Chapter 4: Threads

肖 卿 俊
办公室：九龙湖校区计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：https://csqjxiao.github.io/PersonalPage
电话：025-52091022
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
What is a thread?

A *thread*, also known as *lightweight process* (LWP), is a basic unit of CPU execution.

A thread has a thread ID, a program counter (instruction pointer), a register set, and a stack. Thus, it is similar to a process has.

However, a thread *shares* with other threads in the *same* process its code section, data section, and other OS resources (e.g., files and signals).

A process, or heavyweight process, has a *single* thread of control after its creation.
Single and Multithreaded Processes

Threads in a same process are ________
- tightly coupled or
- loosely coupled?

How is the heap space shared among threads?
### Items shared by all threads in a process

- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

### Items private to each thread

- Program counter
- Registers
- Stack
- State
Thread Usage (1)

A word processor with three threads

MVC paradigm
Thread Usage (2)

A multithreaded Web server
Thread Usage (3)

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread
Note: An Event-Driven Framework

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page)
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures
Economy for Creation

- Compare timing of fork() and pthread_create()

  Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th></th>
<th>pthread_create()</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
<td>real</td>
</tr>
<tr>
<td>AMD 2.4 GHz Opteron (8cpus/node)</td>
<td>41.07</td>
<td>60.08</td>
<td>9.01</td>
<td>0.66</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8cpus/node)</td>
<td>64.24</td>
<td>30.78</td>
<td>27.68</td>
<td>1.75</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8cpus/node)</td>
<td>104.05</td>
<td>48.64</td>
<td>47.21</td>
<td>2.01</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus/node)</td>
<td>54.95</td>
<td>1.54</td>
<td>20.78</td>
<td>1.64</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus/node)</td>
<td>54.54</td>
<td>1.07</td>
<td>22.22</td>
<td>2.03</td>
</tr>
</tbody>
</table>

Economy for Context Switching

- Process (notes: Process Control Block in OS Kernel)
- Light-weight Process and Kernel Threads
- User Threads

Lower Cost in Creation and Context Switching
User Threads

- Thread management done by user-level threads library
  - Context switching of threads in the same process is done in user mode

- Examples
  - POSIX Pthreads (see scope parameter of pthread_create: PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM)
  - Mach C-threads
  - Solaris UI-threads

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#CREATIONTERMINATION
User Threads (Cont.)

A user-level thread library provides all support for thread creation, termination, joining, and scheduling.

A user-level threads package
Pros and Cons of User Threads

- User threads are supported at the user level. The kernel is not aware of user threads.

- Because there is no kernel intervention, user threads are usually more efficient.

- Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, each other threads of the same process are also blocked since the containing process is blocked.

Question: Can two user threads in a same process run simultaneously on two different CPU cores?
Kernel Threads

- Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux and POSIX Thread
Kernel Threads (Cont.)

Kernel threads are directly supported by the kernel. The kernel does thread creation, termination, joining, and scheduling in kernel space.

Kernel threads are usually slower than the user threads.

However, blocking one thread will not cause other threads of the same process to block. The kernel simply runs other threads.

In a multiprocessor environment, the kernel can schedule threads on different processors.
Implementing Threads in the Kernel

A threads package managed by the kernel (Note: POSIX Pthreads library supports the creation of kernel threads)

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API
Multithreading Models

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

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

- Used on systems that do not support kernel threads.
Many-to-One Model (Cont.)
One-to-One

- Each user-level thread maps to kernel thread

Examples
- Windows 95/98/NT/2000
- OS/2
One-to-one Model (Cont.)

![Diagram of one-to-one model with OS, CPU, scheduler, and processes with threads.]
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.

- Windows NT/2000 with *ThreadFiber* package
- Solaris 2
Many-to-Many Model (Cont.)
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API
Solaris 2 Threads

- Task 1
- Task 2
- Task 3

- User-level thread
- Lightweight process

- Kernel thread
- Kernel

- CPU
Solaris Process

- process id
- memory map
- priority
- list of open files

LWP1 → LWP2 → LWP3 → ...
Windows XP Threads

- Implements the one-to-one mapping.

