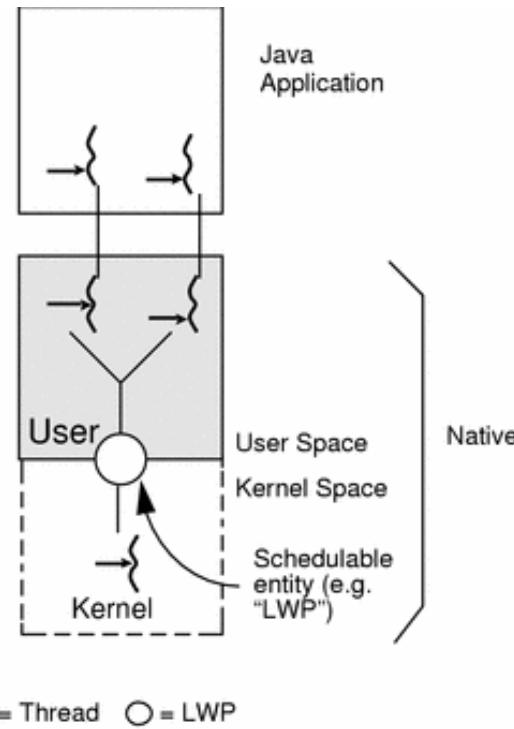


Green Thread

Many to One Mapping



장점

- Threads 간 Context Switching 하는 경우 System Call을 하지 않아 Native Threads 사용 보다 비용이 적게 듈다.
- Multi Threads 사용이 OS에 종속적이지 않다.

단점

- Kernel이 Thread에 대해 모른다. 따라서 **멀티 코어 제대로 활용 불가**
- User Level Threads 중 하나가 **Blocking I/O**를 하게되면, Process 자체가 Wait(Blocked)상태에 들어가기 때문에 **모든 Thread도 Block된다**.

구현

- Application에서 구현한 Thread이다. 따라서, OS는 해당 Thread에 대해서 알지 못한다
- 다수의 User Level Threads가 하나의 Kernel-Level(Native) Thread로 Mapping

Green Thread – 궁금한 점

구현

☞ 이것이 정확히 무슨 의미이지?

- Application에서 구현한 Thread이다. 따라서, OS는 해당 Thread에 대해서 알지 못한다
- 다수의 User Level Threads가 하나의 Kernel-Level(Native) Thread로 Mapping

생각 정리



STEP1

OS가 Process를 관리하기 위해 PCB를 생성하듯 마찬가지로 Thread를 관리하기 위해 TCB를 생성한다.

STEP2

OS가 알지 못한다는 것은 User에서 Thread를 생성한다고 해서 TCB가 생성되지 않는다는 것을 의미하는 것 같다.

STEP3

OS에서 TCB가 생성되지 않으므로, 당연히 kernel에 의해 scheduling 및 context switching이 발생하지 않을 것이다.

STEP4

그러면 User Code에서 Kernel의 dispatcher와 같은 Code를 구현해서 Thread를 관리해야 한다. 즉, App이 사용하는 Thread Library가 해당 기능을 구현해주어야 한다.

Green Thread – 궁금한 점

구현

- Application에서 구현한 Thread이다. 따라서, OS는 해당 Thread에 대해서 알지 못한다
- 다수의 User Level Threads가 하나의 Kernel-Level(Native) Thread로 Mapping

❖ 어떻게 Mapping이 이루어지는 거지?

생각 정리

STEP1 Thread란?

Process 내에서 실행의 흐름이 되는 단위이다.

실행의 흐름이 되기 위해서는 Program Counter 및 다른 Register Value 그리고 Stack이 필요하다.

STEP2 Thread의 생성이란?

PC, Register Value를 보관할 장소 그리고 Stack을 생성하는 것을 의미한다.

"PC, Register Value를 보관할 장소"가 TCB가 된다.

