

IV. 스레드 (Thread)

Topics

1. 개요
2. 다중 스레드 모델
3. 스레드 라이브러리
4. 관련 이슈들
5. 운영체제 사례

1. Introduction

❑ Most software applications that run on modern computers are **multi-threaded**.

- A thread) 하나의 실행 경로
- word processor, web browser, web server 등등

Word processor 그래픽 디스플레이, 키보드 입력, 철자/문법 검사 등등.

Web browser network으로부터 정보 수취, 정보 디스플레이 등등.

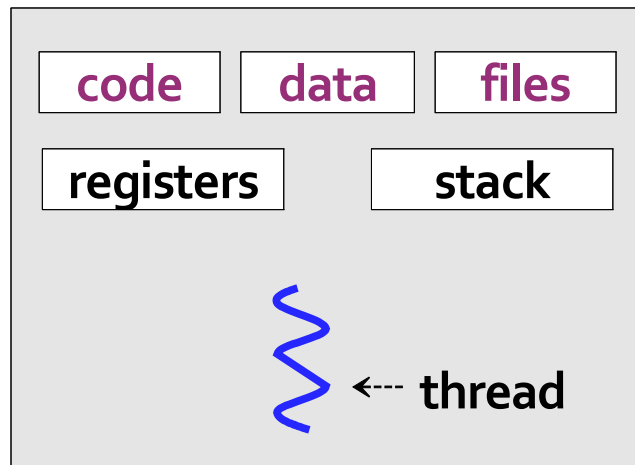
Web server 여러 클라이언트 요청 동시 처리.

❑ Mostly multicore or multiprocessor systems.

❑ Single vs. Multi-threaded Process

Single-threaded Process

하나의 실행경로를 가지는 프로세스

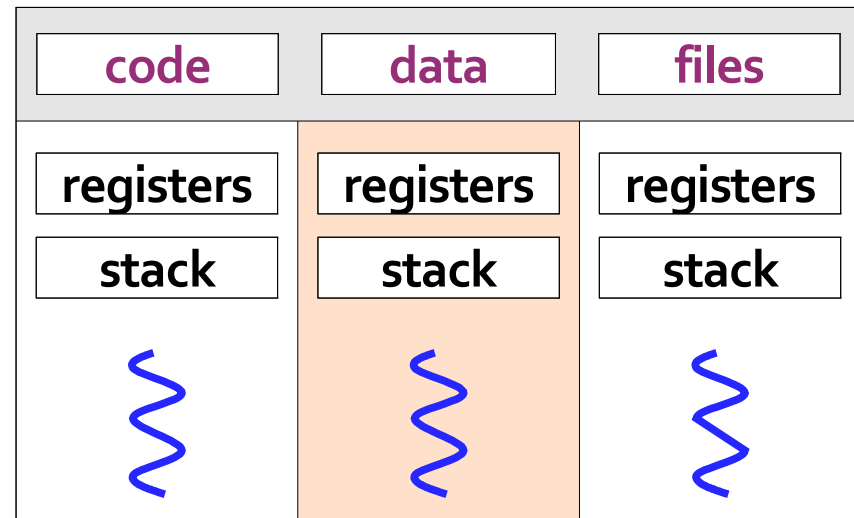


Heavy-weight process.

하나의 process 하나의 작업.

