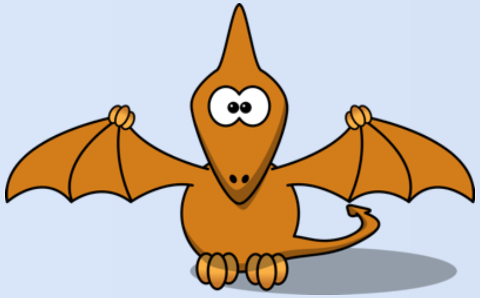
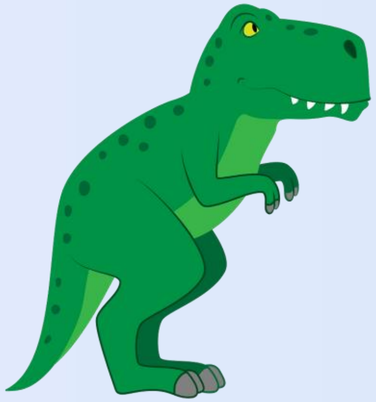


Chapter 4.

Threads & Concurrency



Operating System Concepts (10th Ed.)

4.1 Overview

- So far, we assumed that
  - a *process* was an executing program *with a single thread of control*.
  - however, *a process* is able to contain *multiple threads of control*,
  - isn't it?

4.1 Overview

- A ***thread*** is
  - a lightweight process.
  - a basic unit of CPU utilization.
  - comprises a *thread ID*, a *program counter*, a *register set*, and a *stack*.

