4장. 스레드와 병행성 (Thread & Concurrency)

순천향대학교 컴퓨터공학과
이 상 정

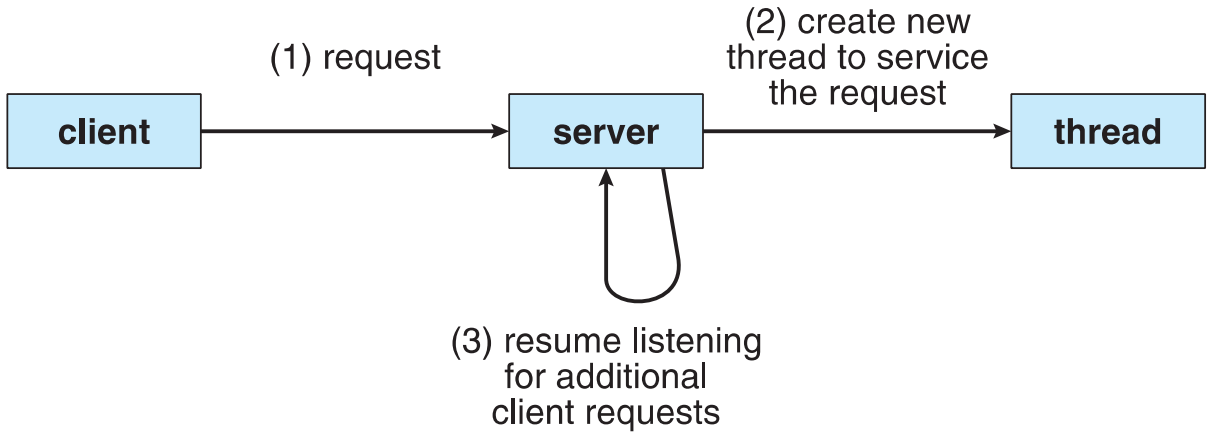
강의 목표 및 내용

- 목표
  - 다중 스레드 컴퓨터 시스템의 기초를 이루는 CPU 이용의 기본 단위인 스레드를 소개
  - Pthreads API 및 Win32와 Java 스레드 라이브러리 소개
- 내용
  - 개요
  - 다중코어 프로그래밍
  - 다중 스레드 모델
  - 스레드 라이브러리
  - 암묵적 스레드
  - 스레드 관련 문제들
  - 운영체제 사례

