操作系统原理与设计

第4章 Threads 1（线程1）

陈香兰

中国科学技术大学计算机学院

October 28, 2009

提纲

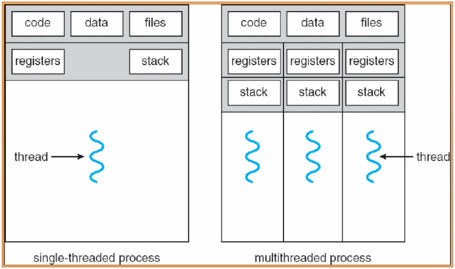
Outline

Thread concept I

- A thread is a basic unit of CPU utilization
    - a thread ID
    - a program counter
    - a register set
    - and a stack
- It shares with other threads belonging to the same process
    - code section
    - data section
    - and other OS resources
        - open files, signals, etc

Thread concept II

- Single threaded VS. Multithreaded processes



Motivation I

- On modern desktop PC, **many APPs are multithreaded**.
  - **a seperate process + several threads**
- Example 1: A web browser
  - one for displaying images or text;
  - another for retrieving data from network
- Example 2: A word processor
  - one for displaying graphics;
  - another for responding to keystrokes from the user;
  - and a third for performing spelling & grammer checking in the background

Motivation II

- **Motivation**, think about
  - a web server,
  - an RPC server
  - and Java's RMI systems
- **PARTICULAR, many OS systems are now multithreaded.**
  - Solaris, Linux(伪)

Benefits

1. **Responsiveness (响应度高)**
    - Example: an interactive application such as web browser, while one thread loading an image, another thread allowing user interaction
2. **Resource Sharing**
    - address space, memory, and other resources
3. **Economy**
    - Solaris:
      creating a process is about 30 times slower then creating a thread;
      context switching is about 5 times slower
4. **Utilization of MP Architectures**
    - parallelism and concurrency ↑

Outline

Two Methods I

- Two methods to **support** threads
  - User threads
  - Kernel threads
- **User threads**
  - Thread management done by **user-level threads library** without kernel support
    - Kernel may be multithreaded or not.
  - **Three primary thread libraries**:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

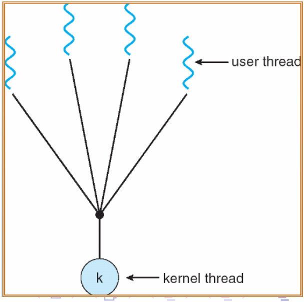
Two Methods II

- **Kernel Threads**
  - Supported by the Kernel, usually may be slower then user thread
  - Examples
    - Windows XP/2000
    - Solaris
    - Linux (伪)
    - Tru64 UNIX (formerly Digital UNIX)
    - Mac OS X

Multithreading Models I

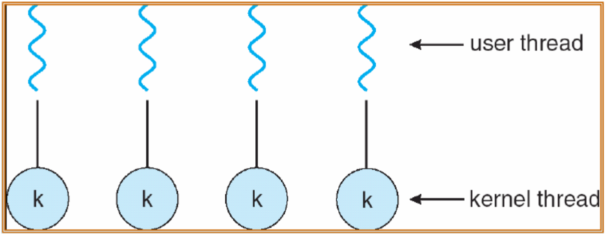
- The relationship between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

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



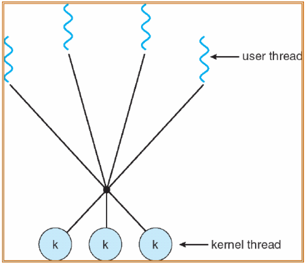
Multithreading Models II

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



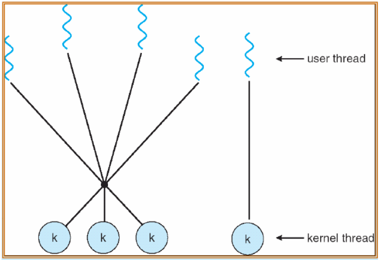
Multithreading Models III

- **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
  - Examples
    - Solaris prior to version 9
    - Windows NT/2000 with the ThreadFiber package



Multithreading Models IV

- **Two-level Model**, a popular variation on many-to-many 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



