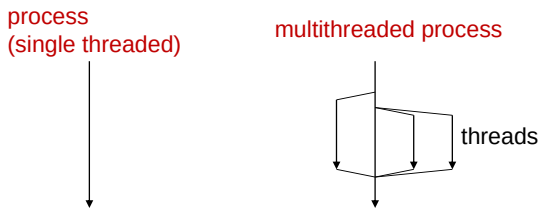
4장. 쓰레드

목표

- **쓰레드(thread)** 개념 소개
- Thread API
- Multithreaded 프로그래밍 관련 이슈

4.1 개요

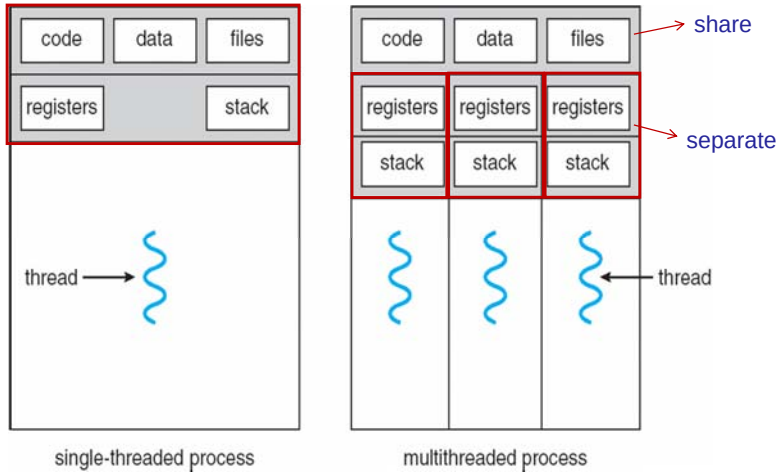
- 쓰레드(Thread)
  - CPU 이용의 기본 실행 단위
- 단일 쓰레드(Single threaded) Processes
  - 전통적인 프로세스 – 한 개의 실행 단위로 구성
- 다중 쓰레드(Multithreaded) Process
  - 여러 개의 실행 쓰레드를 갖는 프로세스
    → 한 프로세스가 동시에 하나 이상의 작업 수행 가능
  - 전통적인 프로세스의 확장



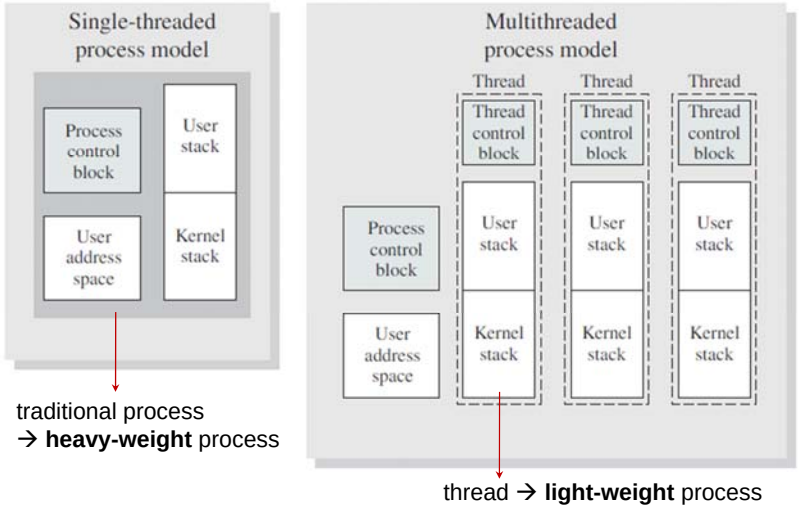
쓰레드

- Thread 사용 자원
  - 같은 프로세서에 속한 다른 thread들과
    code, data, OS 자원(예: open file, signal) 들을 공유함
  - stack, CPU register 저장공간은 thread 전용공간 사용
- Thread Control Block (TCB)
  - 쓰레드에 대한 정보를 보관 (프로세스의 PCB와 유사)
    - thread ID
    - thread 실행상태
    - program counter, register set → thread context
    - stack
    - thread specific 메모리 공간 (static 메모리)