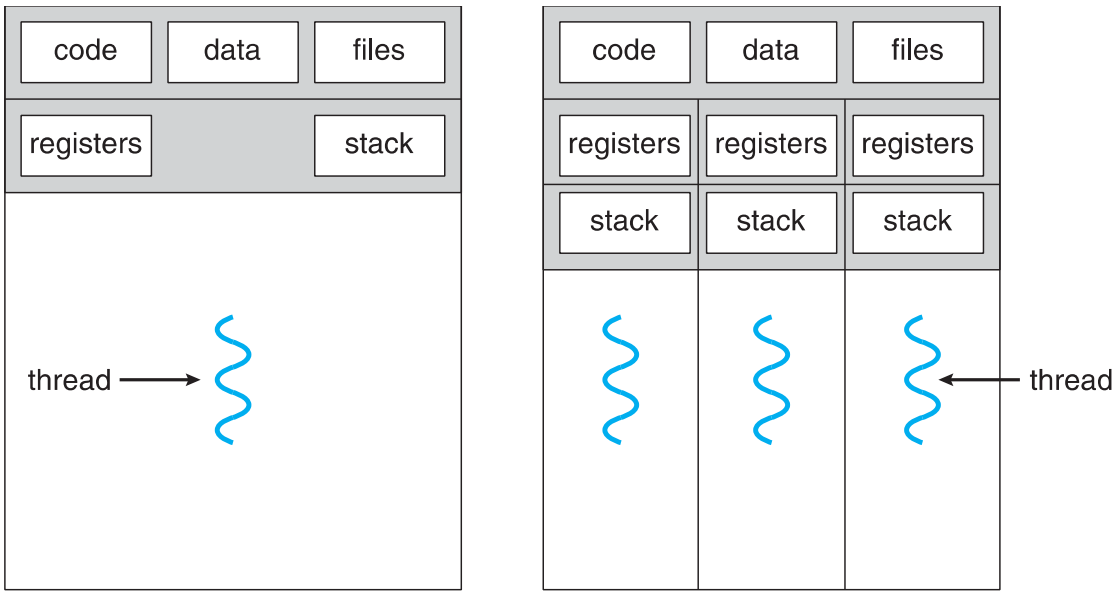
스레드 개요

- 스레드는 CPU 이용의 기본 단위
  - 스레드는 스레드 ID, 프로그램 카운터, 레지스터집합, 스택으로 구성
  - 스레드는 같은 프로세스에 속한 다른 스레드와 코드, 데이터 섹션 그리고, 열린 파일이나 신호와 같은 운영체제 자원들을 공유
- 최근의 응용들은 멀티스레드로 동작
  - 응용에서의 멀티 태스크 동작을 여러 스레드로 구현
    - 디스플레이 갱신
    - 데이터 페치
    - 단어 스펠링 체크
    - 네트워크 요청 응답
- 프로세스 생성 보다 가볍게 스레드 생성
- 스레드 사용하면 코드를 간소화하면서 효율 증가
- 일반적으로 커널은 멀티스레드로 동작

다중 스레드 서버 구조 (Multithreaded Server Architecture)



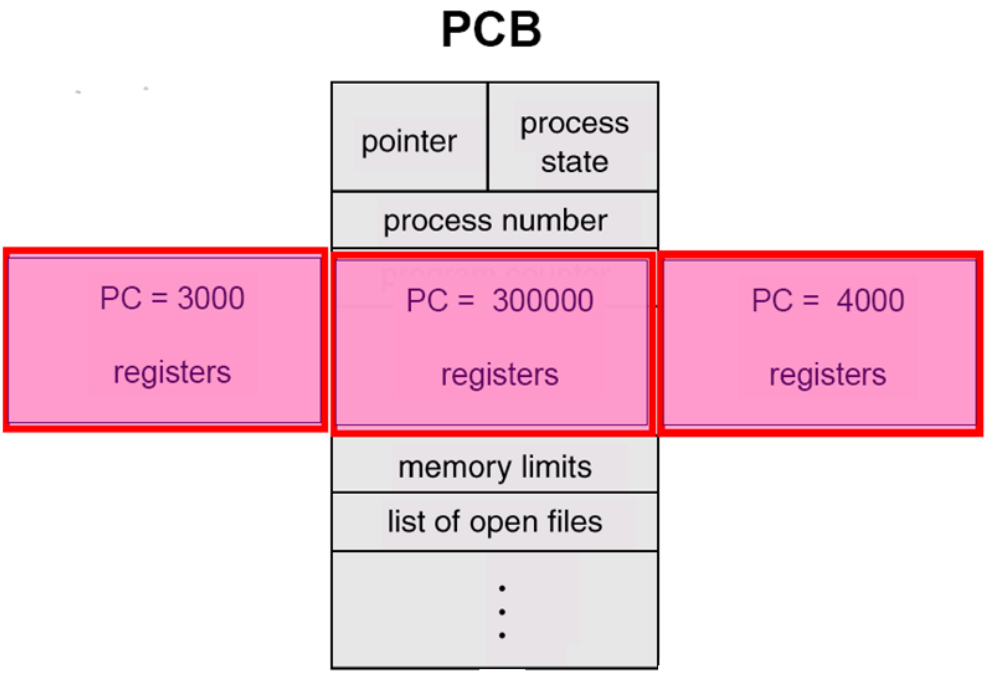
단일 스레드와 다중 스레드 프로세스



single-threaded process

multithreaded process

한 프로세스 – 3개 스레드 PCB 예



다중 스레드 장점

- 응답성 (responsiveness)
  - 사용자에 대한 응답성을 증가
  - 프로세스의 일부분이 봉쇄되거나 긴 작업을 수행해도, 프로그램의 수행의 계속 수행을 허용, 사용자 인터페이스 설계에 유용
    - 다중 스레드 웹 브라우저는 한 스레드가 이미지 파일을 적재하고 있는 동안, 다른 스레드에서 사용자와의 상호 작용이 가능
- 자원 공유 (resource sharing)
  - 스레드는 프로세스의 자원들과 메모리를 공유
  - 한 응용 프로그램이 같은 주소 공간 내에 여러 개의 다른 작업을 하는 스레드 수행 가능
- 경제성 (economy)
  - 스레드 생성과 문맥교환 오버헤드가 프로세스 보다 적음
- 규모 적응성 (scalability)
  - 다중 처리기 구조 (multiprocessor architectures)에서 각각의 스레드가 다른 처리기(CPU)에서 병렬로 수행