Multi-threaded Process

여러 개의 실행경로를 가지는 프로세스

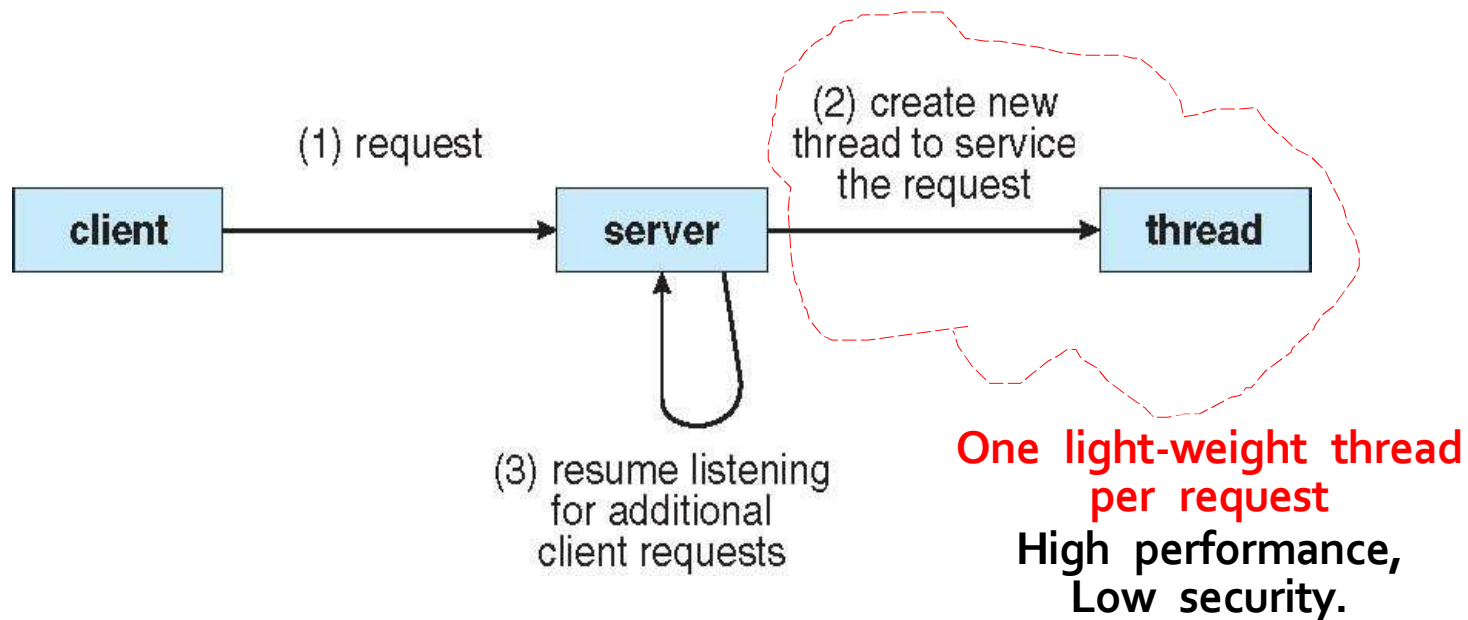


Light-weight thread. ※ 자원 공유

하나의 process 여러 개의 작업.

[교재 p190 참고] Responsiveness, Resource sharing, Economy, Scalability

- Multi-threaded server architecture



(cf) Single-threaded process로 구현하는 경우

```
do forever {  
    get a request;  
    if( pid=fork() == 0 ) { // one heavy-weight process per request  
        exec("요청처리프로그램");  
    }  
}
```

2. Multi-threading Models

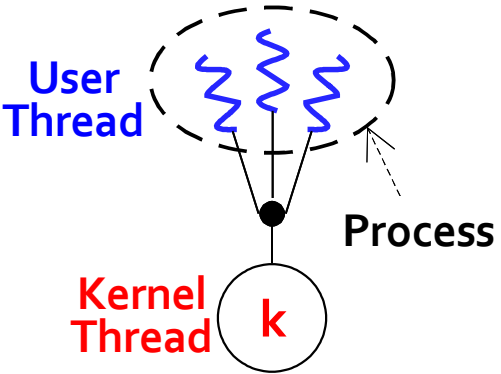
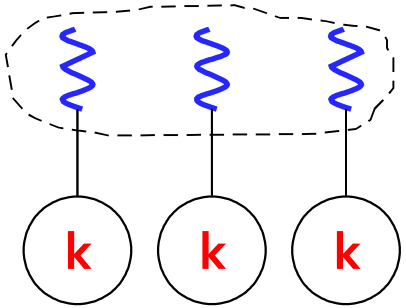
□ 두 가지 유형의 스레드