4.1 Overview

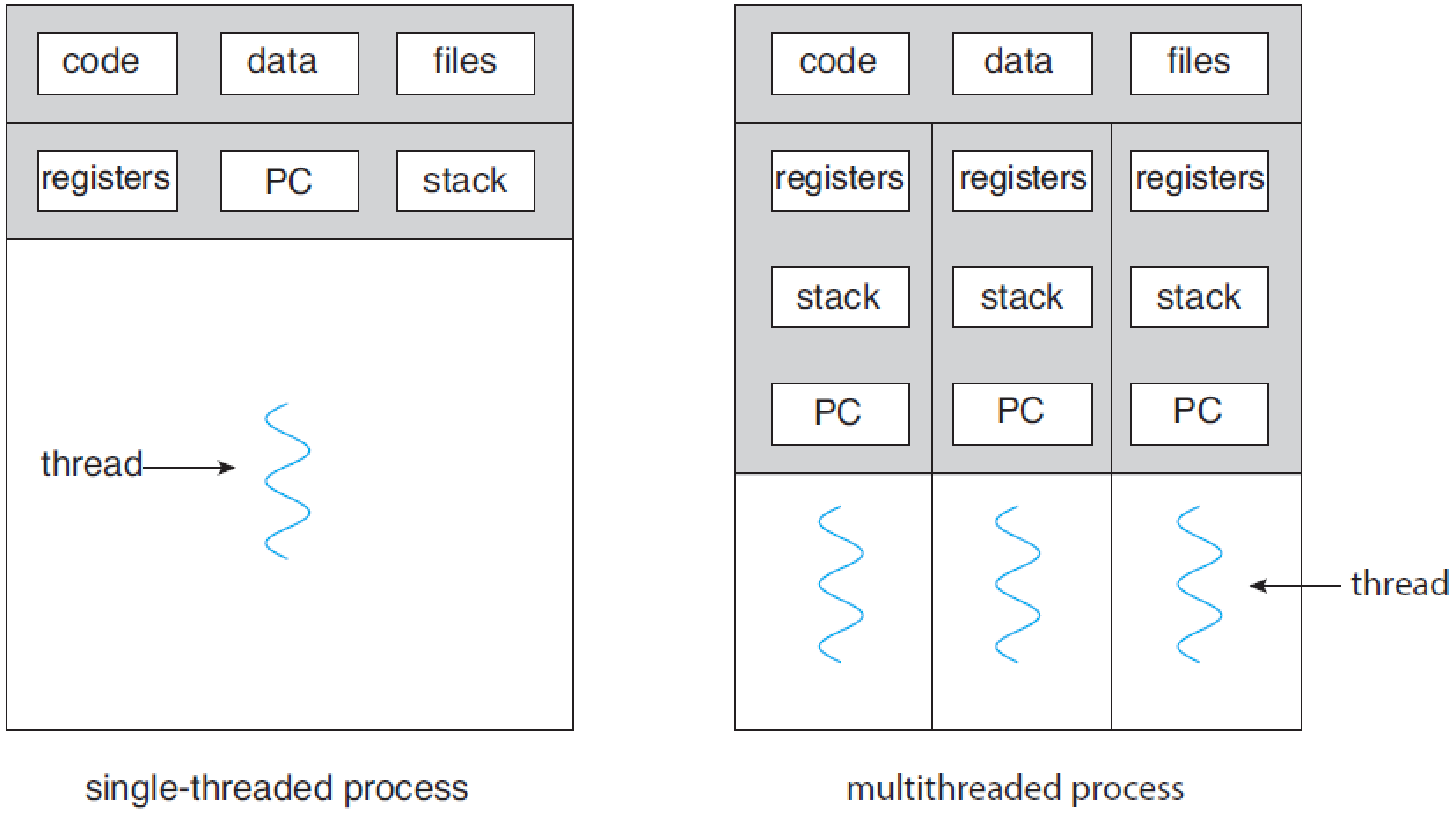


Figure 4.1 *Single-threaded and multithreaded processes.*

4.1 Overview

- Motivation for multithreading
  - Let us consider the case of client-server system, e.g., a *web server*.

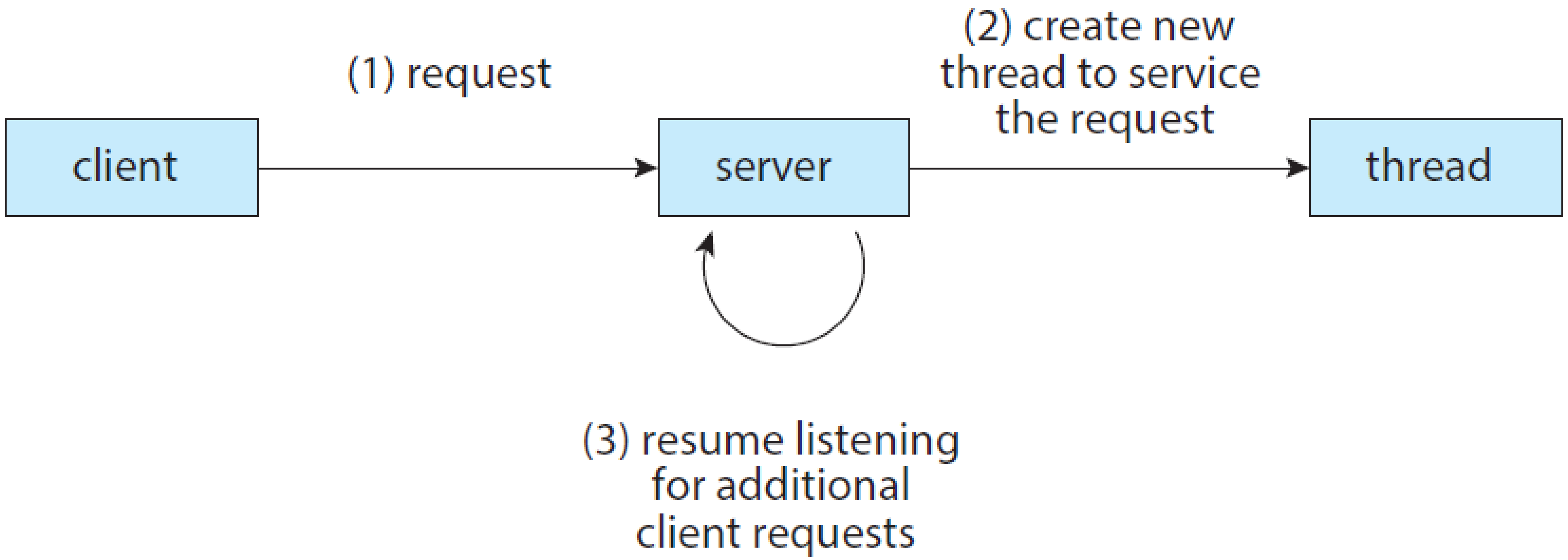


Figure 4.2 *Multithread server architecture.*

4.1 Overview

- The benefits of multithreaded programming:
  - *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

4.4 Thread Library

- Threads in Java
  - In a Java program,
    - threads are the fundamental model of program execution.
  - Java provides a rich set of features
    - for the creation and management of threads

4.4 Thread Library

- Three techniques for explicitly creating threads in Java.
  - *Inheritance* from the `Thread` class
    - create a new class that is derived from the `Thread` class.
    - and override its `public void run()` method.
  - *Implementing* the `Runnable` interface.
    - define a new class that implements the `Runnable` interface.
    - and override its `public void run()` method.
  - Using the *Lambda* expression (beginning with Java Version 1.8)
    - rather than defining a new class,
    - use a *lambda expression* of `Runnable` instead.