다중 코어 프로그래밍 (1)

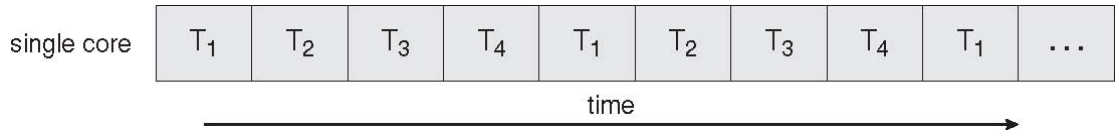
- 다중코어 시스템에서 프로그래밍을 하기 위해 극복해야 할 과제
  - 태스크 인식 (identifying tasks)
    - 응용을 분석하여 독립된 병행 태스크로 분리
  - 균형 (balance)
    - 전체 태스크들이 전체 작업에 균등한 기여도를 갖는 것이 중요
  - 데이터 분리 (data splitting)
    - 데이터도 개별 코아에서 사용할 수 있도록 분리
  - 데이터 종속성 (data dependency)
    - 둘 이상의 태스크가 동시에 접근하는 데이터에 종속성 여부 검토
    - 종속성이 있는 경우 태스크들 간의 동기화 필요
  - 시험 및 디버깅 (testing and debugging)
    - 병렬로 실행되는 다양한 실행 경로로 인해 프로그램 시험과 디버깅이 어려움

다중 코어 프로그래밍 (2)

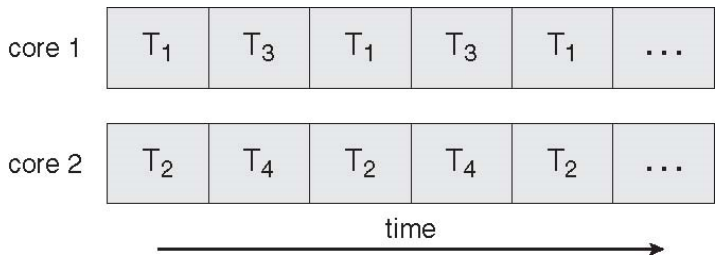
- 병행성과 병렬성 (Concurrency and Parallelisim)
  - 병행성(Concurrency)는 프로세스를 만드는 하나 이상의 태스크를 지원
    - 단일 프로세서/코어, 스케줄러가 병행성 제공
  - 병렬성(parallelism)은 동시에 하나 이상의 태스크를 수행할 수 있는 시스템을 의미
- 병렬 실행의 유형 (Types of Parallelism)
  - 데이터 병렬 실행과 태스크 병렬 실행
  - 데이터 병렬 실행 (data parallelism)
    - 동일한 데이터의 부분집합을 다 수의 계산 코어에 분배
    - 각 코어에서 동일한 연산 수행
  - 태스크 병렬 실행 (task parallelism)
    - 데이터가 아닌 태스크(스레드)를 다 수의 코어에 분배
    - 각 코어의 스레드는 각각 고유 연산을 수행
- 스레드 수가 커지면서 하드웨어 아키텍처가 스레드 지원
  - 오라클 SPARC T4는 8개 코어를 가짐
  - 각 코어 당 8개 하드웨어 스레드 지원

병행성과 병렬성 (Concurrency and Parallelisim)

- 단일 코어에서 병행 실행



- 멀티코어에서 병렬실행



사용자 및 커널 스레드

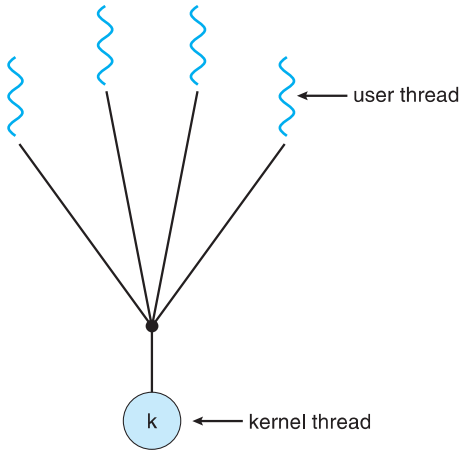
- 사용자 스레드 (user thread)
  - 사용자 수준 스레드 라이브러리로 관리
  - 주요 스레드 라이브러리 예
    - POSIX Pthreads, Win32 스레드, Java 스레드
- 커널 스레드 (kernel thread)
  - 운영체제가 직접 지원하고 관리
  - 예
    - Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