STEP3 User-level Thread vs Kernel-level Thread

User-level Thread :: Application이 Thread를 관리 하므로, TCB 또한 User Space에 존재하게 된다. 만약 Kernel Space에 존재하면 Application이 직접적으로 접근하지 못한다.

Kernel-level Thread :: TCB가 Kernel에 존재한다. 따라서 당연히 OS가 Thread 관리를 해준다. Application이 kernel-level의 Thread에 접근하기 위해서는 System Call을 통해 접근 해야 한다.

STEP4 왜 Mapping되어야 하지?

결국 OS를 통해 CPU를 할당 받는 것은 Kernel-level Thread이다. 따라서 User-level Thread라 하더라도 실행되기 위해서는 kernel-level Thread(native Thread)와 Mapping되어야 한다.

Green Thread – 궁금한 점

구현

- Application에서 구현한 Thread이다. 따라서, OS는 해당 Thread에 대해서 알지 못한다
- 다수의 User Level Threads가 하나의 Kernel-Level(Native) Thread로 Mapping

☞ 어떻게 Mapping이 이루어지는 거지?

생각 정리

STEP5

결국은 하나의 Native Thread를 사용해야 한다. 왜냐하면 결국 CPU를 차지하는 것은 Native Thread이기 때문이다.

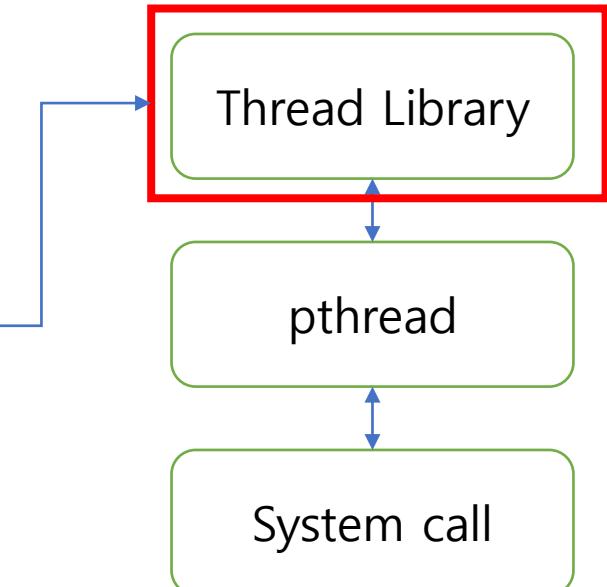
STEP6 Native Thread를 어떻게 사용하지?

결국 Linux의 경우 Linux가 제공하는 pthread를 사용해서 Native Thread를 다룰 것이다.

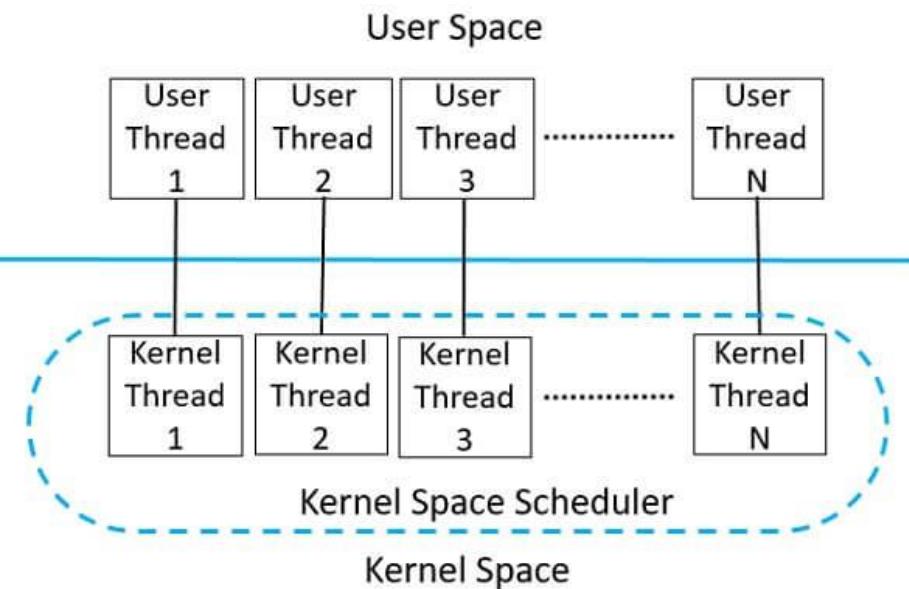
STEP7 결국 어떻게 Mapping이 되는가?

