Ch4. Threads (완)

Created	@May 9, 2020 4:29 PM
Tags	운영체제

What is Thread?

- Smallest unit of execution.
- **Real unit of execution!**
  - Windows, Linux 등 대부분의 Kernel의 execution unit (실행의 기본 단위)는 프로세스가 아니고 스레드임
- 스케쥴링의 단위.
- 한 프로세스 내의 스레드들은 모든 Memory content를 공유한다. 각 스레드들은 그 주소로 가서 그냥 읽고, 변경하면 된다.
- 어떤 모델을 써야 하는가? Multiprocess? Multithread? 각각 장단점이 있다. Multithread가 항상 좋은 것은 아니다.
- 여러 개의 실행에서 프로세스끼리 무언가 해야 할 때가 있다. 하지만 프로세스끼리 무언가를 공유하는 것은 어렵다. OS는 프로세스를 독립 (Isolate)시킨다. Isolation은 굉장히 강하다. 깨는 것은 프로그래머에게도 힘들고 오버헤드가 크다.

Why use thread?

- **Data를 공유하는 것이 더 쉽다!**
  - IPC(InterProcess Communication)를 사용할 필요가 없다. Pipes, Named pipes, Signal, Socket communcation, ... 이런 거 다 필요 없다.
- **Higher parallelism으로부터 오는 더 나은 성능**
- **Higher parallelism →→→ Better perf.!**
  - 새 스레드를 만드는 것은 새 프로세스를 만드는 것보다 더 적은 노력이다. **스레드 생성의 오버헤드가 프로세스 생성의 오버헤드보다 훨씬 적다.**
  - 여러 개의 스레드를 빠르게 만들 수 있다 → Higher parallelism.

    - 프로세스 생성은?
        - Remind: fork() system call을 호출하면 어떤 일이 벌어지는가?
        - 메모리 내용 전체를 한 프로세스에서 다른 프로세스로 복사. Memory copy가 일어난다.
        - 이 작업은 Kernel에게는 아주 heavy한 작업이다. 시간도 오래 걸리고, 리소스도 많이 필요하다.
    - 반면, 스레드 생성은?
        - Kernel 안에 스레드를 나타내는 자료구조 하나만 더 만들면 된다.
        - (Process의 PCB처럼, 스레드의 Running state 등을 나타내는 그런 자료구조가 있다.)
        - Memory copy 등 리소스 복사는 일절 필요 없다. 이미 메모리와 프로세스는 생성되어 있기 때문에.
- **자원을 더 효율적으로 사용 가능**
    - 프로세스 생성 : resource-intensive (자원을 많이 사용함)
        - Remind: 프로세스를 생성하려면 Memory copy해야 하고, 이는 heavy한 작업.
    - 어떠한 작은 작업을 병렬로 수행하고 싶을 때, 짧은 병렬 작업을 위해서 프로세스를 계속 만들고 종료하고 반복한다면? → 계속해서 메모리 복사가 일어나기 때문에, 아주 작은 작업을 위해 프로세스를 계속 만드는 것은 낭비.
    - 반면, 스레드 생성은 이에 비하면 아주 작은 작업. → 따라서 자원을 더 효율적으로 사용하는 것이라 할 수 있다.
- **Multi-threading is common now.**
    - 대세다. 멀티스레딩을 배우는 걸 피할 수 없다.
- Benefits:
    - **Responsiveness** : 프로그램이 Disk I/O 때문에 Blocking되어 있거나 하더라도 다른 User input 등의 요청으로부터 바로 반응할 수 있다.
        - What means 'more responsive?' : 프로그램이 요청으로부터의 반응이 더 즉각적이다.
    - **Resource Sharing** : *한 프로세스 내의 스레드들은 Memory content를 공유한다.*

- **Economy** : *Memory copy가 일어나지 않으므로, 스레드 생성이 더 경제적이다.*
- **Scalability** : 스레드를 얼마나 많이 추가하든, 시스템은 계속 작동한다.
    - 각 프로세스는 각자만의 기본적인 메모리 영역이 필요하다. 프로세스를 계속 생성한다 해도 더 많은 메모리를 소모할 것이고, 프로세스가 소모하는 메모리가 적다고 해도 물리적으로 한계가 있다. Process is not scalable.

Multicore Programming

멀티스레드 하다가 왜 갑자기?

- Multithread : 추상적인 개념
- Multicore : 물리적인, CPU 하드웨어에 관한 개념

Multicore processor

- CPU 하나에 여러 코어
- 싱글코어로는 성능 향상의 한계가 있다 → 멀티코어 등장.
- 이제 프로그래머는 멀티코어 하드웨어를 최대로 활용하려면 멀티코어 프로그래밍 (= 멀티스레드 프로그래밍)을 해야 한다. 기존의 싱글코어 프로그래밍으로는 전체 하드웨어를 utilize하지 못하는 것.

