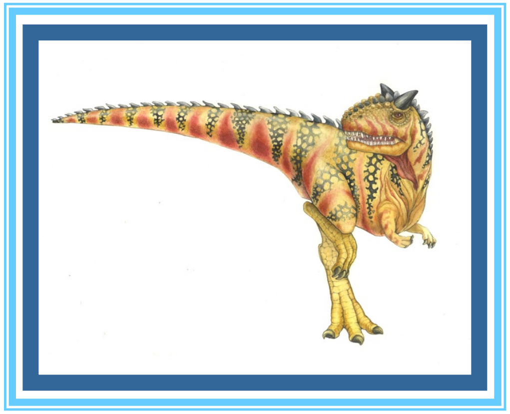
Chapter 4: Threads & Concurrency



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading (Thread Pool)
- Threading Issues
- Operating System Example

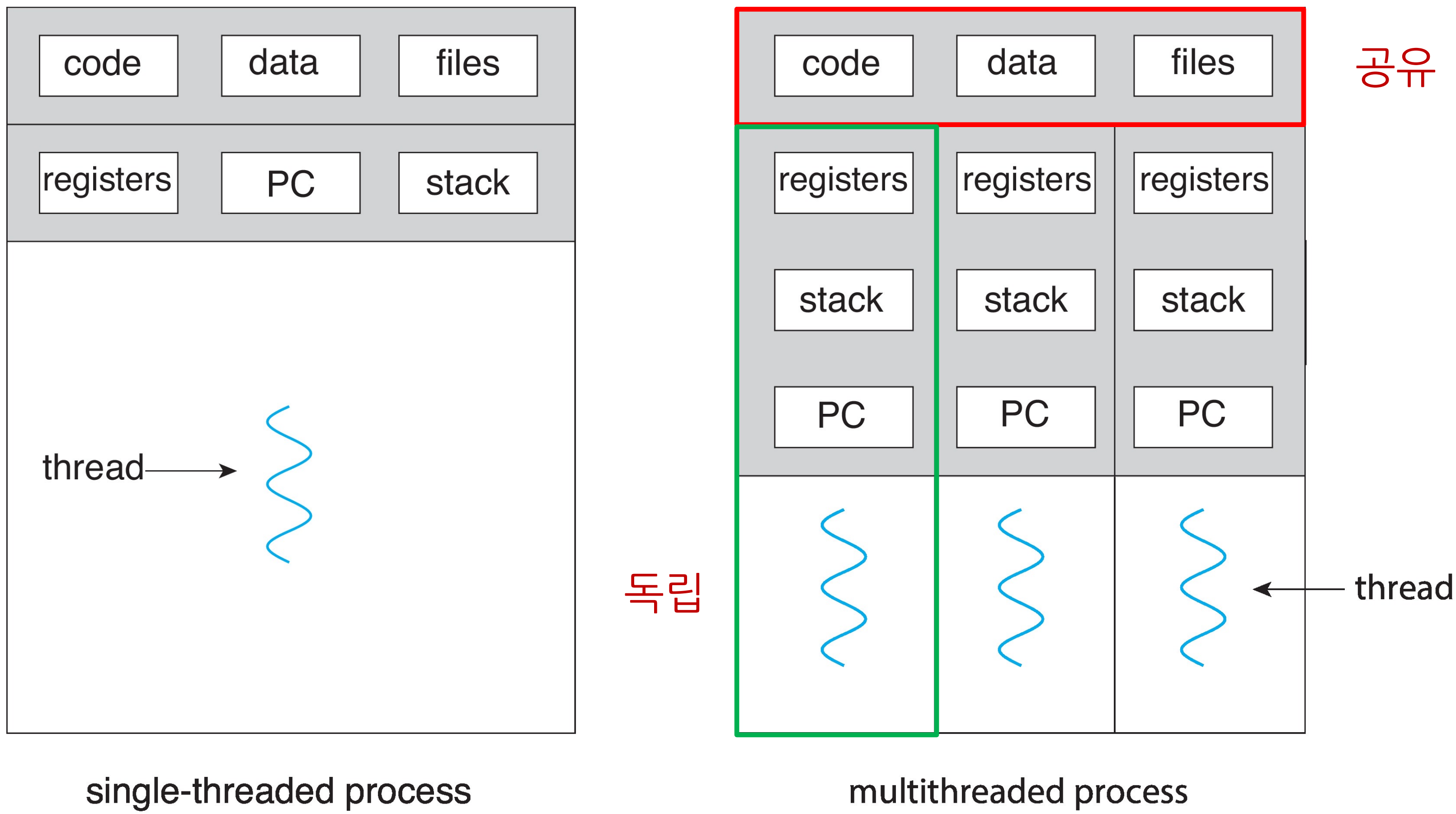
Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designng multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Linux operating systems represent threads