4.4 Thread Library

▪ 방법 1: Thread 클래스 상속받기

```
class MyThread1 extends Thread {
    public void run() {
        try {
            while (true) {
                System.out.println("Hello, Thread!");
                Thread.sleep(500);
            }
        }
        catch (InterruptedException ie) {
            System.out.println("I'm interrupted");
        }
    }
}
```

4.4 Thread Library

```java
public class ThreadExample1 {

    public static final void main(String[] args) {
        MyThread1 thread = new MyThread1();
        thread.start();
        System.out.println("Hello, My Child!");
    }
}
```

4.4 Thread Library

▪ 방법 2: Runnable 인터페이스 구현하기

```
class MyThread2 implements Runnable {
    public void run() {
        try {
            while (true) {
                System.out.println("Hello, Runnable!");
                Thread.sleep(500);
            }
        }
        catch (InterruptedException ie) {
            System.out.println("I'm interrupted");
        }
    }
}
```

4.4 Thread Library

```java
public class ThreadExample2 {

    public static final void main(String[] args) {
        Thread thread = new Thread(new MyThread2());
        thread.start();
        System.out.println("Hello, My Runnable Child!");
    }
}
```

4.4 Thread Library

- 방법 3: Runnable 람다 표현식 사용하기

```
public class ThreadExample3 {
    public static final void main(String[] args) {
        Runnable task = () -> {
            try {
                while (true) {
                    System.out.println("Hello, Lambda Runnable!");
                    Thread.sleep(500);
                }
            }
            catch (InterruptedException ie) {
                System.out.println("I'm interrupted");
            }
        };
        Thread thread = new Thread(task);
        thread.start();
        System.out.println("Hello, My Lambda Child!");
    }
}
```

4.4 Thread Library

- **부모 쓰레드의 대기**: wait? join!

```
public class ThreadExample4 {
    public static final void main(String[] args) {
        Runnable task = () -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Hello, Lambda Runnable!");
            }
        };
        Thread thread = new Thread(task);
        thread.start();
        try {
            thread.join();
        }
        catch (InterruptedException ie) {
            System.out.println("Parent thread is interrupted");
        }
        System.out.println("Hello, My Joined Child!");
    }
}
```

4.4 Thread Library

- 쓰레드의 종료: stop? interrupt!

```
public class ThreadExample5 {
    public static final void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            try {
                while (true) {
                    System.out.println("Hello, Lambda Runnable!");
                    Thread.sleep(100);
                }
            }
            catch (InterruptedException ie) {
                System.out.println("I'm interrupted");
            }
        };
        Thread thread = new Thread(task);
        thread.start();
        Thread.sleep(500);
        thread.interrupt();
        System.out.println("Hello, My Interrupted Child!");
    }
}
```

4.2 Multicore Programming

- Multithreading in a Multicore system.
  - more efficient use of multiple cores for improved concurrency.

  - Consider an application with four threads.
    - single-core: threads will be interleaved over time.
    - multiple-cores: some threads can run in parallel.