Application이 관리하는 TCB에 대한 정보를 Pthread를 통해 Mapping 시키는 것 같다.

☞ 하지만 사용하기 편리하도록 언어자원에서 Thread API를 만들어 제공할 것이다.



Java의 Thread



장점

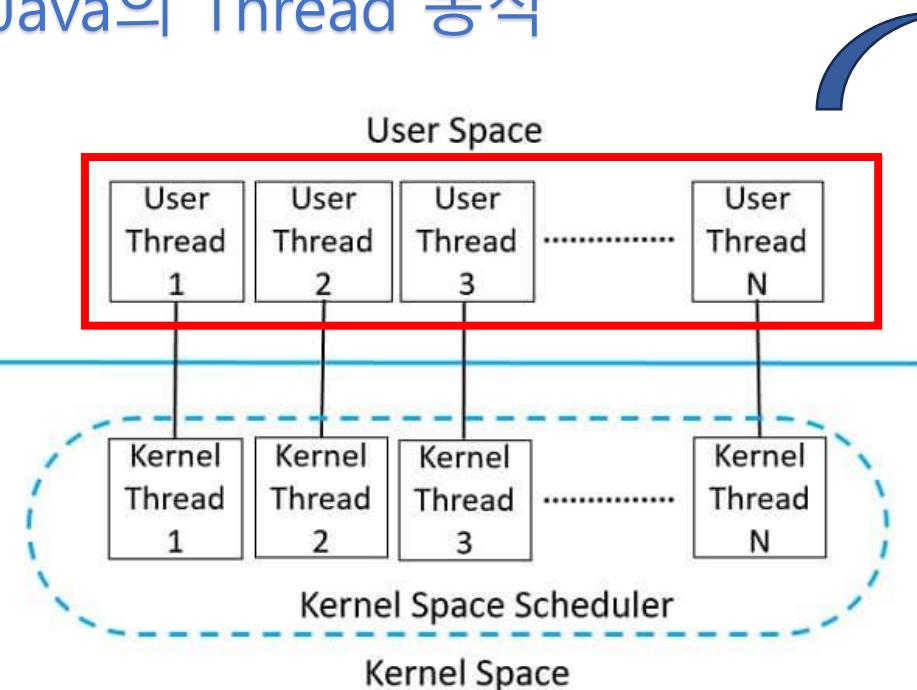
- OS에서 Thread를 관리하므로, Multi Core를 효율적으로 활용 가능
- Blocking I/O를 하더라도 해당 Thread만 Block된다.

단점

- Thread간 Context switching이 kernel에 의해 수행되므로, 상대적으로 느리다.

One to one (1:1) Multithreading Model

Java의 Thread 동작



☞ 질문:: User Thread가 뭐지?

☞ 내가 생각한 결론

- 쉽게 생각하면 Java API로 결국은 OS의 System Call을 쉽게 사용할 수 있도록 구현해 놓은 것이라고 생각해도 될 것 같다.
※ 어떻게 생각하면 printf()와 비슷한 것 같다.
- Java에서 Thread 객체를 하나 생성하면, 우리는 생성된 Thread 객체를 통해 kernel level의 Thread와 연결된다.
- 생성된 Thread 객체를 통해 Thread 관련 Operation을 편리하게 사용할 수 있다. 만약 Thread 객체가 없다면, 우리는 Pthread 연산을 직접 사용해야 할 것이다.

One to one (1:1) Multithreading Model

Java API → JNI (Native Code) → System Call