단일 및 다중 쓰레드 프로세스



중량 프로세스와 경량 프로세스



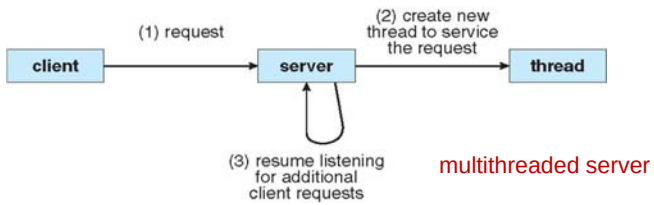
traditional process
→ **heavy-weight** process

thread → **light-weight** process

thread 생성에 필요한 자원이 process 생성에 필요한 자원보다 작다 → 경량

동기

- 단일 응용 프로그램이 여러 개의 작업을 동시에 실행할 필요가 있음
  - word processor: 화면 출력 + 키보드 입력 + 스펠링 검사 …
  - web browser: 화면출력 + 네트워크에서 데이터 수신 …
  - web server: 여러 개(수천 개도 가능)의 client의 요청 처리
  - OS kernel: 장치 관리, 인터럽트 처리 등의 여러 작업 …
- 해결책
  - 다중 프로세스(multiple processes) → 프로세스 생성 오버헤드
  - 다중 쓰레드 프로세스 → 쓰레드 생성에 필요한 자원이 적어서, 효율적



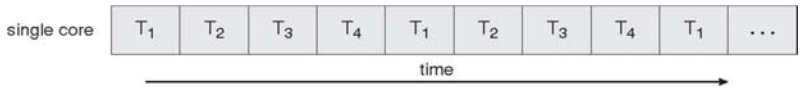
multithreaded server

장점(Benefits)

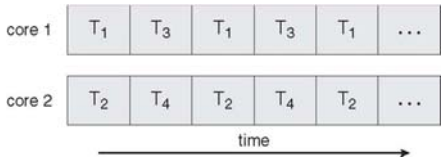
- 응답성(Responsiveness)
  - 프로그램의 일부가 block되거나, 긴 작업을 수행하여도, 대화형 (interactive) 작업의 계속 진행을 허용 → 사용자에 대한 응답성 증가
- 자원 공유(Resource Sharing)
  - 기본적으로 프로세스의 메모리와 자원을 공유함
  - 같은 주소 공간에서 여러 개의 다른 작업을 수행하는 thread 허용
    - 커널 서비스 호출없이 thread 간에 통신 가능
- 경제성(Economy)
  - thread 생성, 종료, context switch에 소요되는 오버헤드가process에 비해서 적음 – 자원, 시간 절약
  - (예) Solaris : 생성 → 30배 빠름, context switching → 5배 빠름
- 규모 확장성(Scalability), 적응성
  - 다중 프로세서(멀티코어) 구조에서 thread들을 **병렬 처리** 가능

4.2 멀티코어 프로그래밍

- 단일 코어 시스템에서의 병행(concurrent) 실행



- 멀티 코어 시스템에서의 병렬(parallel) 실행



병렬(Parallel)과 병행(Concurrency)

- 병렬 실행과 병행 실행의 구분
  - **병렬시스템** (parallel system) 은 동시에 1개 이상의 작업 수행
  - **병행시스템** (concurrent system)은 1개 이상의 task를 지원하여 모든 작업이 진행되게 함
- Amdahl's Law
  - N개의 프로세서를 사용하여 얻을 수 있는 가능한 성능 이득

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

S : 순차(비병렬) 실행 요소 비율
1-S : 병렬 실행 요소 비율
N : 프로세서 개수

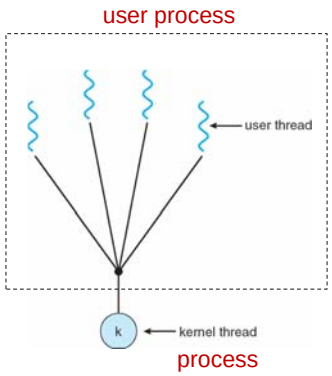
  - N → ∞이면, speedup → 1/S
    (예) S=0.25 이면 최대 4배의 성능향상이 가능
- 병렬 실행(parallelism)의 유형
  - data parallel – 데이터의 부분 집합에 대해서 동일 연산 수행
  - task parallel – 각 프로세서마다 고유 연산 수행
  - 대부분 두 병렬 실행 방법이 혼용된다.

