



Soongsil Univ.  
<http://design.ssu.ac.kr>


Digital  
System  
Design Lab.

# Ch. 04 Threads & Concurrency


Chanho Lee  
Soongsil University

Copyright 2023. (차세대반도체 혁신공유대학 사업단) all rights reserved

1

**Objectives**

- Basic components of a thread
  - ▶ contrast threads and processes.
- Benefits and challenges of designing multithreaded processes.
- Implicit threading,
  - ▶ thread pools, fork-join, and Grand Central Dispatch.
- Threads in Linux operating systems
  - ▶ Design multithreaded applications using the Pthreads threading APIs.


Soongsil Univ.  
Digital System Design Lab.

Operating Systems

Copyright 2023. (차세대반도체 혁신공유대학 사업단) all rights reserved 2

2


Soongsil Univ.  
<http://design.ssu.ac.kr>


Digital  
System  
Design Lab.

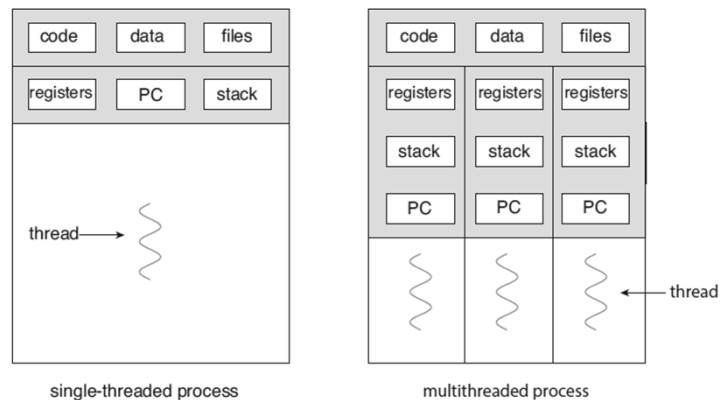
## 6주차 Threads

Copyright 2023. (차세대반도체 혁신공유대학 사업단) all rights reserved

3

## 4.1 Overview

- A thread: a basic unit of CPU utilization
  - ▶ comprises a thread ID, a PC, a register set, and a stack.
  - ▶ Threads share code, data, and other OS resources, such as open files and signals.
- Multiple threads of control → more than one task at a time.
  - ▶ A traditional process has a single thread of control



4

## 4.1 Overview

### ● Motivation

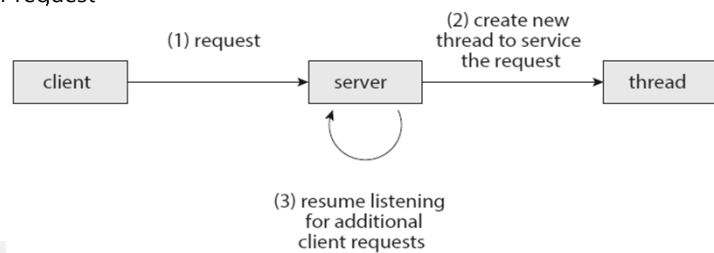
#### ▶ A few examples of multithreaded applications

- Creating photo thumbnails: a separate thread for each separate image.
- Web browser: a thread to display images or text, another thread to retrieve data.
- Word processor: threads for displaying graphics, for responding to keystrokes, and for spelling and grammar checking in the background

#### ▶ Performing several CPU-intensive tasks in parallel across the multiple computing cores

#### ▶ Web server

- a single process accepts requests
- creating a separate process to service each request : the same task  
→ creating a separate thread for each request



## 4.1 Overview

### ● Benefits of multithreaded programming

#### ▶ Responsiveness.

- continue running even if part of it is blocked or is performing a lengthy operation,
- especially useful in designing UI.

#### ▶ Resource sharing.

- threads share the memory and the resources by default.