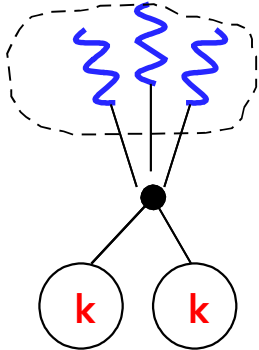
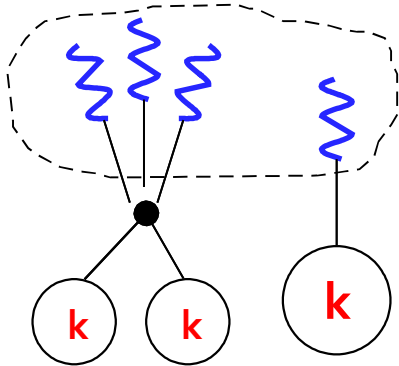
- **Kernel thread** 운영체제가 직접 관리하는 스레드
(also called *Kernel-supported thread* or *Lightweight process*)
거의 모든 OS가 지원함: Windows, Linux, Mac OS, Solaris.
- **User thread** 사용자가 생성•관리하는 스레드 (using Thread Library)
* Thread library: POSIX, Win32, Java thread API

□ Mapping user threads **to** kernel threads

- 운영체제는 kernel thread 만 scheduling하므로
실행되기 위해서 user thread는 **kernel thread에 mapping(사상)**되어야 함.
- 3가지 mapping model - 다대일, 일대일, 다대다 모델

□ Multithreading Model

다대일 모델	일대일 모델
 <p>여러 user thread를 하나의 kernel thread에 사상함.</p>	 <p>하나의 user thread를 하나의 kernel thread에 사상함.</p>
<ul style="list-style-type: none"> • 한 스레드 block은 process 전체 block으로 이어짐. 즉 병행성 낮음. • multi-core system에서 병렬실행 불가능함. 	<ul style="list-style-type: none"> • 다대일 보다 높은 병행성. • 병렬실행 가능함. • 커널 스레드 개수 증가 (성능 저하 야기) • 시스템 內 thread 개수 제한함.
<p>Solaris Green threads, GNU Portable Threads 지금은 거의 사용하지 않음.</p>	<p>Solaris 9 부터, Linux, Windows</p>

다대다 모델	Two-level Model
 <p>n user thread를 $m(\leq n)$ kernel thread에 사상함. (<i>multiplexing</i> 됨)</p>	 <p>다대다에 일대일 결합한 방식</p>
<p>다대일(병행실행)과 일대일(커널스레드 개수) 문제를 개선함.</p>	<p>다대다 모델의 변형</p>
<p>Solaris 9 이전, Windows NT/2000</p>	<p>Solaris 9 이전, Tru64UNIX</p>

3. Thread Library

□ Thread Library

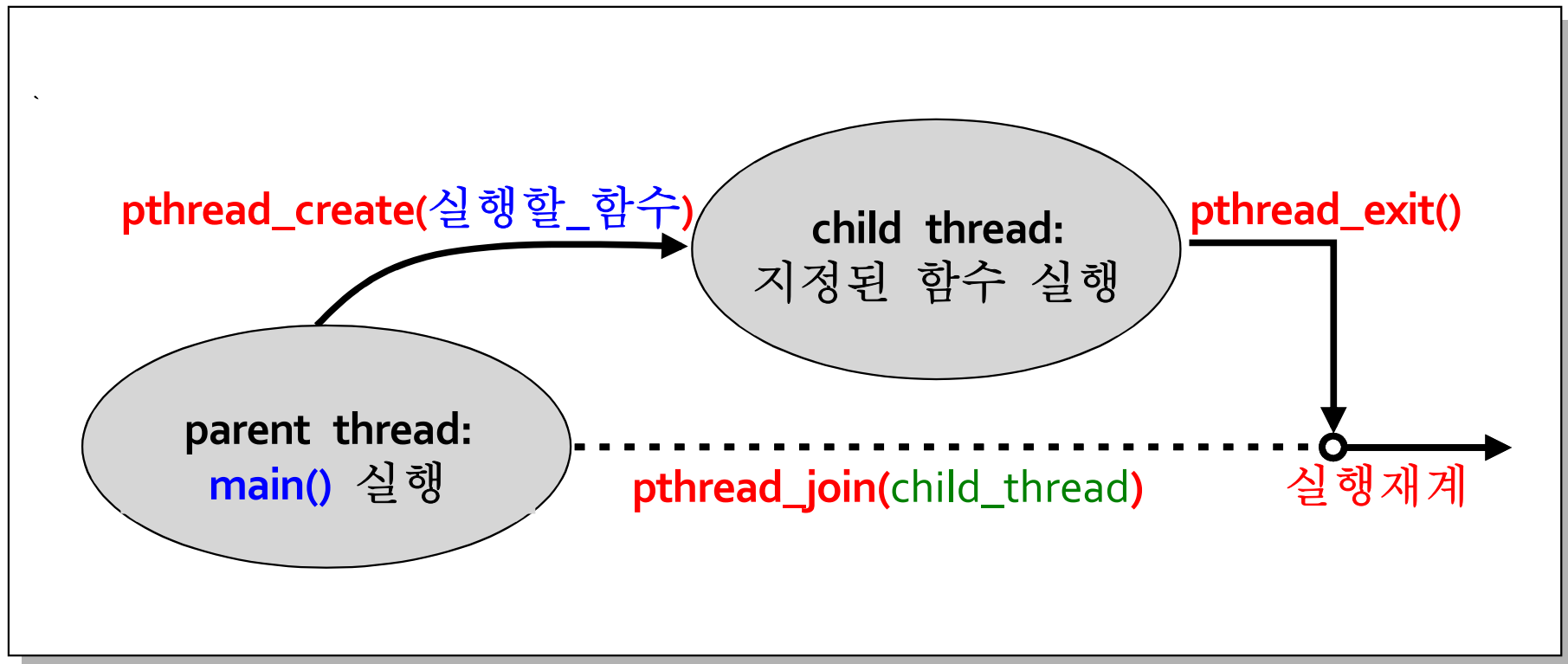
- 프로그래머에게 스레드 생성·관리 API를 제공하는 library.
 - library 구현 방법: User-level library, Kernel-level library
- 주요 library
 - **POSIX Pthreads** Thread extension of POSIX standard (note) Spec (구현 아님) user-level, kernel-level library 모두 가능.
 - **Windows** kernel-level library
 - **Java Thread API** 대부분 host의 library를 사용하여 구현됨.