4.3 다중 쓰레드 모델

- 사용자 쓰레드와 커널 쓰레드
  - **User threads** : 커널 지원 없이 사용자 수준 (커널 위) 에서 **thread library**에 의해서 지원됨
  - **Kernel threads** : OS 커널에서 직접 지원되고 관리됨.
- 모든 현대 운영체제는 kernel threads를 지원함
  - Windows, Solaris, Linux, Mac OS X
- 사용자 프로그램의 쓰레드와 커널 쓰레드 간에 연관 관계가 존재
  - Many-to-One 모델
  - One-to-One 모델
  - Many-to-Many모델

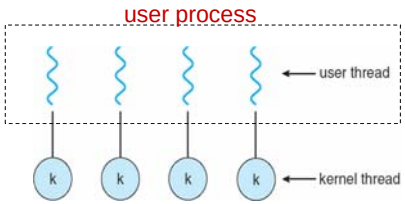
Many-to-One 모델

- 다수의 user-level thread가 한 개의 kernel thread (process)에 연관
- thread 스케줄링과 동기화가 사용자 공간의 thread library에서 수행
- 장점
  - context 스위칭과 동기화 overhead가 작아서 효율적
- 단점
  - 한 thread가 blocking system call을 호출하여 block되면 전체 process가 block됨
    ← 커널이 user-level thread의 존재를 알지 못함
  - Multiprocessor 시스템에서 thread들의 병렬 실행 불가
- 예:
  - Solaris Green Threads
  - GNU Portable Thread



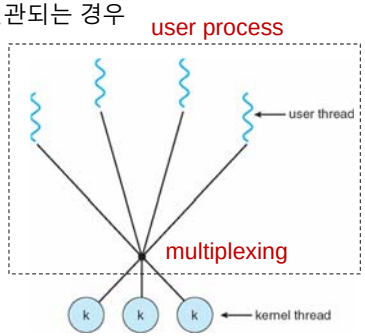
One-to-One 모델

- 각 user-level thread가 한 개의 kernel thread에 연관
- 장점 :
  - 더 많은 병행 실행, 병렬 실행(multiprocessor)
    - 한 thread가 blocking system call을 호출하여 block되면 커널은 같은 process의 다른 thread로 스케줄링
- 단점 :
  - 쓰레드 생성 및 context 스위칭 오버헤드
    - 커널 시스템 호출을 사용하여 커널 쓰레드 생성해야 함
    - 커널에서 thread context switching이 이루어짐
    - 최대 thread 수에 제한이 있음
- 예
  - Windows
  - Linux
  - Solaris 9 이후 버전

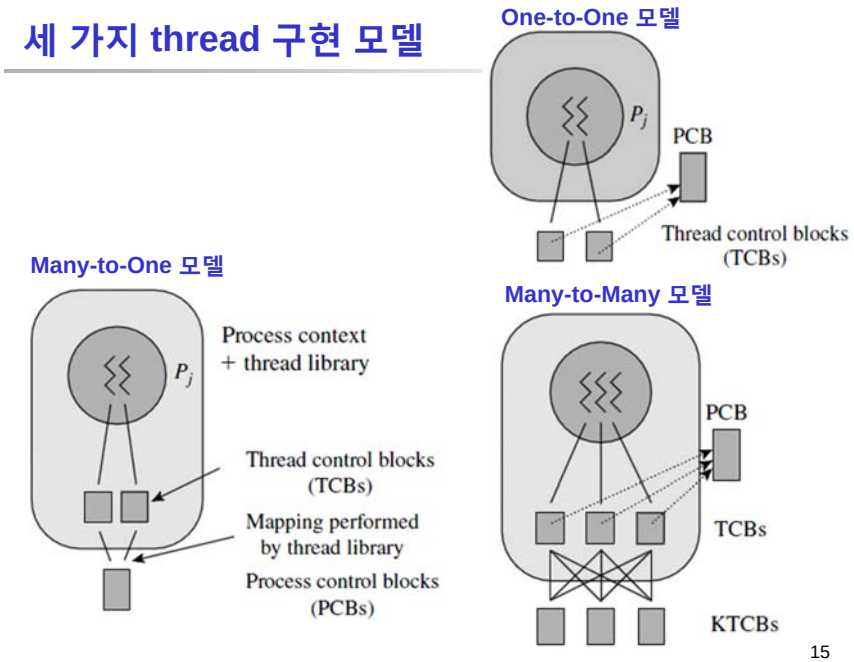


Many-to-Many 모델

- 다수의 user-level thread가 다수의 kernel thread에 연관
  - kernel thread의 수는 user-level thread의 수와 작거나 같음
  - kernel thread들을 user-level thread들이 multiplex하여 사용
  - kernel thread 개수는 응용 프로그램이나 machine에 따라서 결정됨
- 장점
  - one-to-one 모델과 같은 병행/병렬성
    - 필요한 만큼의 kernel thread와 연관되는 경우
    - blocking system call을 호출하여 block되었을 때에 다른 thread를 스케줄
  - user-level thread개수보다 적은 kernel thread를 사용할 수 있어서 kernel thread 수의 제한에 영향 없음