#### ▶ Economy.

- thread creation consumes less time and memory than process
- context switching is typically faster

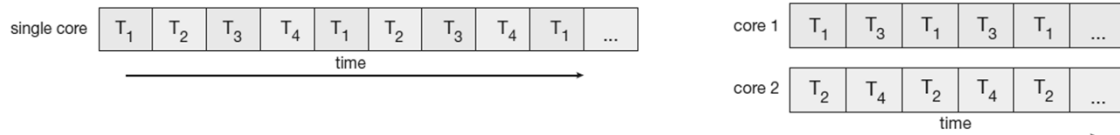
#### ▶ Scalability.

- threads may be running in parallel on different processing cores.

## 4.2 Multicore Programming

### ● Multicore: multiple computing cores on a single processing chip

- ▶ Multithreaded programming provides a mechanism for more efficient use of multiple cores and improved concurrency
- ▶ Concurrency: all the tasks to make progress
  - Concurrency with multicore: some threads can run in parallel (parallelism: simultaneous)
  - Concurrency with a single core: execution of the threads interleaving over time



## 4.2 Multicore Programming

### ● Programming Challenges

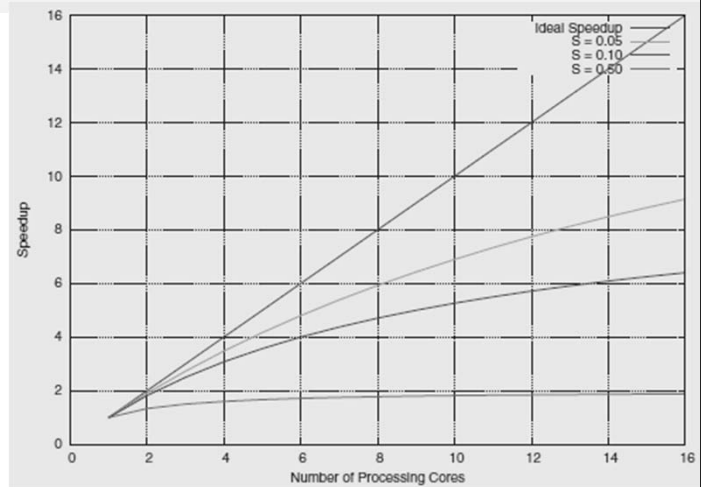
- ▶ Identifying tasks.
  - Divide into separate, concurrent tasks: independent tasks can run in parallel
- ▶ Balance.
  - Tasks with equal work of equal value.
- ▶ Data splitting.
  - The data splitting to run on separate cores.
- ▶ Data dependency.
  - Synchronization of the execution of the tasks to accommodate the data dependency. (ch.6)
- ▶ Testing and debugging.
  - More difficult testing and debugging for running programs in parallel on multiple cores

### ● Parallel programming is important

## 4.2 Multicore Programming

### ● Amdahl's Law

- ▶  $\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}} \rightarrow 1/S \text{ as } N \rightarrow \infty$ 
  - S: portion for serial execution
  - N: # of processing cores



## 4.2 Multicore Programming

### ● Types of Parallelism

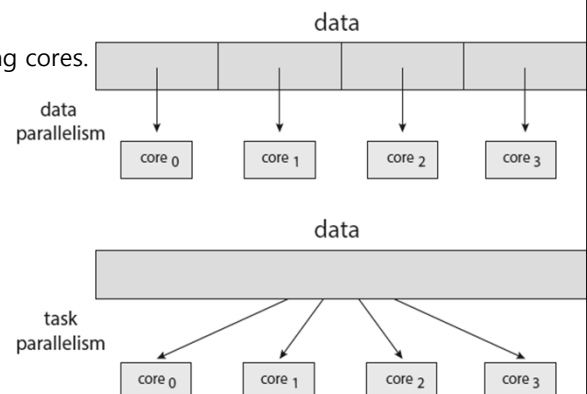
#### ▶ Data parallelism

- focuses on data distribution across multiple cores and performing the same operation on each core.
- ex. summation of array elements of size N on a dual-core system:
  - thread A on core 0, sum the elements [0] . . . [N/2 - 1]
  - thread B, on core 1, sum the elements [N/2] . . . [N - 1].

