

LG전자 BS팀

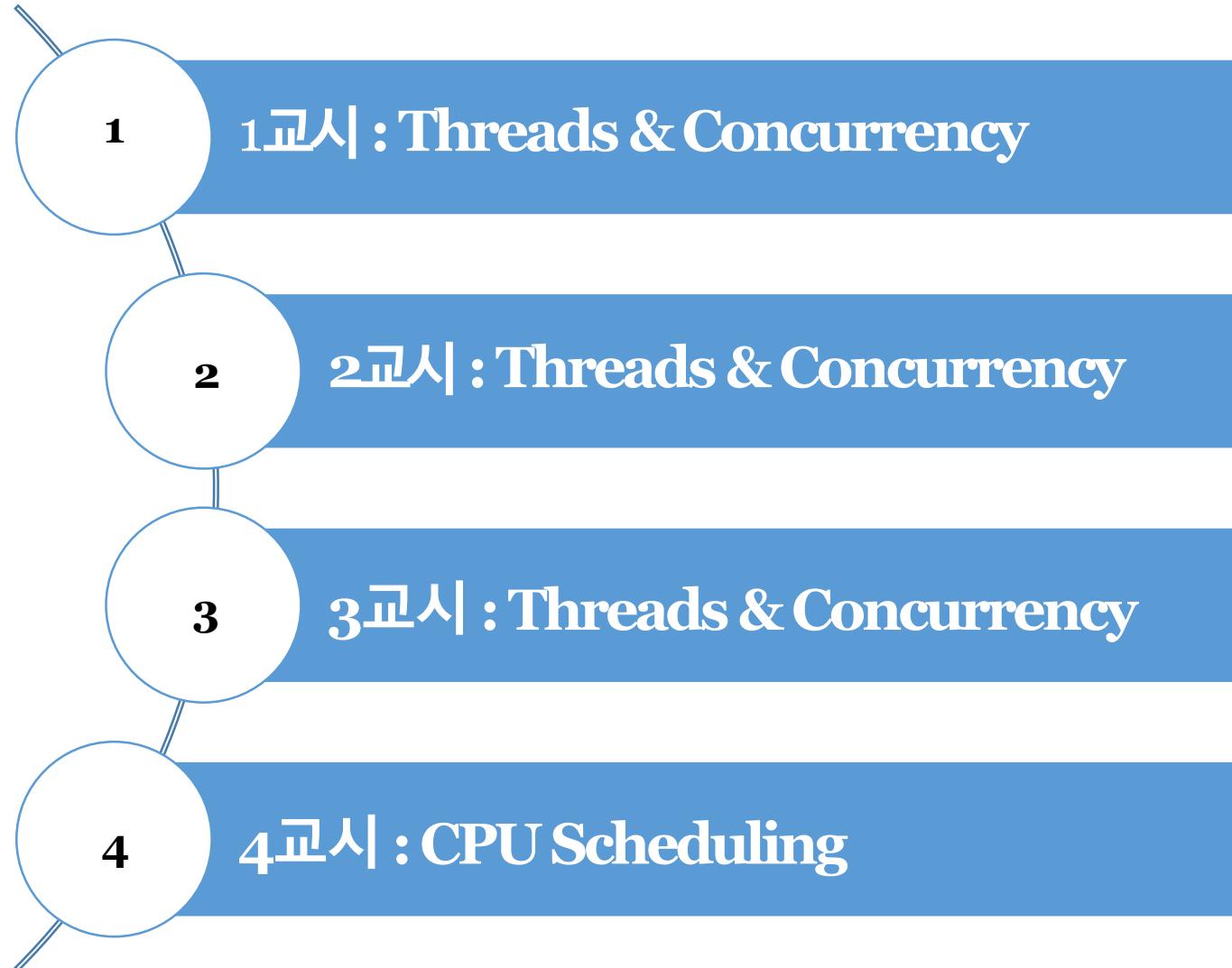
『2일차』: 오전

- ◆ 훈련과정명 : OS 기본
- ◆ 훈련기간 : 2023.05.30 ~ 2023.06.02

Copyright 2022, DaeKyeong all rights reserved



목차



『4과목』

1-3교시 :

Threads & Concurrency





Syllabus

학습목표

- 이 워크샵에서는 스레드의 기본 구성 요소를 식별하고 스레드와 프로세스를 대조할 수 있다.
- 다중 스레드 응용 프로그램 설계의 이점과 과제 설명할 수 있다.
- 스레드 풀, fork-join 및 Grand Central Dispatch를 포함하여 암시적 스레딩에 대한 다양한 접근 방식 설명할 수 있다.
- Windows 및 Linux 운영 체제가 스레드를 나타내는 방법 설명할 수 있다.
- Pthread, Java 및 Windows 스레딩 API를 사용하여 다중 스레드 애플리케이션 설계할 수 있다.



Syllabus

눈높이 체크

- 프로세스 개념을 알고 계신가요?
- 멀티코어 프로그래밍을 알고 계신가요?
- 멀티스레딩 모델을 알고 계신가요?
- 스레드 라이브러리를 알고 계신가요?
- 암시적 스레딩을 알고 계신가요?
- 스레딩 문제을 알고 계신가요?
- 운영 체제 예



1. Overview

동기 부여

- 대부분의 최신 애플리케이션은 다중 스레드.
- 응용 프로그램 내에서 실행되는 스레드
- 응용 프로그램의 여러 작업을 별도의 스레드로 구현할 수 있다.
- 디스플레이 업데이트
 - 데이터 가져오기
 - 맞춤법 검사
 - 네트워크 요청에 응답
- 프로세스 생성은 무겁고 스레드 생성은 가볍다.
- 코드를 단순화하고 효율성을 높일 수 있다.
- 커널은 일반적으로 다중 스레드.



1. Overview

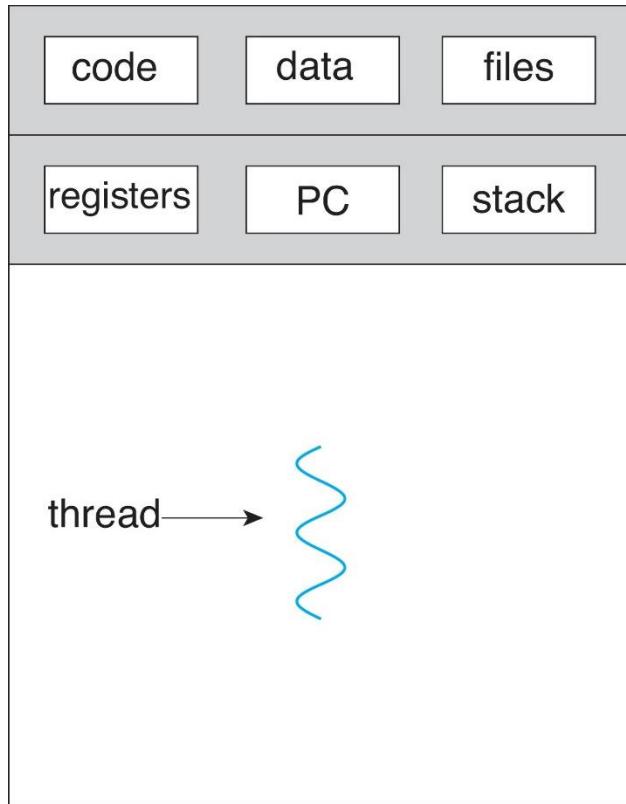
스레드 :

- 프로세스가 생성되면 CPU 스케줄러는 프로세스가 해야 할 일을 CPU에 전달하고, 실제 작업은 CPU가 수행한다. 이때 CPU 스케줄러가 CPU에 전달하는 일 하나가 스레드이다.
 - 그러므로 CPU가 처리하는 작업의 단위는 프로세스로부터 전달받은 스레드인 것이다.
 - Thread(operation) < Process(task) < Job
 - 운영체제의 입장에서 작업 단위는 프로세스이고, CPU 입장에서의 작업 단위는 스레드인 것이다. 따라서 프로세스 입장에서는 스레드를 다음과 같이 정의할 수 있다.
-
- 프로세스의 코드에 정의된 절차에 따라 CPU에 작업 요청을 하는 실행 단위이다.

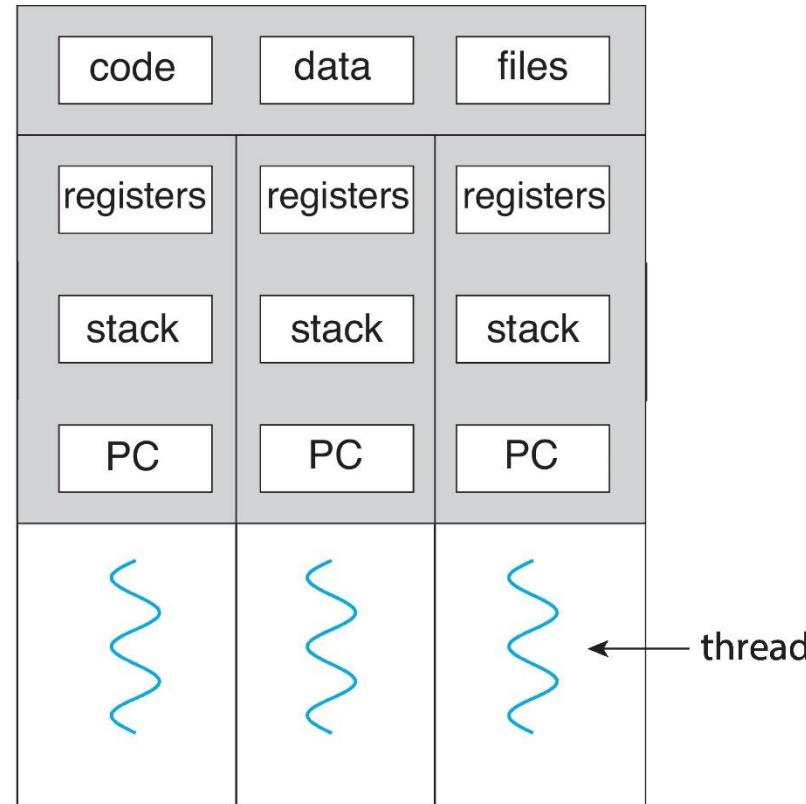


1. Overview

Single and Multithreaded Processes



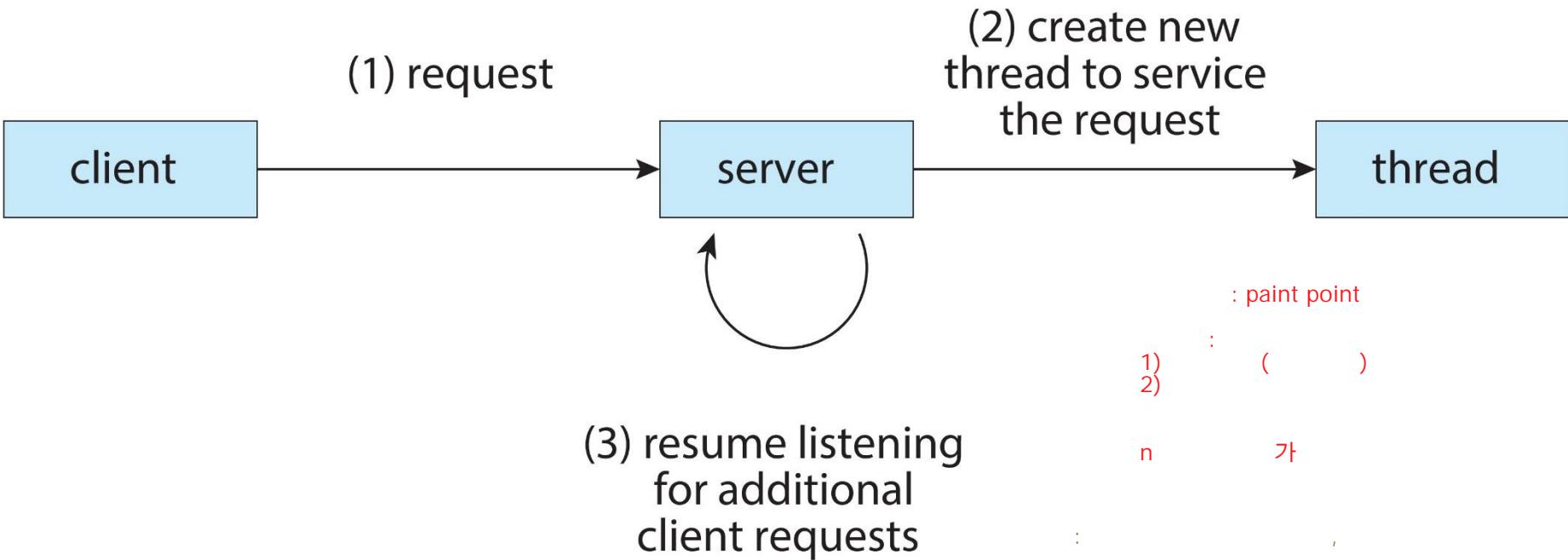
single-threaded process



multithreaded process

1. Overview

Multithreaded Server Architecture





1. Overview

Benefits

- 응답성 **Responsiveness** – 프로세스의 일부가 차단된 경우 지속적인 실행을 허용할 수 있으며 특히 사용자 인터페이스에 중요.
- 리소스 공유 **Resource Sharing** – 스레드는 공유 메모리나 메시지 전달보다 쉽게 프로세스의 리소스를 공유.
- 경제성 **Economy** – 프로세스 생성보다 저렴하고 스레드 전환이 컨텍스트 전환보다 낮은 오버헤드
- 확장성 **Scalability** – 프로세스는 멀티코어 아키텍처를 활용할 수 있다.