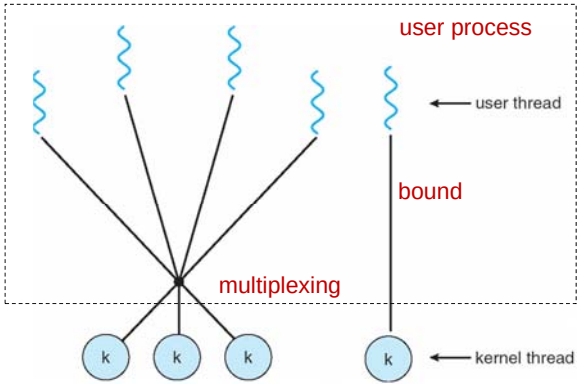


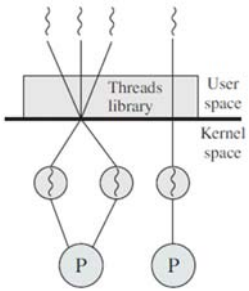
세 가지 thread 구현 모델



Two-level 모델

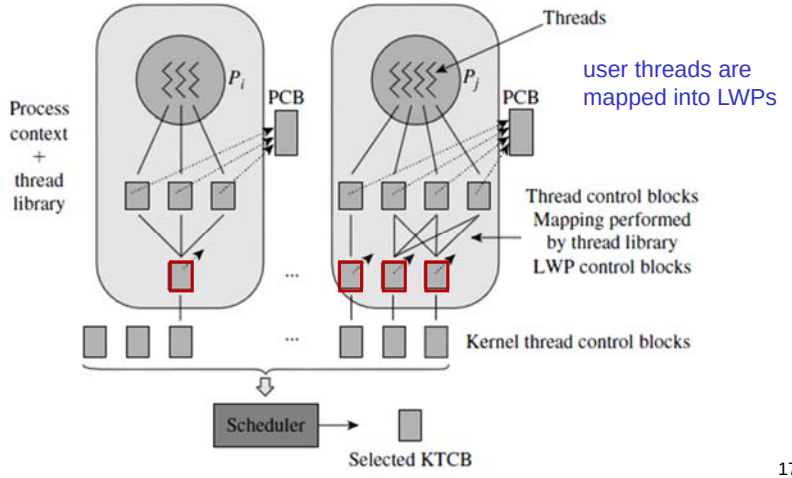
- many-to-many 모델의 변형 모델
- 일부 user-level thread에 대해서 one-to-one 연관 허용
  - 하나의 user-level thread가 하나의 kernel thread에만 연관될 수 있음
- 예: IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier





Solaris에서의 Many-to-Many 모델 (Two-level 모델)

- LWP(lightweight process)가 user thread와 kernel thread 간의 중개 역할 수행



4.4 Thread 라이브러리

- Thread 라이브러리
  - thread를 생성하고 관리하기 위한 API 제공
- thread 라이브러리 구현
  - user-level library : 완전히 user space에서 구현
    - library 함수 호출은 user space에서의 함수 호출로 이어짐
  - kernel-level library : library의 코드와 데이터가 kernel space에 존재
    - library 함수 호출은 커널에 대한 system call 호출로 이어짐
- 주로 사용되는 3가지 Thread libraries
  1. POSIX **Pthreads**: user- or kernel-level library
  2. Windows thread: kernel-level library
  3. Java Thread: Java thread API → 호스트 시스템에서 사용 가능한 thread library로 구현

Pthreads

- Thread 생성 및 동기화를 위한 POSIX 표준 API (IEEE 1003.1c)
  - thread library의 동작에 대한 명세(specification)
  - 구현방법을 명시한 것은 아님
  - UNIX 계열 OS에서 일반적으로 제공됨 (Solaris, Linux, Mac OS X)
- Pthread library 함수
  - pthread_create() - thread 생성
  - pthread_join() – thread 종료를 기다림
  - pthread_exit() – thread 종료
  - pthread_attr_init() – thread attribute를 default 값으로 초기화
  - …

