東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 4. Threads

A/Prof. Kai Dong

Warm-up

Thread — A New Abstraction

- Why we need this abstraction?
- Think of: Some programs have more than one point of execution (i.e., multiple PCs).

Warm-up

Thread — A New Abstraction

- Why we need this abstraction?
- Think of: Some programs have more than one point of execution (i.e., multiple PCs).
  - Example #1: adding two large arrays together
  - How to speed up execution?

Warm-up

Thread — A New Abstraction

- Why we need this abstraction?
- Think of: Some programs have more than one point of execution (i.e., multiple PCs).
  - Example #1: adding two large arrays together
  - How to speed up execution?
  - Example #2: some elements need input
  - How to speed up execution?

Warm-up

Thread — A New Abstraction

- Why we need this abstraction?
- Think of: Some programs have more than one point of execution (i.e., multiple PCs).
  - Example #1: adding two large arrays together
  - How to speed up execution?
  - Example #2: some elements need input
  - How to speed up execution?
- Why multiple threads, not multiple processes?
  - We are now detailing the reasons.

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- 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

Contents

Contents

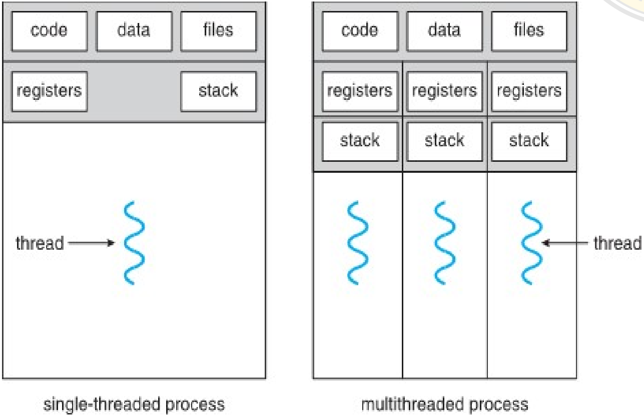
Overview

Motivation

- Most modern applications 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
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Overview

Single and Multithreaded Processes



Overview

Benefits

- Responsiveness — may allow continued execution if part of process is blocked, especially important for user interfaces

Overview

Benefits

- Responsiveness — may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing — threads share resources of process, easier than shared memory or message passing

Overview

Benefits

- Responsiveness — may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing — threads share resources of process, easier than shared memory or message passing
- Economy — cheaper than process creation, thread switching lower overhead than context switching

Overview

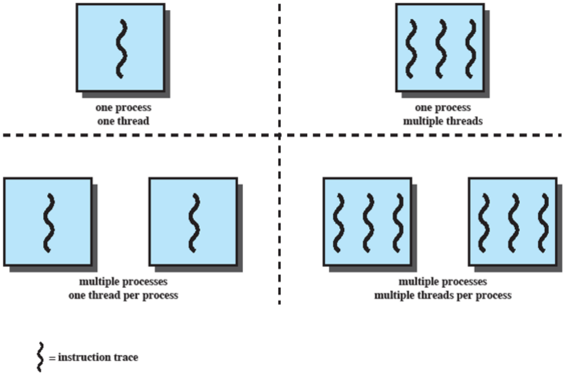
Benefits

- Responsiveness — may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing — threads share resources of process, easier than shared memory or message passing
- Economy — cheaper than process creation, thread switching lower overhead than context switching
- Scalability — process can take advantage of multiprocessor architectures

Overview

Multithreading

MS-DOS	JRE
UNIX	most OS



Contents

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Multicore Programming

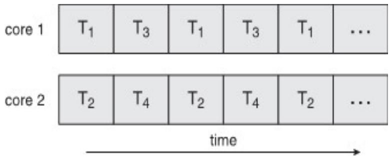
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming

Concurrency Vs. Parallelism

- **Concurrent** execution on single-core system:

- **Parallelism** on a multi-core system:



Multicore Programming

- Types of parallelism
  - Data parallelism — distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism — distributing threads across cores, each thread performing unique operation

Multicore Programming

- Types of parallelism
  - Data parallelism — distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism — distributing threads across cores, each thread performing unique operation
- As ♯ of threads grows, so does architectural support for threading
  - CPUs have cores as well as hardware threads
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Multicore Programming

- Types of parallelism
  - Data parallelism — distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism — distributing threads across cores, each thread performing unique operation
- As ♯ of threads grows, so does architectural support for threading
  - CPUs have cores as well as hardware threads
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core
- How much performance can we gain from adding additional cores? Suppose we have N processing cores.

Multicore Programming

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components

Multicore Programming

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

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

Multicore Programming

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

Contents

Multithreading Models

User Threads and Kernel Threads

- Who/How to provide support for threads?