#### ▶ Task parallelism

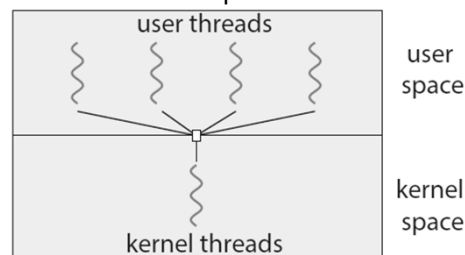
- distributing tasks (threads) across multiple computing cores.
- Each thread is performing a unique operation.
- Data may be the same data, or not
- ex. statistical operation on the array
  - thread A on core 0, average
  - thread B, on core 1, standard deviation

#### ▶ not mutually exclusive; a hybrid approach



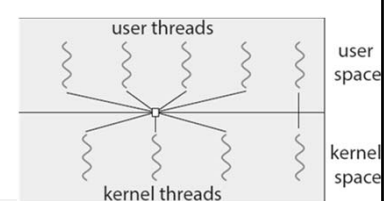
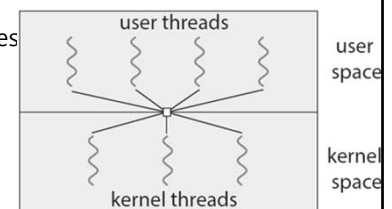
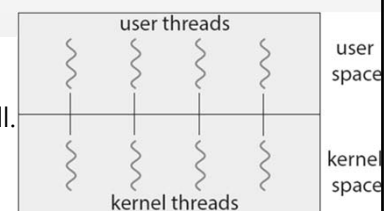
### 4.3 Multithreading Models

- User threads: supported above the kernel and are managed without kernel support,
- Kernel threads: supported and managed directly by the OS.
  - ▶ Virtually all contemporary OSes—Windows, Linux, and macOS
- Many-to-One Model
  - ▶ maps many user-level threads to one kernel thread
  - ▶ managed by the thread library in user space → efficient
  - ▶ entire process will block if a thread makes a blocking system call
  - ▶ one thread at a time: no parallel running on multicore systems
  - ▶ not limited in the number of threads: no parallelism



### 4.3 Multithreading Models

- One-to-One Model
  - ▶ maps each user thread to a kernel thread.
  - ▶ another thread to run when a thread makes a blocking system call.
  - ▶ multiple threads to run in parallel on multiprocessors.
  - ▶ the same number of use and kernel thread → may burden the performance of a system.
    - limited in the number of threads; less important as the increase of cores in processors
  - ▶ Linux and Windows
- Many-to-Many Model
  - ▶ multiplexes many user-level threads to a smaller or equal number of kernel threads
    - # of kernel threads depends on application and machine (# of cores)
  - ▶ Not limited in the number of threads; parallelism
  - ▶ difficult to implement
- Two-level model
  - ▶ many-to-many + one-to-one



## 4.4 Thread Libraries

- API for creating and managing threads
  - ▶ User-level library: local function call
  - ▶ kernel-level library: system call
- POSIX Pthreads
  - ▶ either a user-level or a kernel-level library
- Windows thread library: a kernel-level library
- Java thread API: threads to be created and managed directly in Java programs on the host system
  - ▶ Windows API on Windows, Pthreads on UNIX, Linux, and macOS
- Global variables are shared among all threads for POSIX and Windows
- Asynchronous threading
  - ▶ parent and child execute concurrently and independently of one another
  - ▶ typically little data sharing
- Synchronous threading
  - ▶ parent must wait for all of its children to terminate before it resumes
  - ▶ Typically, involves significant data sharing among threads

## 4.4 Thread Libraries

- Pthreads (IEEE 1003.1c): a *specification* for thread behavior, not an *implementation*
  - ▶ OS designers may implement the specification