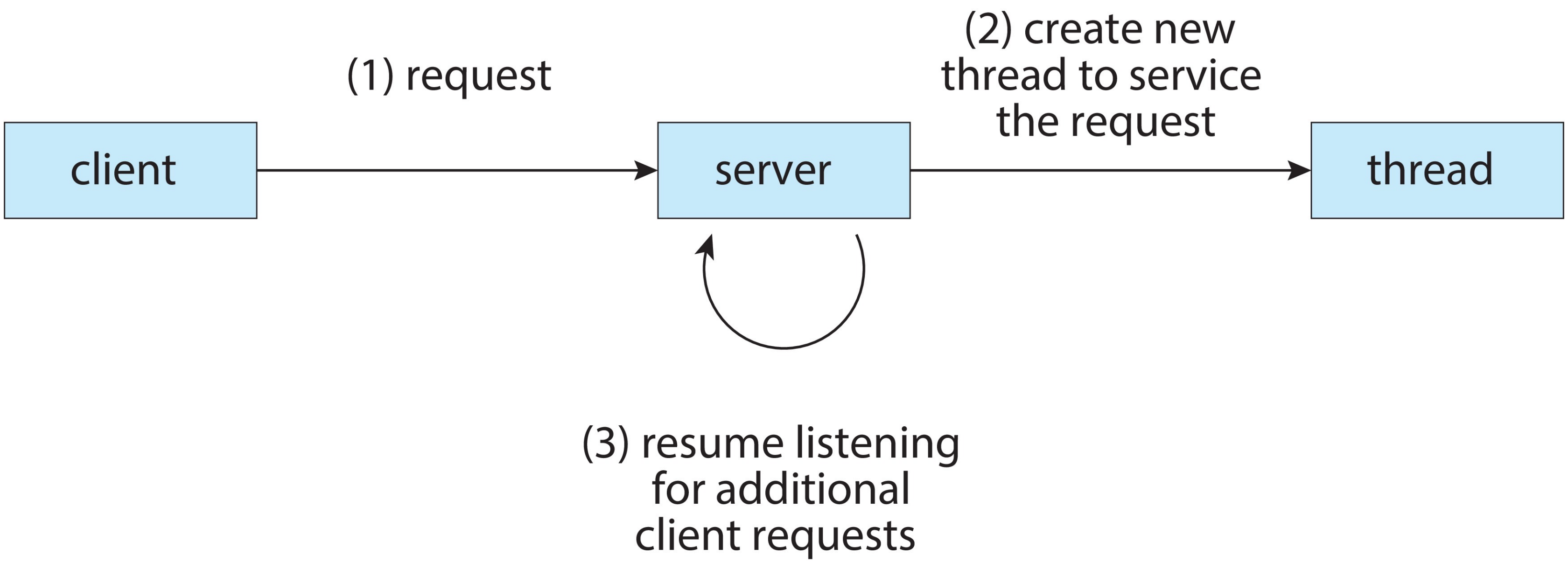
Motivation

- 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

Single and Multithreaded Processes



Multithreaded Server Architecture



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 multicore architectures

Multicore Programming

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

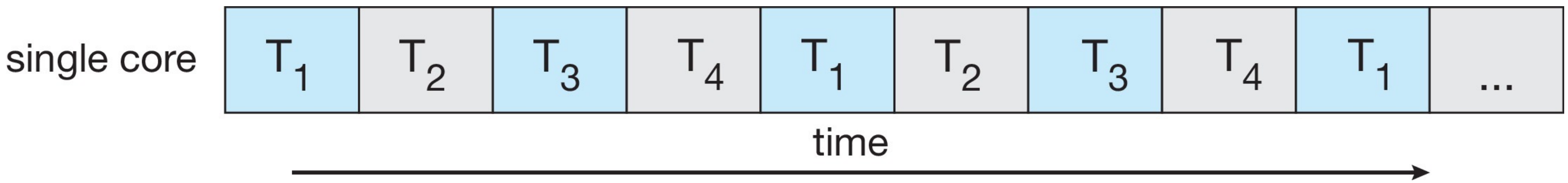
병렬

- ***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

