4장. Threads & Concurrency
0. 개요
1. 다중 코어 프로그래밍
2. 다중 쓰레드 모델
-User level에서 지원하는 MultiThread
-Kernel level에서 지원하는 MultiThread
-U-K 간 매핑 관계에서 동작하는 Thread
3. 쓰레드 라이브러리
Kernel level에서 쓰레드가 제공되나, 라이브러리 level에서도 쓰레드 제공됨
4. 암묵적 쓰레딩
더 쉽게 쓰레드를 이용할 수 있도록 위한 방법 제공
5. 쓰레드 이슈
6. 관련 OS 예시
-OS에서 어떠한 쓰레드가 제공되고 어떻게 쓰레드를 구현했는지

Threads 장점
-process 자원 공유하여, multiprocess에 비해 multiThreads는 훨씬 경량화, 좋은 성능과 동시성을 제공할 수 있음, 따라서 Threads는 Multicore Programming에 최적화되어 있음
H/W적으로 multicore : 하나의 CPU 칩 안에 여러 개의 CPU 역할하는 core들이 여러 개 있는 것이고, multiCPU : 여러 개의 CPU가 존재하는 것

0. 목적
-쓰레드의 기본적인 사항 및 프로세스와의 차이점
-OS나 라이브러리에서 제공되는 쓰레드 API를 이용해서 multiThreaded AP를 어떻게 작성할 것인가? (Concurrency한 프로그래밍 구현)
-Implicit Threading를 자동으로 생성해주는 유용한 라이브러리 기능들
(thread pool, fork-join, Grand Central Dispatch 등)
-Windows나 Linux에서 Thread를 어떻게 작성하고 구현하였는지
-MultiThreaded AP을 작성하기 위해 라이브러리 레벨(OS 레벨 X)에서 제공하는 쉬운 인터페이스(API - Pthreads, Java, Windows Threading APIs 등)

1. MultiThreads
MultiThreads 사용 이유
-현재 대부분 AP는 multiple tasks를 동시에 수행하기 위해서 multithread 이용
ex. AP이 Update display, Fetch data, Spell checking, Answer a network request를 동시 제공하면, 하나의 Thread로 처리하기 어려움(각 task를 독립적 Thread 만들어 수행)
-Multiprocess에 비해 MultiThreads의 장점
Multiprocess 또한 여러 개의 process를 띄워 동시에 여러 일을 수행할 수 있으나, 각 proc는 proc table, proc 간 통신 비용(시간) 등 컴퓨팅 자원을 많이 차지한다.
그런데 MultiThreads에서는 thread 생성하는 것이 신속하고 같은 프로세스 안 여러 개의 쓰레드가 자원(프로세스 내 전역 데이터)을 공유함으로써 쓰레드 간 통신에서 컴퓨팅 자원(시간)

또한 적게 차지한다. (+ MultiThreads를 이용함으로써 코드 단순화 및 효율성 향상 가능)
커널과 라이브러리에서도 MultiThread 제공됨
-Process vs. Thread
Single-Threaded Process : 프로세스 내 쓰레드 하나 존재 (지금까지 본 것)
MultiThreaded process : 프로세스 내 여러 쓰레드 공존
-Thread도 prog 수행하나 단지 한 process 안에 code(명령어, instruction), data(크기가 큼), files(access하려는 파일)은 공유하고, 각자 가지고 있어야 하는 PC, stack, register 등의 자원은 개별로 보유함.

-Multithreaded를 서버에 활용한 예시(Multithreaded Server Architecture)
웹서버, 파일서버등 다양한 서버가 존재하는데, 여기에 여러 Client가 동시에 다양한 형태의 서비스 요청하고 이를 서버가 처리하고 응답함.
이때, 서버를 하나의 쓰레드로 구현하면 여러 client의 요청을 하나씩 순차적으로 처리해야함
이는 client들에게 제공하는 서비스의 속도를 매우 느리게 만들 것임
multithread로 구현하는 것이 좋음
동시 접속하는 Client 숫자만큼 쓰레드를 생성함으로써 동시에 여러 Client 요청 수용 가능

-MultiThreads 사용 장점
a. Responsiveness(응답성)
하나의 프로세스가 블록 당하더라도, 다른 쓰레드가 계속해서 움직일 수 있으므로
b. Resource Sharing
프로세스 내 자원을 공유함으로써 자원 적게 차지, 전역 데이터 공유해 쓰레드 간 통신 가능
c. Economy
Thread 생성이 process 생성에 비해 싸고, proc간 Context switching보다 Thread switiching이 더 적은 오버헤드를 가지고 있음
d. Scalability
Thread를 많이 만들어서 multicore architecture 이용하는 것과 같은 이점을 챙길 수 있다.