둘 사이의 관계는?

- **각 코어는 1개의 스레드만 돌릴 수 있다.**
    - 4코어 CPU에서는 최대 4개 스레드만 동시에 실행 가능하다는 것.
- 프로그램을 멀티스레드로 짜두면 OS가 알아서 코어에 스레드 분배한다.
    - 그럼 프로그램에 4개 스레드만 만들어야 하는가? 아니다.
    - 스레드 100개 만들 수도 있고, 2개만 만들 수도 있다. 하지만 동시에 실행 가능한 스레드는 오직 CPU 코어 개수 = 4개뿐이라는 것.

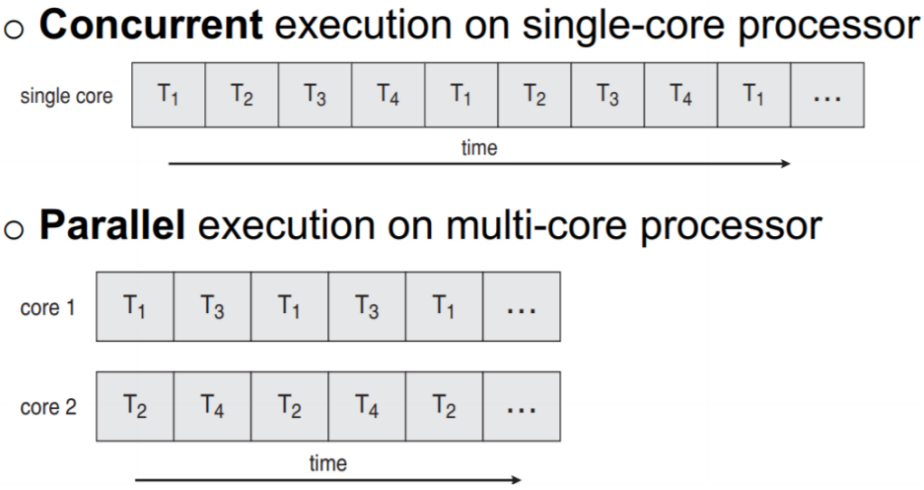
Concurrency vs. Parallelism

- **Concurrency**
    - 여러 작업을 진행할 수 있는 능력.

    - 실제로 동시에 여러 작업이 처리되는 것은 아니다.
    - 논리적 레벨. 싱글코어, 멀티코어 상관없이 Time-sharing으로 CPU를 나눠 사용함으로써 여러 작업이 병렬로 처리되는 것처럼 보이게 할 수 있다.
    - 가짜 Parallelism.
- **Parallelism**
    - 여러 작업을 동시에 수행할 수 있는 능력.
    - 실제로 동시에 여러 작업이 처리되는 것.
    - 물리적 레벨. 오직 멀티코어에서만 가능.
    - 진짜 Parallelism.

> Concurrency is about dealing with lots of things at once.
> Parallelism is about doing lots of things at once.

4개의 작업 T1~T4가 있다고 할 때:



Concurrent하지만, Parallel하지 않은 경우는 있을 수 있다.

- 서로 다른 여러 작업을 진행하기는 하지만, 한 번에 하나의 작업만 수행하는 것. 즉, 동시에 여러 작업을 수행하지는 않는 것.

Parallel이지만 Concurrent하지 않은 경우도 있을 수 있다!

- 하나의 task만 진행하는데, 그 task가 여러 개의 subtask로 쪼개어져서 동시에 여러 subtask를 수행하는 경우.
  다루는 task는 하나지만, 동시에 수행되는 task는 여러 개의 subtask들이다.

Concurrent하지도 않고, Parallel하지도 않은 경우:

- 한 번에 하나의 작업만 수행하고, 작업이 여러 개로 쪼개지지도 않는 경우.

Concurrent하면서 Parallel한 경우:

- 두 코어 모두 여러 작업을 진행하면서, 각 작업을 여러 개의 subtask로 쪼개는 경우.

Hyper-threading (인텔의 상표명)
: 1코어가 2개의 다른 thread를 돌릴 수 있도록 해 줌.

Hyper-threading이 켜지면, OS는 4코어 CPU를 실제 코어 수의 2배로, 8코어가 있는 것으로 인식.
OS는 이 8개 코어들이 진짜 코어인지 가짜 코어인지 모름. (HW level에서 일어나는 'illusion'이다. HW level은 Encapsulated, 추상화되어있기 때문에 OS는 실제 코어가 몇 개인지 모른다.)

따라서 OS는 진짜 코어인지 가짜 코어인지 상관하지 않고 그냥 8개의 서로 다른 스레드를 코어에 분배함. OS 입장에서는 Hyper-threading이 켜졌는지 꺼졌는지 상관하지 않아도 됨.

HW level에서 이루어지는 Parallelism 향상이라 할 수 있다.