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes including stack size and scheduling information */
    pthread_attr_init(&attr); /* set the default attributes of the thread */
    pthread_create(&tid, &attr, runner, argv[1]); /* create the thread with integer parameter N(argv[1]) */
    pthread_join(tid, NULL); /* wait for the thread to exit */
    printf("sum = %d \n", sum);
}
/* The thread will execute in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 1; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

$$sum = \sum_{i=1}^N i$$

```
#define NUM_THREADS 10
/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

## 4.5 Implicit Threading

### ● Creation and management of threading by compilers and run-time libraries

- ▶ To identify tasks (not threads) that can run in parallel.
- ▶ Parallel tasks written as a function
- ▶ Run-time libraries determine the specific details of thread creation and management

### ● Thread Pools

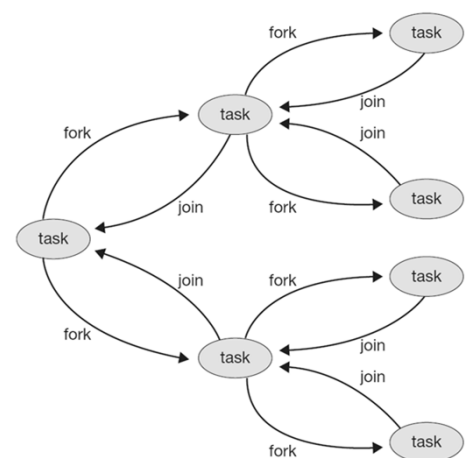
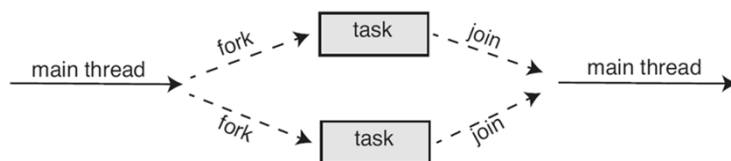
- ▶ Issues in multithreaded system
  - the amount of time required to create the thread + the thread discarded after completion of the work
  - Unlimited threads could exhaust system resources
- ▶ Thread pools
  - to create a number of threads at start-up and place them into a pool
  - A request is submitted to a thread in the thread pool
  - The request waits in the queue if an idle thread is unavailable
  - Works well for asynchronous execution
- ▶ Benefits
  - Faster: existing thread
  - limits the number of threads: the number can be set based on resources or dynamically.
  - Separation of creation and execution: different strategies



## 4.5 Implicit Threading

### ● Fork Join model

- ▶ synchronous model as explicit thread creation
- ▶ synchronous version of thread pools (implicit) : library manages the number of threads





## 4.5 Implicit Threading

### ● OpenMP

- ▶ set of compiler directives as well as an API for programs written in C, C++, or FORTRAN
- ▶ support for parallel programming in shared memory environments
- ▶ identifies parallel regions as blocks of code running in parallel
- ▶ insert compiler directives at parallel regions → instruct the OpenMP runtime library to execute the region in parallel
- ▶ to choose among several levels of parallelism: set the number of threads manually
- ▶ to identify whether data are shared between threads or not

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    /* sequential code */
    #pragma omp parallel //creates as many threads as there are processing cores
    {
        printf("I am a parallel region.");
        #pragma omp parallel for
        for (i = 0; i < N; i++) {
            c[i] = a[i] + b[i];
        }
        /* sequential code */
    }
    return 0;
}
```

OpenMP divides the work among the threads

## 4.5 Implicit Threading

### ● Grand Central Dispatch (GCD) for macOS, iOS

- ▶ Combination of a run-time library, an API, and language extensions
- ▶ Developers identify sections code (tasks) to run in parallel.
- ▶ GCD manages most of the details of threading.
- ▶ Scheduling: dispatch queue and thread pool
  - Serial queue: FIFO order, sequentially removed, must complete execution before another task
  - concurrent queue: FIFO order, several tasks at a time, multiple tasks to execute in parallel

### ● Intel Thread Building Blocks

- ▶ Template library for parallel applications in C++.
- ▶ Specify parallel tasks, and the TBB task scheduler maps them onto underlying threads.
- ▶ Task scheduler provides load balancing and is cache aware

