프로세스(process)란?

: 운영체제에 실행 중인 하나의 애플리케이션.

멀티태스킹

: 두가지 이상의 작업을 동시에 처리하는 것을 말한다.

자바의 프로세싱 순서

: main() 메소드의 첫 코드부터 아래로 순차적으로 실행하여 return 문을 만나면 종료한다.

(싱글 스레드 일 경우, 메인 스레드가 종료하면 프로세스도 종료된다.

But, 멀티 스레드일 경우, 실행 중인 스레드가 하나라도 존재할 경우 프로세스는 종료되지 않는다.

작업 스레드 생성방법

1. Thread 클래스로부터 직접 생성

Thread thread = new Thread(Runnable target);

Class Task implements Runnable{

Public void run(){

}

}

(람다식 사용)

Thread thread = new Thread( () -> {

});

thread.start();

1. Thread 하위 클래스로부터 생성(Runnable 생성X, Thread클래스의 run()을 오버라이딩)

Public class WorkerThread extends Thread{

@Override

Public void run(){

}

}

(람다식 사용가능)

스레드 명명(이름 set, get)

Thread.setName(“스레드 이름”);

Thread.getName();

\*\* setName과 getName은 현재 스레드를 선택하여 얻을수록 목적에 부합한 기능을 수행

현재 스레드 참조 메서드

Thread thread = Thread.getcurrentThread();

스레드 우선순위

: 스레드의 개수가 코어의 수보다 많을 경우, 스레드를 어떤 순서에 의해 실행할 것 인가를 결정해야 하는데, 이것을 스레드 스케줄링이라고 한다.

\*\* 자바의 스케줄링은 우선순위(Priority) 방식과 순환할당(Round-Robin) 방식을 사용한다.

\*\* 우선순위 방식은 개발자가 코드로 제어할 수 있지만, 순환할당 방식은 JVM이 도맡아 한다.

우선순위 설정

Thread.setPriority(우선순위);

우선순위의 종류

1. Thread.MAX\_PRIORITY : 10
2. Thread.NORM\_PRIORITY : 5
3. Thread.MIN\_PRIORITY : 1

동기화 메소드와 동기화 블록

: 멀티 스레드 프로그램에서 여러 스레드들이 객체를 공유해서 작업해야 하는 경우, 사용하는 일종의 잠금 방식

\*\* 임계 영역이란?(critical section)

: 멀티 스레드 프로그램에서 단 하나의 스레드만 실행할 수 있는 코드 영역

동기화메소드

: 메소드 전체 내용이 임계 영역이므로 스레드가 동기화 메소드를 실행하는 즉시, 객체

에는 잠금이 일어나고, 스레드가 동기화 메소드를 종료하면 잠금이 풀린다.

Public synchronized void method(){}

동기화 블록

Public void method(){

Synchronized(공유객체){

}

}

스레드의 대기상태

1. 객체 생성 : NEW(아직 start() 메소드가 호출되지 않은 상태)
2. 실행 대기 : RUNNABLE(실행 상태로 언제든지 갈 수 있는 상태)
3. 일시 정지 : WATING(다른 스레드가 통지할 때까지 기다리는 상태)

TIMED\_WAITING(주어진 시간 동안 기다리는 상태)

BLOCKED(사용하고자 하는 객체의 락이 풀릴 때까지 기다리는 상태)

1. 종료 : TERMINATED

스레드 상태 제어

Ex) 미디어 플레이어 스레드의 일시정지, 재생 기능

Interrupt() : 일시 정지 상태의 스레드에서 InterruptedException 예외를 발생시켜, 예외 처리 코드에서 실행 대기 상태로 가거나 종료 상태로 갈 수 있도록 한다.

Notify(), notifyAll() : 동기화 블록 내에서 wait() 메소드에 의해 일시 정지 상태에 있는 스레드를 실행 대기 상태로 만든다.

Sleep(long millis), Sleep(long millis, int nanos) : 주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.

Join(), join(long millis), join(long millis, int nanos)

: join() 메소드를 호출한 스레드는 일시 정지 상태가 된다. 실행 대기 상태로 가려면 join()메소드를 멤버로 가지는 스레드가 종료되거나, 매개값으로 주어진 시간이 지나야 한다.