2. Multicore Programming

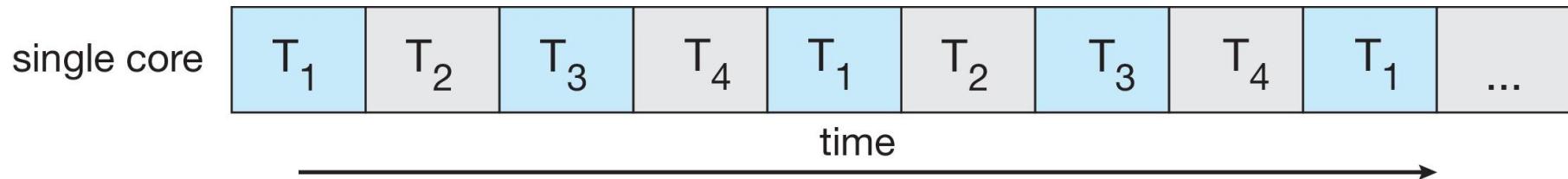
Multicore Programming

- 멀티코어 Multicore 또는 멀티프로세서 multiprocessor 시스템은 프로그래머에게 압력을 가하며 다음과 같은 과제를 안고 있다.
 - 활동 나누기
 - 균형
 - 데이터 분할
 - 데이터 종속성
 - 테스트 및 디버깅
- 병렬성 Parallelism 은 시스템이 둘 이상의 작업을 동시에 수행할 수 있음을 의미.
- 동시성 Concurrency 은 하나 이상의 작업 진행을 지원.
 - 단일 프로세서/코어, 동시성을 제공하는 스케줄러

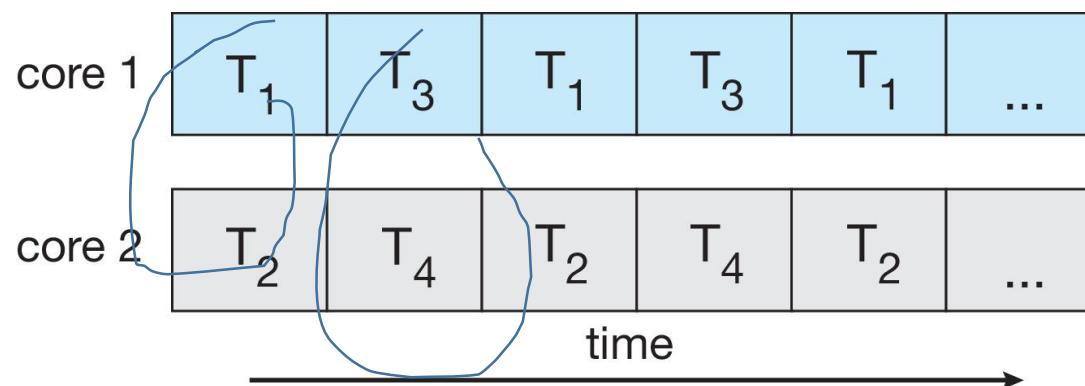
2. Multicore Programming

Concurrency vs. Parallelism Programming

- 단일 코어 시스템에서 동시 실행:



- 다중 코어 시스템의 병렬성:

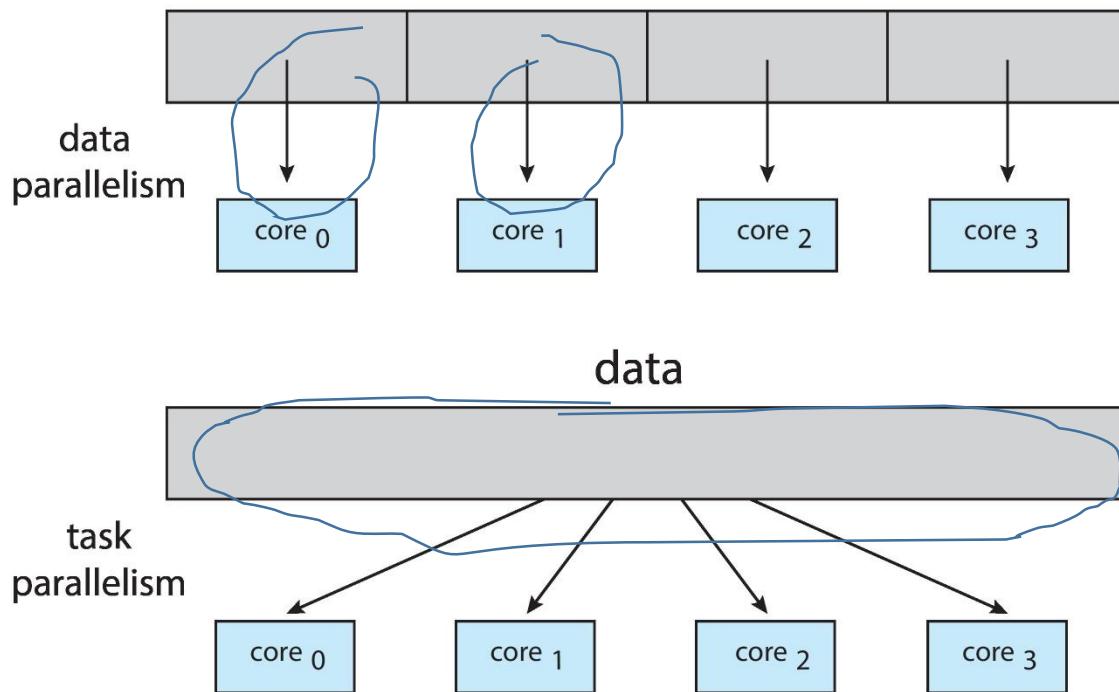


2. Multicore Programming

병렬 처리 유형

- 데이터 병렬성 Data parallelism – 동일한 데이터의 하위 집합을 여러 코어에 분산하고 각 코어에서 동일한 작업을 수행.
- 작업 병렬성 Task parallelism - 코어에 스레드를 분산하고 각 스레드는 고유한 작업을 수행.

Data and Task Parallelism



2. Multicore Programming

암달의 법칙 Amdahl's Law

가

가

- 직렬 및 병렬 구성 요소가 모두 있는 응용 프로그램에 추가 코어를 추가하여 성능 향상을 식별.

- S는 직렬 부분.
- N 프로세싱 코어

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

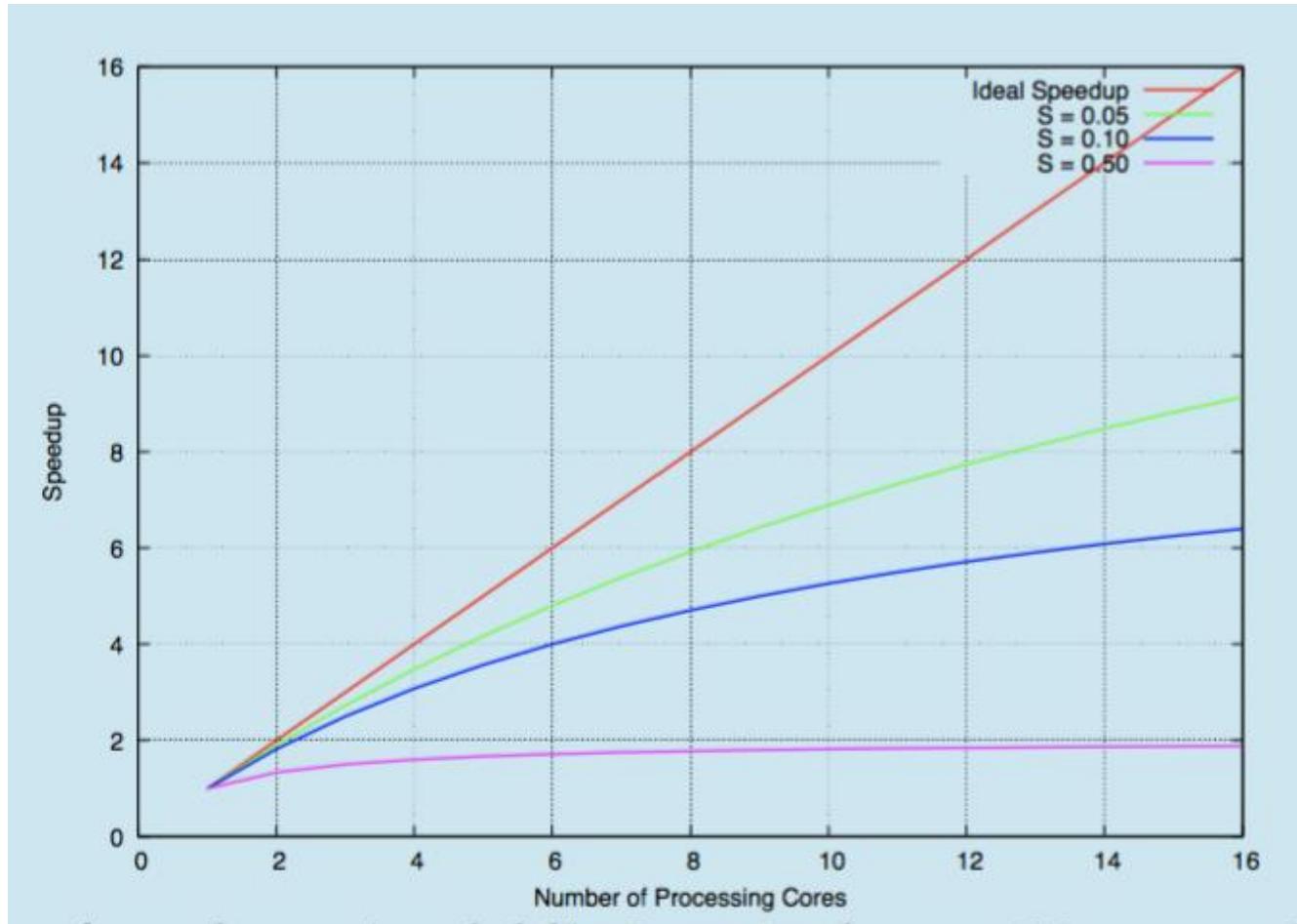
- 즉, Application이 병렬 75% / 직렬 25%일 경우 1코어에서 2코어로 이동하면 1.6배의 속도 향상

$$S=0.25, N=2, speedup = 1.6$$

- N이 무한대에 가까워지면 속도가 $1/S$ 에 가까워짐.
- 응용 프로그램의 직렬 부분은 추가 코어를 추가하여 얻은 성능에 불균형한 영향을 미친다.

2. Multicore Programming

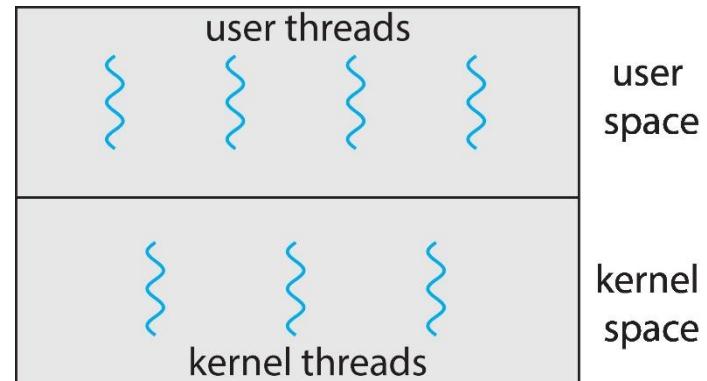
암달의 법칙 Amdahl's Law



2. Multicore Programming

사용자 스레드 및 커널 스레드

- **사용자 스레드 User threads** - 사용자 수준 스레드 라이브러리에서 수행하는 관리
- 세 가지 기본 스레드 라이브러리:
 - POSIX P스레드 Pthreads
 - Windows 스레드
 - 자바 스레드
- **커널 스레드 Kernel threads** - 커널에서 지원
- 예 – 다음을 포함한 거의 모든 범용 운영 체제:
 - 윈도우
 - 리눅스
 - 맥 OS X
 - 아이폰 OS
 - 기계적 인조 인간





3. Multithreading Models

Multithreading Models

- User threads와 Kerner threads 사이의 관계

- Many-to-One
- One-to-One
- Many-to-Many

one-to-Many

thread: (,) >>

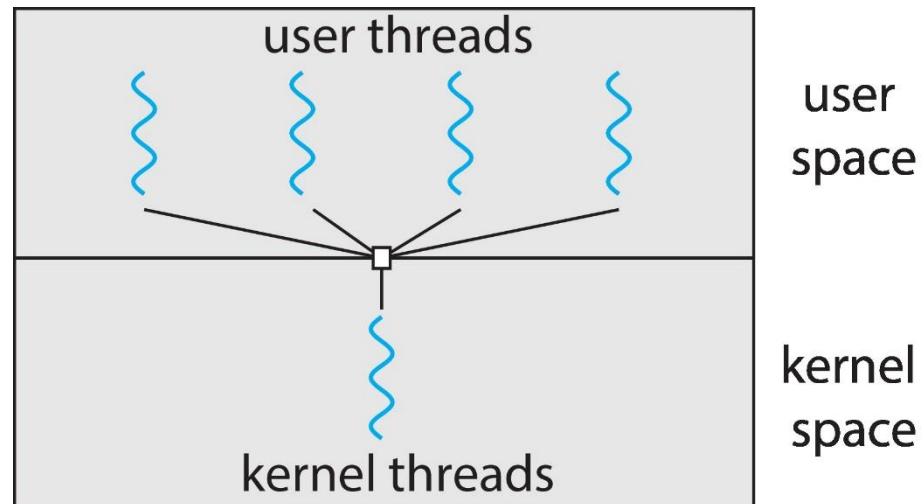
User threads

Kerner threads

3. Multithreading Models

Many-to-One

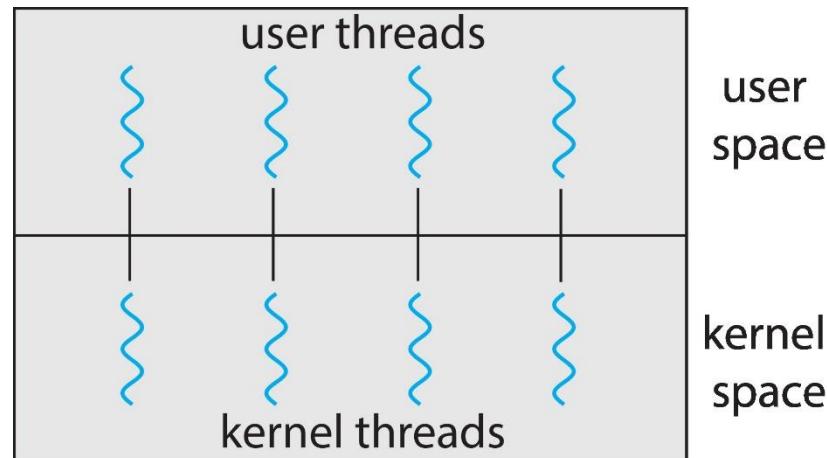
- 단일 커널 스레드에 매핑된 많은 사용자 수준 스레드
- 하나의 스레드 차단으로 인해 모두 차단됨
- 한 번에 하나만 커널에 있을 수 있으므로 다중 스레드가 다중 코어 시스템에서 병렬로 실행되지 않을 수 있습니다.
- 현재 이 모델을 사용하는 시스템은 거의 없습니다.
- 예:
 - 솔라리스 그린 스레드
 - GNU 휴대용 스레드



3. Multithreading Models

One-to-One

- 각 사용자 수준 스레드는 커널 스레드에 매팅.
- 사용자 수준 스레드를 생성하면 커널 스레드가 생성.
- 다대일보다 더 많은 동시성
- 오버헤드로 인해 때때로 프로세스당 스레드 수가 제한됨
- 예
- 윈도우
- 리눅스

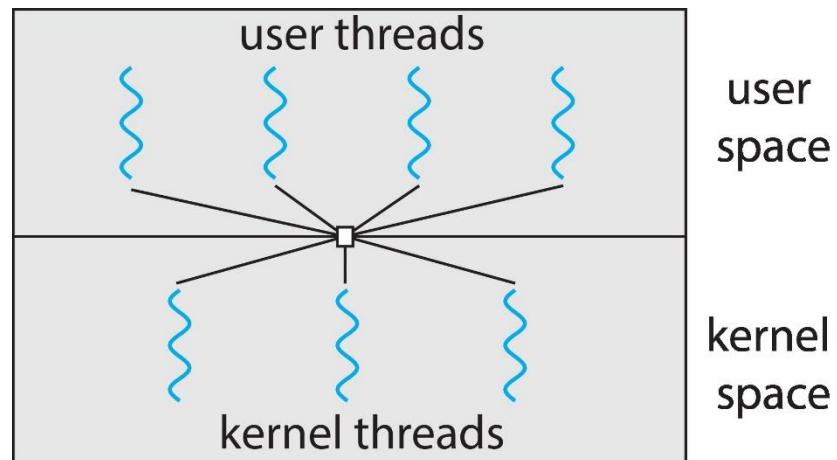




3. Multithreading Models

Many-to-Many Model

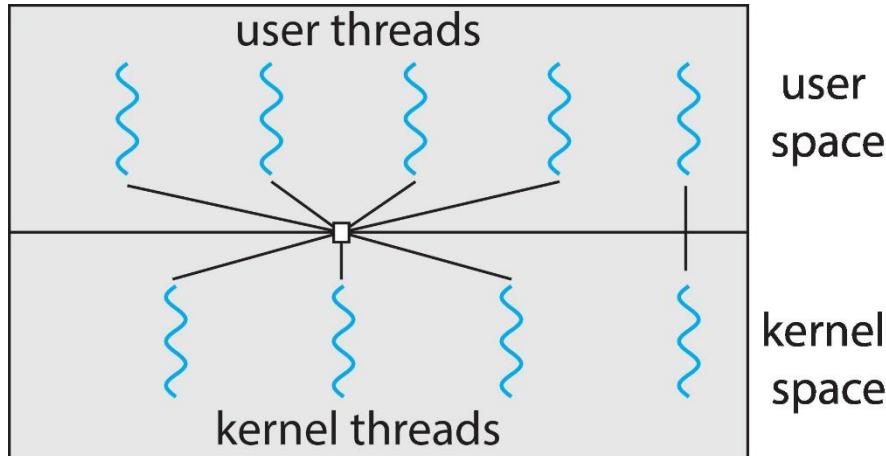
- 많은 사용자 수준 스레드를 많은 커널 스레드에 매핑할 수 있다.
- 운영 체제가 충분한 수의 커널 스레드를 생성하도록 허용
- **ThreadFiber** 패키지가 있는 Windows
- 그렇지 않으면 일반적이지 않습니다.