4.2 Multicore Programming

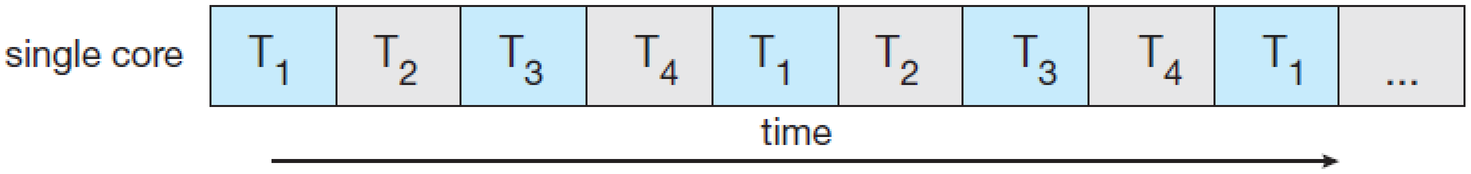


Figure 4.3 *Concurrent execution on a single-core system.*

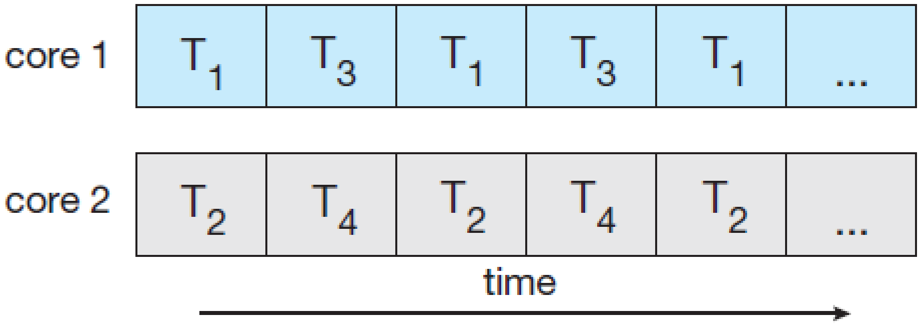


Figure 4.4 *Parallel execution on a multicore system.*

4.2 Multicore Programming

- Programming Challenges in Multicore systems.
  - *Identifying tasks*: find areas can be divided into separate tasks.
  - *Balance*: ensure the tasks to perform equal work of equal value.
  - *Data splitting*: data also must be divided to run on separate cores.
  - *Data dependency*: ensure that the execution of tasks is synchronized to accommodate the data dependency
  - *Testing and debugging*: more difficult than single-thread.

4.2 Multicore Programming

- Types of parallelism:

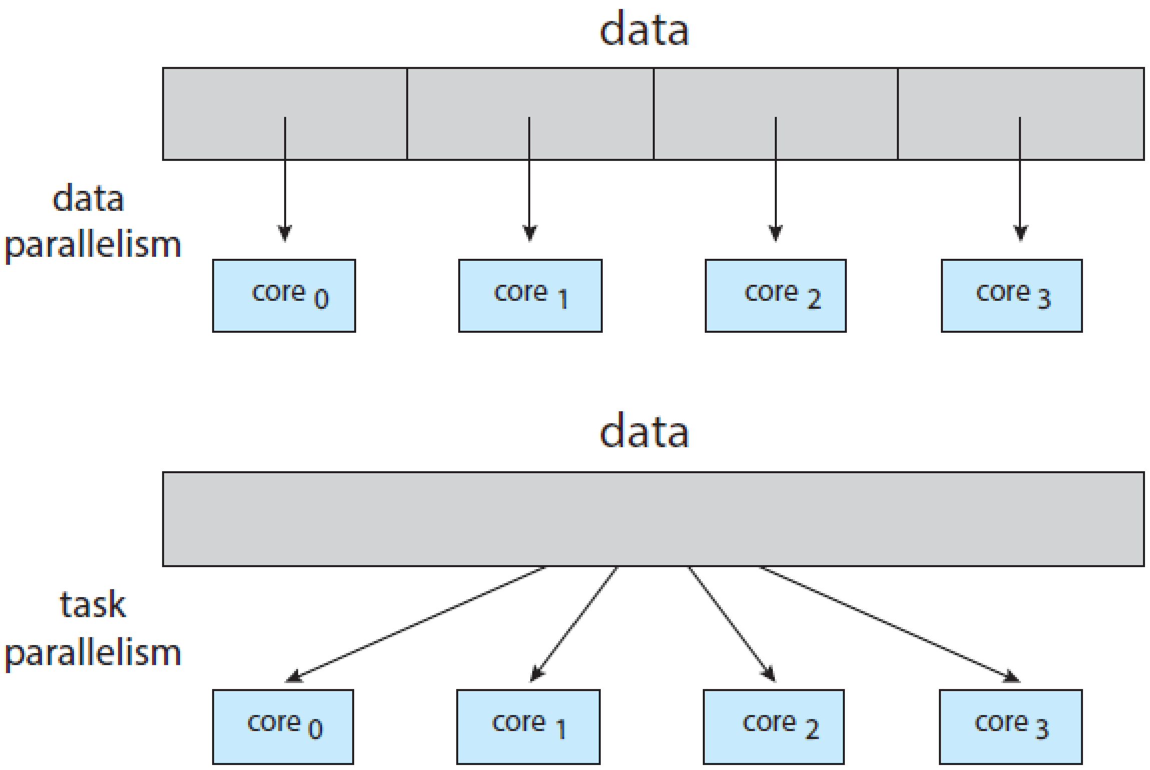
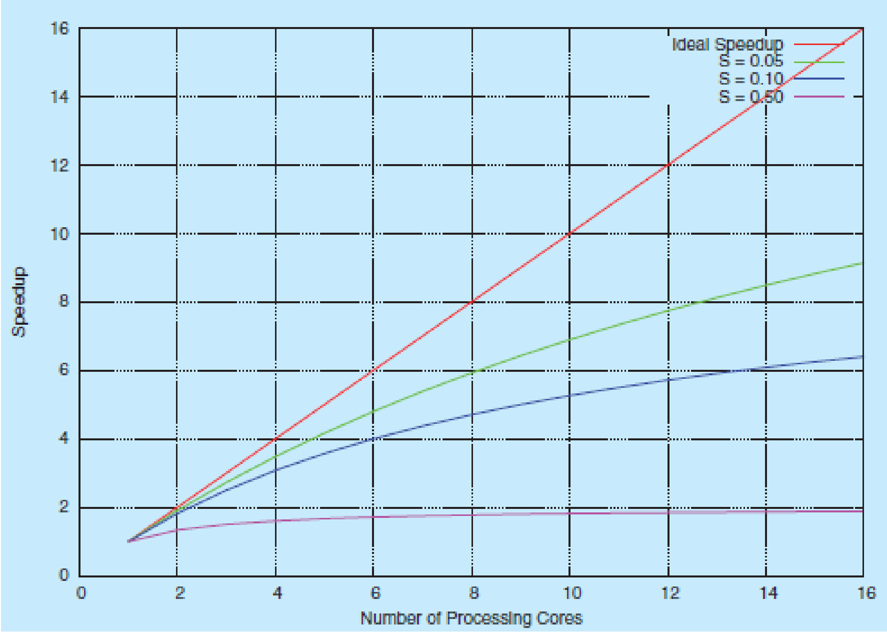