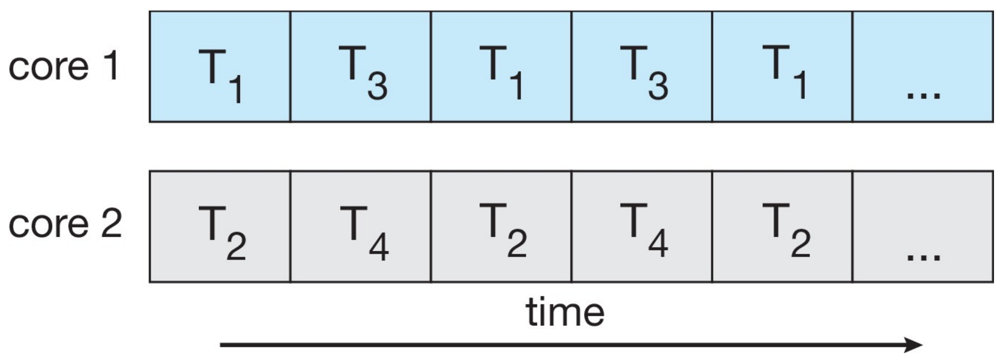
병행

Concurrency vs. Parallelism

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



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



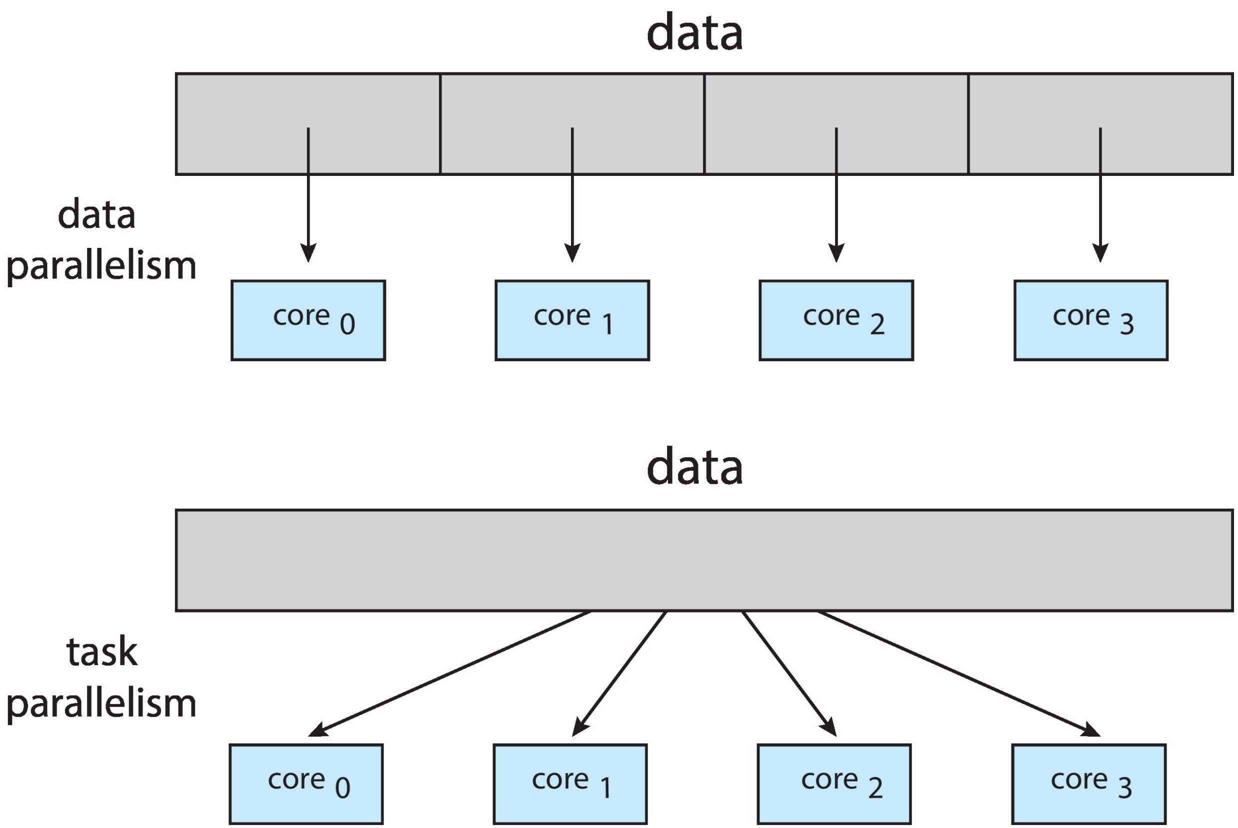
Multicore Programming

- 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

각 코어가 데이터를 나누어서 덧셈을 하는 경우 → data parallelism

같은 데이터에 대해 각 코어가 다른 통계 연산을 하는 경우 → task parallelism