3. Multithreading Models

Two-level Model

- 사용자 스레드가 커널 스레드에 바인딩될 수 있다는 점을 제외하면 M:M과 유사.





4. Thread Libraries

Thread library

- 스레드 라이브러리 Thread library 는 프로그래머에게 스레드 생성 및 관리를 위한 API를 제공.
- 두 가지 기본 구현 방법
 - 1) 완전히 사용자 공간에 있는 라이브러리
 - 2) OS에서 지원하는 커널 수준 라이브러리



4. Thread Libraries

Pthreads

- 사용자 수준 또는 커널 수준으로 제공될 수 있다.
- 스레드 생성 및 동기화를 위한 POSIX 표준(IEEE 1003.1c) API
- 구현 **implementation** 이 아닌 사양 **Specification**
- API는 스레드 라이브러리의 동작을 지정하고 구현은 라이브러리 개발에 달려 있다.
- UNIX 운영 체제(Linux 및 Mac OS X)에서 공통



4. Thread Libraries

Pthreads

- Pthreads Example

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

#include <stdlib.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 */

    /* set the default attributes of the thread */
    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);
}
```



4. Thread Libraries

Pthreads

- Pthreads Example

```
/* The thread will execute 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);
}
```



4. Thread Libraries

Pthreads

- Pthreads Code for Joining 10 Threads

```
#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);
```

- gcc -pthread pthread.c



4. Thread Libraries

Pthreads

파일

실습환경

소스코드

```
pthread.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > pthread.c
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum;
void *runner(void *param);

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

    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid,NULL);

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

void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

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

    pthread_exit(0);
}
```

소스코드

결과값1

비고



4. Thread Libraries

Pthreads

파일

소스코드

실습환경

pthread.c

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -pthread pthread.c  
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
```

Segmentation fault (core dumped)

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out 10
```

SUM = 55

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out 100
```

SUM = 5050

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out 1000
```

SUM = 500500

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```

결과값1

비고



4. Thread Libraries

Windows Multithreaded C Program

- Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```



4. Thread Libraries

Windows Multithreaded C Program

- Windows Multithreaded C Program

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

    Param = atoi(argv[1]);
    /* create the thread */
    ThreadHandle = CreateThread(
        NULL, /* default security attributes */
        0, /* default stack size */
        Summation, /* thread function */
        &Param, /* parameter to thread function */
        0, /* default creation flags */
        &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

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



4. Thread Libraries

Java Threads

- Java 스레드는 JVM에서 관리.
- 일반적으로 기본 OS에서 제공하는 스레드 모델을 사용하여 구현
- Java 스레드는 다음에 의해 생성될 수 있다.
 - 스레드 클래스 확장
 - Runnable 인터페이스 구현

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

- 표준 관행은 Runnable 인터페이스를 구현하는 것.



4. Thread Libraries

Java Threads

- Implementing Runnable interface:

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

- Creating a thread:

```
Thread worker = new Thread(new Task());  
worker.start();
```

- Waiting on a thread:

```
try {  
    worker.join();  
}  
catch (InterruptedException ie) { }
```



4. Thread Libraries

Java Threads

● Thread 클래스 상속

파일

소스코드

실행환경

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > MyThread1.java
public class MyThread1 extends Thread{
```

```
    @Override
    public void run() {
        try
        {
            while(true)
            {
                System.out.println("Hello, Thread!");
                Thread.sleep(500);
            }
        }
        catch(InterruptedException ie) {
            System.out.println("I'm interruptedException");
        }
    }

    public static void main(String[] args)
    {
        MyThread1 thread = new MyThread1();
        thread.start();
        System.out.println("Hello, my child");
    }
}
```

결과값1

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac MyThread1.java
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java MyThread1
Hello, my child
Hello, Thread!
Hello, Thread!
Hello, Thread!
```

비고



4. Thread Libraries

Java Threads

● Runnable 인터페이스 구현

파일

소스코드

실습환경

```
^Ck8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > MyThread2.java
public class MyThread2 implements Runnable{
    @Override
    public void run() {
        try {
            while(true)
            {
                System.out.println("Hello, Runnable!");
                Thread.sleep(500);
            }
        } catch(InterruptedException ie) {
            System.out.println("I'm interruptedException");
        }
    }

    public static void main(String[] args)
    {
        Thread thread = new Thread(new MyThread2());
        thread.start();
        System.out.println("Hello, My Runnable Child");
    }
}
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$
```

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac MyThread2.java
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java MyThread2
Hello, Runnable!
Hello, Runnable Child
Hello, Runnable!
Hello, Runnable!
Hello, Runnable!
Hello, Runnable!
Hello, Runnable!
```

결과값1

비고

4. Thread Libraries

Java Threads

● Runnable 람다 표현식 사용하기

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > MyThread3.java
```

```
public class MyThread3{  
    public static void main(String[] args)  
    {  
        Runnable task = ()->{  
            try  
            {  
                while(true)  
                {  
                    System.out.println("Hello, Lambda Runnable!");  
                    Thread.sleep(500);  
                }  
            }  
            catch(InterruptedException ie)  
            {  
                System.out.println("I'm interrupted");  
            }  
        };  
  
        Thread thread = new Thread(task);  
        thread.start();  
        System.out.println("Hello, My Lambda Child!");  
    }  
}
```

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac MyThread3.java
```

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java MyThread3
```

```
Hello, My Lambda Child!
```

```
Hello, Lambda Runnable!
```

결과값1

비고



4. Thread Libraries

Java Threads

● 부모 쓰레드의 대기 : join

파일

소스코드

실습환경

```
^Ck8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > MyThread4.java  
public class MyThread4{
```

```
    public static void main(String[] args)  

```

```
        Thread thread = new Thread(task);  

```

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac MyThread4.java  
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java MyThread4
```

```
Hello, Lambda Runnable!  
Hello, My Joined C
```

결과값1

비고

- 자식 쓰레드가 종료될때까지 대기했다가 부모 쓰레드의 나머지 명령어를 수행합니다.



4. Thread Libraries

Java Threads

● 쓰레드의 종료

파일

소스코드

실습환경

```
8s@DESKTOP-RoEQ2U6:~/java_workspaces$ cat > MyThread5.java
public class MyThread5{
```

```
    public static void main(String[] args) throws InterruptedException
    {
        Runnable task = ()->{
            try
            {
                while(true)
                {
                    System.out.println("Hello, Lambda Runnable!");
                    Thread.sleep(100);
                }
            } catch(InterruptedException ie)
            {
                System.out.println("I'm Interruped");
            }
        };

        Thread thread = new Thread(task);
        thread.start();
        Thread.sleep(500);
        thread.interrupt();
        System.out.println("Hello, My Interruped Child!");
    }
}
```

```
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ javac MyThread5.java
k8s@DESKTOP-RoEQ2U6:~/java_workspaces$ java MyThread5
```

```
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, My Interruped Child!
I'm Interruped
```

결과값1

비고

- 쓰레드를 명시적으로 종료시킵니다. 그러나 사용하지 않는것을 권고합니다.



4. Thread Libraries

Java Executor Framework

- 스레드를 명시적으로 생성하는 대신 Java는 Executor 인터페이스 주변에서 스레드 생성도 허용.

```
public interface Executor
{
    void execute(Runnable command);
}
```

- Executor는 다음과 같이 사용.

```
Executor service = new Executor;
service.execute(new Task());
```



4. Thread Libraries

Java Executor Framework

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
    private int upper;
    public Summation(int upper) {
        this.upper = upper;
    }

    /* The thread will execute in this method */
    public Integer call() {
        int sum = 0;
        for (int i = 1; i <= upper; i++)
            sum += i;

        return new Integer(sum);
    }
}
```



4. Thread Libraries

Java Executor Framework

```
public class Driver
{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);

        ExecutorService pool = Executors.newSingleThreadExecutor();
        Future<Integer> result = pool.submit(new Summation(upper));

        try {
            System.out.println("sum = " + result.get());
        } catch (InterruptedException | ExecutionException ie) { }
    }
}
```



5. Implicit Threading

Java Executor Framework

- 스레드 수가 증가함에 따라 인기가 높아지고 명시적 스레드에서 프로그램 정확성이 더 어려워짐
- 프로그래머가 아닌 컴파일러 및 런타임 라이브러리가 수행하는 스레드 생성 및 관리
- 다섯 가지 방법 탐색
 1. 스레드 풀 Thread Pools
 2. 포크 조인 Fork-Join
 3. OpenMP
 4. 그랜드 센트럴 파견 Grand Central Dispatch
 5. 인텔 스레딩 빌딩 블록



5. Implicit Threading

Thread Pools

- 작업을 기다리는 풀에서 여러 스레드 생성.
- 이점 Advantages :
 - 일반적으로 새 스레드를 생성하는 것보다 기존 스레드로 요청을 처리하는 것이 약간 더 빠름
 - 응용 프로그램의 스레드 수를 풀 크기에 바인딩.
 - 작업 생성 메커니즘에서 수행할 작업을 분리하면 작업 실행을 위한 다양한 전략이 가능.
 - 즉, 작업이 주기적으로 실행되도록 예약할 수 있다.
- Windows API는 스레드 풀을 지원.

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```



5. Implicit Threading

Java Thread Pools

- **Executors** 클래스에서 스레드 풀을 생성하기 위한 세 가지 팩토리 메서드:

- static ExecutorService newSingleThreadExecutor()
- static ExecutorService newFixedThreadPool(int size)
- static ExecutorService newCachedThreadPool()



5. Implicit Threading

Java Thread Pools

- Executors 클래스에서 스레드 풀을 생성하기 위한 세 가지 팩토리 메서드:

```
import java.util.concurrent.*;

public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

        /* Run each task using a thread in the pool */
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task());

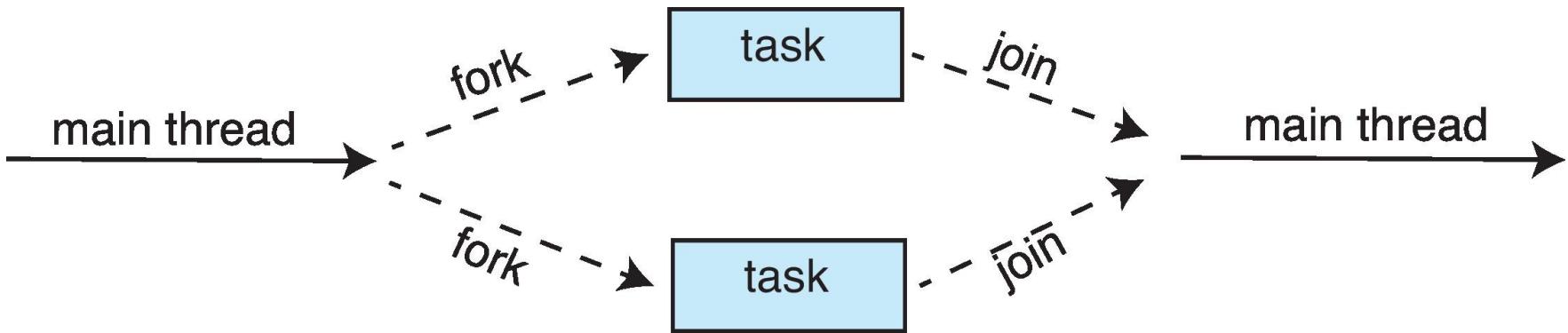
        /* Shut down the pool once all threads have completed */
        pool.shutdown();
    }
}
```



5. Implicit Threading

Fork-Join Parallelism

- 여러 스레드(작업)가 분기 forked 된 다음 결합 joined





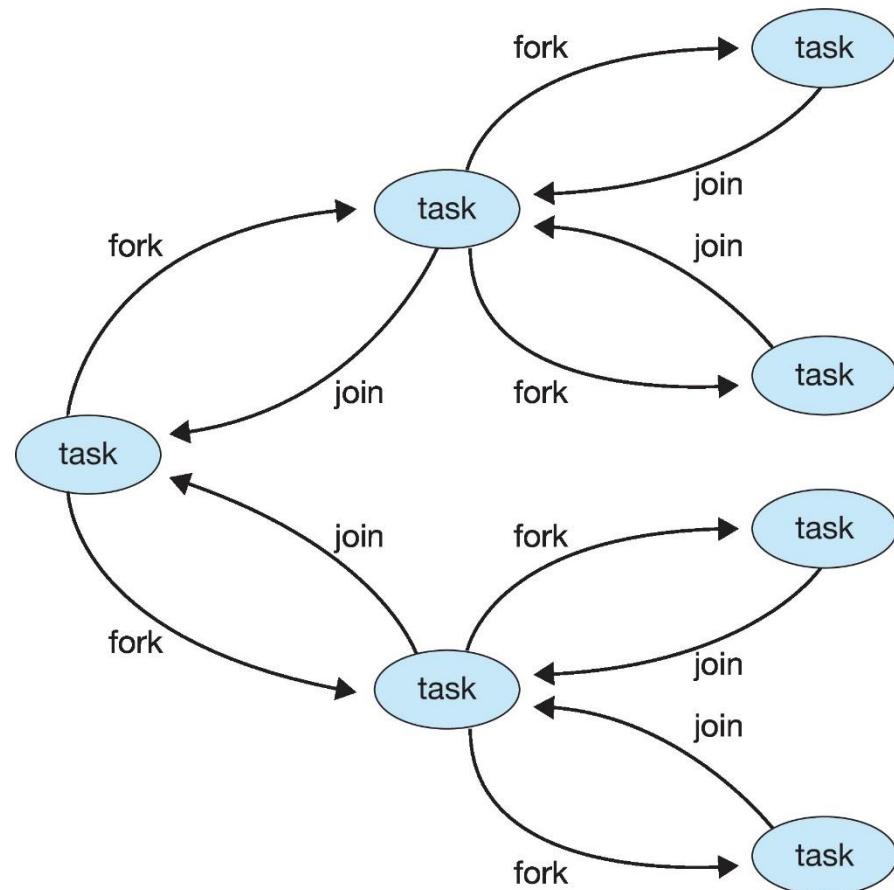
5. Implicit Threading

Fork-Join Parallelism

- fork-join 전략을 위한 일반 알고리즘:

Task(problem)