Figure 4.5 *Data parallelism and task parallelism.*

- Amdahl's Law:
  - 코어는 무조건 많을수록 좋은가?
  - *speedup* $<= \frac{1}{S+\frac{(1-S)}{N}}$, where
    - ***S***: the portion that must be performed serially on a system.
    - ***N***: the number of processing cores
  - For example,
    - S=0.25, N=2, speedup=1.6
    - S=0.25, N=4, speedup=2.28



4.3 Multithreading Models

- Two types of threads:
  - ***user*** threads and ***kernel*** threads

  - User threads are
    - supported above the kernel, and
    - are managed *without kernel support*.
  - Kernel threads are
    - supported and managed directly *by the operating system*.

4.3 Multithreading Models

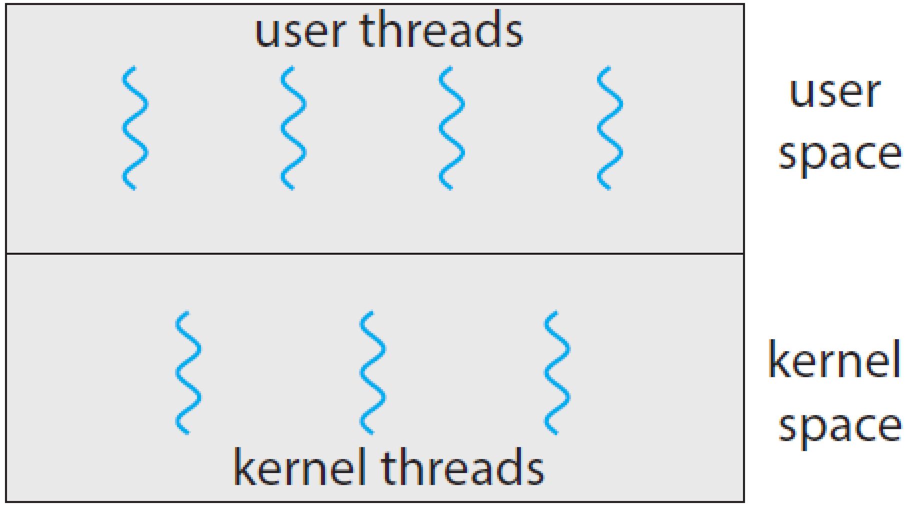


Figure 4.6 *User threads and kernel threads.*

4.3 Multithreading Models

- Three relationships between user and kernel threads:
  - Many-to-One Model
  - One-to-One Model
  - Many-to-Many Model