□ 두 가지 threading 전략

비동기 스레딩	<ul style="list-style-type: none">- 부모는 <u>자식 생성 후 즉시</u> 실행 재개함.- 모든 스레드는 독립적으로 실행되며 (데이터 공유 거의 없음), 부모는 자식의 종료를 알 필요 없음.- multithreading server에서 사용되는 전략.
동기 스레딩	<ul style="list-style-type: none">- 부모는 <u>모든 자식 종료 후</u> 실행을 재개함. (<i>fork-join</i> 전략) (자식 스레드들은 서로 독립적으로 실행됨.)- 보통 데이터 공유 많음.

□ POSIX Pthreads

- POSIX¹⁾ 스레드 표준 API
- Solaris, Linux, Mac OS, Tru64 UNIX 등이 구현함



1) POSIX - Portable Operating System Interface

A family of standards specified by the IEEE for maintaining compatibility between UNIX-like OSs.

(예) $\sum_{i=1}^{upper} i$ 를 계산하는 thread 작성 프로그래밍 - (1)

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

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

```
int sum; // 전역변수는 공유됨
```

```
// child thread가 실행할 코드
```

```
void *runner(void *param) {
```

```
    int i, upper;
```

```
    upper = atoi(param);
```

```
    sum = 0;
```

```
    for (int i=1; i<=upper; i++) {
```

```
        sum += i;
```

```
    }
```

```
    pthread_exit(0); //스레드 종료
```

```
}
```

```
// main thread가 실행함.
```

```
int main(int argc, char *argv) {
```

```
    pthread_t tid;
```

```
    pthread_attr_t attr;
```

```
    pthread_attr_init(&attr); // (디폴트) 스레드 속성 획득
```

```
// child thread 생성 및 실행
```

```
pthread_create(&tid, &attr, runner, argv[1]);
```

```
// child thread 종료 대기 (join-fork 전략)
```

