should a signal be delivered for multi-threaded?

- 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 Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread

Two general approaches:
- Asynchronous cancellation terminates the target thread immediately
- Deferred cancellation allows the target thread to periodically check if it should be cancelled

Pthread code to create and cancel a thread:

```c
pthread_t tid;

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

/* cancel the thread */
pthread_cancel(tid);
```
Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

<table>
<thead>
<tr>
<th>Mode</th>
<th>State</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Disabled</td>
<td></td>
</tr>
<tr>
<td>Deferred</td>
<td>Enabled</td>
<td>Deferred</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Enabled</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

- If thread has cancellation disabled, cancellation remains pending until thread enables it

- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked

- On Linux systems, thread cancellation is handled through signals
Thread-Local Storage

- **Thread-local storage (TLS)** 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).

- Different from local variables:
  - Local variables visible only during single function invocation.
  - TLS visible across function invocations.

- Similar to **static** data:
  - TLS is unique to each thread.
Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.

- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?

- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the upcall handler in the thread library

- This communication allows an application to maintain the correct number kernel threads.