Multithreading Models

User Threads and Kernel Threads

- Who/How to provide support for threads?
- **User threads** — management done by user-level threads library
- **Kernel threads** - Supported by the Kernel

Multithreading Models

User Threads and Kernel Threads

- Who/How to provide support for threads?
- **User threads** — management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel

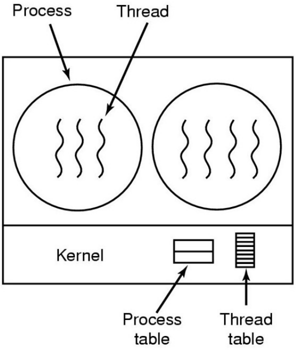
Multithreading Models

User Threads and Kernel Threads

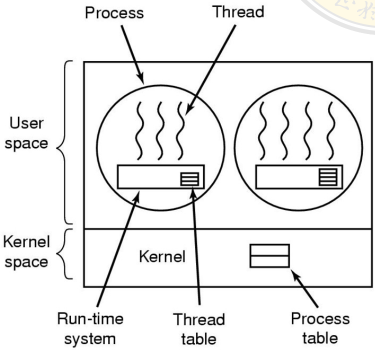
- Who/How to provide support for threads?
- **User threads** — management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples — virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

Implementing Threads in Kernel/User Space



Kernel Threads



User Threads

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.
- Why we need kernel threads?

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.
- Why we need kernel threads?
  - However, blocking one thread will not cause other threads of the same process to block. The kernel simply runs other threads.

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.
- Why we need kernel 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

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.
- Why we need kernel 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
- Why kernel threads are not enough?

Multithreading Models

Motivating Kernel/User Threads

- Why we need user threads?
  - There is no kernel intervention, and, hence, user threads are usually more efficient.
- Why user threads are not enough?
  - Unfortunately, since the kernel only recognizes the containing process (of the threads), if one thread is blocked, every other threads of the same process are also blocked because the containing process is blocked.
- Why we need kernel 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
- Why kernel threads are not enough?
  - Kernel threads are usually slower than the user threads.

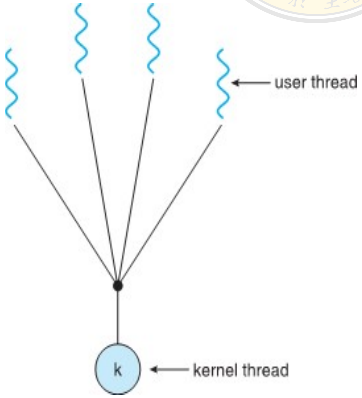
Multithreading Models

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

Multithreading Models

Many-to-One

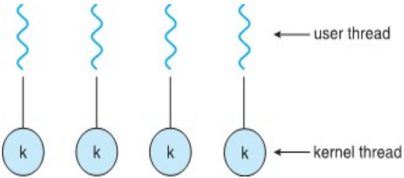
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system
- Used on systems that do not support kernel threads.
- Few systems currently use this model



Multithreading Models

One-to-One

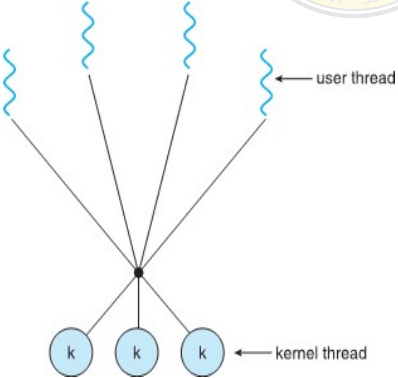
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead



Multithreading Models

Many-to-Many

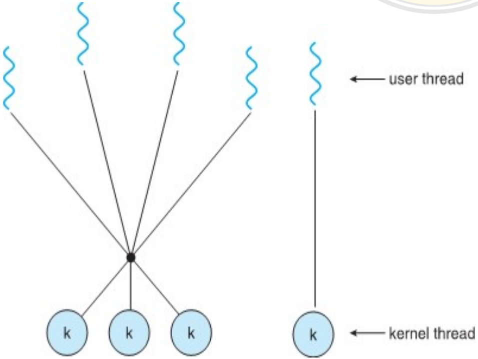
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads



Multithreading Models

Two-Level Model

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



Multithreading Models

Aside

- Among the three models, which model(s) are general in modern systems?
  - M:M for server
  - 1:1 for PC
    - To improve LinuxThreads, two competing projects were started to develop a replacement; NGPT (Next Generation POSIX Threads from IBM) and NPTL (Native POSIX Thread Library from Red Hat). NPTL won out and is today shipped with the vast majority of Linux systems
    - NGPT M:M
    - NPTL 1:1

Contents

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Three main thread libraries
  - **POSIX Pthreads**
  - Windows
  - Java

Thread Libraries

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

Thread Libraries

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library

Thread Libraries

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Libraries

Pthread Creation

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

1. Which thread is to be created?
   - A pointer to structure of this thread.
2. What kind of thread is it?
   - The stack size and priority information, etc.
3. How to run the thread?
   - The thread entry function (function pointer in C).
4. With what arguments?
   - arg.