Data and Task Parallelism



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

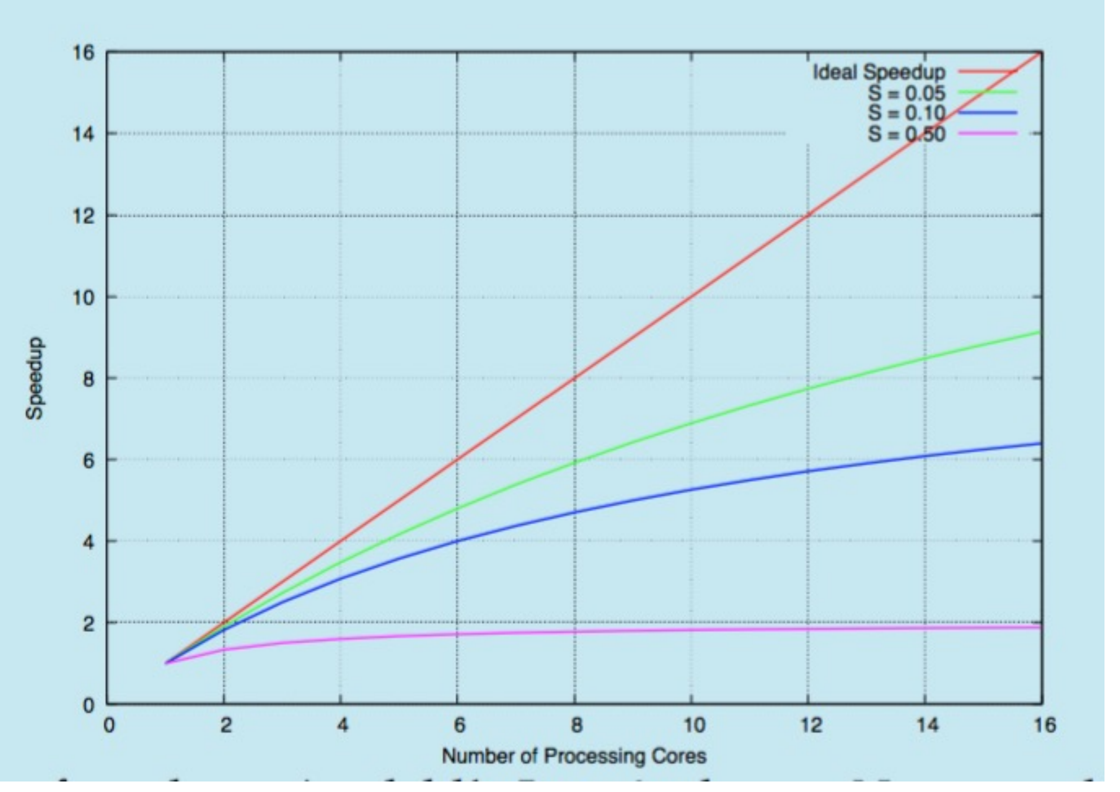
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- 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

- But does the law take into account contemporary multicore systems?

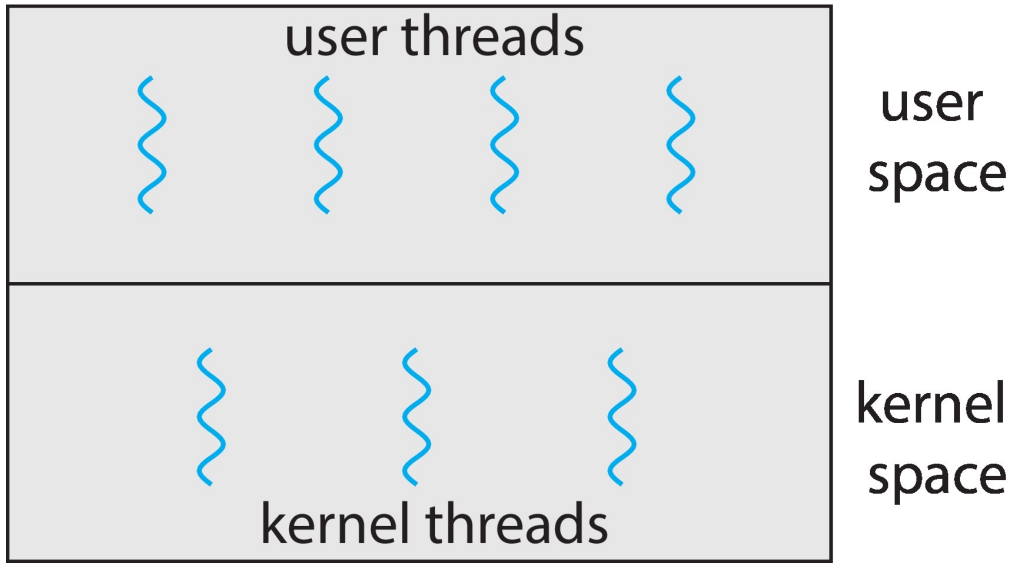
Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