In reality, 코어 개수를 2배 해도 성능 향상은 2배가 되지 않는다. 수행 시간이 딱 절반이 되지 않는다. 가상 코어이기 때문. 무한정 성능 향상이 되는 게 아니다.

New Challenges from Multicore

멀티코어 CPU와 멀티코어 프로그래밍의 등장 → 끝난 게 아니다. 하드웨어를 최대한 utilize하기 위한 새로운 Problem들이 발생.

- **Identifying tasks**
  - **프로그램의 어느 부분이 독립적이고, 병렬화 가능한가? 이를 확인하는 것은 어렵다.**
  - 프로그램 로직 자체를 여러 개로 쪼개는 것이 어려워서, 프로그램이 병렬로 돌아가게 만드는 것(병렬화)이 어렵다.
- **Balance**
  - **여러 스레드에 배정된 각 작업이 밸런스가 맞아야 한다.**
  - 각 스레드에 배정된 작업이 imbalance하다면, 즉 어떤 작업은 일찍 끝나고 또 다른 작업은 늦게까지 계속된다면, 전체적인 시간 향상은 없는 것과 마찬가지.
- **Data splitting (Data imbalance problem)**
  - 각 작업에 필요한, 할당해야 하는 데이터가 어떤 작업은 데이터의 100%를 받아야 하고, 어떤 작업은 10%만 받아야 할 수도 있다. ??
  - 이거 뭔 소린지 모르겠다
- **Data dependency**
  - **서로 다른 스레드의 작업끼리 Data dependency, 데이터 의존성이 있는 경우.**
  - 한 작업이 다른 작업의 output을 기다리는 경우 등.
  - *Data dependency determines the degree of parallelism.*
- **Testing and debugging**
  - Many different execution path : 이거도 모르겠다
  - Race condition : 어떤 병렬 실행의 결과가 매 실행마다 바뀜. (병렬 실행의 경우 instruction끼리 실행 순서가 바뀌는 경우가 있다. 나중에 또 나온다 함)
  - 이러한 병렬 실행의 non-deterministic behaviours 때문에 테스팅과 디버깅이 어렵다.

In reality, 이러한 Challenges 때문에 멀티코어라고 해도 성능 향상을 꾀하는 게 그렇게 쉽지만은 않다.

Types of Parallelism

여기 안 하고 다음 챕터 한다고?