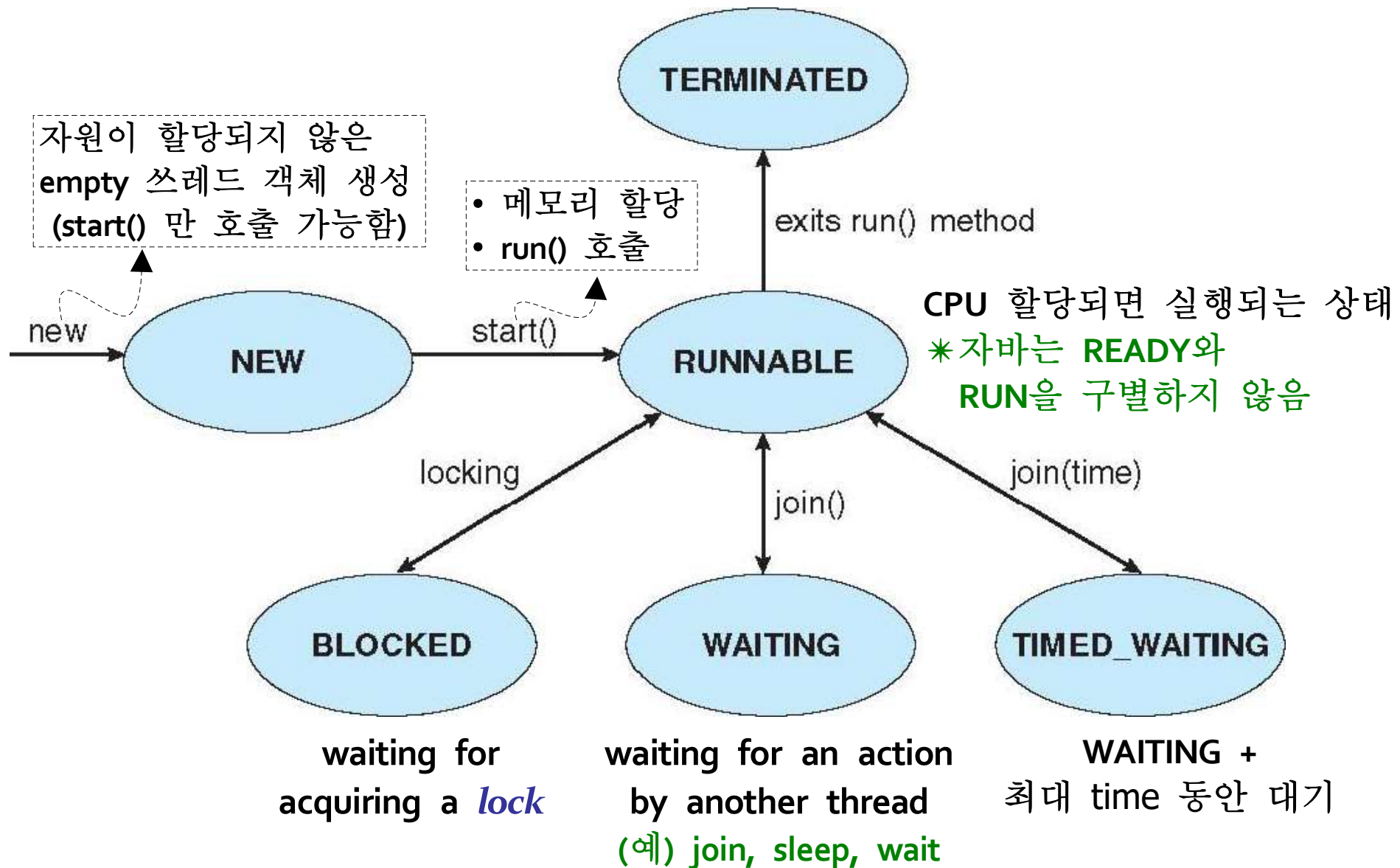
```
pthread_join(tid, NULL);
```

```
printf("총 합: %d\n", sum);
```

```
}
```

□ Java Thread

• 스레드 생애 (lifecycle)