User Threads and Kernel 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
  - Linux
  - Mac OS X
  - iOS
  - Android

User and Kernel Threads



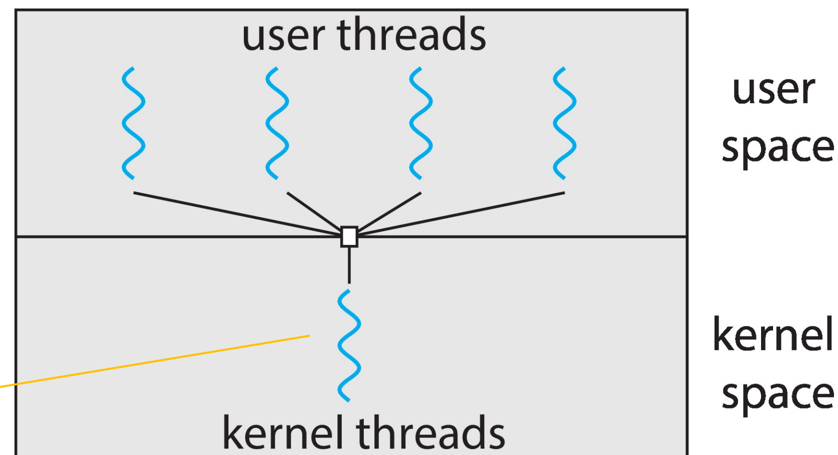
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block in a process
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

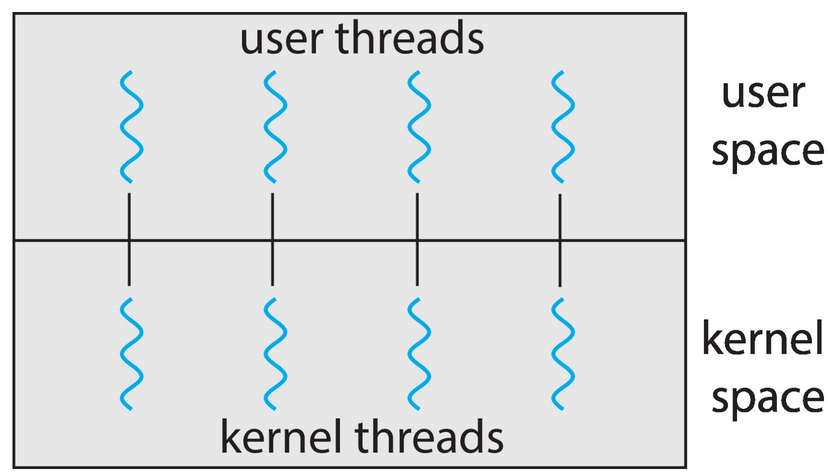
장점: 원하는 만큼 사용자 스레드를 생성할 수 있음



주의: 커널 스레드가 운영체제 전체에 하나만 있다는 의미는 아님. 사용자 프로세스에 상응하는 커널 스레드가 하나만 있다는 의미임.

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
- Examples
  - Windows
  - Linux



장점: 스레드가 블로킹되면 다른 스레드를 실행할 수 있음.

장점: 여러 개의 스레드를 다중코어에 매핑할 수 있음.