Outline

Thread Libraries

- A thread library provides the programer an API for creating and managing threads.
- Two primary ways
  1. to provide a library **entirely in user space** with no kernel support
  2. to implement a **kernel-level library** supported directly by the OS

library	code & data	API	invoking method inside API
user-level	entirely in user space	user space	a local function call
kernel-level	kernel space	user space	system call

- **Three main thead libraries**
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Pthreads

- **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 OSes (Solaris, Linux, Mac OS X)

Multithreaded C program using the Pthreads API I

```
#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[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */

    if (argc != 2)
    {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
```

Multithreaded C program using the Pthreads API II

```
    if (atoi(argv[1]) < 0)
    {
        fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
        return -1;
    }

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

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

Multithreaded C program using the Pthreads API III

```
/* 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);
}
```

pthread_attr_init

NAME

pthread_attr_init, pthread_attr_destroy - initialise and destroy threads attribute object

SYNOPSIS

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

DESCRIPTION

The function pthread_attr_init() initialises a thread attributes object attr with the default value for all of the individual attributes used by a given implementation.

. . .

The pthread_attr_destroy() function is used to destroy a thread attributes object.

RETURN VALUE

Upon successful completion, both return a value of 0.
Otherwise, an error number is returned to indicate the error.

. . .

pthread_create()

NAME

pthread_create - thread creation

SYNOPSIS

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);
```

DESCRIPTION

The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. . . . Upon successful completion, pthread_create() stores the ID of the created thread in the location referenced by thread.

The thread is created executing start_routine with arg as its sole argument. . . .
. . .

If pthread_create() fails, no new thread is created and the contents of the location referenced by thread are undefined.

RETURN VALUE

If successful, the pthread_create() function returns zero.
Otherwise, an error number is returned to indicate the error.

pthread_join

NAME

pthread_join - wait for thread termination

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);

DESCRIPTION

The pthread_join() function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. . . .

The results of multiple simultaneous calls to pthread_join() specifying the same target thread are undefined. . . .

RETURN VALUE

If successful, the pthread_join() function returns zero.

Otherwise, an error number is returned to indicate the error.

. . .

pthread_exit

NAME

pthread_exit - thread termination

SYNOPSIS

#include <pthread.h>

void pthread_exit(void *value_ptr);

DESCRIPTION

The pthread_exit() function terminates the calling thread and makes the value value_ptr available to any successful join with the terminating thread. ...

...

RETURN VALUE

The pthread_exit() function cannot return to its caller.

Win32 Threads I

- **Similar** to the Pthreads technique.
- Multithreaded C program using the Pthreads API

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

/* 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;
}
```

Win32 Threads II

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

Win32 Threads III

```
// create the thread
ThreadHandle = CreateThread(NULL,    //default security attribute
                 0,    //default stack size
                 Summation,    //thread function
                 &Param,    //parameter to thread function
                 0,    //default creation flags
                 &ThreadId);
if (ThreadHandle != NULL)
{
   WaitForSingleObject(ThreadHandle, INFINITE);
   CloseHandle(ThreadHandle);

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

Java Threads

- **Threads are the fundamental model** for program execution in a Java program.
- Java threads may be created by:
  - Extending Thread class
    - to create a new class that is derived from the Thread class and override its run() method.
  - Implementing the Runnable interface Java

Example I

```
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;
```

Example II

```
    public 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) {
```

Example III

```
        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());
    }
}
```

Outline

Threading Issues I

- **Semantics of fork() and exec() system calls**
  - Does fork() duplicate only the calling thread or all threads?
  - Some UNIX system have chosen to have two versions
  - Which one version to use? Depend on the APP.
- **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

Threading Issues II

- **Signal Handling**
  - Signals are used in UNIX systems to notify a process that a particular event has occurred :
    - **Synchronous**: illegal memory access, division by 0
    - **Asynchronous**: Ctrl+C
  - All signals follow the same pattern:
    1. Signal is **generated** by particular event
    2. Signal is **delivered** to a process
    3. Signal is **handled**
  - Signal **handler** may be handled by
    - a **default** signal handler
    - a **user-defined** signal handler
  - When multithread, **where should a signal be delivered**?

Threading Issues III

- 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 threa 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

Threading Issues IV

- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

- **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

Outline

小结

谢谢！