Thread Libraries

Pthread Creation — What is the Output?

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf("%d %d\n", m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;
        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        ...
}
```

Thread Libraries

Pthread Completion

```
#include <pthread.h>
int
pthread_join(    pthread_t *      thread,
                 void **          value_ptr);
```

1. Which thread is to be completed?
   - A pointer to structure of this thread.
2. What is the return value?
   - void **value_ptr.

Thread Libraries

Pthread Completion — What is the Output?

```c
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef strct __myarg_t {
  int a;
  int b;
} myarg_t;

typedef struct __myret_t {
  int x;
  int y;
} myret_t;

void *mythread(void *arg) {
  myarg_t *m = (myarg_t *) arg;
  printf("%d %d\n", m->a, m->b);
  myret_t *r = Malloc(sizeof(
        myret_t));
  r->x = 1;
  r->y = 2;
  return (void *) r;
}
```

```c
int main(int argc, char *argv[]) {
  int rc;
  pthread_t p;
  myret_t *m;
  myarg_t args;
  args->a = 10;
  args->b = 20;
  Pthread_create(&p, NULL,
        mythread, &args);
  Pthread_join(p, (void **) &m);
  printf("returned %d %d\n", m->x,
        m->y);
  return 0;
}
```

Thread Libraries

Windows Multithreaded C Program

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum;
DWORD WINAPI Summation(LPVOID 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, "argc!=2\n");
    return -1;
  }
  Param = atoi(argv[1]);
  if (Param < 0) {
    fprintf(stderr, "Param<0\n");
    return -1;
  }
```

```c
/* 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);
  }
}
```

Thread Libraries

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:

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

- Extending Thread class
- Implementing the Runnable interface

Thread Libraries

Java Multithreaded Program

```
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);
  }
}
```

Thread Libraries

Java Multithreaded Program (contd.)

```java
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.getSum());
        } catch (InterruptedException ie) {}
      }
    }
    else
      System.err.println("Usage: Summation <integer value>");
  }
}
```

In Class Exercise

What are Possible Outputs?

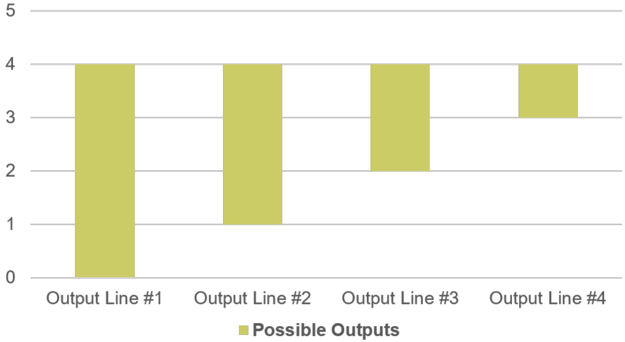
```
void *helloFunc(void *ptr) {
        int *data;
        data = (int *) ptr;
        printf("I'm Thread %d\n", *data);
}

int main(int argc, char *argv[]) {
        pthread_t hThread[4];
        for (int i = 0; i < 4; i ++)
                pthread_create(&hThread[i], NULL, helloFunc, (void *) &i);
        for (int i = 0; i < 4; i ++)
                pthread_join(hThread[i], NULL);
        return 0;
}
```

In Class Exercise

Key

Non-decreasing sequence satisfying ...



Contents

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers

Implicit Threading

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
  - 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
- Windows API supports thread pools:

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

Contents

Threading Issues

- Semantics of *fork*() and *exec*() system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
  - Synchronous and asynchronous
- Thread-local storage
- Scheduler Activations

Threading Issues

Semantics of fork() and exec()

- If one thread in a program calls *fork*(), does the new process duplicate all threads, or is the new process single-threaded?
  - Two versions of *fork*().

Threading Issues

Semantics of fork() and exec()

- If one thread in a program calls *fork*(), does the new process duplicate all threads, or is the new process single-threaded?
  - Two versions of *fork*().
- What about *exec*()?
  - Replace the entire process, including all thread.

Threading Issues

Semantics of fork() and exec()

- If one thread in a program calls *fork*(), does the new process duplicate all threads, or is the new process single-threaded?
  - Two versions of *fork*().
- What about *exec*()?
  - Replace the entire process, including all thread.
- Discussion: If *exec*() is called immediately after forking, which version of *fork*() do you prefer?