다중 스레드 모델 (Multithreading Model)

- 사용자 스레드와 커널 스레드와의 연관관계
  - 다대일 모델 (Many-to-One Model)
  - 일대일 모델 (One-to-One Model)
  - 다대다 모델 (Many-to-Many Model)

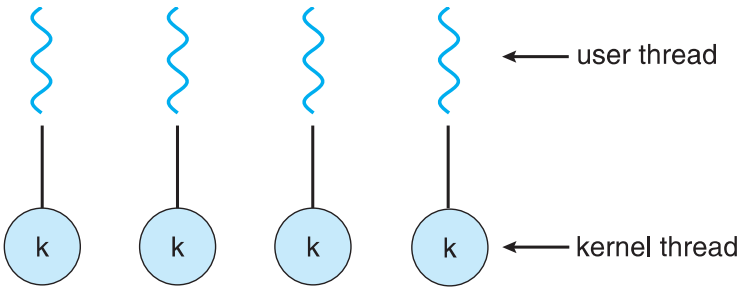
다대일 모델 (Many-to-One Model)

- 많은 사용자 수준 스레드를 하나의 커널 스레드로 사상
  - 사용자 공간의 스레드 라이브러리가 스레드 관리
  - 한 스레드가 봉쇄형 시스템 호출을 할 경우, 전체 프로세스가 봉쇄 (blocking)
  - 다중 스레드가 다중 처리기에서 병렬로 작동할 수 없음
  - 예
    - Solaris의 스레드 라이브러리 (green thread)
    - GNU Portable 스레드



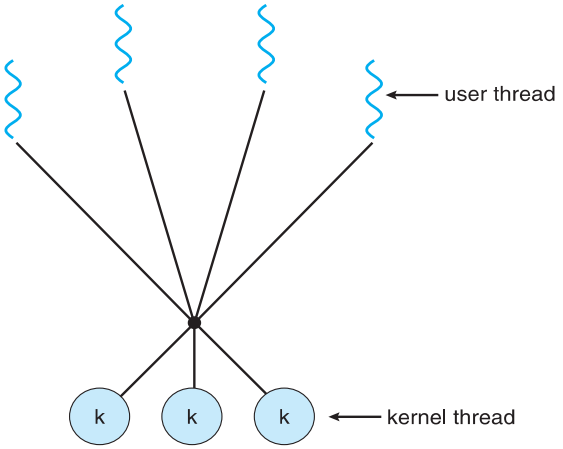
일대일 모델 (One-to-One Model)

- 각 사용자 스레드를 각각 하나의 커널 스레드로 사상
  - 하나의 스레드가 봉쇄적 시스템 호출을 하더라도 다른 스레드가 실행 가능
  - 다중 처리기에서 다중 스레드가 병렬로 수행되는 것을 허용
  - 사용자 수준 스레드를 생성할 때 커널 스레드를 생성해야 하는 오버헤드로 응용 프로그램의 성능저하
  - 예
    - Windows
    - Linux
    - Solaris 9



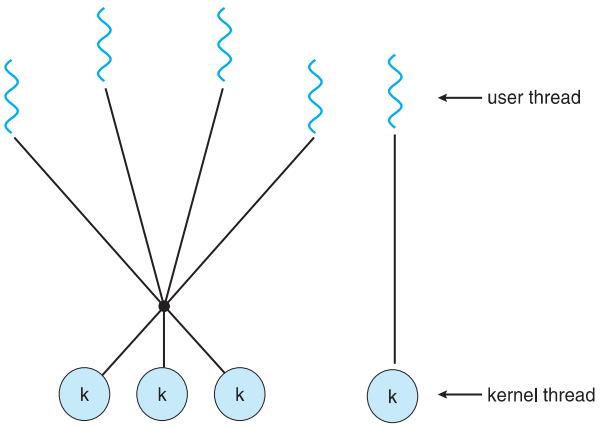
다대다 모델 (Many-to-Many Model)