```
if problem is small enough  
    solve the problem directly  
else  
    subtask1 = fork(new Task(subset of problem))  
    subtask2 = fork(new Task(subset of problem))  
  
    result1 = join(subtask1)  
    result2 = join(subtask2)  
  
return combined results
```





5. Implicit Threading

Fork-Join Parallelism

● Fork-Join Parallelism in Java

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
    static final int THRESHOLD = 1000;

    private int begin;
    private int end;
    private int[] array;

    public SumTask(int begin, int end, int[] array) {
        this.begin = begin;
        this.end = end;
        this.array = array;
    }

    protected Integer compute() {
        if (end - begin < THRESHOLD) {
            int sum = 0;
            for (int i = begin; i <= end; i++)
                sum += array[i];

            return sum;
        }
        else {
            int mid = (begin + end) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

            return rightTask.join() + leftTask.join();
        }
    }
}
```

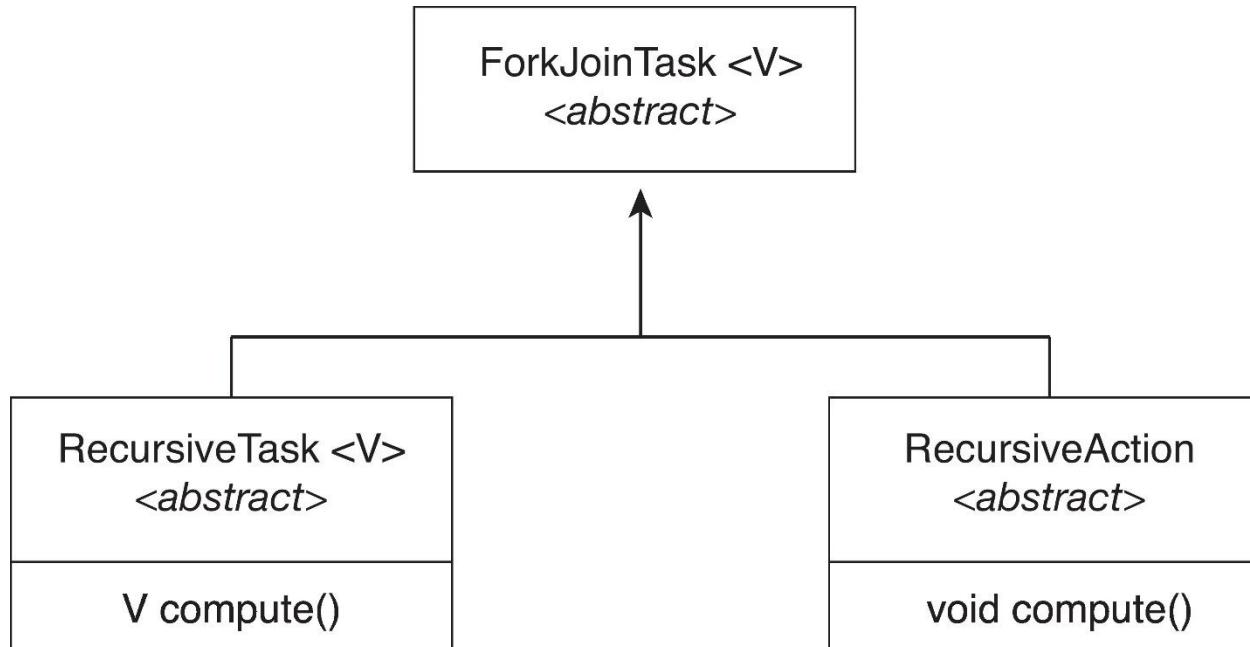


5. Implicit Threading

Fork-Join Parallelism

● Fork-Join Parallelism in Java

- ForkJoinTask는 추상 기본 클래스.
- RecursiveTask 및 RecursiveAction 클래스는 ForkJoinTask를 확장.
- RecursiveTask는 결과를 반환(compute() 메서드의 반환 값을 통해)
- RecursiveAction이 결과를 반환하지 않음





5. Implicit Threading

OpenMP

- C, C++, FORTRAN용 컴파일러 지시문 및 API 세트
- **공유 메모리 환경에서 병렬 프로그래밍 지원 제공**
- **병렬 영역 parallel regions** 식별 - 병렬로 실행할 수 있는 코드 블록

#pragma omp parallel

- 코어 수만큼 스레드 생성

파일

소스코드

실습환경

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > omp_1.c
#include <stdio.h>
#include <omp.h>
```

```
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("There is a parallel region.\n");
    }

    return 0;
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc omp_1.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
There is a parallel region.Hello, My Interruped Child!
I'm Interruped
```

비고



5. Implicit Threading

OpenMP

- 코어 수만큼 스레드 생성

파일

소스코드

실습환경

소스코드

결과값 1

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -fopenmp omp_1.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
There is a parallel region.
There is a parallel region.
```

비고

- **gcc -fopenmp omp_1.c**
- 옵션을 주면 병렬로 여러 번 실행되는 것을 볼 수 있음



5. Implicit Threading

OpenMP

- **for 루프를 병렬로 실행**

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



5. Implicit Threading

OpenMP

- for 루프를 병렬로 실행

파일

실습환경

소스코드

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > omp_2.c
#include <stdio.h>
#include <time.h>
#include <omp.h>

#define SIZE 100000000

int a[SIZE], b[SIZE], c[SIZE];

int main(int argc, char *argv[])
{
    int i;
    double start, end;

    //for 루프 시작 시간
    start = (double)clock() / CLOCKS_PER_SEC;

    for(i=0; i < SIZE; i++){
        a[i]=b[i]=i;
    }

    //for 루프 끝난 시간
    end = (((double)clock()) / CLOCKS_PER_SEC);
    printf("#pragma omp parallel 안 썼을 때 수행 시간 :%lf\n", (end-start));
```

소스코드

결과값1

비고



5. Implicit Threading

OpenMP

- **for** 루프를 병렬로 실행

파일

소스코드

실행환경

```
//for 루프 시작 시간
start = (double)clock() / CLOCKS_PER_SEC;

#pragma omp parallel
for(i=0; i < SIZE; i++){
    c[i]=a[i]+b[i];
}

//for 루프 끝난 시간
end = (((double)clock()) / CLOCKS_PER_SEC);
printf("#pragma omp parallel 썼을 때 수행 시간 :%lf\n", (end-start));

return 0;
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc -fopenmp omp_2.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
#pragma omp parallel 안 썼을 때 수행 시간 :0.000000
#pragma omp parallel 썼을 때 수행 시간 :2.000000
```

결과값1

비고

- **#pragma omp parallel** 안 썼을 때 수행 시간 :0.000000
- **#pragma omp parallel** 썼을 때 수행 시간 :2.000000 커널 모드에서 거의 다 처리해서 시간이 많이 걸리는 것처럼 보이지만, 사용자 모드에서 처리된 시간과 함께 처리되기 때문에 실제 시간은 더 작아지게 됨



5. Implicit Threading

Grand Central Dispatch

- macOS 및 iOS 운영 체제용 Apple 기술
- C, C++ 및 Objective-C 언어, API 및 런타임 라이브러리에 대한 확장
- 병렬 섹션 식별 허용
- 스레딩의 대부분의 세부 사항을 관리.
- 블록은 “^{ }”에 있다.

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

- 디스패치 큐에 배치된 블록
- 대기열에서 제거될 때 스레드 풀의 사용 가능한 스레드에 할당됨



5. Implicit Threading

Grand Central Dispatch

- 두 가지 유형의 디스패치 대기열:
- 직렬 **serial** – FIFO 순서로 제거된 블록, 대기열은 메인 대기열이라고 하는 프로세스당입니다.
- 프로그래머는 프로그램 내에서 추가 직렬 대기열을 생성할 수 있습니다.
- 동시 **concurrent** – FIFO 순서로 제거되지만 한 번에 여러 개가 제거될 수 있음
- 서비스 품질로 나눈 4개의 시스템 전체 대기열:
 - **QOS_CLASS_USER_INTERACTIVE**
 - **QOS_CLASS_USER_INITIATED**
 - **QOS_CLASS_USER.Utility**
 - **QOS_CLASS_USER_BACKGROUND**



5. Implicit Threading

Grand Central Dispatch

- Swift 언어의 경우 태스크는 클로저로 정의됩니다. 캐럿을 뺀 블록과 유사.
- 클로저는 `dispatch_async()` 함수를 사용하여 큐에 제출.

```
let queue = dispatch_get_global_queue  
           (QOS_CLASS_USER_INITIATED, 0)
```

```
dispatch_async(queue, { print("I am a closure.") })
```



5. Implicit Threading

Intel Threading Building Blocks (TBB)

- 병렬 C++ 프로그램 설계를 위한 템플릿 라이브러리

```
for (int i = 0; i < n; i++) {  
    apply(v[i]);  
}
```

- 간단한 for 루프의 직렬 버전

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```



6. Threading Issues

Threading Issues

- fork() 및 exec() 시스템 호출의 의미
- 신호 처리
- 동기 및 비동기
- 대상 스레드의 스레드 취소
- 비동기 Asynchronous 또는 자연 deferred
- 스레드 로컬 저장소
- 스케줄러 활성화

Semantics of fork() and exec()

- fork()는 호출 스레드만 복제할까, 아니면 모든 스레드를 복제할까?
- 일부 UNIX에는 두 가지 버전의 fork가 있다.
- exec()는 일반적으로 정상적으로 작동. 모든 스레드를 포함하여 실행 중인 프로세스를 교체.



6. Threading Issues

Signal Handling

- 신호 **Signals** 는 특정 이벤트가 발생했음을 프로세스에 알리기 위해 UNIX 시스템에서 사용다.
- 신호 처리기 **A signal handler** 는 신호를 처리하는 데 사용.
 - 신호는 특정 이벤트에 의해 생성.
 - 신호가 프로세스에 전달됨
 - 신호는 두 개의 신호 처리기 중 하나에 의해 처리.
- ① 기본 **default**
- ② 사용자 정의 **user-defined**
- 모든 신호에는 신호를 처리할 때 커널이 실행하는 기본 처리기 **default handler** 가 있다.
- 사용자 정의 신호 처리기 **User-defined signal handler** 는 기본 값을 재정의할 수 있다.
- 단일 스레드 **single-threaded** 의 경우 프로세스에 전달된 신호



6. Threading Issues

Signal Handling

- 멀티 스레드를 위한 신호는 어디에 전달되어야 할까?
 1. 시그널이 적용되는 쓰레드에 시그널 전달
 2. 프로세스의 모든 스레드에 신호 전달
 3. 프로세스의 특정 스레드에 신호 전달
 4. 프로세스에 대한 모든 신호를 수신할 특정 스레드 지정



6. Threading Issues

Thread Cancellation

- 완료되기 전에 스레드 종료
- 취소할 스레드는 대상 스레드 **target thread**.
- 두 가지 일반적인 접근법:
 - 비동기식 취소 **Asynchronous cancellation** 는 대상 스레드를 즉시 종료.
 - 지연된 취소 **Deferred cancellation** 를 사용하면 대상 스레드가 취소해야 하는지 주기적으로 확인할 수 있다.
- 스레드를 만들고 취소하는 Pthread 코드:

```
pthread_t tid;

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

. . .

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

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```



6. Threading Issues

Thread Cancellation

- 스레드 취소를 호출하면 취소가 요청되지만 실제 취소는 스레드 상태에 따라 다르다.

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

- 스레드가 취소를 비활성화한 경우 스레드가 활성화할 때까지 취소가 보류 상태로 유지.
- 기본 유형은 지연됨
 - 스레드가 취소 지점에 도달한 경우에만 취소가 발생.
 - ✓ 즉, `pthread_testcancel()`
 - ✓ 그런 다음 정리 핸들러가 호출.
- Linux 시스템에서 스레드 취소는 신호를 통해 처리.



6. Threading Issues

Thread Cancellation in Java

- 자연 취소는 스레드의 중단 상태를 설정하는 `interrupt()` 메서드를 사용.

```
Thread worker;  
.  
.*  
/* set the interruption status of the thread */  
worker.interrupt()
```

- 그런 다음 스레드는 중단되었는지 확인할 수 있다.

```
while (!Thread.currentThread().isInterrupted()) {  
    .  
}  
}
```



6. Threading Issues

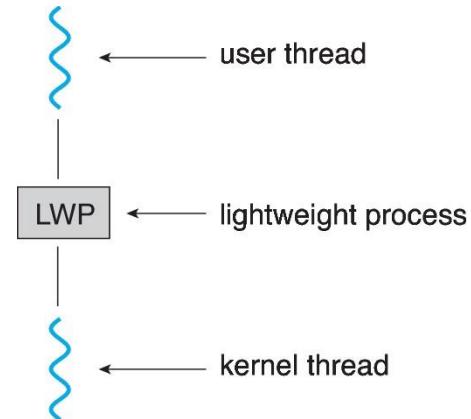
Thread-Local Storage

- 스레드 로컬 스토리지 Thread-local storage (TLS)를 통해 각 스레드는 자체 데이터 사본을 가질 수 있다.
- 스레드 생성 프로세스를 제어할 수 없을 때(즉, 스레드 풀을 사용하는 경우) 유용.
- 지역 변수와 다름
 - 단일 함수 호출 중에만 보이는 지역 변수
 - 함수 호출에서 TLS 표시
- 정적 데이터와 유사
- TLS는 각 스레드에 고유하다.

6. Threading Issues

스케줄러 활성화

- M:M 및 2단계 모델 모두 애플리케이션에 할당된 적절한 수의 커널 스레드를 유지하기 위해 통신이 필요.
- 일반적으로 사용자와 커널 스레드 사이의 중간 데이터 구조 사용 - 경량 프로세스(lightweight process LWP)
 - 프로세스가 사용자 스레드를 실행하도록 예약할 수 있는 가상 프로세서로 나타난다.
 - 커널 스레드에 연결된 각 LWP
 - 생성할 LWP는 몇 개일까?



- 스케줄러 활성화는 업콜 upcalls - 을 제공. 커널에서 스레드 라이브러리의 업콜 핸들러 upcall handler 로의 통신 메커니즘.
- 이 통신을 통해 애플리케이션은 올바른 수의 커널 스레드를 유지할 수 있다.



7. Operating System Examples

Windows Threads

- Windows API – Windows 애플리케이션용 기본 API
- 일대일 매팅, 커널 수준 구현
- 각 스레드에는
 - 스레드 ID
 - 프로세서 상태를 나타내는 레지스터 세트
 - 스레드가 사용자 모드 또는 커널 모드에서 실행될 때 사용자 및 커널 스택 분리
 - 런타임 라이브러리 및 DLL(동적 연결 라이브러리)에서 사용하는 개인 데 이터 저장 영역
- 레지스터 세트, 스택 및 개인 저장 영역은 스레드의 컨텍스트로 알려져 있다.