▶ Ex. 

```
for (int i = 0; i < n; i++) {
    apply(v[i]);
}
```

```
parallel for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

## 4.6 Threading Issues

- The fork() and exec() System Calls
  - ▶ Calling fork() in a thread (UNIX) duplicates
    - all threads or only the thread
  - ▶ exec() system call: replace the entire process—including all threads
  - ▶ fork() and exec() immediately: duplicating only the calling thread
  - ▶ fork() without exec(): duplicating all threads
- Signal Handling
  - ▶ Signal: to notify a particular event
    - synchronously or asynchronously
  - ▶ Signal pattern
    - 1. generated by the occurrence of a particular event.
    - 2. delivered to a process.
    - 3. Once delivered, the signal must be handled.
  - ▶ Ex. Synchronous signals: illegal memory access and division by 0
    - Signal is delivered to the corresponding process to the operation

## 4.6 Threading Issues

- Signal Handling (cont.)
  - ▶ Ex. Asynchronous signals: event external to a running process
    - terminating a process with specific keystrokes (ctrl+c)
    - Typically, an asynchronous signal is sent to another process
  - ▶ Signal handlers
    - A default signal handler
    - A user-defined signal handler
  - ▶ Signal delivery in multithreaded programs
    - 1. to the thread to which the signal applies. (synchronous)
    - 2. to every thread in the process.: ex. Ctrl+c
    - 3. to certain threads in the process: ex. pthread\_kill(pthread\_t tid, int signal)
    - 4. to a specific thread to receive all signals for the process: ex. kill(pid\_t pid, int signal)

## 4.6 Threading Issues

### ● Thread Cancellation

▶ terminating a thread before completion

▶ Cancellation of a target thread

- 1. Asynchronous cancellation: One thread immediately terminates the target thread.
- 2. Deferred cancellation: The target thread periodically checks termination, allowing termination in an orderly fashion.

```
pthread_t tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
. . .
/* cancel the thread */
pthread_cancel(tid);
/* wait for the thread to terminate */
pthread_join(tid, NULL);
```

- A thread sets cancellation state using an API

Mode	State	Type
Off	Disabled	– (pending)
Deferred	Enabled	Deferred (default)
Asynchronous	Enabled	Asynchronous

## 4.6 Threading Issues

### ● Thread Cancellation (cont.)

▶ deferred cancellation at a cancellation point

Most of the blocking system calls  
in the POSIX and standard C library

▶ `pthread_testcancel()` function: establishing a cancellation point

- With pending cancellation request: thread termination
- → cleanup handler function: resources are released

