Chapter 4: Threads
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.
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**
  - **Per process items**
    - Address space
    - Global variables
    - Open files
    - Child processes
    - Pending alarms
    - Signals and signal handlers
    - Accounting information

- **Items private to each thread**
  - **Per thread items**
    - Program counter
    - Registers
    - Stack
    - State
Thread Usage (1)

A word processor with three threads
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

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

```c
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
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></th>
<th>pthread_create()</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
<td>real</td>
<td>user</td>
<td>sys</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>
<td>0.19</td>
<td>0.43</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>
<td>0.69</td>
<td>1.10</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>
<td>1.00</td>
<td>1.52</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>
<td>0.67</td>
<td>0.90</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>
<td>1.26</td>
<td>0.67</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
A user-level thread library provides all support for thread creation, termination, joining, and scheduling.
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?
What is Coroutine?

Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.


```plaintext
var q := new queue

coroutine produce loop
    while q is not full
        create some new items
        add the items to q
    yield to consume

coroutine consume loop
    while q is not empty
        remove some items from q
        use the items
    yield to produce
```
协程（Coroutine）是什么？

协程这个概念近年流行起来。尤其 golang 语言问世之后，内置的协程特性，完全屏蔽了操作系统线程的复杂细节；甚至 go 开发者“只知有协程，不知有线程”

对称协程的执行模型类似于用户态线程

解决问题：

◆ 编写高性能网络服务器程序从来都不是件容易的事情。Linux底层IO框架epoll/kqueue等提供非阻塞、异步编程接口；但事件驱动框架的编程复杂

◆ 协程并发模式可利用同步阻塞式IO 接口，极大简化服务器端的十/百万高并发网络IO的开发过程
Libco: 微信后台大规模使用的C/C++协程库

2013年起，稳定运行在微信后台数万台机器上

Libco特性

- 无需侵入业务逻辑，把多进程、多线程服务改造成协程服务，并发能力得到百倍提升；
- 类pthread接口设计，通过co_create、co_resume等简单清晰接口即可完成协程的创建与恢复；
- 可选的共享栈模式，单机轻松接入千万连接(New)；
- 基于epoll/kqueue实现的小而轻的网络框架，基于时间轮盘实现的高性能定时器；
- 库里面提供了socket族函数的hook，使得后台逻辑服务几乎不用修改逻辑代码就可以完成异步化改造；

https://github.com/Tencent/libco
其他的C++开源协程库

玩具级：实现协程调度，无需用户手动处理协程上下文切换；特点没有HOOK；代表作：libmill

工业级：以部分正确的方式HOOK了网络io相关的syscall，可以少改甚至不改代码的兼容大多数第三方库；特点：没有完整生态，代表作：腾讯libco

框架级：以100%行为模拟的方式HOOK了网络io相关的syscall，可以完全不改代码兼容大多数第三方库；依照专为协程而生的语言的使用经验，提供了协程开发所必须的完整生态；代表作：魅族libgo

语言级：语言级的协程实现；代表作：golang语言；这一层次的协程库，开发者的一切行为都是受限行为，可以实现无死角的完善的协程
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.)

Diagram showing the Many-to-One Model in operating system concepts. The diagram illustrates the relationship between the OS, CPU, Scheduler, Process, and threads.
One-to-One

Each user-level thread maps to kernel thread

Examples

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

OS

CPU  CPU  CPU

Scheduler

Process  Process  Process  Process

threads  threads  threads  threads

USER
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.)

The diagram illustrates the Many-to-Many model in an operating system (OS) context. The OS has multiple CPUs, represented as nodes, which are connected to a central scheduler. The scheduler allocates threads to processes, which are connected to user applications. Each process can have multiple threads, indicated by the arrows pointing from the scheduler to the threads within the processes. This model allows for efficient resource utilization and task management in the system.
Chapter 4: Threads

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

Diagram showing the relationship between tasks, threads, and processes in Solaris 2. The diagram illustrates how multiple tasks can have threads, and these threads can be lightweight processes. The kernel thread is shown at the bottom, with CPU access points at the bottom as well. The user-level thread and lightweight process concepts are also indicated.
Solaris Process

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

LWP_1 → LWP_2 → LWP_3 → ...

Solaris process
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
- Pending I/O Requests

EPROCESS

Access Token

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
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**
  - Transition occurs with `start()` method.

- **runnable**
  - Transition occurs with `sleep()` and `suspend()` methods.
  - Transition occurs with `stop()` method.

- **blocked**
  - Transition occurs with `resume()` method.

- **dead**
  - Transition occurs with `suspend()` method.

States: new, runnable, blocked, dead
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 $1 \times 1$ 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

```c
int pthread_create(pthread_t * tid, const pthread_attr_t *attr, void *(*function) (void*) , void *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

- a single argument sent to function
- 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;
}
```

// The volatile keyword forces the compiler to always read the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested.

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.

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

- Now, you may tell the reason
## Uncontrolled Scheduling

<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>
<tr>
<td>interrupt</td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>save T1’s state</td>
<td></td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>restore T2’s state</td>
<td></td>
<td></td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>mov 0x8049a1c, %eax</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>interrupt</td>
<td></td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>save T2’s state</td>
<td></td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>restore T1’s state</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov %eax, 0x8049a1c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
用gcc -S命令简单验证一下

GCC的选项-S使GCC在执行完汇编后停止

$ gcc -S t1.c -o t1.s // 汇编代码
$ gcc -c t1.s -o t1.o // 二进制代码
$ ld t1.o -o t1 // 链接后可执行代码

看t1.s汇编代码

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

### @main
```
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;
}
```

```
_mythread:
    .cfi_startproc
    ...
    movl    _counter(%rip), %eax
    addl    $1, %eax
    movl    %eax, _counter(%rip)
    ...
    .cfi_endproc
    ## -- End function
    ## @counter
```

```
_main:
    .cfi_startproc
    ...
    .cfi_endproc
    .section __TEXT,__cstring,cstring_literals
    ...
    .globl _counter  ## @counter
```
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?

![Diagram showing the comparison of stack frames and stack states.](image)
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

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

- `lpStartAddr` points to function declared as `DWORD WINAPI ThreadFunc(LPVOID)`
- `lpvThreadParm` is 32-bit argument
- `lpIDThread` points to DWORD that receives thread ID non-NULL pointer!

- `cbStack == 0`: thread's stack size defaults to primary thread's size
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

DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)

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

#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, NULL, NULL, NULL);

    thrdDone = TRUE;

    while (!thrdDone);
}
```

Not a good idea!

while (!thrdDone);
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:

```
Hello from Thread #0
Hello from Thread #1
Hello from Thread #2
Hello from Thread #3
```
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>T₀</td>
<td>i = 0</td>
<td>---</td>
<td>----</td>
</tr>
<tr>
<td>T₁</td>
<td>create(&amp;i)</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>T₂</td>
<td>i++ (i == 1)</td>
<td>launch</td>
<td>---</td>
</tr>
<tr>
<td>T₃</td>
<td>create(&amp;i)</td>
<td>p = pArg</td>
<td>---</td>
</tr>
<tr>
<td>T₄</td>
<td>i++ (i == 2)</td>
<td>myNum = *p</td>
<td>launch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>myNum = 2</td>
<td></td>
</tr>
<tr>
<td>T₅</td>
<td>wait</td>
<td>print(2)</td>
<td>p = pArg</td>
</tr>
<tr>
<td>T₆</td>
<td>wait</td>
<td>exit</td>
<td>myNum = *p</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>myNum = 2</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
Hello Thread: Local Storage solution

```c
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.
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.

![Diagram showing thread activation]