Chapter 4
Threads

Thread of Control -- Fundamental Unit of CPU Utilization

<table>
<thead>
<tr>
<th>Process A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread1</td>
</tr>
<tr>
<td>• Thread ID (TID)</td>
</tr>
<tr>
<td>• PC</td>
</tr>
<tr>
<td>• Register Set</td>
</tr>
<tr>
<td>• Stack</td>
</tr>
<tr>
<td>Thread2</td>
</tr>
<tr>
<td>• Thread ID (TID)</td>
</tr>
<tr>
<td>• PC</td>
</tr>
<tr>
<td>• Register Set</td>
</tr>
<tr>
<td>• Stack</td>
</tr>
<tr>
<td>Thread3</td>
</tr>
<tr>
<td>• Thread ID (TID)</td>
</tr>
<tr>
<td>• PC</td>
</tr>
<tr>
<td>• Register Set</td>
</tr>
<tr>
<td>• Stack</td>
</tr>
</tbody>
</table>

Code Section | Data Section | Open Files | Signals

Single Threaded Process  
Traditional Process  
Heavyweight Process  

versus  
Multithreaded Process  

1:30

For each new process created by a single threaded process to service a need, a Solaris Operating System using a multithreaded process can create 30 new threads to serve as many new needs. Also for each context-switch between processes, Solaris can context-switch between 5 threads.

Most modern operating system kernels are multithreaded; each kernel thread performs a specific task, e.g., Solaris uses a set of threads to support interrupt handling. Also Linux uses a thread to manage free memory, i.e., keep track of all of the free memory available.

Benefits

- Responsiveness
  - even if a thread is suspended, e.g., I/O request, or performing a very long operation, e.g., loading an image, the other threads and hence the process, may keep running
- Resource Sharing
  - all the threads in a process share the same address space provided by the process
- Economy -- avoid additional and unnecessary
  - memory allocation
  - resource allocations
  - process context-switching
- Scalability – a Multithreaded Process running on a Multiprocessor Architecture
  - threads from one process can run on more than one processor
Multicore Programming -- O/S considers each core to be a separate processor

Concurrency

- Single core → execution of threads will be interleaved over time
- Multiple cores → system can assign a separate thread to each core → the threads can run simultaneously

Programming Requirements

- divide activities into separate, concurrent tasks
- balance tasks to ensure that they do work of equal value
- data accessed by, and possibly modified by, the separate tasks must be segregated into separate bundles to be run on the separate cores
- if data assigned to one task is used by another task, the execution of the tasks must be synchronized to accommodate the data dependency
- program running in parallel on multiple cores → many different execution paths → testing & debugging procedures and techniques must be reevaluated for effectiveness

User threads – provided & managed at the user level without kernel involvement

Kernel threads – provided & managed by the O/S

Windows XP
Linux
Mac OS/X
Solaris

Many-To-One Model

- maps many user threads to one kernel thread
- thread library in user space provides thread management -- efficient
- thread makes a blocking call → process is blocked
- only one thread can access the kernel at a time
  multiple threads cannot run in parallel on multiprocessors

GNU Portable Threads implements the many-to-one model

Solaris provides two different methods to implement threads, one of which is the Green Treads Library which implements the many-to-one model
One-To-One Model
• maps each user thread to a separate kernel thread
• thread makes a blocking call
    ➔
    process is only blocked if
    o it is a single threaded process, or
    o all the other threads are also blocked
• allows multiple threads from one process to run on multiple processors
• creating a USER THREAD requires creating A KERNEL THREAD
    ➔
    creating kernel threads burdens the application performance
    ➔
    most implementations of the one-to-one model restrict the number of kernel threads

Many-to-Many Model
• multiplexes many user threads to an equal or smaller set of kernel threads
• number of kernel threads may be specific to particular application or particular machine
• number of user threads created is not limited
• corresponding kernel threads may run in parallel on multiprocessors
• thread performs blocking system call
    ➔
    kernel can schedule another thread from the same process
    ➔
    process does not block ➔ process context-switch is not required

Two-Level Model
• variation of the many-to-many model
• multiplexes many user threads to an equal or smaller set of kernel threads
• allows the binding of a user thread to a kernel thread

Implemented in IRIX, HP-UX, Tru-64 UNIX
Thread Libraries
provides API's for creating & managing threads

Implementation

- User-space Library – no kernel support
  - library programs and data structures reside in user space
  - invoking a function in the API results in a LOCAL FUNCTION CALL to user space