- 여러 개의 사용자 수준 스레드를 그보다 작거나 같은 수의 커널 스레드로 다중화
  - 운영체제가 개발자의 필요한 만큼 많은 커널 스레드 생성을 허용
  - 예
    - Solaris 9 이전 버전



두 수준 모델 (Two-level Model)

- 다대다 모델과 비슷하며, 하나의 사용자 스레드가 하나의 커널 스레드에 종속되는 것을 허용
  - 예
    - IRIX
    - HP-UX
    - Tru64 UNIX
    - Solaris 8과 이전 버전



스레드 라이브러리 (Threads Library)

- 스레드 라이브러리는 프로그래머에게 스레드를 생성하고 관리하기 위한 API를 제공
- 스레드 라이브러리를 구현
  - 커널의 지원 없이 완전히 사용자 공간에서만 라이브러리를 제공
  - 운영체제에 의해 지원되는 커널 수준 라이브러리를 구현
- 주로 사용되는 세 종류 라이브러리
  - POSIX 표준안의 스레드 확장판인 Pthread
    - 사용자 또는 커널 수준 라이브러리로서 제공
  - Win32 스레드 라이브러리
    - Window 시스템에서 사용 가능한 커널 수준 라이브러리
  - Java 스레드 API
    - Java 프로그램에서 직접 스레드 생성과 관리

Pthread

- POSIX(IEEE 1003.1c)가 스레드 생성과 동기화를 위해 제정한 표준 API
  - 스레드의 동작에 관한 명세(specification)일 뿐이지 그것 자체를 구현(implementation)한 것은 아님
  - 대부분의 UNIX 운영체제에서 적용
    - Solaris, Linux, Mac OS X
- 다중 스레드 프로그램 예
  - 명령어 라인에서 N 값 입력

$$sum = \sum_{i=0}^{N} i$$

Pthread API 사용 예 (1)

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

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

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```


Pthread API 사용 예 (2)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

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

   pthread_exit(0);
}
```

10개의 스레드 종료를 기다리는(join) Pthread 예

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```


Java 스레드(Java Thread)

- Java 스레드들은 JVM이 관리
- Java 프로그램에서 스레드를 생성하는 기법
  - Thread 클래스로부터 파생된 새로운 클래스를 생성하고, Thread 클래스의 run() 메서드를 오버라이드(override)하는 것
  - Runnable 인터페이스를 구현하는 클래스를 정의

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

Java 스레드 예 (1)

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
       sum += i;
    sumValue.setSum(sum);
  }
}
```


Java 스레드 예 (2)

```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

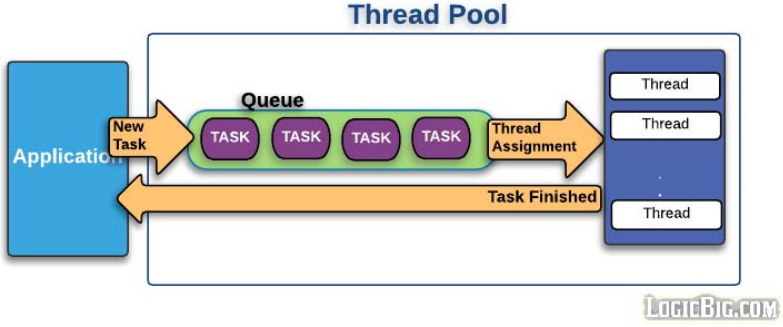
암묵적 스레딩 (Implicit Threading)

- 스레드의 수가 늘어남에 따라 명시적(explicit)으로 프로그램을 작성하고 검증하는 것이 어려워짐
- 프로그래머가 아닌 컴파일러와 실행 라이브러리(run-time library)가 스레드를 생성하고 관리

- 다음 3가지 방식 소개
  - 스레드 풀 (Thread Pool)
  - OpenMP
  - GCD (Grand Central Dispatch)
- 소개되지 않은 방식으로 마이크로소프트의 TBB(Threading Building Blocks), 자바의 java.util.concurrent 패키지 등이 있음

스레드 풀 (Thread Pool)

- 프로세스를 시작할 때 아예 일정한 수의 스레드들을 미리 풀로 생성하고 대기
- 장점
  - 새 스레드를 만들어 주기보다 기존 스레드로 서비스해 주는 것이 더 빠름
  - 스레드 풀은 동시에 존재할 스레드 개수에 제한



OpenMP

- C, C++, Fortran 컴파일러 지시자와 API로 멀티스레드 프로그래밍 지원
- 공유 메모리 환경에서 병렬 프로그래밍 지원
  - 병렬로 수행되는 블록들을 parallel regions로 구분