- Each thread has a corresponding thread control block in kernel, which contains:
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area
Thread Block

ETHREAD

- KTHREAD
  - Create and Exit Time
  - Process ID
  - Thread Start Address
  - Impersonation Information
  - LPC Message Information
  - Timer Information
- EPROCESS
- Access Token
- Pending I/O Requests

KTHREAD

- Dispatcher Header
- Total User Time
- Total Kernel Time
- Thread Scheduling Information
- Trap Frame
- Synchronization Information
- List of Pending APCs
- Timer Block and Wait Blocks
- List of Objects Being Waiting On
- Kernel Stack Information
- System Service Table
- Thread Local Storage
- TEB

Operating System Concepts 4.30 Southeast University
Linux Threads
(not POSIX pthreads Library)

- 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).

- What is the difference between fork() and clone()? 
  
  http://linux.die.net/man/2/clone
Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

- Java threads are managed by the JVM.
Java Thread States

- **new**
  - **Runnable**
    - **Blocked**
      - **Dead**
        - **Start()**
          - **Sleep()**
            - **Suspend()**
              - **I/O**
                - **Resume()**
                  - **Stop()**
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API
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

POSIX 1003.1 Commands: http://www.unix.com/man-page-posix-repository.php

- Common in UNIX operating systems.
- Implemented over Linux operating system by Native POSIX Thread Library (NPTL)
  - NPTL is a 1×1 threads library, in that threads created by the user are in 1-1 correspondence with schedulable entities (i.e., task) in the kernel

https://en.wikipedia.org/wiki/Native_POSIX_Thread_Library
pthread_create

int pthread_create(tid, attr, function, arg);

- pthread_t * tid
  ◆ handle or ID of created thread
- const pthread_attr_t * attr
  ◆ attributes of thread to be created
- void *(*function) (void*)
  ◆ function to be mapped to thread
- void *arg
  ◆ single argument to function
- Integer return value for error code
pthread_create explained

spawn a thread running the function;
thread handle returned via pthread_t structure

- specify NULL to use default attributes
- If no argument to function, specify NULL

check error codes!

EAGAIN – insufficient resources to create thread
EINVAL – invalid attribute
Threads states

- pthread threads have two states
  - joinable and detached

- threads are joinable by default
  - Resources are kept until `pthread_join`.
  - When a joinable thread terminates, some of the thread resources are kept allocated, and released only when another thread performs `pthread_join` on that thread.
  - can be reset with attribute or API call

- detached thread can not be joined
  - resources can be reclaimed at termination
  - cannot reset to be `joinable`
Waiting for a thread

```c
int pthread_join(tid, val_ptr);
```

- `pthread_t *tid`
  - handle of joinable thread

- `void **val_ptr`
  - exit value returned by joined thread
 pthread_join explained

calling thread waits for the thread with handle tid to terminate

- only one thread can be joined
- thread must be joinable

exit value is returned from joined thread

- Type returned is (void *)
- use NULL if no return value expected

ESRCH – thread not found

EINVAL – thread not joinable
Example 1

Q1: Guess what are the possible outputs?

Q2: What if we remove the two Pthread_join() function calls?
Note: the termination of main thread will cause the automatic termination of children threads.
Example 2

```c
volatile int counter = 0; // shared global variable

void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Q1: Guess what is the possible output
Discussion

- Why not deterministic?
- The Heart Of The Problem: Uncontrolled Scheduling
- What happens when executing “counter = counter + 1;”? 
- Understand the code sequence that the compiler generates for the update to counter.

```plaintext
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

- Now, you may tell the reason
# Uncontrolled Scheduling

## OS

<table>
<thead>
<tr>
<th>OS</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>(after instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>before critical section</td>
<td></td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>mov 0x8049a1c, %eax</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>add $0x1, %eax</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>108</td>
</tr>
</tbody>
</table>

**interrupt**

- save T1’s state
- restore T2’s state

|    | mov 0x8049a1c, %eax | 100 | 0 | 50 |
|    | add $0x1, %eax | 105 | 50 | 50 |
|    | mov %eax, 0x8049a1c | 108 | 51 | 50 |
|    |                  | 113 | 51 | 51 |

**interrupt**

- save T2’s state
- restore T1’s state

|    | mov %eax, 0x8049a1c | 108 | 51 | 51 |
|    |                  | 113 | 51 | 51 |
Uncontrolled Scheduling

Race condition

- Several processes (threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.
- Result indeterminate.