(예) $\sum_{i=1}^{upper} i$ 를 계산하는 thread 작성 프로그래밍 - (2)

```
/*
thread 間 공유객체:
Java는 전역변수 없음.
공유데이터(객체)는
parameter를 통해 주고
받아야 함.
*/

class Sum {
    private int sum;

    public int get() {
        return sum;
    }
    public void set(int s) {
        sum = s;
    }
}
```

```
// 총합을 구하는 thread
class Summation implements Runnable {
    private Sum sum; // shared object
    private int upper;

    // 공유객체를 파라미터로 전달 받음.
    public Summation(int upper, Sum sum) {
        this.upper = upper;
        this.sum = sum;
    }

    // run(): thread's main()
    public void run() {
        int s = 0;
        for (int i=0; i<=upper; i++)
            s += i;
        sum.set(s); // 공유객체에 결과 write.
    }
}
```

```

// 실행: java Driver upperValue
public class Driver {

    // main thread가 실행함
    public static void main(String[] args) {

        Sum sumObject = new Sum(); // 스레드 간 공유 객체

        int upper = Integer.parseInt(args[0]);

        // child thread (Summation 스레드 객체) 생성.
        // 이때 child thread와 공유할 객체를 넘겨줌.
        Thread child = new Thread(new Summation(upper, sumObject));

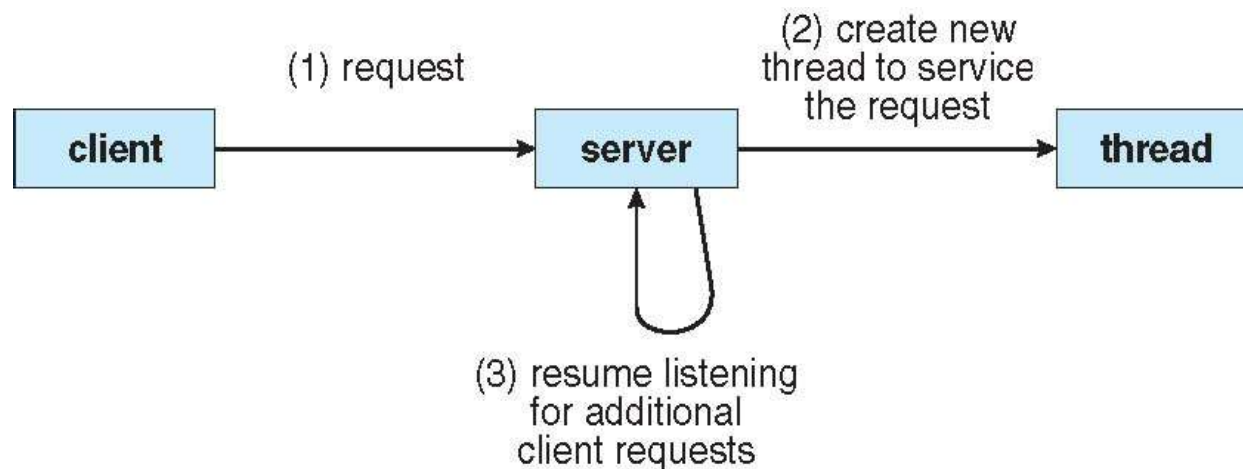
        child.start(); // child thread 실행시킴.

        try {
            child.join(); // child thread 종료를 기다림
            System.out.println("총 합: " + sumObject.getSum());
        } catch (InterruptedException e) {
        }
    }
}

```

4. Implicit Threading

- 수백-수천 개의 thread를 가지는 multi-threaded application 등장
필요할 때마다 thread 생성하면
 - application의 성능 저하 (많은 스레드 생성 비용)
 - system resource 고갈 염려 (unstable)



- **Implicit threading**

thread의 생성과 관리를 application에서 분리하여
compiler와 thread library가 수행하게 함:

Thread pool, OpenMP, Grand Central Dispatch, Thread Building Block,
etc.

❑ Thread Pool

- process 시작 時 적정 수의 thread를 미리 생성하고 (thread pool), 요청時 스레드 할당, 작업완료 時 회수함.
- application performance ↑, system stability ↑
- thread pool의 크기 설정 factor
 - CPU 개수, 메모리 용량, 동시 요청 client 최대 수
 - pool 사용 패턴을 분석하여 동적으로 조정

❑ Java Thread Pool

Thread pool

<<interface>> **Executor**

void **execute**(Runnable command) * No return value from the thread



<<interface>> **ExecutorService**

void **shutdown**() // pool 관리

<V> Future<V> **submit**(Callable<V> task) * Value-returning task

Submits a value-returning task(Callable<V>) for execution
and returns a Future representing the *pending* results of the asynchronous task.