```
#pragma omp parallel
```
코어 개수 만큼 많은 스레드를 생성

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```
병렬로 루프를 실행

```
#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;
}
```


GCD (Grand Central Dispatch)

- Apple Mac OS X, iOS 운영체제에서 지원
- C, C++ 언어의 확장으로 API, 실행 라이브러리 제공

- 병렬 섹션을 구분하고 스레딩 관리를 지원
  - 블록 표시, "^{ }"
    ```
    ^{ printf("I am a block"); }
    ```

- 블록들은 발송 큐(dispatch queue)에 놓임
  - 큐에서 제거되는 가용한 스레드는 스레드 풀에 지정

스레드와 관련된 문제들 – fork(), exec() (Threading Issues)

- 스레드와 관련된 문제들
  - fork(), exec() 시스템 호출
  - 신호 처리 (signal handling)
  - 스레드 취소
  - 스레드 로컬 저장소 (thread-local storage)
  - 스케줄러 액티베이션 (scheduler activation)

- fork() 및 exec() 시스템 호출
  - 한 프로그램의 스레드가 fork()를 호출하면 새로운 프로세스는 모든 스레드를 복제 또는 호출한 스레드만 복제하는가?
  - fork() 후 exec() 호출하는 경우 호출한 스레드만 복사해주는 것이 적절

스레드와 관련된 문제들 – 신호 처리

- 신호 처리 (signal handling)
  - 신호 (signal)는 UNIX에서 프로세스에게 어떤 사건(event)이 일어났음을 알려주기 위해 사용
  - 신호는 다음과 같은 형태로 전달
    - 신호는 특정 사건이 일어나야 생성
    - 신호가 생성되면 프로세스에게 전달
    - 신호가 전달되면 반드시 처리
  - 다중 스레드 프로그램에서는 어느 스레드에게 신호를 전달?
    - 신호가 적용될 스레드에게 전달
    - 모든 스레드에게 전달
    - 몇몇 스레드들에게만 선택적으로 전달
    - 특정 스레드가 모든 신호를 전달받도록 지정
  - 신호를 전달하는 표준 UNIX 함수는 kill(pid_t pid, int signal)
  - POSIX Pthread는 pthread_kill(pthread_t tid, int signal) 함수도 제공

시그널 예

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

int count=0;

// 시그널 핸들러
void catch_sigint(int signum)
{
    printf("₩n(count=%d) CTRL-C pressed₩n", count++);
}

int main()
{
    struct sigaction act;

     // 시그널 핸들러 설정
    act.sa_handler = catch_sigint;
    sigaction(SIGINT, &act, NULL);

    while (1) {
        if (count == 5)
            break;
    }

    return 0;
}
```

```
lee@leeVB: ~/os
lee@leeVB:~/os$ gcc -o signal signal.c
lee@leeVB:~/os$
lee@leeVB:~/os$ ./signal
^C
(count=0) CTRL-C pressed
^C
(count=1) CTRL-C pressed
^C
(count=2) CTRL-C pressed
^C
(count=3) CTRL-C pressed
^C
(count=4) CTRL-C pressed
lee@leeVB:~/os$
```


스레드와 관련된 문제들 – 취소 (1)

- 스레드 취소 (thread cancellation)
  - 스레드 취소(thread cancellation)는 스레드가 끝나기 전에 그것을 강제 종료시키는 작업
  - 2가지 방식
    - 비동기식 취소(asynchronous cancellation)
      한 스레드가 즉시 목적 스레드를 강제 종료
    - 지연 취소(deferred cancellation)
      목적 스레드가 주기적으로 자신이 강제 종료되어야 할지를 점검
  - 스레드 생성 및 취소 pthread 코드 예

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

스레드와 관련된 문제들 – 취소 (2)