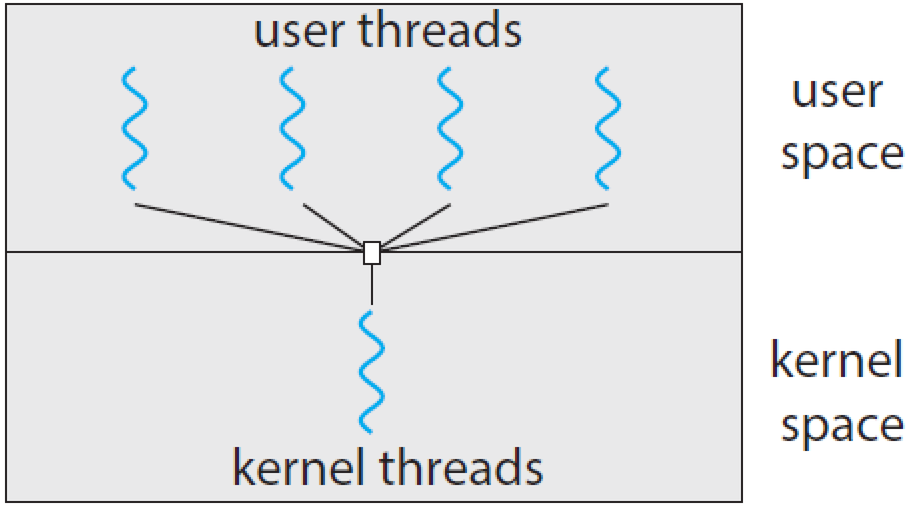


Figure 4.7 *Many-to-one model.*

4.3 Multithreading Models

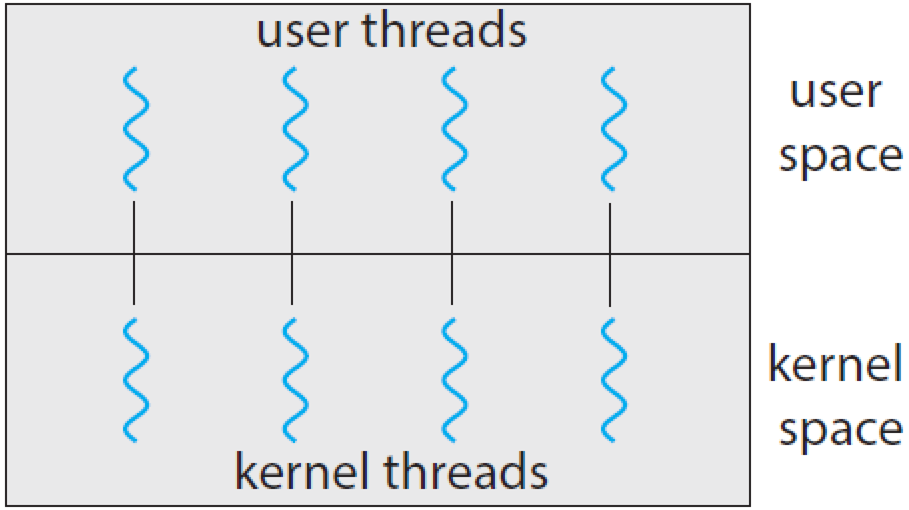


Figure 4.8 *One-to-one model.*

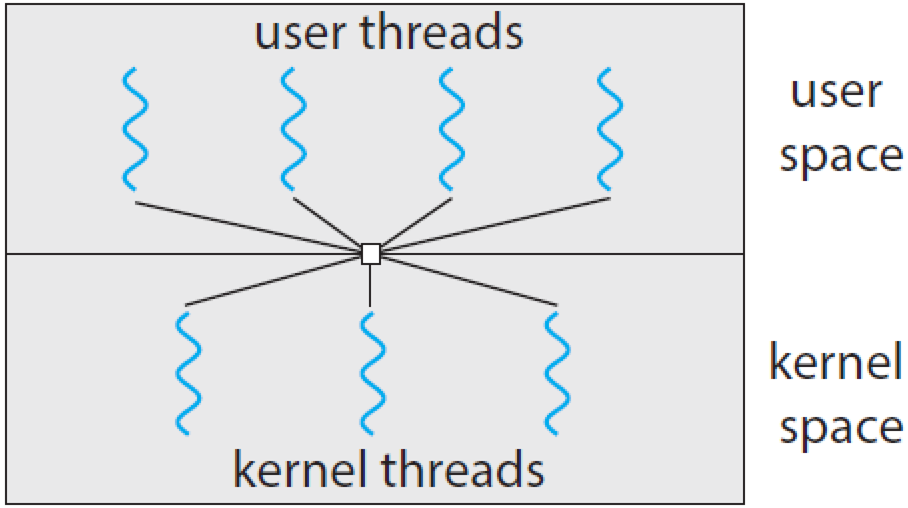


Figure 4.9 *Many-to-Many model.*

4.4 Thread Libraries

- A *thread library* provides
  - an API for *creating* and *managing* threads.

- Three main thread libraries are in use today:
  - POSIX **Pthreads**
  - **Windows** thread
  - **Java** thread.

4.4 Thread Libraries

- ***Pthreads***
  - refers to the POSIX standard (IEEE 1003.1c)
  - just a *specification* for thread behavior, not an implementation.

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

/* the data shared by the threads */
int sum;
/* thread call this function */
void * runner(void *param);
```

4.4 Thread Libraries

```
int main(int argc, char *argv[])
{
    pthread_t tid;          // thread identifier
    pthread_attr_t attr;    // thread attributes

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);

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

void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i = 0; i <= upper; i++)
        sum += i;
    pthread_exit(0);
}
```

```
$gcc -pthread 4.11_pthread.c
```

Figure 4.11 *Multithreaded C program using the Pthread API.*

4.4 Thread Libraries

- Exercise 4.17 (p. 910)
  - Consider the following code segment:

```
pid_t pid;

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

    - a. How many unique processes are created?
    - b. How many unique threads are created?

4.4 Thread Libraries

- Exercise 4.19 (p. 910)

```
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <pthread.h>

int value = 0;
void * runner(void *param);

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;
```

4.4 Thread Libraries

```
    pid = fork();

    if (pid == 0) { // child process
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d\n", value); // LINE C
    }
    else if (pid > 0) { // parent process
        wait(NULL);
        printf("PARENT: value = %d\n", value); // LINE P
    }
}

void *runner(void *param)
{
    value = 5;
    pthread_exit(0);
}
```

4.5 Implicit Threading