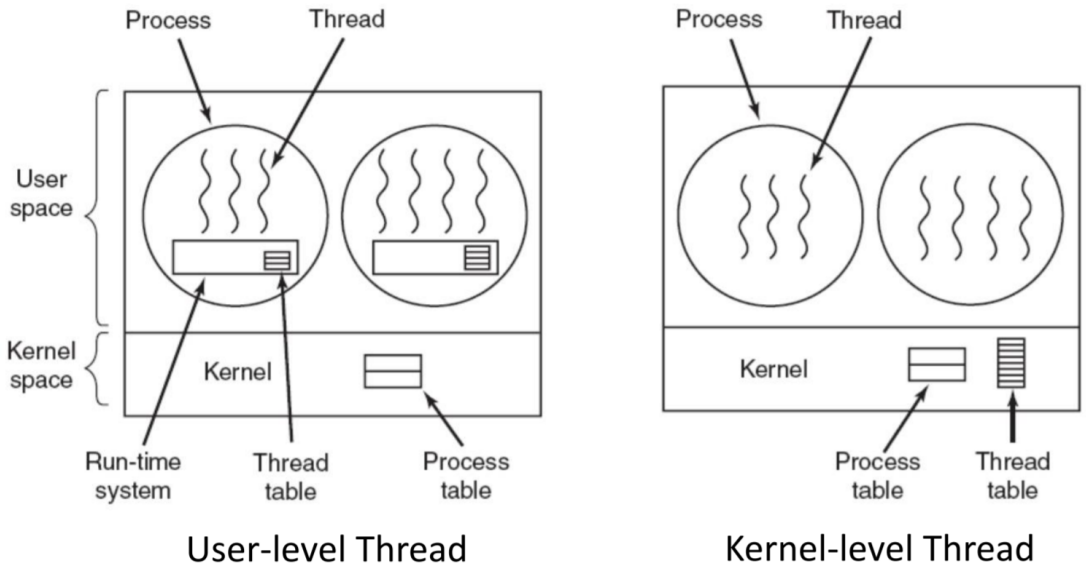
Multithreading Models

= 많은 스레드들을 어떻게 관리하는지. 다수의 스레드들을 어떻게 운영하는지에 대한 방법들

우선 User thread, Kernel thread 두 개념을 알아야 한다. **두 개의 개념이 명확해야 한다. 명확히 알고 있어야 한다.**

- **User thread (User-level thread)**
  - **스레드를 나타내는 자료구조들이 User-level에서 관리됨. User-level space에 저장됨.**
  - **스레드 생성, 삭제, 스케쥴링 등 모든 작업들이 프로그램 내의 library function 임.**
    라이브러리 형태로 프로그램 내에 존재.
  - 모든 스레드 자료구조와와 라이브러리는 프로그램의 일부일 뿐.
    스레드 관리 자체가 OS와 상관없이, 프로세스가 프로그램의 일부분으로 직접 관리.
  - 스레드 상태 나타내는 자료구조 : PCB와 비슷. (TCB 정도가 되겠다)
  - User-level Thread State: 대충 이런 것들이 들어간다
    - Thread ID
    - Register state (PC, Stack pointer, ...)
    - Stack, Signal mask, Priority, Thread-private storage
  - 커널은 프로세스 내에서 User-level thread에 관해 어떤 일이 일어나는지 모른다.
    커널은 프로세스가 실행중인 것만 보이고, 프로세스 내부적으로 User thread를 쓰는지 안 쓰는지, 몇 개인지 이런 것들 다 모른다.
- **Kernel thread (Kernel-level thread)**
  - **모든 스레드 자료구조들, function logic들이 커널 영역에 저장된다.**
  - **커널이 각각의 스레드를 관리할 수 있다.**
  - 커널은 이러한 Kernel thread를 스케쥴링의 대상이 되는 개체로 삼는다.
  - 현존하는 거의 모든 OS는 kernel thread를 지원한다.

- 커널 내부에는 현재 커널 내에 있는 스레드를 나타내는 table/linked list가 있다. 커널은 어떤 스레드가 어떤 프로세스에 속하는지 알 수 있다.



User-level Thread　　Kernel-level Thread

(왼) User-level Thread:

- User space : User space 내에 프로세스 2개. 각 프로세스 내에 스레드 3개, 4개.
  - **Run-time system** : 스레드를 관리하는 로직 모음 (정도로 알고 있으면 됨)
  - **Thread table** : 1 프로세스당 1 Thread table. 각 프로세스마다 하나씩 있다. 각각의 프로세스들이 자신이 만든 User thread를 자체 관리.
- Kernel space : Process table밖에 없다.
  - 커널이 볼 수 있는 개체는 프로세스뿐이고, User thread는 볼 수 없다.

(오) Kernel-level Thread:

- User space: 똑같음. 근데 프로세스마다 Thread table은 없다.
- Kernel space:
  - Kernel space 내부에 Thread table이 하나 있다.
  - 현재 컴퓨터 내에 있는 모든 Kernel thread들을 직접 관리.
  - 프로세스 테이블도 있으니까 프로세스 목록도 알 수 있다.

Pros & Cons

User thread

- Pros
  - **빠르고 가볍다.** 스레드 관련 모든 작업이 단순히 Library call.
    System call이 연관되지 않는다. 단순히 Library function들만 호출하면 된다.

    System call이 call된다면?
    → 현재 프로세스 context 저장, kernel mode 전환, kernel 내부 로직 수행, 나오면서 user mode로 전환, ...
    단순히 함수 호출하는 것보다 더 많은 일들이 일어난다.

- Cons
  - Kernel이 각 프로세스 내의 스레드를 볼 수 없다. 프로세스만 볼 수 있을 뿐이다.
  - Kernel이 스레드에 코어를 더 할당한다거나, 스케쥴링한다거나 하는 선택을 할 수 없다.
    애초에 프로세스 내의 User thread 수를 알 수도 없기 때문에.
  - 프로세스가 I/O Request 등으로 Ready queue에서 Wait queue로 들어가게 되면, 그 프로세스 내의 User thread들이 전부 멈추게 된다.

    프로세스가 Schedule된다 == 프로세스가 돌기 시작
    프로세스가 Descuedule된다 == CPU를 잡고 있다가 뺏기고 나오는 것

Kernel Thread

- Pros
  - **Flexibility : Kernel이 유연한 결정을 할 수 있다.**
    Kernel이 개별 Kernel thread들을 인식할 수 있으므로, Kernel이 한 스레드가 I/O에 들어갔을 때, 다른 스레드에게 CPU를 사용하게 한다든지 해서 CPU를 쉬지 않게 하는 등. 스레드를 더 많이 가지고 있는 프로세스에세 CPU를 더 준다던지.
- Cons
  - **Kernel Thread의 생성은 User thread와 비교했을 때, 느리다.**
    스레드 생성에 System call이 수반되고, mode switch가 필연적이고, system

call handling이 필요하고, ...
커널 내부에서 자료구조가 생성되고, 이 크기는 User thread에서의 그것보다 더 큼. (Field가 더 많음)

Multiplexing User Threads

User thread와 Kernel thread를 어떻게 연관시킬 것인가. 이 둘을 어떻게 'mapping'할 것이냐.

근데 왜 둘을 연관시켜야 하지?