```
while (1) {
/* do some work for a while */
. . .
/* check if there is a cancellation request */
pthread_testcancel();
}
```

The gcc compiler provides the storage class  
keyword `__thread` for declaring TLS data.  
// to assign a unique identifier for each thread  
`static __thread int threadID;`

### ● Thread-Local Storage (TLS)

▶ Thread's copy of certain data instead of sharing

- Ex. Transaction service with unique identifier in a separate thread

▶ visible across function invocations: `static` to each thread

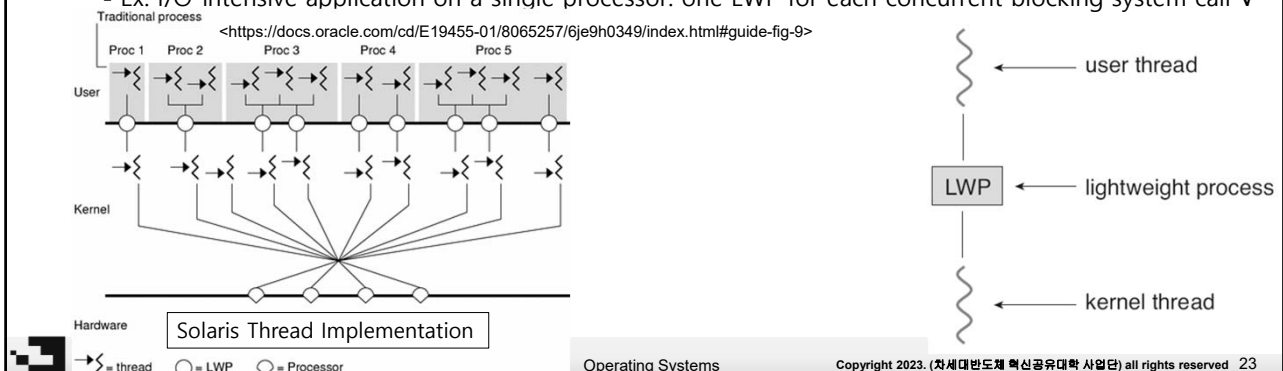
- Cf. local variable: visible only during a single function invocation

▶ Pthread: `pthread_key_t` type

## 4.6 Threading Issues

### ● Scheduler Activations

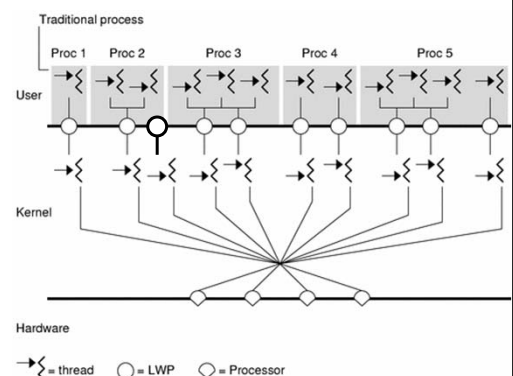
- ▶ Many-to-many/two-level model: communication between the kernel and the thread library
- ▶ Lightweight process (LWP): intermediate data structure between the user and kernel threads
  - Attached to a kernel thread (scheduled by the kernel to physical processor)
  - A virtual processor for a user thread
- ▶ The number of LWP depends on application
  - Ex. CPU-bound application on a single processor: one LWP
  - Ex. I/O-intensive application on a single processor: one LWP for each concurrent blocking system call ✓



23

## 4.6 Threading Issues

- ▶ Scheduler activation: communication
  - Kernel: provides virtual processors (LWPs)
  - Application: schedule user threads onto LWPs
  - Upcall: kernel informs an application about events
    - handled by the thread library with an upcall handler on a virtual processor.
  - Ex. A blocking thread → upcall → upcall handler (by app.) on a new LWP (by kernel)✓
    - saves state of the blocking thread
    - gives up the blocking LWP
    - schedules another thread to the new LWP
  - Ex. Exiting blocking (unblocked) → upcall → upcall handler
    - marks unblocked thread as eligible to run
    - schedules another thread to the new LWP

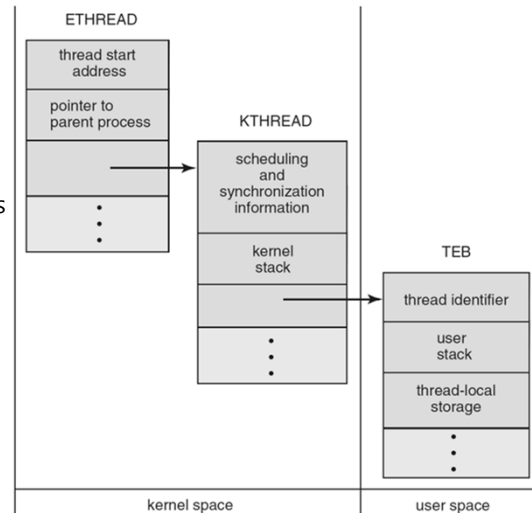


24

## 4.7 Operating-System Examples

### ● Windows Threads

- ▶ one-to-one mapping
- ▶ general components of a thread
  - A thread ID
  - A register set representing the status of the processor
  - A program counter
  - A user stack in user mode, and a kernel stack in kernel mode
  - A private storage area used by various run-time libraries and dynamic link libraries (DLLs)
- ▶ context of a thread: register set, stacks, and private storage area
- ▶ Primary data structures of a thread
  - ETHREAD—executive thread block
  - KTHREAD—kernel thread block
  - TEB—thread environment block



## 4.7 Operating-System Examples

### ● Linux Threads

- ▶ fork() system call: duplicating a process
  - Copy data structures of parent
- ▶ clone() system call: to create threads
  - Passes a set of flags that determine the sharing between the parent and child tasks
  - A unique kernel data structure (struct task\_struct) for each task in the system.
    - contains pointers to other data structures for the list of open files, signal-handling information, and virtual memory and so on.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- ▶ Linux does not distinguish between processes and threads.
  - uses the term task when referring to a flow of control within a program.
  - flags determine a process or a thread based upon the amount of sharing

## Summary

- Thread: a basic unit of CPU utilization
  - ▶ share many of the process resources, including code and data
- Primary benefits to multithreaded applications:
  - ▶ (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- Concurrency: multiple threads are making progress
  - ▶ Only concurrency is possible with a single CPU
- Parallelism: multiple threads are making progress simultaneously.
  - ▶ parallelism requires a multicore system
- Challenges in designing multithreaded applications
  - ▶ dividing and balancing the work: task parallelism
  - ▶ dividing the data between the different threads: data parallelism
  - ▶ identifying any data dependencies
  - ▶ test and debug



## Summary

- User-level threads from user application, mapped to kernel threads to execute on a CPU.
  - ▶ many-to-one model
  - ▶ one-to-one
  - ▶ many-to-many models.
- Thread library API for creating and managing threads.
  - ▶ Windows is for the Windows system only
  - ▶ Pthreads for POSIX-compatible systems such as UNIX, Linux, and macOS.
  - ▶ Java threads on a Java virtual machine.
- Implicit threading: increasingly common technique in developing concurrent and parallel applications
  - ▶ thread pools
  - ▶ fork-join frameworks
  - ▶ Grand Central Dispatch.



## Summary

- Threads termination
  - ▶ Asynchronous cancellation stops a thread immediately,
  - ▶ Deferred cancellation allows the thread to terminate in an orderly fashion. Preferred
- Linux does not distinguish between processes and threads; a task.
  - ▶ clone() system call: to create tasks that behave either more like processes or threads.

## Exercises, problems and projects

### ● Exercises

- ▶ 4.1, 4.3

### ● Problems

- ▶ 4.8, 4.9, 4.16

**4.8** Provide two programming examples in which multithreading does **not** provide better performance than a single-threaded solution.

**4.9** Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

**4.16** A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

## Exercises, problems and projects

**4.17** Consider the following code segment:

```
pid_t pid;
pid = fork();
if (pid == 0) { /* child process */
    fork();
    thread_create( . . . );
}
fork();
```

- How many unique processes are created?
- How many unique threads are created?

**4.19** The program shown in Figure 4.23 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
#include <pthread.h>
#include <stdio.h>
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;
    pid = fork();
    if (pid == 0) { /* child process */
        pthread_attr_t init(&attr);
        pthread_create(&tid,&attr,runner,NULL);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d",value); /* LINE C */
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE P */
    }
}
void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}
```

## Exercises, problems and projects

### ● Programming

**4.22** Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the minimum value. For example, suppose your program is passed the integers

90 81 78 95 79 72 85

The program will report

The average value is 82

The minimum value is 72

The maximum value is 95

The variables representing the average, minimum, and maximum values will be stored globally. The worker threads will set these values, and the parent thread will output the values once the workers have exited. (We could obviously expand this program by creating additional threads that determine other statistical values, such as median and standard deviation.)

**4.23** Write a multithreaded program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.



## Exercises, problems and projects

### ● Project

**4.24** An interesting way of calculating  $\pi$  is to use a technique known as **Monte Carlo**, which involves randomization. This technique works as follows: Suppose you have a circle inscribed within a square, as shown in Figure. (Assume that the radius of this circle is 1.)

- First, generate a series of random points as simple  $(x, y)$  coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle.

- Next, estimate  $\pi$  by performing the following calculation:  $\pi = 4 \times (\text{number of points in circle}) / (\text{total number of points})$  Write a multithreaded version of this algorithm that creates a separate thread to generate a number of random points. The thread will count the number of points that occur within the circle and store that result

in a global variable. When this thread has exited, the parent thread will calculate and output the estimated value of  $\pi$ . It is worth experimenting with the number of random points generated.

As a general rule, the greater the number of points, the closer the approximation to  $\pi$ .