-Multicore Programming
Multicore or multiprocessor 환경에서 programming한다는 것은 기존 배운 것에서보다 난이도가 높음
a. Dividing activities
작업을 어떻게 쪼개 Multicore or multiprocessor에게 배분할 것인지
b. Balance
작업을 어떻게 load balancing해서 일부 core나 processor가 놀지 않도록 만들 것인지
c. Data splitting
큰 data를 처리할 때 어떻게 core, process에게 분배할 것인지
d. Data dependency
로직 상 data 의존성이 존재하는 경우, 동시 처리하지 않도록 따져봄
e. Testing and debugging
multicore나 multiprocessor는 작성하는 것보다 테스트와 디버깅이 더 어려움

#Concurrency vs. Parallel
-Concurrency는 S/W 적인 것, CPU가 하나더라도 multiprogramming, timesharing에 의해 여러 개가 동시에 수행되는 효과가 있음(실제로 한 순간에는 한 proc 수행)
대부분의 proc들은 cpu 사용하는 시간도 있지만, I/O waiting 시간도 많이 걸린다.
한 Thread가 I/O waiting 시간에 다른 proc들을 수행할 수 있기 때문에 S/W 측면에서 concurrency이다.
-Parallel은 실제로 H/W가 여러 개 존재한다는 것이다. core가 많거나, CPU가 많던지, computer가 많은 물리적인 환경을 parallel이라고 한다. (실제로 한 순간에 여러 proc 수행)

#Paralelism 종류 - 11p
a. data parallelism
Data를 쪼개 여러 일로 나누어 동시 수행
b. task parallelism
Data는 동일하지만 수학적, 논리적 연산 등의 해야 할 일을 쪼개 동시 수행

#Amdahl's Law
-컴퓨터 시스템의 일부를 개선할 때 전체적으로 얼마만큼의 최대 성능(시간 기준) 향상이 있는지 계산하는데 사용하는 법칙
-AP에서 어떤 Data나 Task에는 serial(순차적)한 부분과 parallel(병렬)한 부분을 동시에 가지고 있다.

S: serial portion (줄일 수 없는 부분)
P: parallel portion (1 – S) (N개 코어 동시 수행함으로써 줄일 수 있음)
N: processing core 개수일 때,
우리가 속도를 얼마나 높일 수 있는 것인가?에 대한 계산식

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$, 시간의 역수 = 속도

prog마다 serial portion이 있기 때문에, 무조건 core의 수에 비례하여 speedup 불가
(serial portion이 많은 prog의 경우 core 많아도 speedup 거의 없을 수도)

2. User Threads and Kernel Threads
※라이브러리 kernel 할당 관계 (계층 구조 이룸)
CPU는 여러 개의 Kernel Thread를 스케줄링하고 이 Kernel Thread 위에 User Thread가 존재하는데, 이는 kernel 쓰레드를 할당받아야 하는 이런 관계에 있음
User Threads : 라이브러리에서 MultiThreads를 제공하는 것
라이브러리에서 만든 Thread는 kernel의 쓰레드를 할당받아야 실제 해당 kernel Thread가 CPU 할당받았을 때 동작 가능
▶3가지 주요 Thread 라이브러리 : POSIX Pthreads, Windows threads, Java threads
Kernel Threads : 커널에서 MultiThreads를 제공하는 것 (대부분의 OS에서 이를 지원)

User Thread와 Kernel Thread의 Mapping 관계
(1) Many-to-One
여러 User Thread가 하나의 Kernel Thread로 매핑되는 관계
장점 ; 간단
단점 : 하나의 User Thread가 Kernel Thread를 할당받고 blocking 상태에 들어가면 나머지 Thread들이 더 이상 동작 불가(Concurrency 효과 떨어짐),
여러 Thread가 하나의 Kernel Thread를 할당받기 때문에 core가 많더라도 parallel 효과 없음 (한 순간에 동시 수행 X)
(2) One-to-One
User Thread마다 Kernel Thread 하나씩 매핑되는 관계
장점 : many-to-one에 비해 Concurrency 높음(CPU나 Core 여러 개 있을 때 kernel이 독립적 할당받아 여러 User Thread가 동시 수행 가능)
단점 :
(3) Many-to-Many
장점 : 여러 User Thread가 여러 Kernel Thread를 할당받을 수 있는 관계
단점 :
(4) Two-level Model
M:M와 유사하나, 모든 User Thread가 모든 Kernel Thread를 할당받는 것이 아니라, 일부 Thread 할당에 제약이 존재하는 관계