User thread는 실제로 일을 할 능력이 없다!

프로그램에서 병렬적으로 해야 하는 작업의 수를 기준으로, 필요한 병렬성의 개수만큼 자료구조를 만들어둔 것 뿐. User thread는 자료구조일 뿐이다.

실제로 일을 할 수 있는 개체는 Kernel 내부에 있는 Kernel thread이다. Kernel thread만이 실제 일을 할 능력이 있다.

Kernel thread 자체가 Kernel에 의해 Scheduling되어서 실제로 CPU를 할당받고, Kernel thread 내부에 저장된 PC값, 현재 instruction 주소로 가서 instruction을 수행하는 것.

파일에서 단어 몇 개인지 찾는 프로그램을 생각해보자.
파일은 4개 있다. 그럼 4개 User thread를 만드는 게 낫겠지? 그리고 각 스레드는 오직 각 하나의 파일만을 담당하면 된다. 4개 스레드가 끝나면, 다 더하고 끝내면 됨.
→ **Optimal degree of parallelism : 4**. (파일 4개니까)
(Data dependency가 deg. of parallelism을 결정한다고 했는데 이 소리? 애초에 degree of parallelism의 개념은?)
프로그램의 특성에 따라

4개의 User thread가 생성되고, 프로그램 메모리에 4개의 자료구조로서 존재하게 된다.
그러나, 이들은 그저 스레드의 현재 상태를 저장하는 자료구조일 뿐이다.
실제 일을 하려면, Kernel thread의 도움이 있어야 한다. 자료구조들은 단지 상태일 뿐이다.

따라서, User thread를 Kernel thread에 mapping해줘야 한다. 매핑 후에 User thread가 의도한 일을 하는 것.

어떻게 구현하느냐에 따라 다르지만, 보통:

- User thread 내의 상태 정보를 Kernel thread의 내부 정보로 옮겨주고, Kernel thread의 PC를 보고 instruction을 수행하면 된다.
- Kernel thread 내부에 User thread 구조체를 가리키는 포인터를 만들고, 나중에 이 Kernel thread가 Scheduling되면 가리키는 User thread를 따라가서 똑같이 수행하면 됨.

매핑이 끝나면, Kernel thread는 매핑된 User thread의 상태 정보에 접근할 수 있고, CPU 자원을 얻기 위해 기다린다.

Kernel이 많은 커널 스레드를 스케줄링할 것이다.

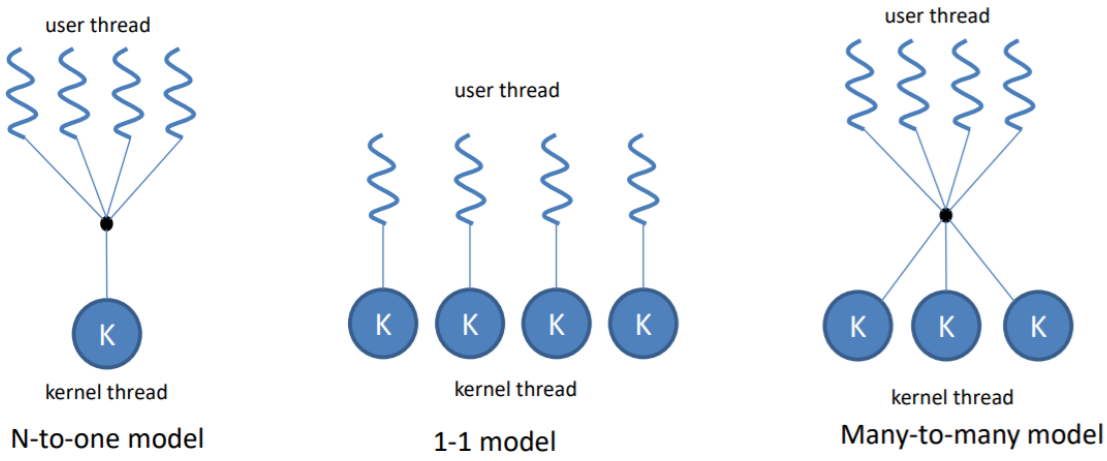
Kernel thread는 Kernel space에 존재하기 때문에, User application은 Kernel의 도움 없이 커널의 아무 것도 변경하지 못하기 때문에

system call을 수반한다.

Clarification:
Multiplexing User Threads : User thread 여러개 중 일부만을 선택해서 Kernel thread에 매핑 시도 *(Multiplexing : 여러 개의 input 중, input의 일부만을 선택해서 출력)*
In general, User thread의 수가 Kernel thread보다 많다는 것도 알 수 있다.

OS는 Kernel thread를 Scheduling한다? Process가 아니고?
CPU를 받는 개체도 프로세스가 아니라 Kernel thread?
Process를 Schedule한다 == Kernel thread를 Schedule한다
사실 리눅스 내부적으로도, 리눅스는 프로세스 기반 스케쥴링을 하지 않는다.
Kernel thread를 대상으로 스케쥴링을 한다. (Remind: task_struct)

어떻게 mapping하느냐에 따라 3가지 model이 있다.