Thread pool factory class

Executors

public static **ExecutorService** newSingleThreadExecutor() // size of one

public static **ExecutorService** newFixedThreadPool(int n) // size of n

public static **ExecutorService** newCachedThreadPool() // unbounded pool


```
import java.util.concurrent.*; 프로그래밍 - (3)
```

```
public class ThreadPool {  
    public static void main(String[] args) {  
  
        // thread pool 생성  
        ExecutorService pool = Executors.newFixedThreadPool(10);  
  
        // pool의 thread를 이용하여 task 실행  
        for (int i=0; i<3; i++) {  
            pool.execute(new Hi(i));  
        }  
  
        pool.shutdown(); // 새로운 thread 불허;  
                          // 모든 current thread 완료 후 pool을 shutdown시킴.  
    }  
}  
  
class Hi implements Runnable {  
    int i;  
    public Hi(int i) { this.i = i; }  
    public void run() { System.out.println("hi!! I am number " + i + "."); }  
}
```

```

interface ArchiveSearcher { String search(String target); }

class App { 프로그래밍 - (4)
    ExecutorService pool = ...
    ArchiveSearcher searcher = ...

    void showSearch(final String target) throws InterruptedException {
        // Submits a value-returning task for execution
        Future<String> future = pool.submit(
            new Callable<String>() {
                public String call() { return searcher.search(target); }
            }
        );
        displayOtherThings();           // do other things while searching
        try {
            displayText(future.get();    // get result of submitted asyn task
        } catch (ExecutionException ex) {
            cleanup();
            return;
        }
    }
}

```

스케줄링 기능을 가지는 Thread pool

<<interface>> ExecutorService



<<interface>> **ScheduledExecutorService**

스레드 스케줄링 기능:

- fixed delay
- fixed period
- periodically with an initial delay

Thread pool factory class

Executors

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

(예) 프로그래밍 - (5)

// scheduled thread pool 생성

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(10);
```

// scheduled execution of task - 3초 후 실행

```
scheduler.schedule(new Hi(), 3, TimeUnit.SECONDS);
```

// scheduled execution of task - 5초 후 부터 3초 주기로 실행

```
scheduler.scheduleAtFixedRate(new Hello(), 5, 3, TimeUnit.SECONDS);
```

```
class Hi implements Runnable {  
    public void run() {System.out.println("hi!!"); }  
}  
  
class Hello implements Runnable {  
    public void run() {System.out.println("hello!!"); }  
}
```

4. 스레드 관련 이슈

□ fork()와 exec() 시스템호출의 의미

- **exec(program)** 프로세스 전체가 새로운 프로그램으로 대체됨.
- **fork()**의 두 가지 의미
 - ① **process** 내의 모든 thread 복제
 - ② **fork()**를 호출한 thread 만 복제 (note) **fork(); exec();** 경우에 유용.
- ※ 일부 UNIX 시스템은 두 가지 버전의 **fork()**를 제공함.

□ 스레드 실행 취소 (Thread Cancellation)

- 한 스레드가 다른 스레드를 강제 종료시키는 것 (예) 병렬 검색 스레드들
- **Cancellation** 時 발생하는 문제: 자원 회수, 공유자원 갱신
- **비동기 취소** **target thread**를 즉시 종료시킴.
- **지연 취소** 스레드는 **target thread**의 취소 예정 flag를 set.
target thread는 주기적으로 **flag**를 검사함.
※ 안전한 시점(*cancellation point*) 자신을 종료시킬 수 있음.

(Java) Deferred Cancellation 프로그래밍 - (6)

A thread interrupts a target thread.	Target thread cancel itself at cancellation point.
<pre>Thread target; target = new InterruptibleThread(new Thread()); // target thread를 실행시킴. target.start(); ... // interrupts the target thread target.interrupt();</pre>	<pre>class InterruptibleThread implements Runnable { public void run() { while() { do some work // 주기적 interruption status 검사 if (Thread.currentThread().isInterrupted()) { System.out.println("I'm interrupted."); break; // while block 탈출 } clean up and terminate } } }</pre>

interrupt() sets the interruption status of the target thread.

isInterrupted() returns true if interruption status is set (interruption status를 그대로 둠)

interrupted() returns true if interruption status is set (interruption status를 clear 함)

□ Thread-Local Storage

- 동일 프로세스 内の 스레드들은 프로세스의 **data**를 공유함.
필요 時 스레드는 자신의 local copy를 가질 수 있어야 함.
이를 *thread-local storage(TLS)*라 함.
- 대부분 스레드 라이브러리는 이를 지원함
(예) Win32, Pthreads, Java
- (Java)
각 thread(object)는 자신의 data(instance fields)를 가지므로 불필요.
thread pool과 같이 스레드 생성 과정을 제어할 수 없는 경우 필요함.