- Kernel-level Library – supported directly by the O/S
  - library programs and data structures reside in kernel space
  - invoking a function in the API for the library results in a SYSTEM CALL to the kernel

POSIX Pthreads – may be implemented as either a user level library or a kernel level library
Solaris, Linux, Mac OS/X, Tru64 Unix

Win32 Thread Library is a kernel level library available only on Windows Operating Systems

Java Thread Library allows threads to be created and managed directly in Java programs; but the JVM runs either on a special Java chip or on top of the host O/S hence the Java Thread API generally implemented using the host system thread library

- Win32 API for a Windows Host System
- pthreads for a Unix, Linux or Mac OS/X Host System

Comparison of the following Thread libraries -- computation of sum as indicated below:

\[
sum = \sum_{i=0}^{N} i
\]
Pthreads
POSIX Standard (IEEE 1003.1c) Specification
Implementation is left to the discretion of the developers

/** Most Unix/Linux/OS X users 
 * gcc thrd.c -lpthread
 * Solaris users must enter 
 * gcc thrd.c -lpthreads
 */
* Figure 4.9
* @author Gagne, Galvin, Silberschatz
* Operating System Concepts – 8th Ed.
* Copyright John Wiley & Sons - 2009.
*/
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[]) /* CREATES A SINGLE THREAD FOR THE MAIN FUNCTION */
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread – STACK SIZE, SCHEDULING INFORMATION, etc */
    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>n"); /*exit(1);*/
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1])); /*exit(1);*/
        return -1;
    }
    /* get the default attributes */
    pthread_attr_init(&attr); /* attr retrieves default attributes; pthread_attr_init(&attr) sets attributes */
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]); /* CREATES A SECOND THREAD */
    /* now 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;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}

causes parent thread to wait for second thread to terminate, i.e., execute the pthread_exit(0) command
name of the procedure or function where the new thread will begin execution
argv[1] parameter passed on the command line
the main thread outputs the computed value
Win32 Threads

/** This program creates a separate thread using the CreateThread() system call.
 * Figure 4.10
 * @author Gagne, Galvin, Silberschatz
 * Operating System Concepts - Eighth Edition
 * Copyright John Wiley & Sons - 2009.
 */

#include <stdio.h>
#include <windows.h>

DWORD Sum; /* global declaration of data to be shared by the thread(s) – unsigned 32-bit integer */

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

int main(int argc, char *argv[])
{
    DWORD ThreadId; /* unique id for the thread throughout the o/s */
    HANDLE ThreadHandle; /* a reference to a kernel thread structure */
    int Param;
    // do some basic error checking
    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, 0, Summation, &Param, 0, &ThreadId);
    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n", Sum);
    }
    return 0;
}
Java Threads

Java supports two independent procedures for creating Java threads

- create a new class derived from the Thread class and override its run() method
- define a class that implements the Runnable interface

```java
// This program creates a separate thread by implementing the Runnable interface.
/* Figure 4.11
* @author Gagne, Galvin, Silberschatz Operating System Concepts - 8th Ed.
* Copyright John Wiley & Sons - 2009.
*/
class Sum
{
    private int sum;
    public int get() { return sum; }
    public void set(int sum) { this.sum = sum; }
}

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

    Summation(int upper, Sum sumValue)
    {
        if (upper < 0)
            throw new IllegalArgumentException();
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run()
    {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.set(sum);
    }

    public class Driver
    {
        public static void main(String[] args)
        {
            if (args.length != 1)
                {
                    System.err.println("Usage Driver <integer> ");
                    System.exit(0);
                }
            Sum sumObject = new Sum();
            int upper = Integer.parseInt(args[0]);
            Thread worker = new Thread(new Summation(upper, sumObject));
            worker.start();
            try { worker.join(); } catch (InterruptedException ie) { }
            System.out.println("The sum of " + upper + " is " + sumObject.get());
        }
    }
}
```

Java shares data between the main thread in the Driver (main) class and the worker thread in the Summation class via the Sum class which includes an accessor method, i.e., get() method, and a mutator method, i.e., set() method, for just such a purpose.