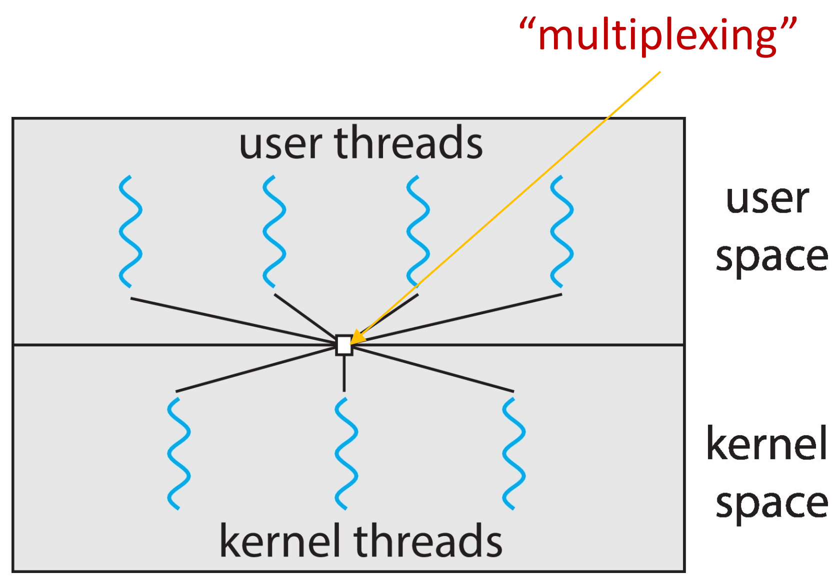
단점: 사용자 스레드와 커널 스레드가 1:1이기 때문에 사용자 스레드를 무한정 생성할 수 없음.

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 with the *ThreadFiber* package
- Otherwise not very common

장점: 사용자 스레드를 마음대로 생성 가능 + 멀티코어에서 병렬로 실행 가능

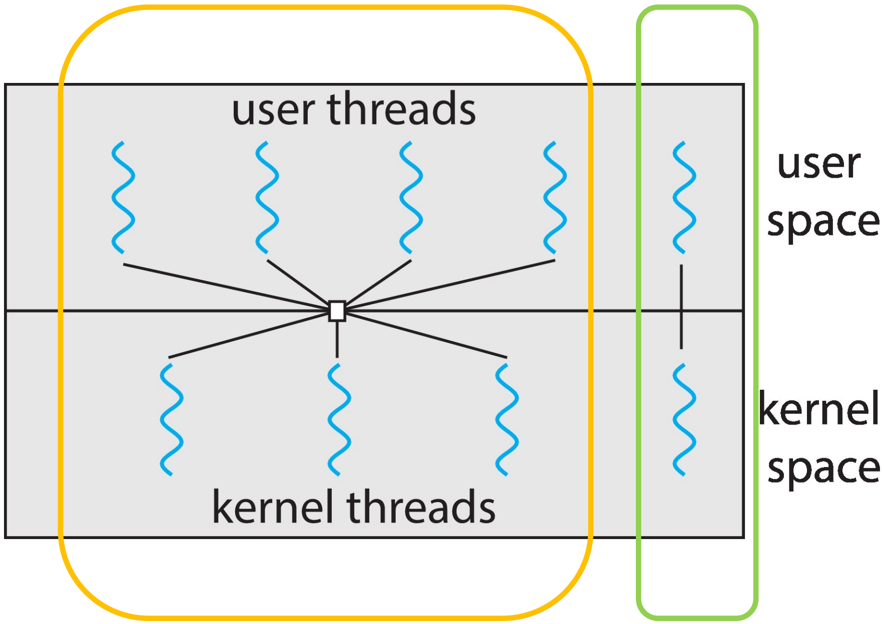
단점: 구현하기 어려움



커널 스레드의 수는 응용 또는 코어의 수에 따라 다를 수 있다

Two-level Model

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



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 (results in a system call to the kernel)

Asynchronous threading: 부모와 자식 스레드가 독립적으로 병행 실행
Synchronous threading: 부모 스레드는 모든 자식 스레드가 종료될 때까지 기다림

POSIX Pthreads: user level or kernel level
Windows: kernel level
Java threads: depending on OS

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 (Linux & Mac OS X)

Pthreads Create

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

thread ID

thread attributes

address of start routine

argument to the start routine

Pthreads Example

```
#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 */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

Pthreads Example (cont)

```
/* 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);
}
```

Pthreads Code for Joining 10 Threads

```
#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);
```

부모가 다수의 자식 스레드가
종료할 때까지 기다리는 경우

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Windows Multithreaded C Program (Cont.)