```
public class java.lang.ThreadLocal<T> extends Object
```

- thread가 thread local variable을 접근할 경우, 자신의 copy를 가지게 됨.
- typically **private static**

```
public class Service {  
    private static volatile int x = 10;
```

```
    private static final ThreadLocal<Integer> uniqueId; // thread-local variable  
    private static final AtomicInteger nextId;  
    static {  
        nextId = new AtomicInteger(0);  
        uniqueId = new ThreadLocal<Integer>() {  
            protected Integer initialValue() {  
                return nextId.getAndIncrement();  
            }  
        };  
    }  
    public static int getUniqueId() { return uniqueId.get(); }
```

```
    public static void transaction() {  
        System.out.println(uniqueId.get() + ": x=" + x++);  
    }  
  
    public static void main(String[] args) {  
        Service svc = new Service();  
        Thread w0 = new Thread(new Worker(svc)); w0.start();  
        Thread w1 = new Thread(new Worker(svc)); w1.start();  
    }  
}
```


Thread:

```
class Worker implements Runnable {  
    private static Service svc;  
  
    public Worker(Service svc) {  
        this.svc = svc;  
    }  
  
    public void run() {  
        System.out.println(svc.UniqueID.getUniqueId() + " started...");  
        svc.transaction();  
        System.out.println(svc.UniqueID.getUniqueId() + " terminated...");  
    }  
}
```

```
public class java.lang.ThreadLocal<T> extends Object
```

- thread가 thread local variable을 접근할 경우, 자신의 copy를 가지게 됨.
- typically **private static**

(ex) thread-specific unique id를 제공하는 클래스 프로그래밍 - (7)

```
import java.util.concurrent.atomic.AtomicInteger;

public class UniqueId {
    private static final AtomicInteger nextId = new AtomicInteger(0);
    private static final ThreadLocal<Integer> uniqueId = // thread-local variable
        new ThreadLocal<Integer>() {
            protected Integer initialValue() {
                return nextId.getAndIncrement();
            }
        };

    public static int getUniqueId() {
        return uniqueId.get(); // returns thread-specific unique id.
    }
}
```

```

public class Service2 {
    private static volatile int x = 10;

    public static class UniqueID {
        private static final AtomicInteger nextId = new AtomicInteger(0);
        private static ThreadLocal<Integer> uniqueId =
            new ThreadLocal<Integer>() {
                protected Integer initialValue() { return nextId.getAndIncrement(); }
            };
        public static int getUniqueId() {
            return uniqueId.get();
        }
    }

    public static void transaction() {
        System.out.println(UniqueID.getUniqueId() + ": x=" + x++);
    }

    public static void main(String[] args) {
        Service2 svc = new Service2();
        Thread wo = new Thread(new Worker(svc)); wo.start();
        Thread w1 = new Thread(new Worker(svc)); w1.start();
    }
}

```

```
class Worker implements Runnable {  
    private static Service2 svc;  
  
    public Worker(Service2 svc) {  
        this.svc = svc;  
    }  
  
    public void run() {  
        System.out.println(svc.UniqueID.getUniqueId() + " started...");  
        svc.transaction();  
        System.out.println(svc.UniqueID.getUniqueId() + " terminated...");  
    }  
}
```

5. 운영 체제 사례

□ Windows XP threads

- An application ↔ A multi-threaded process
- 기본적으로 일대일 모델, 또한 "fiber" library(다대다 지원)도 제공함
- Thread = 식별자 + Thread context
Thread context = register set + user stack + kernel stack
+ thread-specific data

□ Linux threads

- fork() - UNIX fork()와 동일, duplicate task를 생성함.
- clone() - child task를 생성함; parent task와의 자원공유는 *flag*에 의해 결정됨
 - 부모 자원의 공유는 부모 자원에 대한 *pointer*로 구현됨

플래그	의미
CLONE_FS	파일시스템 정보 공유
CLONE_VM	메모리 공간 공유
CLONE_SIGHAND	신호처리기(signal handler) 공유
CLONE_FILES	open file 집합 공유