- **N-to-one model :** 여러 User thread가 하나의 Kernel thread에 매핑
  - Pros
    - **빠르다.** Syscall이 연관되지 않는다.
      Remind: 모든 User thread들은 Userlib에 의해 관리되고, 모든 작업은 단지 library call. Kernel은 User thread의 존재조차도 모른다.
    - **Portable. 커널 의존성이 없다.**
      User thread는 Kernel의 도움을 받아 생성되는 것이 아니라, 프로그램 내에서 생성돼서 User space에서 관리된다.
      Kernel과 무관하다. Kernel의 어느 기능에 의존하지 않는다.
  - Cons
    - **No parallel run.**
      여러 User thread가 하나의 Kernel thread를 공유하므로, Kernel thread가 schedule되면, 그 스케쥴된 시간을 여러 User thread가 다 나누어가져야 한다. 추가적인 코어를 쓰지 않고 하나의 코어에서 하나의 실행 시간을 나눠 갖는 것이기 때문에 진짜 parallel이라 할 수 없다.
    - **Process block.**
      Remind: 여러 User thread 중 하나가 Disk I/O request 등이 일어나면, 그 프로세스 자체가 Ready queue에서 Wait queue로 넘어간다. 모든 User thread가 같이 따라간다. → 모든 스레드가 멈춘다.
- **1-1 model** : User thread 하나당 Kernel thread 하나. User thread 하나가 생길 때마다 전용 Kernel thread 하나가 생겨서 일일이 할당된다.
  (일대일 매핑 이야기는 이론적인 이야기이고, 어떻게 보면 User thread를 쓰지 않고 그냥 Kernel thread만 쓰는 방법에 가깝다.)
  - Pros

        - Kernel이 User thread 개수만큼 Kernel thread를 관리하게 되니까, User thread를 개별 관리하는 효과를 얻을 수 있다.
        - 스레드 하나가 blocked되면, 다른 스레드에게 CPU 줘서 돌리고, 멀티코어 CPU의 경우 User thread들이 진정한 병렬 수행을 할 수 있게 됨. (하나의 코어를 여러 스레드들이 시분할해서 concurrent하게, 병렬 수행되는 것처럼 보이게 하는 '가짜 병렬'이 아니라)
    - Cons
        - User thread 하나 생성될 때마다, Kernel thread도 생성되므로, Kernel에 자료구조가 생성된다. → Kernel의 자원을 소비하는 것.
        - → 스레드 관련 작업이 Kernel에서 이루어지므로, System call을 사용 → 느리다.
        - 프로그램이 User thread를 너무 많이 만들면, Kernel thread도 똑같이 늘어나므로, Kernel memory가 꽉 찰 수 있다. → User thread를 너무 많이 만들 수 없다. 제약이 있다.
- **Many-to-many model** : User thread 개수와 Kernel thread 개수가 서로 연관이 없다. User thread도 필요한 만큼 만들 수 있고, Kernel thread도 커널이 지원하는 개수만큼 만들 수 있다. 각자 자신에게 맞는 개수만큼 만들면 된다.
    - Pros
        - **Flexible.**
          프로그램은 프로그램의 특성을 반영하는 최적의 병렬성을 기준으로 User thread들을 만든다.
          프로그램은 Application concurrency를 지원하기 위해, 그에 맞는 개수만큼 User thread를 만든다.
          OS는 Physical concurrency를 위해
        - 매핑 관계는 프로그램 수행 중간에 동적으로 바꿀 수 있다.
    - Cons
        - (User thread들 중 매핑 안 된 애들도 생긴다.)
        - **Complexity.**

Thread Library

프로그래머가 처음에 스레드를 만들 때, Thread library의 function들을 호출해서 만들게 된다. API가 그렇듯, 파라미터 잘 맞추고 호출해서 thread id 받아오면 된다.

내부적으로 어떻게 구현되어 있는지에 따라, User thread library인지, Kernel thread library인지 나뉜다. 각각 User thread를 만드는 라이브러리, Kernel thread를 만드는 라이브러리.

요즘은 대부분 Kernel thread library이다.

실제로 보면, Kernel thread만 만들고 그걸 직접적으로 사용하는 식으로 구현된다.

User thread를 Kernel thread로 매핑한다고 했는데, 그건 이론적인 이야기이고, 실제 구현에서는 매핑하는 것보다 Kernel thread만 만들고 바로 사용하는 게 더 간단하다. 괜히 User thread 만들고 매핑하는 작업을 거칠 필요가 없다.

User thread library:

- 모든 기능을 User space에서 수행하도록 구현하면, 그게 User thread library이다.