Threading Issues

Semantics of fork() and exec()

- If one thread in a program calls *fork*(), does the new process duplicate all threads, or is the new process single-threaded?
  - Two versions of *fork*().
- What about *exec*()?
  - Replace the entire process, including all thread.
- Discussion: If *exec*() is called immediately after forking, which version of *fork*() do you prefer?
  - If *exec*() is called immediately after forking, then duplicating all threads is unnecessary

Threading Issues

Thread Cancellation

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

Threading Issues

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 canceled

Threading Issues

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 canceled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

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

/* cancel the thread */
pthread_cancel(tid);
```

Threading Issues

Thread Cancellation (contd.)

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

Threading Issues

Thread Cancellation (contd.)

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

Mode	**State**	**Type**
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Threading Issues

Thread Cancellation (contd.)

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

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

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

Threading Issues

Thread Cancellation (contd.)

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

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- 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

Threading Issues

Thread Cancellation (contd.)

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

Mode	**State**	**Type**
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- 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

Threading Issues

Signal Handling

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

Threading Issues

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
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers:
    - default
    - user-defined

Threading Issues

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
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers:
    - default
    - 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

Threading Issues

Signal Handling (contd.)

- 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

Threading Issues

Signal Handling (contd.)

- 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
- Discussion: Can a thread have its own handler?

Threading Issues

Signal Handling (contd.)

- 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
- Discussion: Can a thread have its own handler?
  - All threads in a process share a same signal handler, which can be either a default one or a user-defined one (overridden).

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own data

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own 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

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own 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

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* ++?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own 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
- Why not use the thread stack?

Threading Issues

Thread-Local Storage

- How to achieve parallelism in executing *counter* $++$?
  - Each thread counts, and summarize when needed.
- Is counter shared by all threads in the process, or owned by each thread?
  - Not shared within the process, also not in thread stacks.
- **Thread-local storage** (**TLS**) allows each thread to have its own 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
- Why not use the thread stack?
  - Life cycle reasons. Local variables are in thread stack

Threading Issues

Scheduler Activations

- In M:M or Two-level model, how to maintain a appropriate number of kernel threads allocated to the application?

Threading Issues

Scheduler Activations

- In M:M or Two-level model, how to maintain a appropriate number of kernel threads allocated to the application?
  - Communication, but how?

Threading Issues

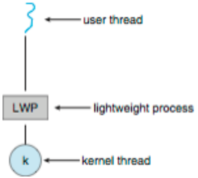
Scheduler Activations

- In M:M or Two-level model, how to maintain a appropriate number of kernel threads allocated to the application?
  - Communication, but how?
- 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?

Threading Issues

Scheduler Activations

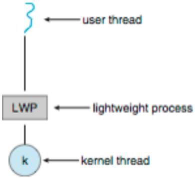
- In M:M or Two-level model, how to maintain a appropriate number of kernel threads allocated to the application?
  - Communication, but how?
- Typically use an intermediate data structure between user and kernel threads — lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?



Threading Issues

Scheduler Activations

- In M:M or Two-level model, how to maintain a appropriate number of kernel threads allocated to the application?
  - Communication, but how?
- 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. (vs. downcalls)
- This communication allows an application to maintain the correct number kernel threads

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?
- At some other moment, one kernel thread blocks (waiting for I/O completion)

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?
- At some other moment, one kernel thread blocks (waiting for I/O completion)
- T/F? This kernel thread makes an upcall, and the attached LWP blocks the current user thread, and schedules another user thread to run.

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?
- At some other moment, one kernel thread blocks (waiting for I/O completion)
- T/F? This kernel thread makes an upcall, and the attached LWP blocks the current user thread, and schedules another user thread to run.
- T/F? The attached LWP also blocks. The kernel makes an upcall and then allocates a new LWP to the application.

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?
- At some other moment, one kernel thread blocks (waiting for I/O completion)
- T/F? This kernel thread makes an upcall, and the attached LWP blocks the current user thread, and schedules another user thread to run.
- T/F? The attached LWP also blocks. The kernel makes an upcall and then allocates a new LWP to the application.
- After returning from upcall handling, how many LWPs now?

Threading Issues

Scheduler Activations (contd.)

- Suppose an I/O-bound application which has five different file-read requests occur simultaneously. At some moment, two kernel threads are allocated to the application.
- How many LWPs are created?
- At some other moment, one kernel thread blocks (waiting for I/O completion)
- T/F? This kernel thread makes an upcall, and the attached LWP blocks the current user thread, and schedules another user thread to run.
- T/F? The attached LWP also blocks. The kernel makes an upcall and then allocates a new LWP to the application.
- After returning from upcall handling, how many LWPs now?
- If we wish to run this application efficiently, how many LWPs are needed for this application.