3. Thread Libraries
라이브러리 레벨에서 Thread를 제공하고 프로그래머가 쉽게 MultiThread 이용 가능한 API 제공
라이브러리는 User space이고, Kernel-level Thread는 OS가 제공함

#여러 가지 User Thread 라이브러리가 존재함
(1)Pthreads
-POSIX 표준(UNIX 계열에서 많이 채택하며, Linux나 Mac OS에서도 많이 사용)
-Pthreads 이용한 프로그래밍 방법(API 예시)
pthread attr_init(&attr); - pthread 속성 초기화
pthread create(&tid, &attr, runner[수행 목표 함수], argv[1]); - pthread 하나 생성
pthread join(tid, NULL); - thread 생성 이후, runner 함수 수행완료를 대기하는 것
마치, fork() 후 exce(runner)이후 wait()하는 것
+ pthread barrier() - 여러 쓰레드가 실행되다가 barrier를 만나면 모든 쓰레드가 barrier에 도달할 때까지 대기하는 것

(2)Windows Multithreaded C program
-윈도에서 제공하는 Multithreads API
-Windows Multithreaded 이용한 프로그래밍 방법(API 예시)
상황. Summation(LPVOID Param) - 쓰레드 만들어서 합계를 내는 일을 시키고 싶을 때

CreateThread(NULL, 0, Summation[수행 목표 함수], &Param, 0, &ThreadId); - Thread 생성
WaitForSingleObject(ThreadHandle, INFINITE); - Thread 함수 수행 완료 대기

(3) Java Threads
-Java에서 제공하는 Multithreads API
Thread 수행 방법 2가지
a. Thread class 만드는 방법
b. Runnable interface 구현하는 방법
-Java 이용한 프로그래밍 방법(API 예시)
Implementing Runnable interface; - 내부에 run()함수 구현
Thread변수.start(); - Thread 생성
Thread변수.join(); - Thread 수행 대기

#Java Executor Framework
지금까지 Pthreads, Windows, Java는 AP 프로그래머가 Thread를 생성하는 개념이었는데, 이를 explicitly creating thread라고 함.

Java Executor Framework 예시
public Integer call();
Executors.newSingleThreadExecutor(); - Thread pool에 미리 생성
pool.submit(); - pool에 있는 것에게 Task 할당
result.get(); -

4. Implicit Threading(암묵적 쓰레딩)
-목적 : MultiThread를 이용하는 것은 (쉬운 API를 제공함에도) 일반 프로그래밍보다 난이도가 있는데, 이를 더 쉽게 MultiThread 환경을 이용할 수 있도록 해주는 것

#AP은 OS를 직접 이용할 수 있지만, (OS에서 제공하는 API가 어려우므로) 라이브러리 등에서 제공하는 조금 더 쉬운 API를 이용하기도 함.
ex. TCP/IP 프로토콜 - Socket(이것의 쉬운 이용 도와주는 API) - RPC(socket 이용해, remote에서 제공하는 function을 바로 호출하는 것, 더 쉬운 이용)
-> 위로 Layer가 올라감으로써, AP이 더 쉽게 programming할 수 있는 환경 제공 받음

-정의 : 우리가 직접 쓰레드를 생산, 관리하는 것이 아니라 운영체제에게 쓰레드 생산, 관리를 맡기는 것 (프로그래머가 직접 만드는 것 X라, 컴파일러 or 라이브러리에 의해 쓰레드가 관리됨)
-5가지 메소드 : Thread Pools, Fork-Join, OpenMP, Grand Central Dispatch, Intel Threading Building Blocks

(1) Thread Pools
-미리 쓰레드를 많이 만들어 놓는 것
-interface Executor를 통해, Thread pool(여러 개 Thread 미리 생성)을 만들고, 이 thread들이 해야 할 일이 생겼을 때, 이런 task를 수행하겠다는 개념
-task의 생성과 수행을 분리한 개념
장점 : thread 생성 후 수행하면 지연 시간 생기는 것을 줄일 수 있음,
생성, 수행을 분리함으로써 더 유연한 MultiThread 환경 챙길 수 있음
-필요에 따라 Pool의 크기를 동적으로 조정 가능함
-Windows API에서 제공하는 Thread Pools 관련 함수 : PoolFunction
-Java에서 제공하는 Thread Pools 관련 함수 : Executor class 내 3가지 메소드 존재함
newSingle(단일)ThreadExecutor(), new(Fixed, Cached 쓰레드 수 변동 유무)ThreadPool()
ex. 실제 사용 사례
ExecutorService pool = **Executors.newCachedThreadPool**();
pool.**execute**(new Task());
pool.**shutdown**();