\*\*\* Main 스레드에서 특정 스레드의 작업에 join을 사용한다면, 마찬가지로 Main스레드가 해당 스레드의 작업이 완료될 때까지, 대기상태에 들어가게 된다.

Wait(), wait(long millis), wait(long millis, int nanos)

: 동기화 블록 내에서 스레드를 일시 정지 상태로 만든다 매개값으로 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다. 시간이 주어지지 않으면 notify(), notifyAll()메소드에 의해 실행 대기 상태로 갈 수 있다.

Yield() : 실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

(while문을 활용한 부울린 값의 체킹 스레드의 경우, 같은 우선순위의 다른 스레드보다 평균 작업이 무의미한 경우가 많기에, yield()를 사용하여 다른 스레드에게 순서를 양보한다.)

스레드의 안전한 종료(stop()플래그, interrupt())

: Stop() 메소드는 스레드의 갑작스런 종료로 인한 자원들의 불안전한 제거로 인해 deprecated 되었다. 그렇지만 코드 상 종료는 필요한 기능이다. 이를 위해 stop()플래그를 이용하여 정상적으로 종료하게끔 하였다.

* stop() 플래그

public class 클래스명 extends Thread{

public void run(){

while(!stop){

}}}

* interrupt() 메소드

public class 클래스명 extends Thread{

try{

}catch(InterruptedException e){

}

Thread.interrupt();}}

데몬 스레드

: 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드

, 주 스레드가 종료되면 데몬 스레드가 자동 종료된다.

Ex) 워드프로세서의 자동 저장, 미디어 플레이어의 동영상 및 음악재생, 가비지 컬렉터 등

Public static void main(String[] args){

AutoSaveThread thread = new AutoSaveThread();

Thread.setDaemon(true);

Thread.start(); // start() 메소드 호출 후에 setDaemon() 호출시,

IllegalThreadStateException이 발생한다.

스레드 그룹

: 관련된 스레드를 묶어서 관리할 목적으로 이용한다.

JVM이 실행되면 default system 스레드 그룹이 생성된다.

명시적으로 스레드 그룹에 포함시키지 않으면 기본적으로 자신을 생성한 스레드와 같은 스레드 그룹에 속하게 된다.

스레드 그룹 이름 얻기

ThreadGroup group = Thread.currentThread().getThreadGroup();

String groupName = group.getName();

모든 스레드 그룹 정보 얻기

Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();

Set<Thread> threads = map.keySet(); // keyset()메서드 : 스레드 객체들을 참조

// Set이란 집합을 의미하는 영어이다.

스레드 그룹 생성

ThreadGroup tg = new ThreadGroup(String name);

ThreadGroup tg = new ThreadGroup(ThreadGroup parent, String name);

Thread t = new Thread(ThreadGroup group, Runnable target, String name, long stackSize);

ㄴ> 이와 같이 부모 그룹 및 스레드 이름, 스택 사이즈를 동시에 입력하면서 생성할 수 있다.

스레드 그룹의 일괄 interrupt()

: 그룹명.interrupt()

스레드풀

: 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고, 작업 큐에 들어오는 스레드풀은 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐에 들어오는 작업들을 하나씩 스레드가 맡아 처리한다.

(자바에서의 스레드풀 생성은 java.util.concurrent 패키지의 ExecutoService 인터페이스와 Executors 클래스를 이용하여 처리한다.)

스레드풀 생성 및 종료 초기 스레드 수 코어 스레드 수 최대 스레드 수

newChachedThreadPool() 0 0 Integer.MAX\_VALUE

newFixedThreadPool((int nThreads) 0 nThread nThread

ExecutorService executorService = Executors.newChachedThreadPool();

ExecutorService executorService = Executors.newFixedThreadPool(

Runtime.getRuntime().availableProcessors());

// cpu 코어 개수를 런타임 변수로 불러와 고정 스레드 수로 생성하는 FixedThreadPool

스레드 풀 종료

Shutdown() : 현재 처리 중인 작업뿐만 아니라 작업 큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드풀을 종료시킨다.

shutdownNow() : 현재 작업 처리 중인 스레드를 interrupt해서 작업 중지를 시도하고 스레드풀을 종료시킨다. 리턴값은 작업 큐에 있는 미처리된 작업 (Runnable)의 목록이다

awaitTermination(long timeout, TimeUnit unit)

: shutdown() 메소드 호출 이후, 모든 작업 처리를 timeout 시간 내에 완료하면 true를 리턴하고, 완료하지 못하면 작업 처리 중인 스레드를 interrupt하고 false를 리턴한다.