```c
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* 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 */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

갯수 배열

WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

모두 기다림 시간 무제한

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
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP (Open Multi-Processing)
  - Grand Central Dispatch (Apple's thread pool)
  - Intel Threading Building Blocks (C++ template library)

개발자는 병렬로 수행할 수 있는 task를 식별하는데 주력하고,
스레드는 컴파일러나 런타임 라이브러리가 담당

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:

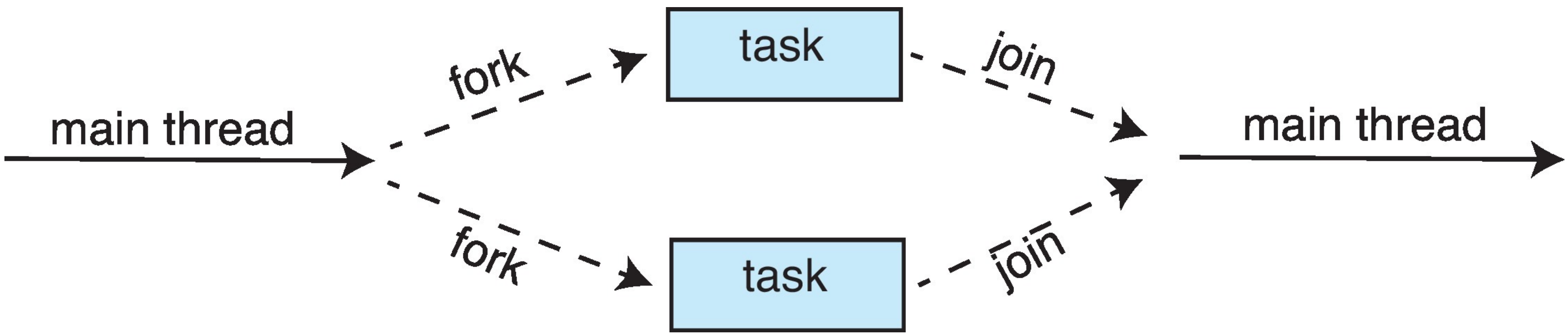
Task 실행과 task 생성을 분리

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

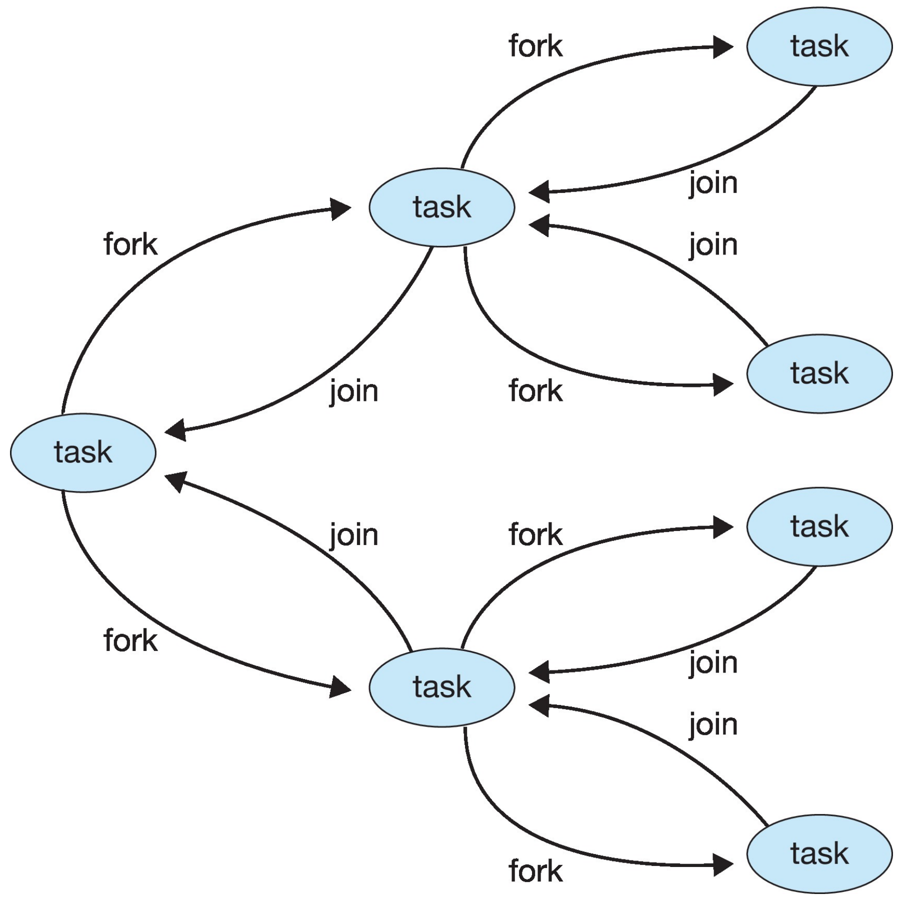
QueueUserWorkItem() 함수를 통해 pool에
있는 스레드가 PoolFunction을 실행하게 함