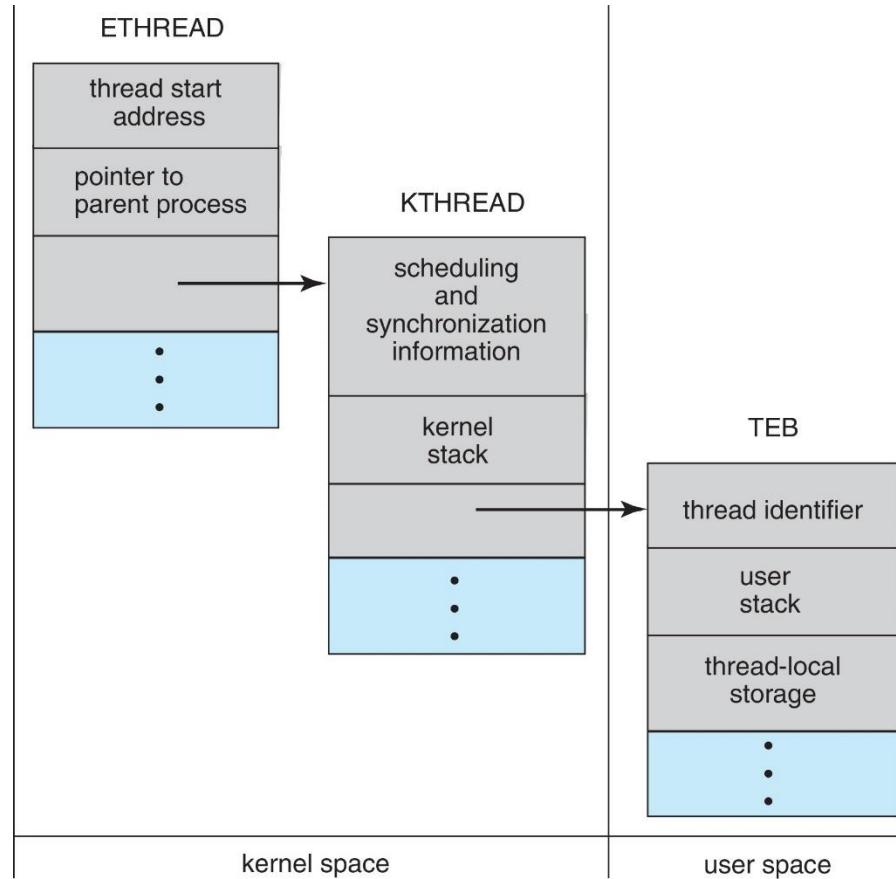
7. Operating System Examples

Windows Threads

- 스레드의 기본 데이터 구조는 다음과 같다.
 - ETHREAD(실행 스레드 블록) - 커널 공간에서 스레드가 속한 프로세스 및 KTHREAD에 대한 포인터를 포함.
 - KTHREAD(커널 스레드 블록) – 스케줄링 및 동기화 정보, 커널 모드 스택, TEB에 대한 포인터, 커널 공간
 - TEB(스레드 환경 블록) – 스레드 ID, 사용자 모드 스택, 스레드 로컬 저장소, 사용자 공간

7. Operating System Examples

Windows Threads Data Structures





7. Operating System Examples

Linux Threads

- Linux는 스레드 tasks 가 아닌 작업 threads 으로 참조.
- 스레드 생성은 clone() 시스템 호출을 통해 수행.
- clone()을 사용하면 자식 작업이 부모 작업(프로세스)의 주소 공간을 공유할 수 있다.
- 플래그 제어 동작

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- struct task_struct는 처리 데이터 구조(공유 또는 고유)를 가리킨다.

『5과목』

4-6교시 :

CPU Scheduling





학습목표

- 이 워크샵에서는 다양한 CPU 스케줄링 알고리즘 설명할 수 있다.
- 스케줄링 기준에 따라 CPU 스케줄링 알고리즘 평가할 수 있다.
- 다중 프로세서 및 다중 코어 스케줄링과 관련된 문제 설명할 수 있다.
- 다양한 실시간 스케줄링 알고리즘 설명할 수 있다.
- Windows, Linux 및 Solaris 운영 체제에서 사용되는 스케줄링 알고리즘 설명할 수 있다.
- 모델링 및 시뮬레이션을 적용하여 CPU 스케줄링 알고리즘 평가할 수 있다.



Syllabus

눈높이 체크

- 기본 개념
- 스케줄링 기준을 알고 계신가요?
- 스케줄링 알고리즘을 알고 계신가요?
- 스레드 스케줄링을 알고 계신가요?
- 다중 프로세서 스케줄링을 알고 계신가요?
- 실시간 CPU 스케줄링을 알고 계신가요?
- 운영 체제 예
- 알고리즘 평가



1. Basic Concepts

기본 개념

- 다중 프로그래밍으로 얻은 최대 CPU 사용률
- CPU-I/O 버스트 주기 CPU-I/O Burst Cycle – 프로세스 실행은 CPU 실행 주기와 I/O 대기 주기로 구성.
- CPU 버스트 이후 I/O 버스트
- CPU 버스트 분포가 주요 관심사.



1. Basic Concepts

목적

1. **공평성** : 모든 프로세스가 자원을 공평하게 배정받아야 하며 자원 배정 과정에서 프로세스가 배제되어서는 안된다.
2. **효율성** : 시스템 자원이 유휴 시간 없이 사용되도록 스케줄링을 하고, 유휴 자원을 사용하려는 프로세스에는 우선권을 주어야 한다.
3. **안정성** : 우선순위를 사용하여 중요 프로세스가 먼저 작동하도록 배정함으로써 시스템 자원을 점유하거나 파괴하려는 프로세스로부터 자원을 보호해야 한다.
4. **확장성** : 프로세스가 증가해도 시스템이 안정적으로 작동하도록 조치해야 한다. 또한 시스템 자원이 늘어나는 경우 이 혜택이 시스템에 반영되게 해야한다.
5. **반응 시간 보장** : 응답이 없는 경우 사용자는 시스템이 멈춘 것으로 가정하기 때문에 시스템은 적절한 시간 안에 프로세스의 요구에 반응해야 한다.
6. **무한 연기 방지** : 특정 프로세스의 작업 무한히 연기되어서는 안된다.

1. Basic Concepts

Histogram of CPU-burst Times

- 다수의 짧은 버스트
- 적은 수의 더 긴 버스트

⋮

load store
add store
read from file

wait for I/O

store increment
index
write to file

wait for I/O

load store
add store
read from file

wait for I/O

⋮

load store
add store
read from file

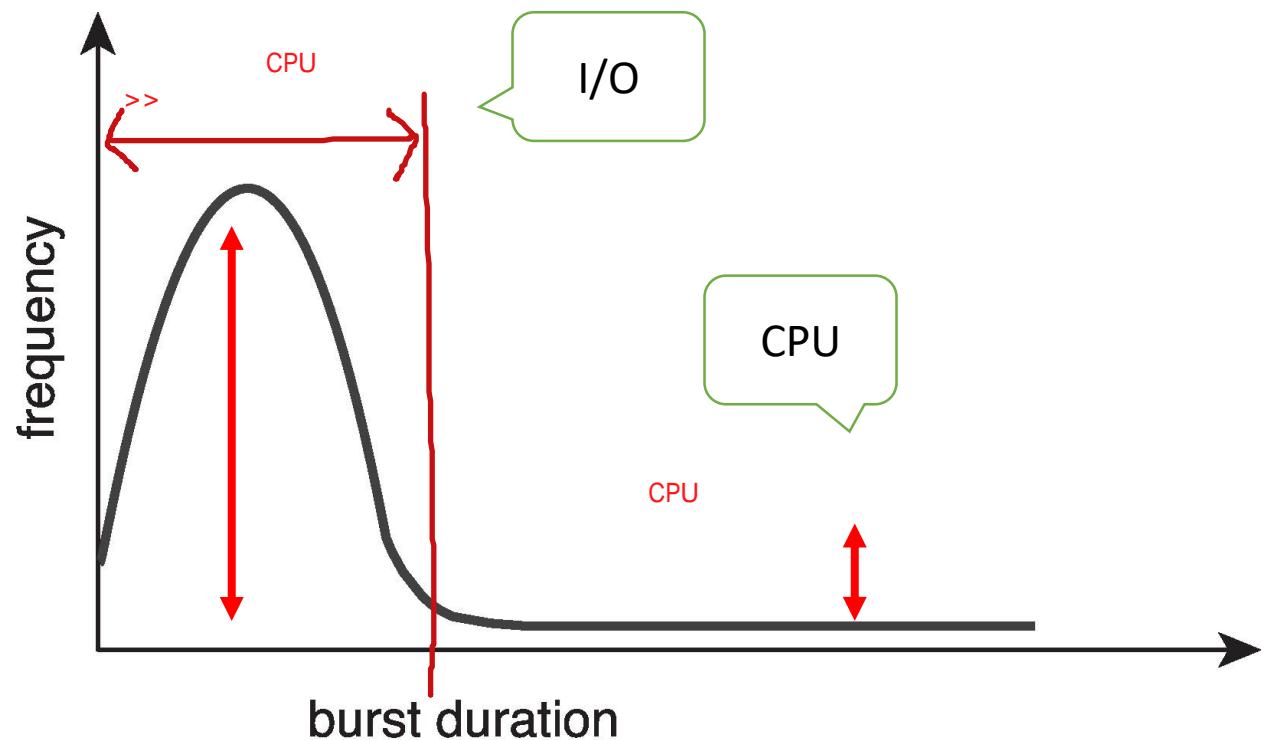
wait for I/O

store increment
index
write to file

wait for I/O

load store
add store
read from file

wait for I/O





1. Basic Concepts

CPU Scheduler

- CPU 스케줄러는 Ready Queue의 프로세스 중에서 선택하고 그 중 하나에 CPU 코어를 할당.
 - 대기열은 다양한 방법으로 주문할 수 있다.
- CPU 스케줄링 결정은 프로세스가 다음과 같은 경우에 발생할 수 있다.
 1. 실행 running 상태에서 대기 상태 waiting state로 전환
 2. 실행 running 상태에서 준비 상태 ready state로 전환
 3. 대기 waiting에서 준비 ready로 전환
 4. 종료 Terminates
- 상황 1과 4의 경우 일정 측면에서 선택의 여지가 없다. 실행을 위해 새 프로세스(준비 대기열에 있는 경우)를 선택해야 한다.
- 그러나 상황 2와 3의 경우 선택 사항이 있다.



1. Basic Concepts

선점형 및 비선점형 스케줄링



1. Basic Concepts

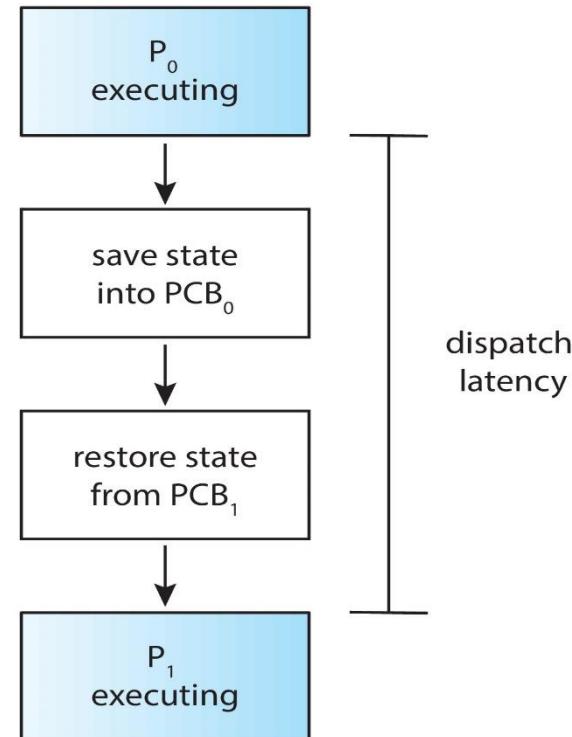
선제적 스케줄링 및 경쟁 조건

- 선점형 스케줄링은 데이터가 여러 프로세스 간에 공유될 때 경쟁 조건을 초래할 수 있다.
- 데이터를 공유하는 두 프로세스의 경우를 고려.
 - ① 한 프로세스가 데이터를 업데이트하는 동안 두 번째 프로세스가 실행될 수 있도록 선점.
 - ② 그런 다음 두 번째 프로세스는 일관성 없는 상태에 있는 데이터를 읽으려고 시도.

1. Basic Concepts

Dispatcher

- Dispatcher 모듈은 CPU 스케줄러가 선택한 프로세스에 CPU 제어권을 부여. 여기에는 다음이 포함.
 - 컨텍스트 전환 **Switching context**
 - 사용자 모드로 전환
 - 해당 프로그램을 다시 시작하기 위해 사용자 프로그램의 적절한 위치로 이동
- 디스패치 대기 시간 **Dispatch latency** – 디스패처가 한 프로세스를 중지하고 다른 프로세스를 시작하는 데 걸리는 시간



1. Basic Concepts

Dispatcher

```
k8s@DESKTOP-RoEQ2U6:~$ ls /proc/
```

```
1 14 bus cmdline filesystems loadavg mounts self sys uptime version_signature  
13 786 cgroups cpufreq interrupts meminfo net stat tty version
```

```
k8s@DESKTOP-RoEQ2U6:~$
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat /proc/1/status | grep ctxt  
voluntary_ctxt_switches: 150  
nonvoluntary_ctxt_switches: 545
```

자발적_ctxt_스위치 >
Nonpreemptive

! : 가
>> OS가

비자발적_ctxt_스위치 >
Preemptive

1. Basic Concepts

Dispatcher

```
k8s@DESKTOP-RoEQ2U6:~$ cat /proc/1/status
```

```
Name: init
State: S (sleeping)
Tgid: 1
Pid: 1
PPid: 0
TracerPid: 0
Uid: 0 0 0 0
Gid: 0 0 0 0
FDSize: 10
...
CapPrm: ooooooo1fffffffff
CapEff: ooooooo1fffffffff
CapBnd: ooooooo1fffffffff
Cpus_allowed: 3
Cpus_allowed_list: 0-1
Mems_allowed: 1
Mems_allowed_list: 0
```

```
voluntary_ctxt_switches: 150
nonvoluntary_ctxt_switches: 545
k8s@DESKTOP-RoEQ2U6:~$
```

자발적 ctxt 스위치 >
Nonpreemptive

비자발적 ctxt 스위치 >
Preemptive



2. Scheduling Criteria

스케줄링 기준

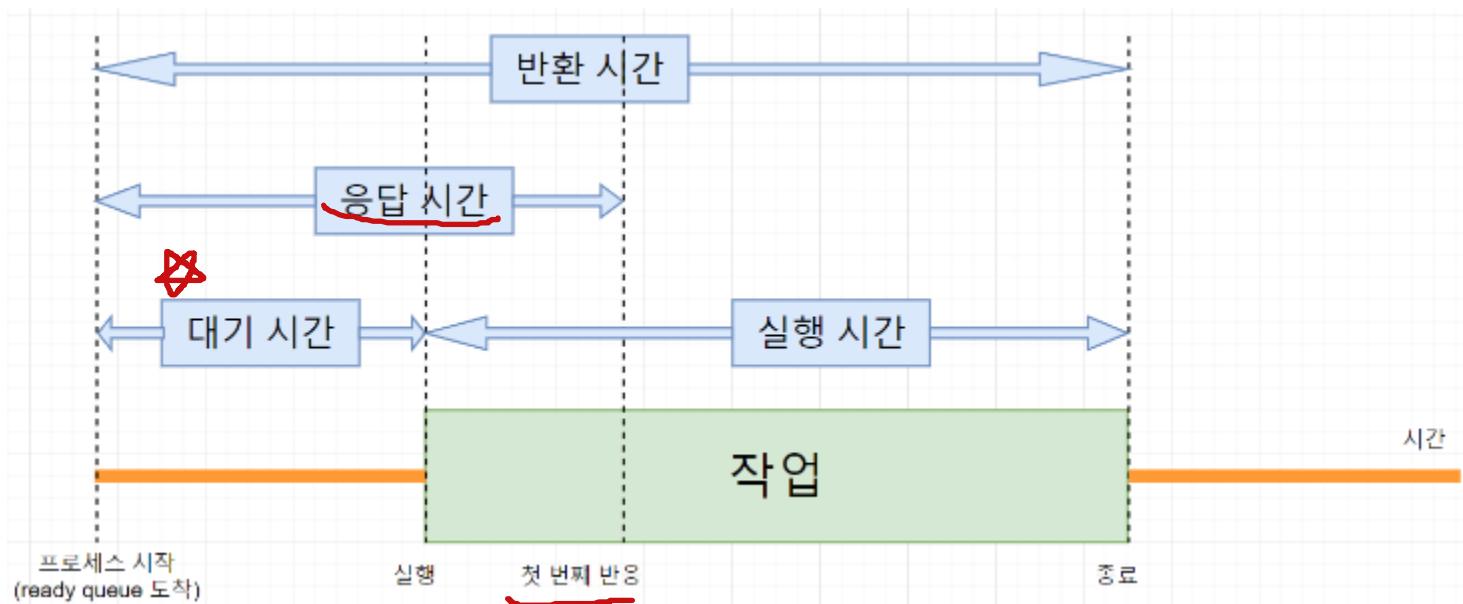
- CPU 사용률 CPU utilization – CPU를 최대한 바쁘게 유지
- 처리량 Throughput – 시간 단위당 실행을 완료하는 프로세스 수
 - 클수록 좋다.
- 처리 시간 Turnaround time – 특정 프로세스를 실행하는 데 걸리는 시간
 - 짧을 수록 좋다.
- 대기 시간 Waiting time – 프로세스가 준비 대기열에서 대기한 시간
 - 짧을 수록 좋다.
- 응답 시간 Response time – 요청이 제출된 시점부터 첫 번째 응답이 생성될 때까지 걸리는 시간입니다.
 - 짧을 수록 좋다.