Pthreads API를 사용한 Multithreaded C program

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);
   printf("sum = %d\n",sum);
}

void *runner(void *param) {
   int upper = atoi(param);
   int i;
   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;
   }
   pthread_exit(0);
}
```

Windows Threads

- CreateThread() – thread 생성
- WaitForSingleObject() – 한 thread 종료 기다림
- WaitForMultipleObjects() - 여러 thread 종료 기다림

- Example code

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

DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD *)Param;
    DWORD i;

    printf("start Summation Thread ...\n");
    for (i = 0; i <= Upper; i++)
        Sum += i;

    return 0;
}
```


Windows Threads (계속)

```
int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    // do some basic error checking
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }

    Param = atoi(argv[1]);

    if (Param < 0) {
        fprintf(stderr, "an integer >= 0 is required \n");
        return -1;
    }

    // create the thread
    ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);

    if (ThreadHandle != NULL) {
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n",Sum);
    }
}
```


Java Threads

- Java threads
  - 운영체제가 아닌 Java 언어 수준에서 지원하고 JVM이 관리함
  - main() method만으로 구성된 Java program은 JVM에서 단일 thread로 실행됨

- Thread 생성 방법
  1. Extending Thread class (파생 클래스 생성)
  2. Implementing the Runnable interface (인터페이스 구현)

```
public interface Runnable
{
        public abstract void run();
}
```


Extending the Thread Class

```
class Worker1 extends Thread
{
    public void run() {     // do not call run() directly
        System.out.println("I am a Worker Thread");
    }
}

public class First
{
    public static void main(String args[]) {
        Thread thrd = new Worker1();
        thrd.start();       // start a new thread
        System.out.println("I am the main thread");
    }
}
```

- Thread's **start()** method

Implementing the Runnable Interface

```
class  Worker2 implements Runnable
{
    public void run() {
        System.out.println("I am a Worker Thread");
    }
}

public class Second
{
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();
        System.out.println("I am the main thread");
    }
}
```

Thread thrd = new Thread(new Worker2())

```
public interface Runnable
{
    public abstract void run();
}
```


Joining Threads

```
class JoinableWorker implements Runnable
{
   public void run() {
      System.out.println("Worker working");
   }
}

public class JoinExample
{
   public static void main(String[] args) {
      Thread task = new Thread(new JoinableWorker());
      task.start();

      try { task.join(); }
      catch (InterruptedException ie) { }

      System.out.println("Worker done");
   }
}
```

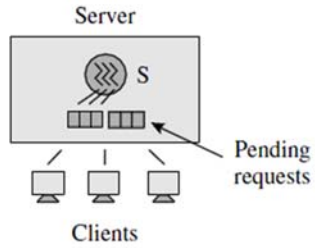
- Thread's **join()** method

4.5 묵시적 쓰레딩

- 묵시적 쓰레딩(Implicit Threading)
  - Thread 생성과 관리를 응용 프로그램 개발자가 아닌 **compiler**와 **runtime library**에게 넘겨주는 것
  - 멀티코어 병렬 처리를 사용하는 프로그램 설계에 사용
- 컴파일러에서 멀티쓰레딩 지원
  - Open MP
  - GCD (Grand Central Dispatch)
- 멀티쓰레딩 관리 방법
  - Thread Pool

Thread Pools

- Multithreaded server에서의 잠재적 문제점
  - thread 생성 오버헤드 : 요청마다 thread를 생성하는 데 시간이 소요됨
  - thread 수가 증가에 따른 자원 고갈 가능성 → thread 수에 제한이 필요
- 해결책 ➔ Thread pool
  - 프로세스를 시작할 때에 일정한 수의 thread들을 thread pool에 미리 생성하여 대기함
  - 요청을 받을 때마다 pool에 있는 한 thread를 깨워서 사용
- 장점
  - 빠른 속도 - 새 thread 생성 소요시간보다 기존 thread로 서비스하는 것이 더 빠름
  - 동시 존재 thread 수 제한 – pool의 크기
  - task 생성 방법을 task에서 분리하면 task 실행을 다른 방식으로 할 수 있음 – 주기적 실행, 일정시간 후 실행 등



OpenMP

- OpenMP (Open Multi-Processing)
  - C, C++, Fortran 컴파일러의 **directive**와 **API**로 multithreaded 프로그래밍 지원.
  - 공유 메모리 환경에서의 병렬 프로그래밍 지원