(2) Fork-Join Parallelism
-Fork와 Join API를 통해 새로운 Task를 생성시키겠다는 것
-main Thread에서 fork를 통해 새로운 Thread를 만들고 Task를 시킨 이후, Join을 통해 이 Thread들의 종료를 기다리며, main Thread로 합쳐지도록 하는 것
-필요한 만큼 fork 가능하며, 많이 fork 할수록 병렬성 효과 더 볼 수 있음
ex. 실제 사용 사례

	C언어	Java

사례	Task(problem)
	if ()
	else {
	subtask1 = fork(new Task(subset of problem))
	subtask2 = fork(new Task(subset of problem))
	result1 = join(subtask1)
	result2 = join(subtask2)
	return combined results
	}
		ForkJoinPool pool = new ForkJoinPool();
		int[] array = new int[SIZE];
		
		SumTask task = new SumTask(0, SIZE - 1, array);
		int sum = pool.invoke(task);

#ForkJoinTask는 abstract base class (in Java)
#RecursiveTask, RecursiveAction classes extends ForkJoinTask
#RecursiveTask는 반환값 O(compute함수 통해서), RecursiveAction는 반환값 X

(3) OpenMP(C, C++, FORTRAN에서 제공됨)
-멀티스레드 기반의 공유 메모리 병렬 프로그램을 위한 표준 API
-parallel regions로 식별(ex.#pragma omp parallel 지역 내 명령어 병렬로 수행해주세요)

예시는 compiler directive(컴파일러 지시문)에 속함
-구성: 컴파일러 지시어, 런타임 라이브러리(함수), 환경 변수
a. 컴파일러 지시어: omp parallel
b. 런타임 라이브러리: omp_set_num_threads(n)
c. 환경변수: OMP_NUM_THREADS=n

(4) Grand Central Dispatch
-OpenMP에서처럼 블록 내("^{}")에 병렬 수행할 부분을 지정
-두 가지 종류의 dispatch queue 존재 : serial, concurrent
serial : 할 일을 순차적으로 진행해야 하는 것(FIFO, 병렬성 X)
concurrent : 할 일을 동시에 수행해도 되는 것(서비스 질에 따라 구분하여 우선순위 O)
동시 수행하더라도, INTERACTIVE, INITIATED, UTILITY, BACKGROUND 4가지로 구분됨
-Swift언어에서 task는 closure로 정의되었으며, dispatch_async() 함수(API) 사용해 제시됨
dispatch_async(queue,{ print("I am a closure!") })

(5) Intel Threading Building Blocks (TBB)
-다중 코어 프로세서의 이점을 취하는 소프트웨어 프로그램을 작성할 목적으로 인텔이 개발한 C++ Template 라이브러리
-장점 : 원시 스레드보다 높은 수준에서 작동하지만 이국적인 언어나 컴파일러 필요 X
-serial 버전은 일반적인 순차 접근 방법
-parallel 버전도 동시 수행하는 일의 일반적 접근 방법
parallel for (size_t(0), n, [=](size_t I) {apply(v[i]);});

라이브러리는 다음과 같은 점에서 다른 라이브러리와 다릅니다. TBB를 사용하면 스레드 대신 논리적 병렬 처리를 지정할 수 있습니다. TBB는 성능을 위해 스레딩을 목표로 합니다. TBB는 다른 스레딩 패키지와 호환됩니다. TBB는 확장 가능한 데이터 병렬 프로그래밍을 강조합니다. TBB는 일반 프로그래밍에 의존합니다

단순히 기존의 스레드를 대체하는 것이 아니라, 플랫폼의 세부 사항과 스레드 처리 메커니즘을 추상화한 '태스크' 개념을 기반으로 하는 높은 수준의 병렬처리 기술
목표 : 성능(performance), 조정성(scalability)

5. Threading Issues
(1) fork(), exec() system calls의 의미[AP 목적에 따라 조정 가능]
a. process를 fork()하면, 그 내부의 모든 Thread가 Child에게 다 복제가 되는지 아닌지?
->조정 가능 (fork()의 두 가지 version 존재)
b. process를 exec()하면, 그 내부의 모든 Thread가 새로운 프로그램 수행하는 것인지?
-> //

(2) Signal handling(Synchronous and asychronous)

Signal : 일종의 S/W interrupt
-Signal이 왔을 때, INT handler routine과 같이 Signal handler routine에 따라 처리
(OS에서 미리 작성한 default 값 있음, 사용자가 정의한 Signal handler routine도 있음)
ex. Signal 왔을 때, Multi-Threaded 인 경우 반응하는 방법 (default로 정해져 있음)
모든 Thread가 다 signal을 받는 것인지? or 특정 Thread가 signal을 받는 것인지?