3. Scheduling Algorithm

스케줄링 알고리즘 최적화 기준

- 최대 CPU 사용률 Max CPU utilization
- 최대 처리량 Max throughput
- 최소 처리 시간 Min turnaround time
- 최소 대기 시간 Min waiting time
- 최소 응답 시간 Min response time



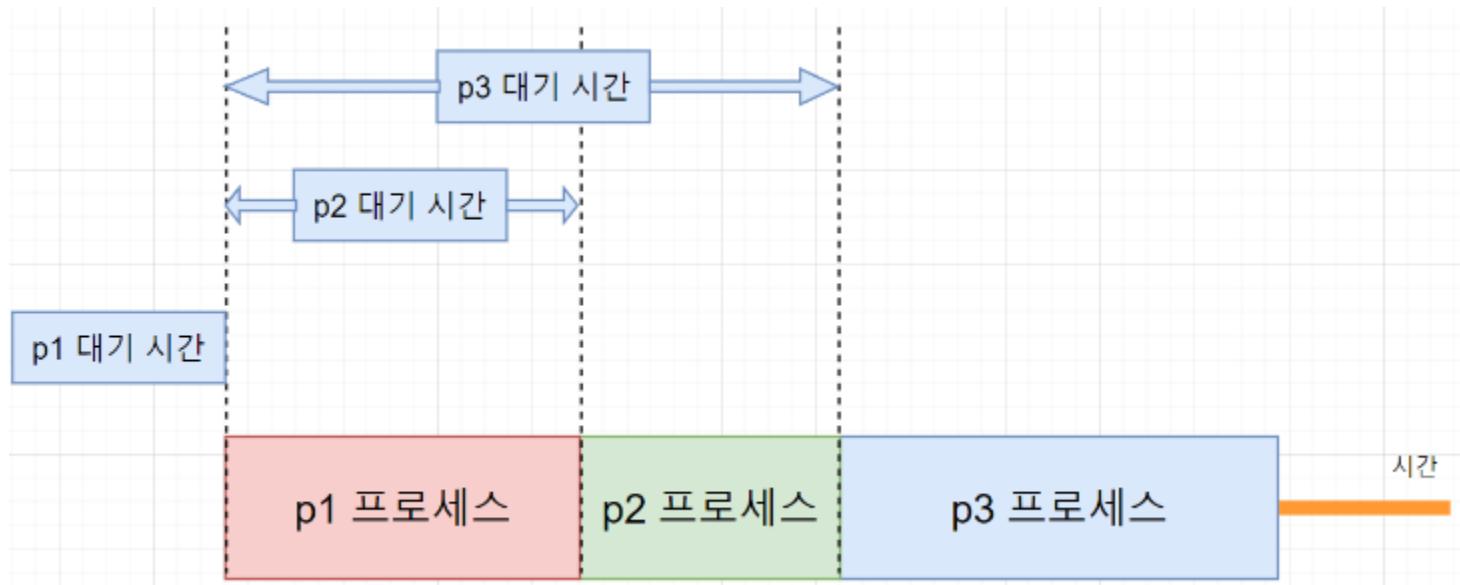


3. Scheduling Algorithm

평균 대기 시간

- 평균 대기 시간은 (p_1 대기시간 + p_2 대기시간 + p_3 대기시간) / 3
- p_1 은 대기 시간없이 바로 실행되므로 대기 시간이 0

>> 0 !





3. Scheduling Algorithm

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

- 프로세스가 P₁, P₂, P₃ 순서로 도착한다고 가정. 일정에 대한 Gantt 차트는 다음과 같다.



- P₁의 대기 시간 = 0; P₂ = 24; P₃ = 27
- 평균 대기 시간: (0 + 24 + 27)/3 = 17



3. Scheduling Algorithm

First-Come, First-Served (FCFS) Scheduling

- 프로세스가 순서대로 도착한다고 가정.

P_2, P_3, P_1

- 일정에 대한 Gantt 차트는 다음과 같다.



- P_1 의 대기 시간 = 6; P_2 = 0; P_3 = 3
- 평균 대기 시간: $(6 + 0 + 3)/3 = 3$
- 이전 사례보다 훨씬 낫다.
- **호송 효과 Convoy effect - 긴 프로세스 뒤의 짧은 프로세스**
- 하나의 CPU 바운드 프로세스와 많은 I/O 바운드 프로세스를 고려. FCFS로는 좋은 효과를 보기 어려움.

도착 순서만 바꿔도 평균 대기 시간이 짧아짐



3. Scheduling Algorithm

Shortest-Job-First (SJF) Scheduling

- 다음 CPU 버스트의 길이를 각 프로세스와 연결
 - 이 길이를 사용하여 가장 짧은 시간으로 프로세스를 예약.
- SJF는 최적. 주어진 프로세스 집합에 대한 최소 평균 대기 시간을 제공.
- **shortest-remaining-time-first**라는 선점형 버전
- 다음 CPU 버스트의 길이를 어떻게 결정할까?
- 사용자에게 물어볼 수 있음
- **추정 Estimate**

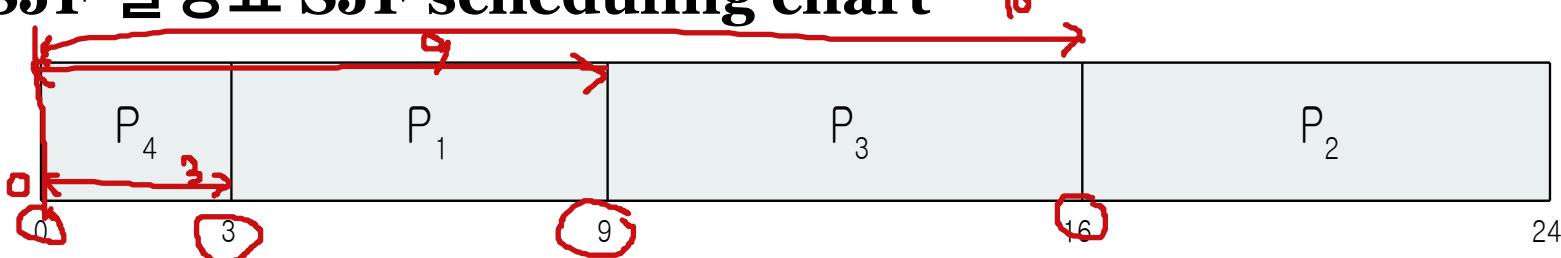


3. Scheduling Algorithm

Example of SJF

프로세스	버스트 시간
P1	6
P2	8
P3	7
P4	3

- SJF 일정표 SJF scheduling chart



24

- 평균 대기 시간 Average waiting time

$$= (3 + 16 + 9 + 0) / 4 = 7$$

P1 P3 P2



3. Scheduling Algorithm

다음 CPU 버스트의 길이 결정

- 길이 추정만 가능 – 이전 길이와 유사해야 함
 - 그런 다음 예측된 다음 CPU 버스트가 가장 짧은 프로세스를 선택.
- 지수 평균을 사용하여 이전 CPU 버스트의 길이를 사용하여 수행할 수 있다.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α , $0 \leq \alpha \leq 1$
4. Define
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

- 일반적으로 가중치 파라미터 α 는 $1/2$ 로 설정

가 :

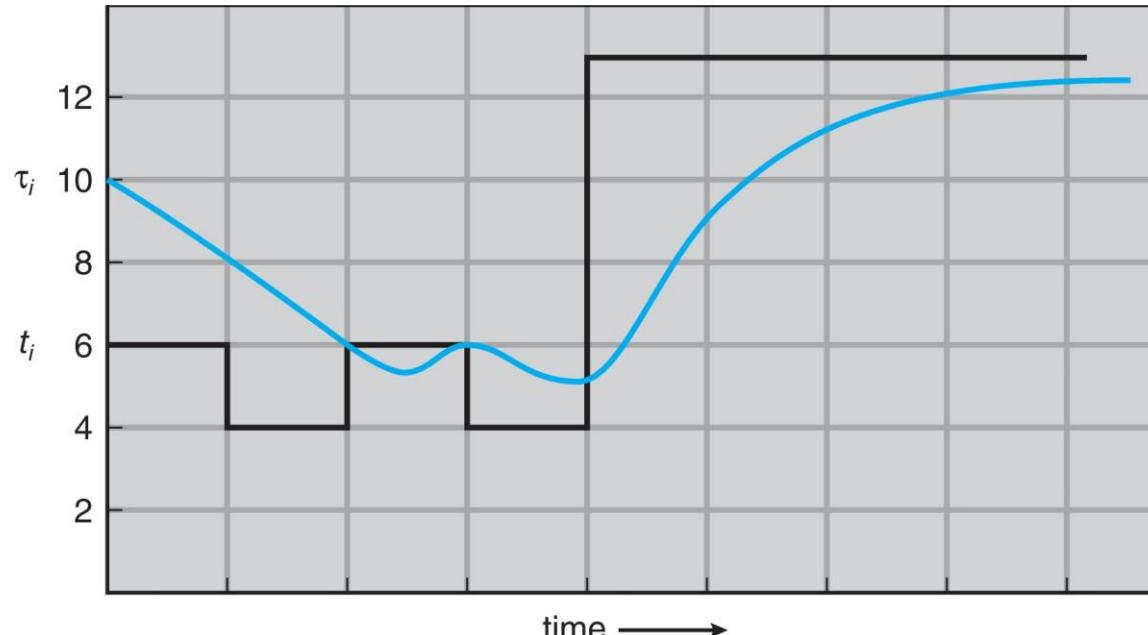
>>

가 0.5



3. Scheduling Algorithm

다음 CPU 버스트의 길이 예측



- 일반적으로 α 는 $1/2$ 로 설정

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$



3. Scheduling Algorithm

지수 평균의 예

- $\alpha = 0$
- $\tau_{n+1} = \tau_n$
- 최근 기록은 포함되지 않는다.
- $\alpha = 1$
- $\tau_{n+1} = \alpha t_n$
- 실제 마지막 CPU 버스트만 카운트
- 공식을 확장하면 다음을 얻는다.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

$$\begin{aligned}\tau_{n+1} &= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ &\quad + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ &\quad + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- α and $(1 - \alpha)$ 이 모두 1보다 작거나 같기 때문에 각 후속 선행 조건은 선행 조건보다 가중치가 적다.



3. Scheduling Algorithm

| 최단 남은 시간 우선 스케줄링

- SJN의 선점형 버전
- 새 프로세스가 준비 대기열에 도착할 때마다 SJN 알고리즘을 사용하여 다음에 일정을 잡을 프로세스에 대한 결정이 다시 수행.
- 주어진 프로세스 집합에 대한 최소 평균 대기 시간 측면에서 SRT가 SJN보다 "최적"일까?

3. Scheduling Algorithm

() >>

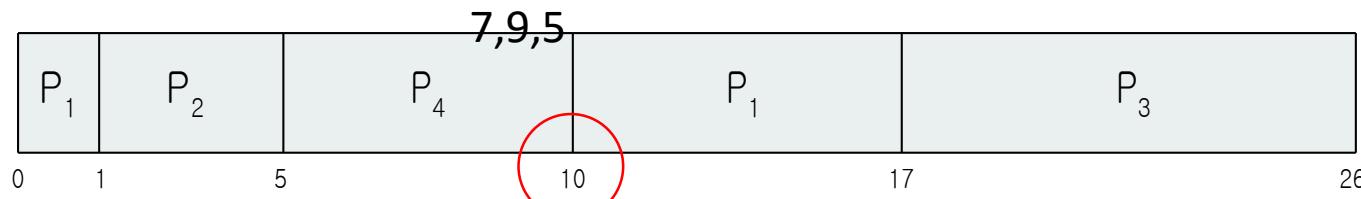
ㅋㅋ

Example of Shortest-remaining-time-first

- 이제 다양한 도착 시간 및 선점의 개념을 분석에 추가.