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

코어 개수의 쓰레드 생성

GCD

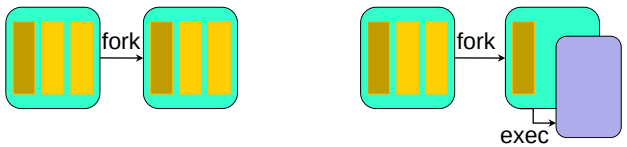
- GCD (Grand Central Dispatch)
  - Apple Mac OS X, iOS 운영체제에서 지원
  - C/C++ 언어의 확장, API, runtime library로 제공
  - 병렬 섹션을 구분하여 쓰레딩 관리 지원 – 블록 표시 ^{ }

```
^{ printf("I am a block"); }
```

  - 블록들은 dispatch queue에 넣고, queue에서 제거될 때에 thread pool에서 가용 thread에 선택하여 할당함

4.6 쓰레드 관련 이슈

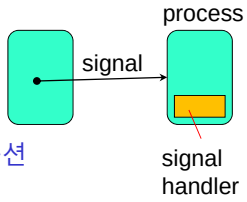
- fork(), exec() 시스템 호출
  - thread에서 fork()를 호출할 때의 동작은?
- 두 가지 fork() 구현 방법
  - process의 **모든 thread들을 복제** : fork – no exec의 경우에 사용
    - 생성된 프로세스는 multi-threaded
  - **해당 thread만 복제** : fork – exec의 경우에 사용
    - 생성된 프로세스는 single threaded



UNIX에서의 Signal handling

- Signal (Process에 대한 signal은 CPU에 대한 interrupt와 비슷함)
  - UNIX에서 프로세스에게 특정 event 발생을 알려주기 위해서 사용
  - 동기식 signal –(ex) illegal memory access, division by zero
  - 비동기식 signal (ex) Ctrl-C로 종료
    signal handler – signal(→ signal number)을 처리하기 위해 사용됨
  - signal은 특정 event에 의해 생성되어 process에게 전달됨
  - process의 signal handler에서 처리됨
- 두 종류의 signal handler
  - A default signal handler
  - A user-defined signal handler
- Multithreaded process에서의 signal 전달 옵션
  - signal이 적용될 thread에게 전달
  - 모든 thread에게 전달
  - 일부 thread에게 선택적으로 전달
  - signal을 전달받는 thread 지정

signal의 유형에 따라서 signal 전달 방법이 정해짐



Thread 취소

- Thread 취소(cancellation)
  - **target thread**가 종료되기 전에 thread를 강제로 중단시키는 것
- 두 가지 취소 방식
  - **비동기(asynchronous) 취소**: 즉시 target thread를 취소
  - **지연(deferred) 취소** : 주기적으로 점검하여 target thread를 취소
- thread 취소의 어려운 점
  - 자원이 할당되어 있거나 공유 데이터를 갱신하고 있는 thread를 취소할 때에 처리에 어려움이 발생함
  - 비동기 취소 방식에서 자원 회수에 문제 발생 가능성
  - 지원 취소 방식에서는 **취소 시점(cancellation point)**에서 안전한 취소 가능 여부를 검사하여 취소

Thread-Local 저장소

- Thread-Local Storage(TLS)
  - thread 자신 만이 접근 가능한 저장 공간
- 대부분의 thread library가 TLS 지원기능 제공
- TLS, local variable, static variable의 비교
  - local 변수 – 함수 호출 동안에만 일시적으로 사용
  - static 변수 – 프로그램 실행 동안 함수 내의 기억장소를 유지
    (같은 함수를 사용하는 thread들은 static 기억장소 공유)
  - TLS – static 변수와 비슷하지만, 각 thread별로 분리된 기억장소 사용

Scheduler Activations

- 경량 프로세스(lightweight process: LWP)
  - Many-to-many 모델에서는 user-level thread와 kernel thread 사이에 LWP(lightweight process)라고 하는 중간 자료구조 사용하여 연결
  - 각 LWP는 kernel thread에 연결되며 user-thread library에게 virtual processor 같이 보여짐 → 가용 LWP에 user-level thread들을 스케줄링
- 각 응용에 필요한 LWP 개수
  - CPU-bound 응용: 1개의 LWP이면 충분
  - I/O intensive 응용: 여러 개의 LWP 필요
- Scheduler Activation
  - **upcall**: thread가 block되거나 event가 발생하면 커널에서 signal을 보내어 thread library의 **upcall handler**를 호출함
  - upcall handler는 새 LWP를 할당 받아 실행되고 다른 thread에게 스케줄링 함
- 운영체제는 kernel thread를 스케줄링 함