- 스레드 취소 요청의 처리는 스레드의 상태에 따라 다름
- Pthread의 3 가지 취소 모드

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

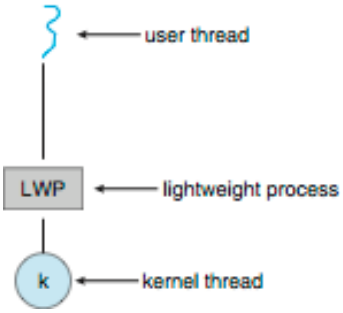
  - 모드가 off 인 경우 스레드 취소할 수 없음
  - 디폴트 모드는 지연 (deferred) 취소
    - 취소는 스레드가 취소점(cancellation point)에 도달한 경우 취소 수행
      - 예, pthread_testcancel() 호출
        이 후 정리 처리기 (cleanup handler) 호출하여 수행
  - Pthread에서는 비동기식 취소는 권장하지 않음
- Linux 시스템에서는 신호를 사용하여 스레드를 취소

스레드와 관련된 문제들 – 스레드 국지 저장소 (Thread-Local Storage)

- 스레드-로컬 저장소(Thread-Local Storage, TLS)
  - 스레드 자신만 접근 가능한 데이터를 허용
    - 스레드별 데이터
  - 지역 변수 (local variable)와 차이
    - 지역 변수는 하나의 함수가 호출되는 동안 가용
    - TLS는 전체 함수 호출 동안 가용
    - 정적 데이터 (static data)와 유사
  - 대부분의 스레드 라이브러리들은 어떤 형태로든 이와 같은 스레드 국지 저장소를 지원

스레드와 관련된 문제들 – 스케줄러 액티베이션 (Scheduler Activations)

- 다대다와 두 수준 모델은 사용자 스레드와 커널 간의 통신 조정이 필요
  - 응용 프로그램의 성능을 위해 커널 스레드 수를 동적으로 조정하는 것이 필요
- 일반적으로 사용자와 커널 스레드 사이에 중간 자료 구조를 사용, LWP(lightweight process, 경량 프로세스)
  - 응용이 사용자 스레드를 실행하기 위해 스케줄할 가상 처리기(virtual processor)처럼 보임
  - 각 LWP는 하나의 커널 스레드에 연결
- 스케줄러 액티베이션은 업콜(upcall)을 제공
  - 커널에서 스레드 라이브러리의 업콜 처리기(upcall handler)로 통신 메커니즘
- 이러한 통신 기법으로 한 응용이 올바른 수의 커널 스레드를 유지



과 제

실습과제 – Pthread 예 1

1. 다음과 같은 p.187 그림 4.11의 수정된 버전을 아래와 같이 두 개의 터미널에서 실행하고 결과 분석
   - 사용된 pthread API 설명 분석
   - 컴파일 시 다음과 같이 –pthread 옵션을 기술
     $ gcc –pthread prog.c
   - 터미널1에서 소스 작성하고 실행 시작 후 문자 입력 전 대기
   - 터미널2를 생성하고 다음을 실행하고 출력 분석
     $ ps –am –L
   - 터미널 1에서 문자 입력하고 실행 결과 분석


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

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

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
            fprintf(stderr,"usage: a.out <integer value>₩n");
            return -1;
   }
   if (atoi(argv[1]) < 0) {
            fprintf(stderr,"%d must be >= 0₩n",atoi(argv[1]));
            return -1;
   }
```

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d₩n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);

    sum = 0;
    for (i = 1; i <= upper; i++)
            sum += i;

    printf("Input any character to continue₩n");
    getchar();

    pthread_exit(0);
}
```


실습과제 – Pthread 예 2

2. 다음 프로그램을 아래와 같이 두 개의 터미널에서 실행하고 결과 분석
   - 터미널1에서 소스 작성하고 실행 시작 후 문자 입력 전 대기
   - 터미널2를 생성하고 다음을 실행하고 출력 분석
     $ ps –am –L
   - 터미널 1에서 문자 입력하고 실행 결과 분석

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  int pid;
  pthread_t tid;
  pthread_attr_t attr;

  pid = fork();

  if (pid == 0) { /* child process */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d₩n", value); /* LINE C */
  }
  else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d₩n", value); /* LINE P */
  }
 }
```



```
void *runner(void *param)
{
  value = 5;

  printf("Input any character to continue₩n");
  getchar();

  pthread_exit(0);
}
```

특별 실습 과제 – 생산자/소비자 프로그램

- 3장의 멀티프로세스 유한 버퍼의 생산자, 소비자 프로그램을 멀티스레드 프로그램으로 변환하여 작성
  - 생산 및 소비되는 데이터는 임의의 응용 데이터
  - 소스 프로그램 및 설명
  - 실행 예