Critical section

- Multiple threads executing a segment of code, which can result in a race condition.

What we want: Mutual exclusion

- The property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.
Revisit the Threading Model

- “Data” is a public memory segment shared by all threads, which may incur race condition.
- Stack is a private memory segment of a thread.
- Question: What if a thread accesses the data variables on the stack of another thread?
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Thread APIs
Windows Thread APIs

- CreateThread
- ExitThread
- TerminateThread
- GetExitCodeThread

- GetCurrentThreadId - returns global ID
- GetCurrentThread - returns handle
- SuspendThread/ResumeThread
- GetThreadTimes
Windows API Thread Creation

HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm,
    DWORD fdwCreate,
    LPDWORD lpIDThread)

* cbStack == 0: thread’s stack size defaults to primary thread’s size

- lpStartAddr points to function declared as
  
  \[
  \text{DWORD WINAPI ThreadFunc(LPVOID)}
  \]

- lpvThreadParm is 32-bit argument

- lpIDThread points to DWORD that receives thread ID non-NULL pointer!
Windows API Thread Termination

**VOID ExitThread( DWORD devExitCode )**

- When the last thread in a process terminates, the process itself terminates

**BOOL GetExitCodeThread (HANDLE hThread, LPDWORD lpdwExitCode)**

- Returns exit code or STILL_ACTIVE
Suspending and Resuming Threads

- Each thread has suspend count
- Can only execute if suspend count == 0
- Thread can be created in suspended state

```c
DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)
```

- Both functions return suspend count or 0xFFFFFFFF on failure
Example: Thread Creation

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

DWORD WINAPI helloFunc(LPVOID arg) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hThread = CreateThread(NULL, 0, helloFunc,
                                 NULL, 0, NULL);
}
```

What’s Wrong?
Example Explained

- Main thread is process
- When process goes, all threads go
- Need some methods of waiting for a thread to finish
Waiting for Windows* Thread

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

BOOL thrdDone = FALSE;

DWORD WINAPI helloFunc(LPVOID arg) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hThread;
    CreateThread(NULL, 0, helloFunc, arg, 0, &hThread);
    while (!thrdDone);
}
```

Not a good idea!
Waiting for a Thread

Wait for one object (thread)

```c
DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds);
```

Calling thread waits (blocks) until
- Time expires
  - Return code used to indicate this
- Thread exits (handle is signaled)
  - Use INFINITE to wait until thread termination

Does not use CPU cycles
Waiting for Many Threads

Wait for up to 64 objects (threads)

```
DWORD WaitForMultipleObjects(
    DWORD nCount,
    CONST HANDLE *lpHandles, // array
    BOOL fWaitAll, // wait for one or all
    DWORD dwMilliseconds)
```

Wait for all: fWaitAll==TRUE

Wait for any: fWaitAll==FALSE

- Return value is first array index found
Notes on WaitFor* Functions

- Handle as parameter
- Used for different types of objects
- Kernel objects have two states
  - Signaled
  - Non-signaled
- Behavior is defined by object referred to by handle
  - Thread: signaled means terminated
Example: Waiting for multiple threads

```c
#include <stdio.h>
#include <windows.h>
const int numThreads = 4;

DWORD WINAPI helloFunc(LPVOID arg) {
    printf("Hello Thread\n");
    return 0;
}

main() {
    HANDLE hThread[numThreads];
    for (int i = 0; i < numThreads; i++)
        hThread[i] =
            CreateThread(NULL, 0, helloFunc, NULL, 0, NULL);
    WaitForMultipleObjects(numThreads, hThread,
                           TRUE, INFINITE);
}
```
Example: HelloThreads