프로세스	도착 시간	버스트 시간	remaining-time
P1	0	8	7
P2	1	4	
P3	2	9	
P4	3	5	

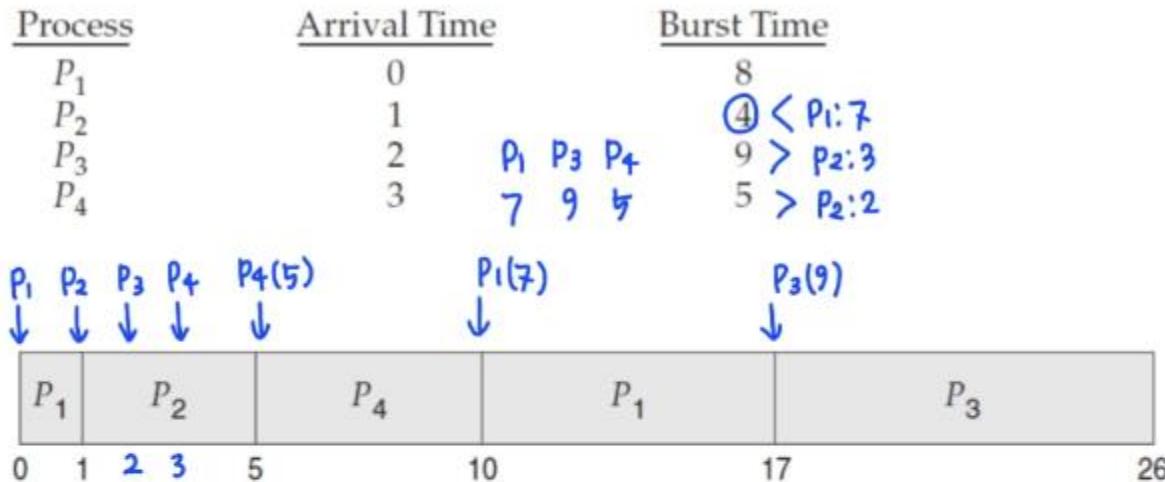
- 선점형 SJF 간트 차트



- 평균 대기 시간 = $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$

3. Scheduling Algorithm

선점형 SJF 스케줄링



$$\text{Waiting time: } P_1 = 10 - 1 = 9, P_2 = 1 - 1 = 0, P_3 = 17 - 2 = 15, P_4 = 5 - 3 = 2$$

$$\text{average waiting time: } (9 + 0 + 15 + 2) / 4 = 6.5$$

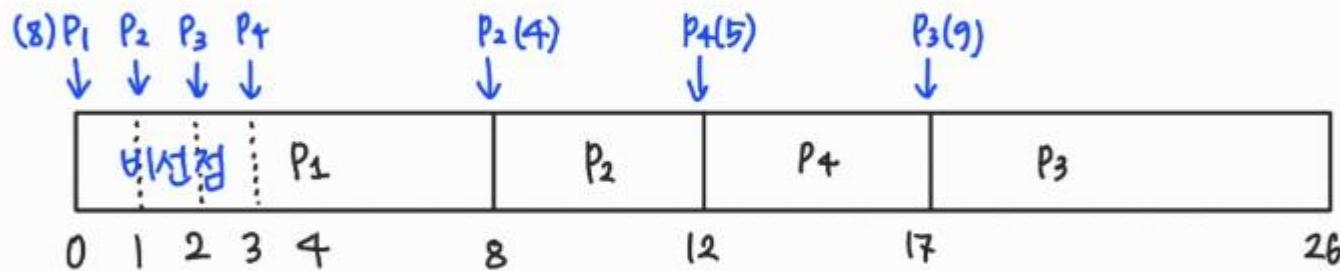
$$\text{Turnaround time: } P_1 = 10 - 0 = 10, P_2 = 5 - 1 = 4, P_3 = 26 - 2 = 24, P_4 = 10 - 3 = 7$$

$$\text{average turnaround time: } (10 + 4 + 24 + 7) / 4 = 11.25$$

3. Scheduling Algorithm

비선점형 SJF 스케줄링

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



$$\text{average waiting time} : \{0 + (8-1) + (17-2) + (12-3)\} / 4 = 7.75$$

$$\text{Average turnaround time} : \{8 + (12-1) + (26-2) + (17-3)\} / 4 = 14.25$$

3. Scheduling Algorithm

Round Robin (RR)

- 각 프로세스는 일반적으로 10-100밀리초인 작은 CPU 시간 단위 (시간 양자 q time quantum q)를 얻는다. 이 시간이 지나면 프로세스가 선점되어 준비 대기열의 끝에 추가.
- 준비 대기열에 n 개의 프로세스가 있고 시간 할당량이 q 인 경우 각 프로세스는 한 번에 최대 q 시간 단위 청크로 CPU 시간의 $1/n$ 을 얻습니다. 어떤 프로세스도 $(n-1)q$ 시간 단위 이상 대기하지 않는다.
- 타이머는 다음 프로세스를 예약하기 위해 모든 양자를 중단한다.
- 성능
 - q large \Rightarrow FIFO (FCFS)
 - q small \Rightarrow RR
- 컨텍스트 스위치와 관련하여 q 가 커야 합니다. 그렇지 않으면 오버 헤드가 너무 높다.

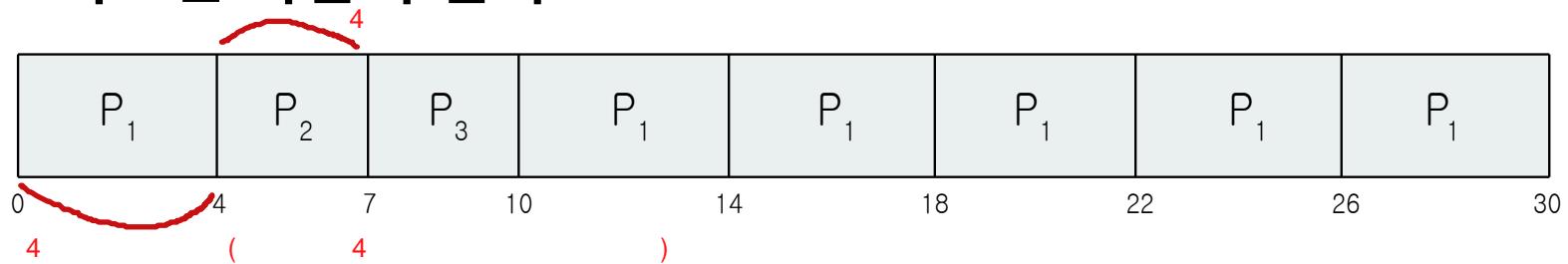


3. Scheduling Algorithm

시간 양자 = 4인 RR의 예

Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

- 간트 차트는 다음과 같다.



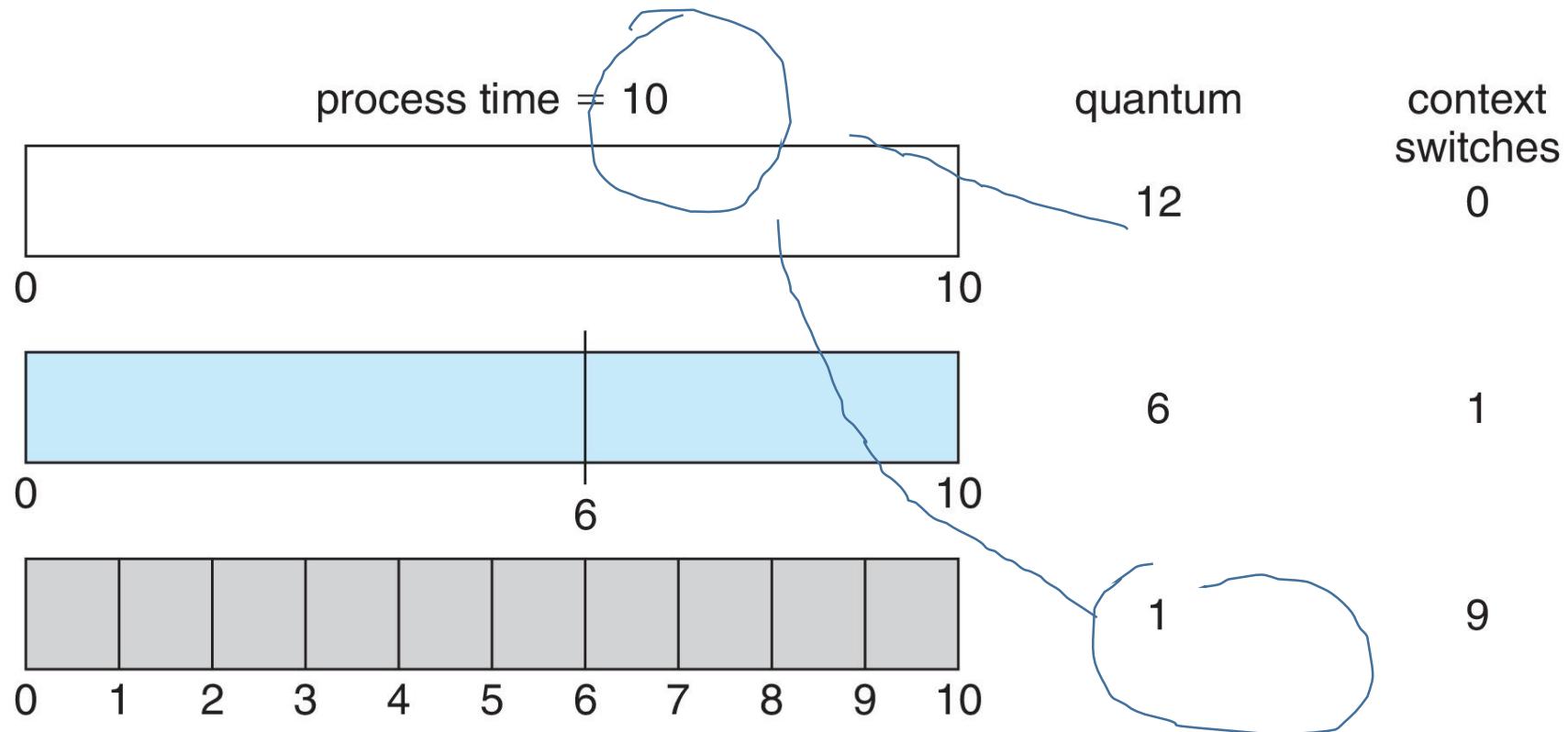
- 일반적으로 SJF보다 평균 턴어라운드가 높지만 응답성이 더 좋다.
- q는 컨텍스트 전환 시간에 비해 커야 함.
 - q 일반적으로 10밀리초에서 100밀리초,
 - 컨텍스트 스위치 < 10마이크로초
 - Waiting time : P₁=10-4=6, P₂=4, P₃=7



3. Scheduling Algorithm

시간 양자 = 4인 RR의 예

- 시간 양자 및 컨텍스트 전환 시간

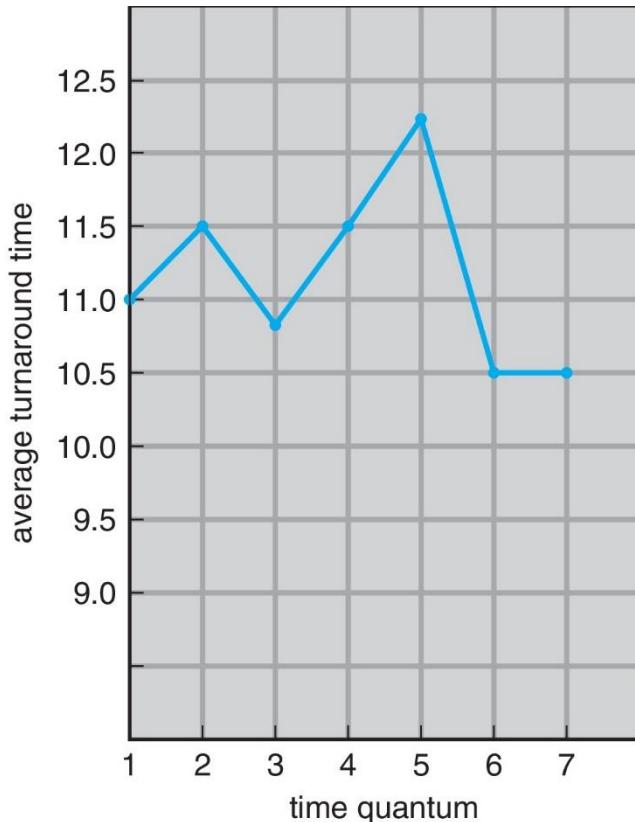




3. Scheduling Algorithm

시간 양자 = 4인 RR의 예

- 처리 시간은 시간 양자에 따라 다르다.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q



3. Scheduling Algorithm

Priority Scheduling

- 우선 순위 번호(정수)는 각 프로세스와 연결.
- 우선순위가 가장 높은 프로세스에게 CPU가 할당됨(가장 작은 정수
≡ 우선순위가 가장 높음)
- 선제적 Preemptive > SRTF
- 비선점형 Nonpreemptive > SJF
- SJF는 우선순위가 예측된 다음 CPU 버스트 시간의 역수인 우선순위 스케줄링.
- 문제 ≡ 기아 Starvation – 우선 순위가 낮은 프로세스는 절대 실행되지 않을 수 있다.
- 솔루션 ≡ 에이징 Aging – 시간이 지남에 따라 프로세스의 우선 순위를 높인다.

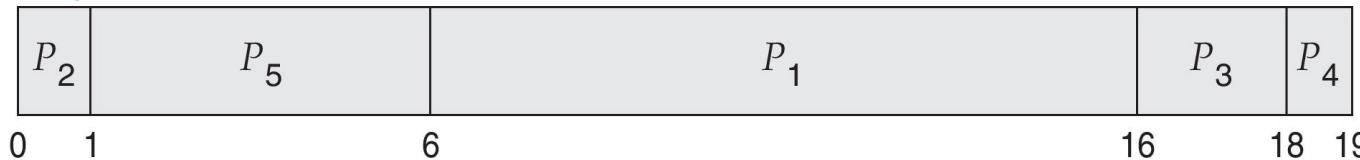


3. Scheduling Algorithm

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- 우선순위 일정 간트 차트



- 평균 대기 시간 = 8.2



3. Scheduling Algorithm

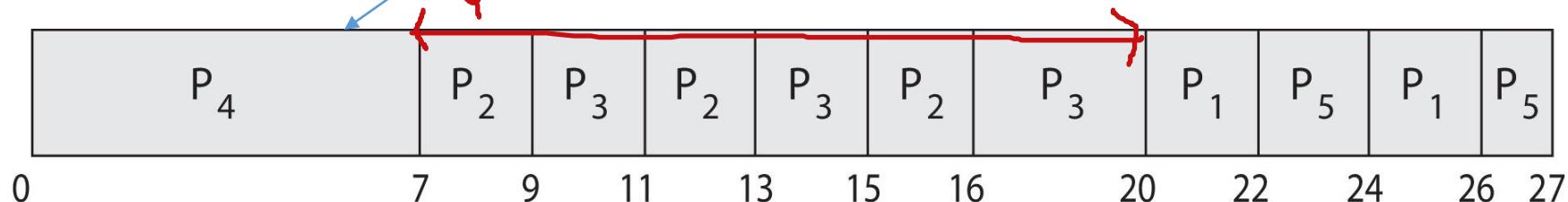
Example of Priority Scheduling

- 우선 순위가 가장 높은 프로세스를 실행. 우선 순위가 같은 프로세스는 라운드 로빈으로 실행. ()

- 예:

Process	Burst Time	Priority
P1	4	3
P2	5	2
P3	8	2
P4	7	1
P5	3	3

- 시간 퀀텀이 있는 간트 차트 = 2





3. Scheduling Algorithm

Multilevel Queue

- 준비 대기열은 여러 개의 대기열로 구성.
- 다음 매개변수로 정의된 다단계 대기열 스케줄러:
- 대기열 수 Number of queues
- 대기열별 스케줄링 알고리즘
- 프로세스가 서비스를 필요로 할 때 프로세스가 입력할 대기열을 결정하는 데 사용되는 방법
- 대기열 간 스케줄링



3. Scheduling Algorithm

Multilevel Queue

- 우선 순위 스케줄링을 사용하면 각 우선 순위에 대해 별도의 대기열이 있다.
- 우선 순위가 가장 높은 대기열에서 프로세스를 예약.