(3) Thread cancellation of target thread(Asynchronous: 즉시 or deferred : 연기)
Thread Cancellation : Thread를 종료시키는 것 (INT의 일종)
Target Thread : Thread Termination을 받은 쓰레드
-Target Thread가 하는 두 가지 접근 방법 (Asynchronous vs. Deffered)
a. Asynchronous cancellation(아무때나 cancel 받음)
쓰레드 종료시킬 때, Target 쓰레드가 즉시 종료되는 것
b. Deffered cancellation(취소가 disable(INT 받으면)->pending 없어지는 것 X라, 보류됨)
쓰레드 종료시킬 때, Target 쓰레드가 cancel을 연기하여 특정 시점까지 가서 종료되는 것
#Deffered의 경우, cancellation point에 도달할 경우에만 취소가 발생함
ex. pthread_testcancel()으로 cancel이 왔는지 확인하는 지점에서 취소를 받음
-실제 사용 사례

	POSIX pthread	Java

사례	pthread_t tid;
	pthread_create(&tid, 0, worker, NULL);
	pthread_cancel(tid);
	pthread_join(tid, NULL);
		Thread worker;
		worker.interrupt();
		while (!Thread.currentThread().isInterrupted()){
		…
		}

(4) Thread-local storage (TLS)
쓰레드마다 독립적으로 가지고 있는 공간(저장소)
-쓰레드 간 구별해야 하는 Data가 존재할 때, 이 TLS에 저장함
-쓰레드 생성 제어하지 않을 경우(쓰레드 pool 이용할 때), 유용함
-지역 변수(단일 함수 호출 시에만 확인 가능)와는 다름
-TLS는 함수 호출 사이에서 확인 가능한 특징이 있고, 정적 데이터와 유사함

(5) Scheduler Activations
-사용자 스레드 레벨과 커널 스레드 레벨과의 통신을 하는 매커니즘
(Scheduler Activations은 upcalls 제공)
Upcalls : 아래에서 위로 신호를 주는 것 (위에서 아래로는 function call)
LightWeight process(LWP) : 사용자 스레드를 수행하기 위한 가상 프로세서
(User Thread는 수행을 위해, LWP를 할당받아야 하며, 이는 커널 쓰레드마다 하나씩 존재)
*Kernel 쓰레드가 수행하기 위해서, CPU나 core를 할당받아야 함

6. Operating System Examples

실제로 OS에서 Thread가 어떻게 구현되어 있는지

(1) Windows Threads
-Windows API : 이를 통해 programmer가 쉽게 multi-thread 구현할 수 있는 환경 제공
-one-to-one 매핑 관계 및 kernel-level로 구현하여, Concurrency를 쉽게 확장 가능함
-Window에서 제공하는 thread 특징
a. Thread id
b. Thread마다 register 값, stack 등을 가지고 있음
c. user stack과 kernel stack은 독립적임
d. private data storage도 제공(Thread마다 독립적으로 가져야 할 data 저장 공간)
e. thread의 context(현 상태라는 의미) : register set, stacks, private storage가 속함
f. Windows Thread를 관리하기 위한 주요 자료구조(thread table) 3가지 존재
ETHREAD, KTHREAD, TEB (Windows는 process의 경우, PCB을 통해 관리하였음)
#Kernel 영역
-ETHREAD(executive thread block)
쓰레드 시작 주소, 자신이 담겨 있는 parent process를 가리키는 pointer

-KTHREAD(kernel thread block)
커널 쓰레드 자료구조이며, scheduling, 동기화, kernel stack에 대한 것을 저장함
또한, user Thread 자료구조를 가리킴
#User 영역
-TEB(thread environment block) : user Thread 자료구조
Thread id, user stack, thread-local storage를 가지고 있음

(2) Linux Threads
Linux는 Threads라는 말 대신 Tasks라는 표현 사용
-Thread 생성을 위한 clone()이라는 sys.call 제공
-clone()의 경우, child task가 parent task의 어느 수준까지 물려받는지, flag로 제어 가능