  > 💡 User space : 애플리케이션이 사용하는 메모리 영역

- 커널의 도움이 필요 없음 (= System call을 사용하지 않음)
- 스레드를 나타내는 자료구조가 User space에 저장됨

Kernel thread library:

- 라이브러리 function들이 커널 내부의 System call을 호출해서, 커널 내부에 스레드가 만들어지게 됨
- 스레드를 나타내는 자료구조가 Kernel space에 저장됨

라이브러리는 인터페이스일 뿐. 내부적으로 어떻게 구현되어 있는지만 차이가 있다. 사용자(프로그래머) 입장에서는 디테일은 몰라도 되지만, 적어도 이게 User thread library인지 Kernel thread library인지 알고 사용해야 한다.

Existing Thread libraries

대표적으로 세 가지가 있다.

- POSIX pthread

- Windows thread API
- Java thread API

각각의 구현 방식

- pthread : 구현체는 user-level, kernel-level 둘 다 존재. 보통은 kernel-level 사용
- Windows thread : kernel-level library
- Java thread : higher-level library. 약간 좀 다름
  - low-level library를 호출한다.
    low-level library로 pthread나 Windows thread library를 사용한다.
  - low-level library로 어떤 걸 사용하느냐에 따라 특성이 달라진다.

Pthread Overview

pthread : POSIX 표준

> 💡 구현이 아니라, Specification이 표준인 것.
> 어떤 라이브러리 코드를 구현해서 제공하는 것이 아니라, 어떤 식으로 구현되어야 하고, 어떤 함수가 있어야 하고 파라미터는, 리턴 값들은 어떻게 되어야 한다 등 사항만 정의를 해 둔 것.

pthread API의 함수들 : 세 그룹으로 나눌 수 있다

- **Thread management** : create, exit, join, detach, ...
- **Mutexes** : lock, unlock
- **Condition variables** : init, signal, wait, destroy, ...

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

int sum;

void *runner(void *param) {
  int i, upper = atoi(param);
  sum = 0;
```

```
  for (i = 1; i <= upper; i++) sum += i;

  pthread_exit(0);
}

int main(int argc, char* argv[]) {
  pthread_t tid;  // pthread 변수
  pthraed_attr_t attr;  // pthread의 속성 저장하는 변수

  pthread_attr_init(&attr);  // 속성 변수 초기화
  pthread_create(&tid, &attr, runner, argv[1]);  // 스레드 생성
  pthread_join(tid, NULL);  // tid 기다리기