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



Fork-Join Parallelism



OpenMP

컴파일러 지시문

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

코어의 수만큼

`#pragma omp parallel` fork

Create as many threads as there are cores

join

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

코어의 수만큼 메시지가 중복되서 출력됨.
모든 코어가 동일한 코드를 실행하기 때문에
출력은 겹쳐서 보일 수 있음

OpenMP

- Run the for loop in parallel

코어의 수만큼 스레드를 생성하고
N개의 iteration을 스레드에 분할 매핑

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

☛ 이 예제에서는 data의 종속성이 없어서 병렬 실행이 가능함. 그러나 종속성이 있는 경우에는 주의해야 함

☛ 스레드의 수를 지정할 수 있음 num_threads(N)

☛ 매핑 순서에 변화를 줄 수 있음 schedule(dynamic)

Linux, Windows, macOS용 컴파일러 존재

Grand Central Dispatch

Thread pool의 크기를 자동 조절
하부는 POSIX pthread로 구현

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections 코드의 어느 부분을 병렬로 실행시킬 수 있는지 개발자가 지정할 수 있다
- Manages most of the details of threading GCD는 스레딩의 세세한 것을 관리함
- Block is in "^{ }" : 이 부분을 스레드에 할당해서 실행

```
^{ printf("I am a block"); }
```

- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

Block은 dispatch queue에 들어갔다가
사용 가능한 스레드가 있으면 할당함

Grand Central Dispatch

Process 내에 있기 때문에
private dispatch queue라 부름

- Two types of dispatch queues:
  - **serial** – blocks removed in FIFO order one at a time, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - **concurrent** – removed in FIFO order but several may be removed at a time
    - Four system wide queues divided by quality of service:
      - QOS_CLASS_USER_INTERACTIVE 신속한 응답이 요구될 때
      - QOS_CLASS_USER_INITIATED 적당한 시간의 응답이 요구될 때
      - QOS_CLASS_USER_UTILITY 긴 시간이 요구될 때
      - QOS_CLASS_USER_BACKGROUND 시간에 관계없는 task일 때

빠른
응답

System-wide queue이기 때문에
global dispatch queue라 부름

Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

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

- The same for loop written using TBB with **`parallel_for`** statement:

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

Windows
Linux
macOS

Open-source
or
Commercial

Threading Issues

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

내부에서 발생 ex) illegal memory access, divide by zero

외부에서 발생 ex) Ctrl-C, timer

Semantics of fork() and exec()

- Does **`fork()`** duplicate ① only the calling thread or ② all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads

fork() 후에 exec()가 바로 수행되는
경우 ①이 유리함

Linux는 ①을 지원함

“Don't use both threads and forks”

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
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. 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

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies → Ex) divide by zero
  - Deliver the signal to every thread in the process → Ex) Ctrl-C
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Unix signal 생성 함수

kill(pid_t pid, int signal);
pthread_kill(pthread_t tid, int signal);

대부분의 Unix 계열의 OS에서는 스레드마다 받는 또는 거부하는 signal을 설정할 수 있음. 따라서 다중 스레드인 경우 signal을 허용하는 첫 번째 스레드가 처리함

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 → 할당된 자원을 모두 free하지 못할 경우가 생길 수 있음
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- 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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Cancellation point에서 스레드를 캔슬함.
일반적으로 read()와 같은 blocking system call이 cancellation point가 됨.

Thread Cancellation (Cont.)

- 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**

    인위적으로 cancellation point를 지정할 때 사용
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked

    Q: 이 함수를 호출하면
    다시 이 자리로 리턴될까?
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) 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)
- 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

스레드는 자신이 속한 프로세스의 데이터를 공유한다. 그러나
경우에 따라서는 자신만의 데이터가 필요할 수 있다. → TLS

스레드 pool을 사용하면 사용자는 스레드 생성에
관여하지 않기 때문에 TLS가 유용함.

Linux Threads

- 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)
  - Flags control behavior

fork() and pthread_create() eventually call clone().

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.

- **`struct task_struct`** points to process data structures (shared or unique)

fork()는 이 자료구조를 copy함 (복사로 처리)
clone()은 이 자료구조를 point함 (링크로 처리)

End of Chapter 4