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

```
definition of Summation constructor
run() overrides the default run() method, e.g., Runnable interface run()
```

```
start() method execution
- allocates memory in kernel space
- initializes the new thread in the kernel
- calls the run method directly, hence making the thread eligible to be run in the JVM
Java user code never directly calls the run() method
```

```
first thread created
second thread defined
second thread created & started
main thread waits for the termination of the second thread
first thread outputs the value
```
Thread Issues

- if a single thread in multithreaded process issues fork( ) system call, should the new process
  - duplicate all the threads of the parent process and thus be multithreaded
  - or duplicate only the single thread and thus be single threaded

<table>
<thead>
<tr>
<th>Two Versions of fork( )</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g., one version duplicates only the thread that invoked the fork( ) call</td>
<td></td>
</tr>
<tr>
<td>the other duplicates all the threads of the parent process</td>
<td></td>
</tr>
</tbody>
</table>

- thread cancellation
  - asynchronous cancellation
    - downloading images on the web
    - may leave some resources un-reclaimed
  - deferred cancellation – synchronous cancellation
    - notice issued of target thread cancellation, e.g., cancellation flag is set
    - cancellation occurs after target thread checks the cancellation flag
    - check is performed at cancellation point
    - cancellation point – point of execution where thread may be safely cancelled

Signals

- Synchronous Signals
  - process operation causes signal
  - signal delivered to the process
- Asynchronous Signals
  - signal generated by an event external to the running process

Signal Handlers

- Default
- User Defined

Multithreaded Processes – deliver the signal to

- the thread to which the signal applies
- every thread in the process
- a specified set of threads
- designated thread that handles all signals for the process

Unix

allow threads to block signals
signal is delivered to the first thread not blocking it
command delivering a Unix signal kill(pid_t tid, int signal)

Windows

Asynchronous Procedure Calls (APC)

- user thread specifies a function to call when the user thread receives notification of a particular event
- notification delivered to a particular thread via the function
Thread Pools
- limit the total number of threads in system at any one time
- service requests with existing threads from the thread pool
- eliminate constant thread creation & deletion overhead
- when pool is empty, stack incoming thread requests; service those requests as threads become available
- dynamically adjust the number of threads in the pool depending upon traffic demands

Thread Specific Data
- one transaction
- one transaction identifier
- one set of transaction data
- one thread

Scheduler Activations
- Many-to-Many Model
- Two-Level Model -- binding user thread to kernel thread

Light Weight Process LWP
User library ➔ LWP is virtual processor, i.e.,
user library can schedule user threads on the LWP
- each LWP is attached to a kernel thread
- If the kernel thread blocks ➔ LWP block ➔ user thread blocks
- efficient execution requires that there be one LWP for each potential, concurrent blocking call

- kernel provides to each process a set of LWP’s
- each process schedules threads on any available LWP in the set

Upcall
- kernel informs the process about certain events, e.g., impending blocks of specific thread

Upcall Handler
- allocates a new LWP to the process
- saves the state of the blocking thread
- schedules an eligible thread to run on the new LWP,
  hence it allows the process to continue
- when the blocking event ends, an upcall is made to the thread library informing it that the thread is now eligible to run
- allocates another new LWP to the process
- process can schedule an eligible thread to execute
**Windows XP Threads**
- Win32 API → One-to-One Model
- Fiber Library → functionality of the Many-to-Many Model

**Thread Components**
- tid
- register set – processor status
- user stack – used when thread is in user mode
- kernel stack – used when thread is in kernel mode
- private storage area – private data – used by
  - run-time libraries
  - dynamic link libraries DLL

**Thread Data Structures**

**Executive Thread Block**
- ETHREAD
  - thread start address
  - Pointer to parent process

**Kernel Thread Block**
- KTHREAD
  - scheduling & synchronization information
  - kernel stack

**Thread Environmental Block**
- TEB
  - thread identifier
  - user stack
  - thread local storage

thread context

kernel space

user space
Linux Threads

- task
  - process
  - thread
- fork() creates new process
- clone() creates new entity which varies between a new process and a new thread
  - flags
    - CLONE_FS share file-system information
    - CLONE_VM share memory space
    - CLONE_SIGHAND share signal handlers
    - CLONE_FILES share open files
- all flags set ➞ new process is created
- no flags set ➞ new thread is created
- task data structure, i.e., struct task_struct

fork()
  - create new task data structure, i.e., struct task_struct
  - copy all data from original structure to new structure
  - create new structures for open files, signal handling information, virtual memory
  - copy information from old structures to the new structures

clone()
  - create new task data structure, i.e., struct task_struct
  - depending upon what flags are set, use pointers in the new task structure to
    ★ reference the old structures
    ★ create new structures
      - copy the data from the old structures to the new structures
      - set the task pointers to reference the new structures

Native POSIX Thread Library (NPTL)
- POSIX-compliant Thread Model
- thread creation cheaper than task creation