  printf("sum=%d\n", sum);
  return 0;
}
```

Java Thread

자바에서 일반적으로 클래스를 생성하는 두 가지 방법:

- Thread 클래스 상속받아서 새 클래스 만들고 run() 메소드 구현
- Runnable 인터페이스 implement하는 클래스 만들고 run() 메소드 구현

// 코드 있는데 공부 시작할 때 잠깐 보고 베껴두자

Thread Pool

Multithread 구조를 가지는 서버 프로그램에서, Client의 요청을 처리하려면?

→ **요청 하나당 스레드 하나를 만들어서, 그 리퀘스트를 전담해서 처리** 후 클라이언트에게 응답하고, 끝내면 된다.

이렇게 **하나의 스레드가 하나의 리퀘스트만을 전담**하게 하면 된다.

→ 리퀘스트마다 프로세스를 만드는 것보다 훨씬 더 빠르게 많은 수의 클라이언트를 상대할 수 있다.
프로세스는 스레드보다 만들 수 있는 개수가 훨씬 적으므로, 서버가 더 이상 리퀘스트를 처리할 수 없는 상태가 너무 빠르게 도달하더라.

그런데, 프로세스에 비하면 스레드 생성/삭제는 작은 일이 맞지만, 어느 수준 이상으로 올라가면 이 부하도 무시할 수 없다.

갑자기 사용자 리퀘스트가 폭발적으로 늘어나게 되면?

→ 매 리퀘스트마다 스레드를 만드는 식으로 구현하면 감당할 수 없게 된다. 서버가 다운될 수도 있다.
(스레드 생성 개수는 제한이 없다. 그래서 문제가 발생하는 것)

→ **Thread Pool 구조를 사용하자!**

- 개념 자체는 간단하다. 서버 프로그램 자체가 **처음에 정해진 개수만큼 스레드를 생성해두고, Pool에 넣고, 서비스를 시작하는 것.**
- 요청이 오면, 풀에서 스레드 하나를 꺼내서 쓰고, 처리가 끝나면 스레드를 다시 풀로 돌리고.
- 스레드의 개수는 처음에 만들어 둔 스레드 개수 이상으로 늘어나지 않는다. (새로 생성하지 않으므로.)
- 아무리 요청이 많이 모여도 서버에서는 단위 시간당 처리하는 최대 스레드 개수가 유지된다. 다운될 일이 없다.
- 단, 리퀘스트가 몰리면 요청을 스레드에 할당하기 위해 대기하는 시간이 길어지므로 사용자가 대기하는 시간이 길어진다.

> 💡 그래도 사용자 대기시간 길어지는 게 서버가 다운되는 것보다는 낫다.

Dispatcher : 리퀘스트 받아서 여유 스레드가 생기면 일을 주는 행위를 하는 애.

그럼 스레드 풀 사이즈 n은 얼마로 정할 것이냐? 적정 스레드 풀 사이즈는 경험을 바탕으로 적절히 설정해야 할 것.

Threading Issues

멀티스레드 프로그램을 코딩할 때 고려해야 할 사항들.

- **The fork() and exec().**
  - 하나의 프로세스에서 여러 스레드가 만들어졌는데, 그 스레드 중 하나가 fork를 call할 경우
  - 그렇게 해서 만들어진 child process는, thread를 어떻게 가져야 하는가?
    - parent가 가지고 있던 스레드 모두 가져가야 하는가? **(모든 스레드 복제)**
    - fork를 call한 그 스레드 하나만 child process 내에 만들어져야 하는가? **(fork를 call한 스레드만 복제)**
  - 이는 fork 후에 그 child process에서 exec를 부르냐 마냐에 따라 달라짐.
    - exec를 부르면, 스레드가 이전에 몇 개가 있었는가 아무 의미가 없어진다.
    - exec가 바로 call될 거면, 굳이 모든 스레드를 parent에서 child로 복제할 이유가 없다.
  - 실제로 리눅스에서는 fork 시스템 콜이 두 종류가 있다 - **fork, vfork**
    - **vfork는 parent process의 메모리 복사를 안 한다.**
    - 곧바로 exec할 건데 불필요한 메모리 복제 수고를 덜기 위함. 따라서 곧바로 exec할 것을 전제로 vfork를 사용해야 한다.
- **Signal Handling.**

  Remind : **Signal : IPC(InterProcess Communication)의 일종** (Pipes도 있었고 그랬다)

  - 특정한 상황이 발생했을 때, 일종의 신호를 주는 것. 그냥 number를 던져주는 것. 단순한 형태의 communication.
    - ^C (SIGTERM), SIGKILL, ...
  - Signal handler : 기본으로 Default handler가 존재하고, 사용자가 User-defined handler를 정의할 수도 있다.
  - 그래서, **어떤 Multithread process가 시그널을 받으면 어떻게 해야 하는가?** 시그널은 프로세스에게 전달된 것이고, 처리하려면 스레드 중 하나가 처리해야 하는데...
    - 시그널을 발생시킨 스레드에게 시그널을 전달해주자.

        - 프로세스 내의 모든 스레드에게 시그널을 전달해주자.
        - 특정 스레드에게만 시그널을 전달해주자.
        - 시그널 처리만 전담하는 스레드 하나를 띄워놓고, 그 스레드에게만 시그널을 전달해주자.

- **Thread Cancellation.**
    - 스레드를 취소하는 것 : 스레드가 완료되고 저절로 종료되기 전에 일찍 끝내는 것.

    한 프로세스에서 스레드를 여러 개 만들어서 특정한 파일 내에서 데이터를 찾으려 했는데,
    특정 스레드가 결과를 찾았다면, 나머지 스레드들은 돌 필요가 없다.
    답을 찾자마자 다른 스레드들은 의미가 없으므로 종료시켜야 한다. 이럴 때 Thread cancellation을 사용.

    - 두 종류의 Cancellation을 생각해볼 수 있다.
        - **Asynchronous cancellation** : 비동기. **캔슬 명령이 떨어지는 그 순간 바로 캔슬.**
            - Data consistency(데이터 일관성 문제) : **스레드가 데이터를 업데이트 하고 있었다면?** Data consistency가 깨진다.
            - **스레드가 메모리 할당했거나, 특정 부분에 lock을 걸어놨다면?** 그 스레드를 캔슬하게 되면 그 메모리를 반환하거나 lock을 풀지 못한다.
        - **Deferred cancellation** : **바로 캔슬되는 것이 아니고, 종료되기 전에 적당한 시간을 주고 뒷처리를 할 수 있게.**
            - pthread는 기본적으로 Deferred cancellation을 사용한다.
            - 코드 내에 캔슬해도 괜찮은 포인트들인 **Cancellation point**을 넣어뒀다.
              pthread_testcancel() 함수를 콜하면 그 지점이 cancellation point가 된다.
            - 스레드 내에서 그 라인으로 코드가 도달하기 전에 캔슬이 떨어지면,
              쭉 진행하다가 뒷처리할 수 있는 함수가 하나 콜돼서 메모리 반환하고 다 해서,
              그렇게 하고 나서 Cancellation point에 도달하면 스레드가 cancel되는 것.

Ch4. Thread 끝!