- Modify the previous example code to print out
  - appropriate “Hello Thread” message
  - Unique thread number
    - use for-loop variable of CreateThread loop

- Sample output:

```plaintext
Hello from Thread #0
Hello from Thread #1
Hello from Thread #2
Hello from Thread #3
```
```c
DWORD WINAPI threadFunc(LPVOID pArg) {
    int* p = (int*)pArg;
    int myNum = *p;
    printf( "Thread number %d\n", myNum);
}

... // from main():
for (int i = 0; i < numThreads; i++) {
    hThread[i] =
        CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
}
```

What is printed for myNum?
## Hello Threads Timeline

<table>
<thead>
<tr>
<th>Time</th>
<th>main</th>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$i = 0$</td>
<td>---</td>
<td>----</td>
</tr>
<tr>
<td>$T_1$</td>
<td><code>create(&amp;i)</code></td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$i++ (i == 1)$</td>
<td><code>launch</code></td>
<td>---</td>
</tr>
<tr>
<td>$T_3$</td>
<td><code>create(&amp;i)</code></td>
<td>$p = pArg$</td>
<td>---</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$i++ (i == 2)$</td>
<td><code>myNum = *p</code></td>
<td><code>launch</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>myNum = 2</code></td>
<td></td>
</tr>
<tr>
<td>$T_5$</td>
<td><code>wait</code></td>
<td><code>print(2)</code></td>
<td>$p = pArg$</td>
</tr>
<tr>
<td>$T_6$</td>
<td><code>wait</code></td>
<td><code>exit</code></td>
<td><code>myNum = *p</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>myNum = 2</code></td>
</tr>
</tbody>
</table>
Race Conditions

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Most common error in concurrent programs
- May not be apparent at all times
- How to avoid data races?
  - Local storage
  - Control shared access with critical regions
DWORD WINAPI threadFunc(LPVOID pArg)
{
    int myNum = *((int*)pArg);
    printf( "Thread number %d\n", myNum);
}
...

// from main():
for (int i = 0; i < numThreads; i++) {
    tNum[i] = i;
    hThread[i] = CreateThread(NULL, 0, threadFunc, &tNum[i], 0, NULL);
}
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Pthreads
- Windows Threads API
Threading Issues

1. Semantics of fork() and exec() system calls.
2. Thread cancellation.
3. Signal handling
4. Thread pools
5. Thread specific data
6. Scheduler Activations
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
- In a Pthreads-compliant implementation, the fork() call always creates a new child process with a single thread, regardless of how many threads its parent may have had at the time of the call.
- Furthermore, the child's thread is a replica of the thread in the parent that called fork.
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

✓ The point a thread can terminate itself is a cancellation point.
With **asynchronous cancellation**, if the target thread owns some system-wide resources, the system may not be able to reclaim all resources.

With **deferred cancellation**, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.

Most systems implement asynchronous cancellation for processes (e.g., use the `kill` system call) and threads.

**Pthread** supports **deferred cancellation**.
Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.

- All signals follow the same pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled

- A **signal handler** is used to process signals.
Signal Handling (Cont.)

- How to handle a signal when its target process has multiple threads?

- Options:
  1. Deliver the signal to the thread to which the signal applies
  2. Deliver the signal to every thread in the process
  3. Deliver the signal to certain threads in the process
  4. 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

https://en.wikipedia.org/wiki/Thread_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)
- Pthreads library supports thread specific data

https://en.wikipedia.org/wiki/Thread-local_storage#Pthreads_implementation
Thread Scheduler Activations

**Background:** Server-version operating systems often use many-to-many and two-level thread models.

- The thread library needs to maintain the appropriate number of kernel threads allocated to the process.
- Requires kernel-user space communication to do it.
Thread Scheduler Activations

- Scheduler activations provide **upcalls**: a communication mechanism from the kernel to the user-mode thread library.
- When the kernel knows a thread has blocked/resumed, it notifies the process’ run-time system about this event.
- This communication allows an application to maintain the correct number of available kernel threads.