스레드 풀의 작업 생성

1. Runnable task = new Runnable(){ // Runnable 생성

@Override

Public void run(){

}}

1. Callable<T> task = new Callable<T>(){ // Callable 생성

@Override

Public T call() throws Exception{

Return T;

}}

스레드 풀에 작업 처리 요청

: ExecutorService의 작업 큐에 Runnable 또는 Callable 객체를 넣는 행위

1. Execute(Runnable command) : Runnable 용, 작업 처리 결과를 받지 못함 = 콜백 X
2. Submit(Runnable task), submit(Runnable task, V result), submit(Callable<V> task)

: Runnable 또는 Callable을 작업 큐에 저장, 리턴된 Future를 통해 작업 처리 결과를 얻을 수 있다.(리턴 타입들은 Future이다.)

( execute는 작업 처리 도중 예외가 발생하면 스레드가 종료되고 해당 스레드는 스레드풀에서 제거된다. But, submit은 예외가 발생하면, 다음 작업을 위해 재사용된다.)

* 오버헤드를 줄이기 위해서 submit()을 사용하는 것이 좋다.

\*\* Future는 비동기적인 연산의 결과를 표현하는 클래스입니다. Future를 이용하면 멀티쓰레드 환경에서 처리된 어떤 데이터를 다른 쓰레드에 전달할 수 있습니다.

Future 내부적으로 Thread-Safe하도록 구현되었기 때문에 synchronized block을 사용하지 않아도 됩니다. \*\*

블로킹 방식의 작업 완료 통보

Future 객체는 작업 결과가 아니라 작업이 완료될 때까지 기다렸다가 최종 결과를 얻는데 사용된다.( 그래서 지연 완료 객체라고 한다.)

1. Get() : 작업이 완료될 때까지 블로킹되었다가 처리 결과 V를 리턴
2. Get(long timeout, TimeUnit unit)

: timeout 시간 전에 작업이 완료되면 결과 V를 리턴하지만, 작업이 완료되지 않으면 TimeoutException을 발생시킴

\*\* UI처리 스레드가 작업 완료 전 get()메소드를 호출할 경우, 예외처리로 인해, 이후의 스레드를 처리할 수 없게 된다. 그러므로 get()메소드를 호출하는 스레드는 새로운 스레드이거나 스레드풀의 또다른 스레드가 되어야 한다. \*\*

Future 객체의 메소드 종류

* Boolean cancel(Boolean mayInterruptIfRunning) : 작업 처리가 진행 중일 경우 취소시킴
* Boolean isCancelled() : 작업 취소 여부
* Boolean isDone() : 작업 완료 여부

작업 처리 결과를 외부 객체에 저장

: Future<Result> future = executorService.submit(task, result);

Result result = future.get();

( 스레드에서 결과를 저장하기 위해 외부 Result 객체를 사용해야 하므로 생성자를 통해 Result 객체를 주입받도록 해야 한다.)

Ex) 생성자(Result result){

This.result = result;

작업 완료 순으로 통보(CompletionService)

: 처리 결과도 순차적으로 이용할 필요가 없다면 작업 처리가 완료된 것부터 결과를 얻어 이용하면 된다.

* CompletionService의 메소드

1. poll() : 완료된 작업의 Future를 가져옴(완료된 작업이 없다면 즉시 null을 리턴함
2. poll(long timeout, TimeUnit unit)

: 완료된 작업의 Future를 가져옴, 만약 없다면 timeout까지 블로킹됨

1. take() : 완료된 작업의 Future를 가져옴, 없다면 있을때까지 블로킹됨
2. submit(Callable<V> task) : 스레드풀에 Callable 작업 처리 요청
3. submit(Runnable task, V result) : 스레드풀에 Runnable 작업 처리 요청