- The Strategy of **Implicit Threading**
  - The design of *concurrent* and *parallel* applications,
    - i.e., the design of *multithreading* in *multicore* systems,
    - is too difficult for application developers.
  - So, *transfer the difficulty* to compiler and run-time libraries.

4.5 Implicit Threading

- Four alternative approaches using implicit threading:
  - **Thread Pools**
    - create a number of threads in a pool where they await work.
  - **Fork & Join**
    - *explicit* threading, but an excellent candidate for *implicit* threading.
  - **OpenMP**
    - a set of compiler directives and an API for programs written in C/C++.
  - **Grand Central Dispatch (GCD)**
    - developed by Apple for its macOS and iOS operating system.

4.5 Implicit Threading

- OpenMP
  - identifies parallel regions as blocks of code that may run in parallel.
  - insert compiler directives into source code at parallel regions.
  - these directives instruct OpenMP runtime library to execute the region in parallel.

4.5 Implicit Threading

```
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    #pragma omp parallel // compiler directive
    {
        printf("I am a parallel region.\n");
    }

    return 0;
}
```

```
$gcc -fopenmp 4.20_OpenMP1.c
```

4.5 Implicit Threading


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

int main(int argc, char *argv[])
{
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        printf("OpenMP thread: %d\n", omp_get_thread_num());
    }

    return 0;
}
```

4.5 Implicit Threading

```
#include <stdio.h>
#include <omp.h>

#define SIZE 100000000

int a[SIZE], b[SIZE], c[SIZE];

int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < SIZE; i++)
        a[i] = b[i] = i;

    #pragma omp parallel for
    for (i = 0; i < SIZE; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

4.5 Implicit Threading

```
joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ time ./sum_not_parallel

real    0m0.586s
user    0m0.364s
sys     0m0.223s

joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ time ./sum_with_openmp

real    0m0.423s
user    0m1.091s
sys     0m0.441s
```

Any Questions?