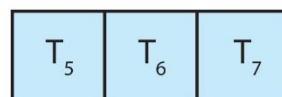
우선순위

priority = 0



각각 Ready
queue

priority = 1



priority = 2

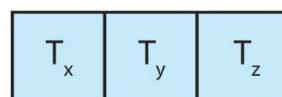


•

•

•

priority = n

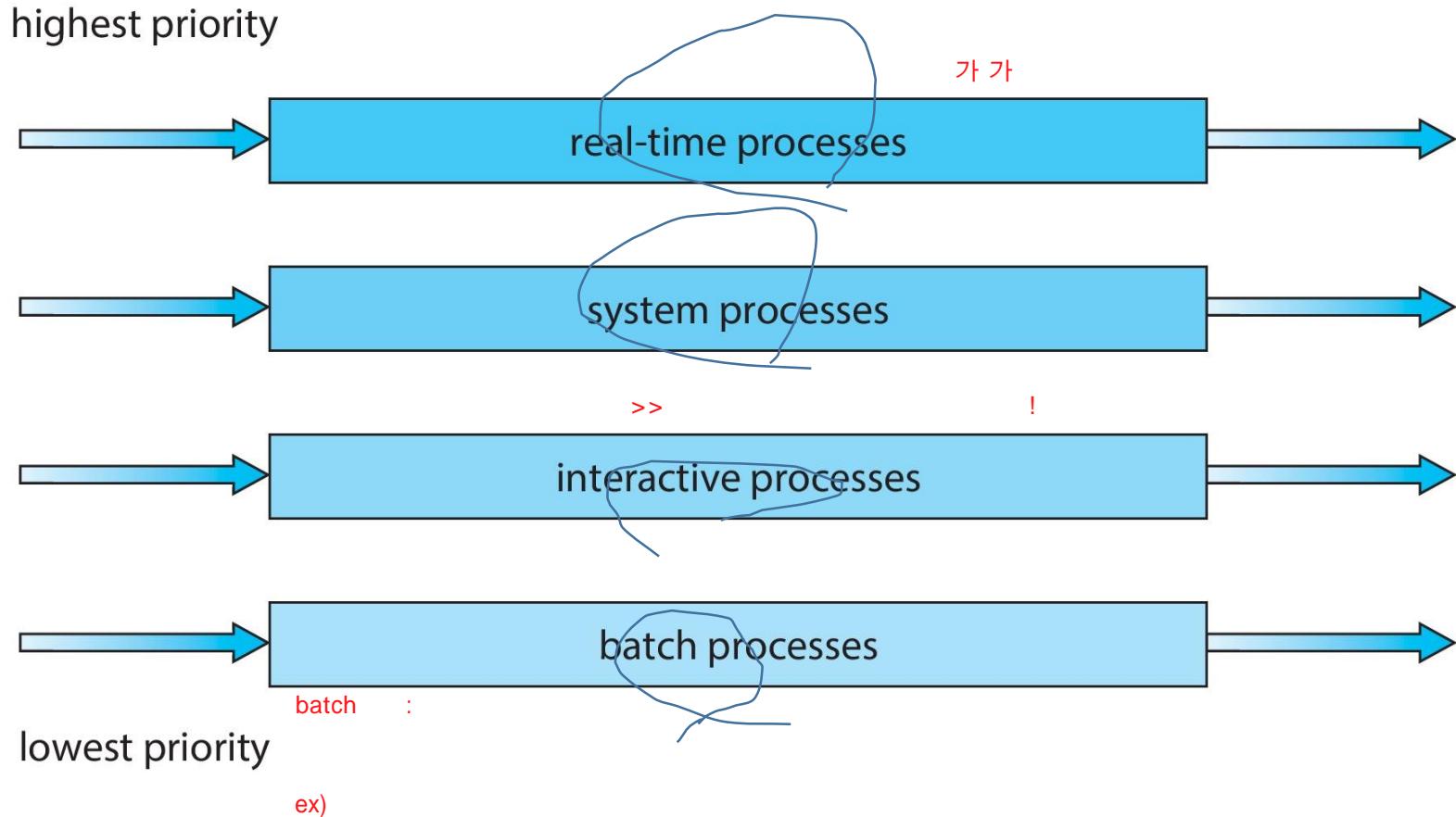




3. Scheduling Algorithm

Multilevel Queue

- 프로세스 유형에 따라 우선 순위 지정





3. Scheduling Algorithm

다단계 피드백 대기열 Multilevel Feedback Queue

- 프로세스는 다양한 대기열 사이를 이동할 수 있다.
- 다단계 피드백 대기열 스케줄러는 다음 매개변수로 정의.
 - 대기열 수 Number of queues
 - 대기열별 스케줄링 알고리즘
 - 프로세스를 업그레이드할 시기를 결정하는 데 사용되는 방법
 - 프로세스 수준을 내릴 시기를 결정하는 데 사용되는 방법
 - 프로세스가 서비스를 필요로 할 때 프로세스가 입력할 대기열을 결정하는 데 사용되는 방법
- 다단계 피드백 큐를 사용하여 에이징을 구현할 수 있다.

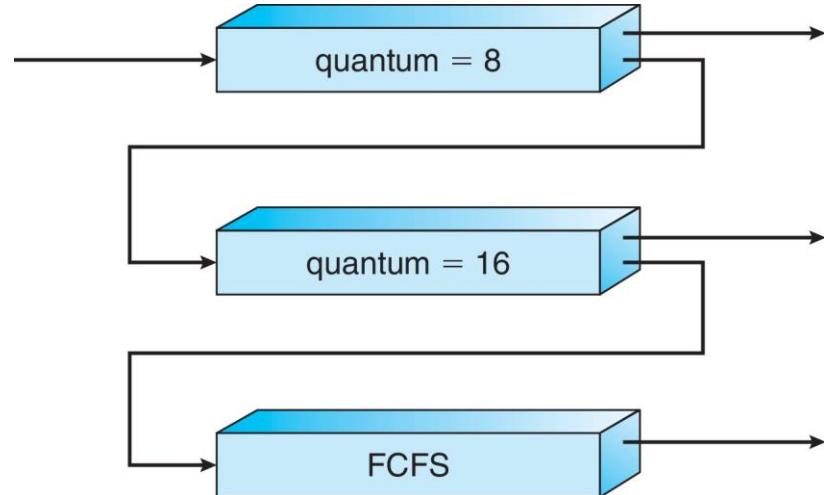


3. Scheduling Algorithm

다단계 피드백 대기열의 예

- 3개의 대기열:

- Q₀ – 시간 양자 8밀리초의 RR
- Q₁ – RR 시간 양자 16밀리초
- Q₂ – FCFS



- 스케줄링

- 새로운 프로세스가 RR에서 제공되는 대기열 Q₀에 들어간다.
 - ✓ CPU를 확보하면 프로세스는 8밀리초를 수신.
 - ✓ 8밀리초 안에 완료되지 않으면 프로세스는 큐 Q₁로 이동.
- Q₁에서 작업이 다시 RR로 제공되고 추가로 16밀리초를 받는다.
 - ✓ 그래도 완료되지 않으면 선점되어 큐 Q₂로 이동.



4. Thread Scheduling

Thread

- 사용자 수준 스레드와 커널 수준 스레드의 구별
 - **user-level threads**
 - **kernel-level threads**
- 스레드가 지원되는 경우 프로세스가 아닌 스레드가 예약됨
- 다대일 및 다대다 모델, 스레드 라이브러리는 LWP에서 실행할 사용자 수준 스레드를 예약.
 - 일정 경쟁이 프로세스 내에 있기 때문에 프로세스 경합 범위(process-contention scope PCS)로 알려져 있다.
 - 일반적으로 프로그래머가 설정한 우선 순위를 통해 수행됨
- 사용 가능한 CPU에 예약된 커널 스레드는 시스템 경합 범위(system-contention scope SCS) - 시스템의 모든 스레드 간의 경쟁.



4. Thread Scheduling

Pthread Scheduling

- API를 사용하면 스레드 생성 중에 PCS 또는 SCS를 지정할 수 있다.
 - PTHREAD_SCOPE_PROCESS는 PCS 스케줄링을 사용하여 스레드를 스케줄.
 - PTHREAD_SCOPE_SYSTEM은 SCS 스케줄링을 사용하여 스레드를 스케줄.
- OS에 의해 제한될 수 있음 – Linux 및 macOS는 PTHREAD_SCOPE_SYSTEM만 허용



4. Thread Scheduling

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

가



5. Multi-Processor Scheduling

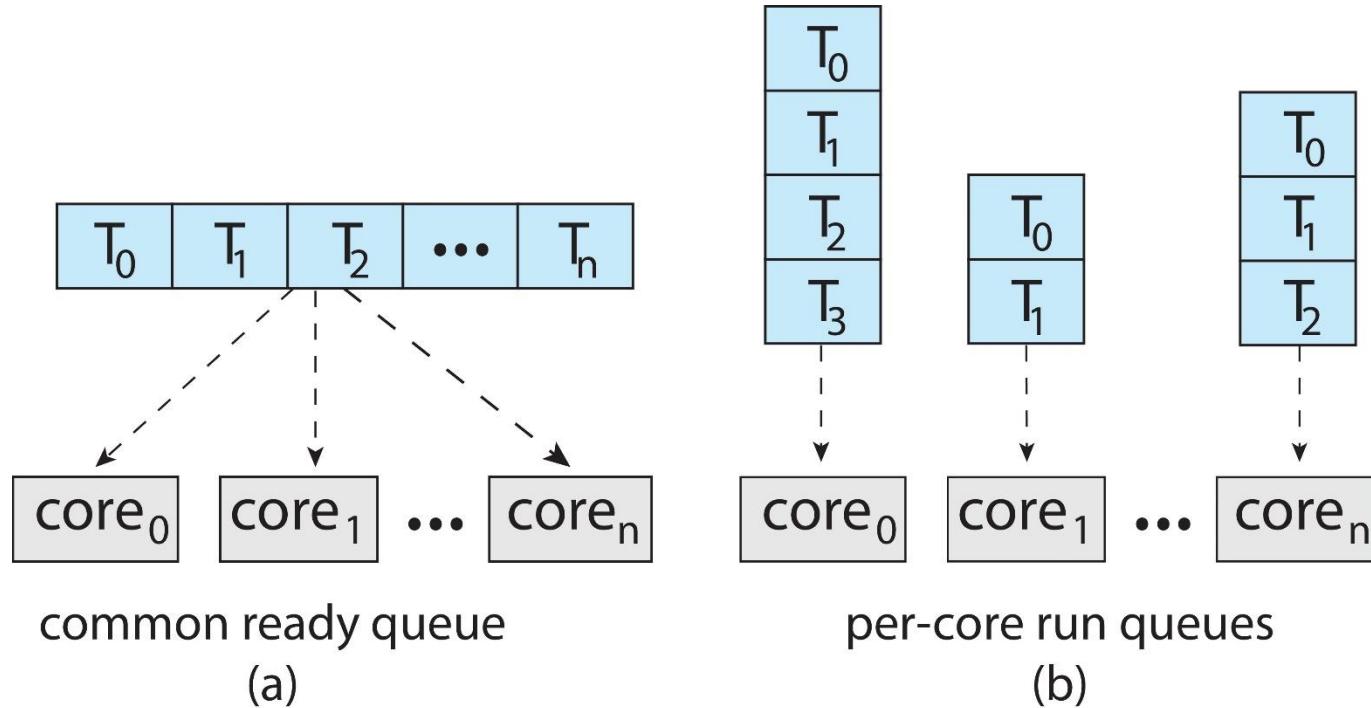
Multiprocess architectures

- 여러 CPU를 사용할 수 있는 경우 CPU 스케줄링이 더 복잡해짐
- 다중 프로세스는 다음 아키텍처 중 하나일 수 있다.
 - 멀티코어 CPU
 - 다중 스레드 코어
 - NUMA 시스템
 - 이기종 다중 처리 Heterogeneous multiprocessing

5. Multi-Processor Scheduling

Multiprocess architectures

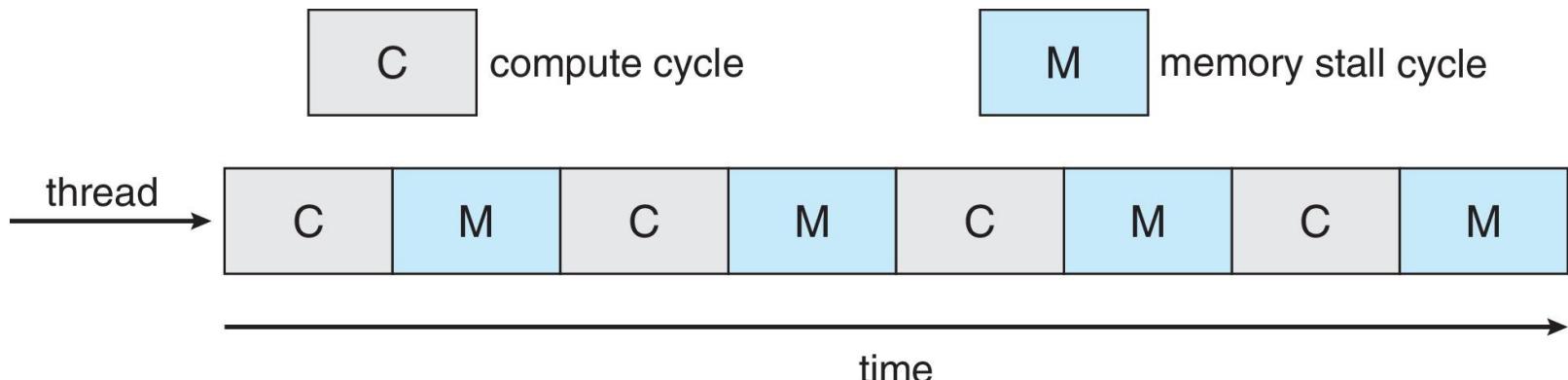
- **SMP(Symmetric multiprocessing)**는 각 프로세서가 자체적으로 스케줄링되는 곳.
 - 모든 스레드는 공통 준비 큐(a)에 있을 수 있다.
 - 각 프로세서는 스레드의 자체 개인 큐를 가질 수 있다(b).



5. Multi-Processor Scheduling

Multiprocess architectures

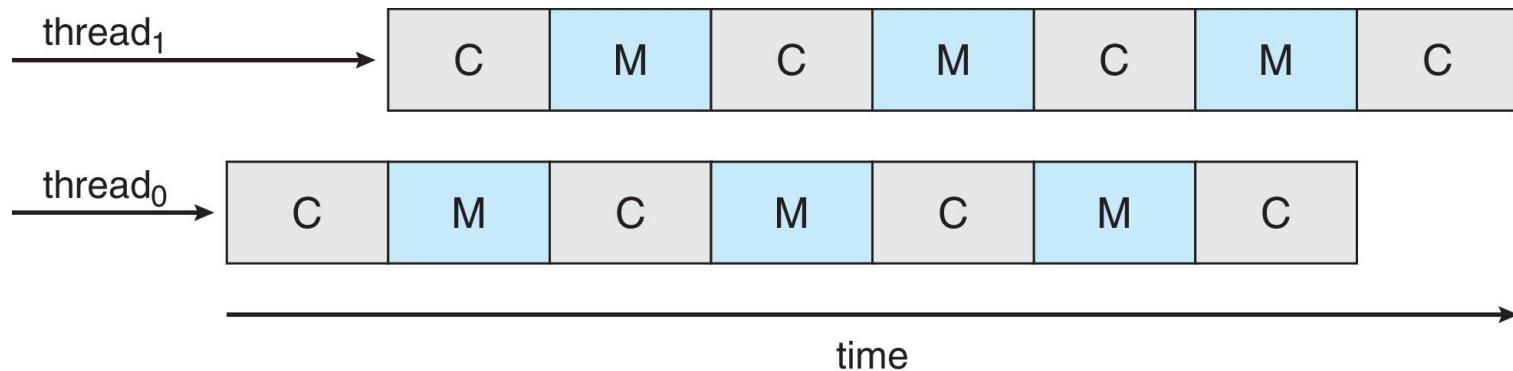
- 동일한 물리적 칩에 여러 프로세서 코어를 배치하는 최근 추세
- 더 빠르고 더 적은 전력 소비
- 코어당 여러 스레드도 증가
- 메모리 검색이 발생하는 동안 다른 스레드에서 진행하기 위해 메모리 정지 를 이용.
- Figure



5. Multi-Processor Scheduling

Multithreaded Multicore System

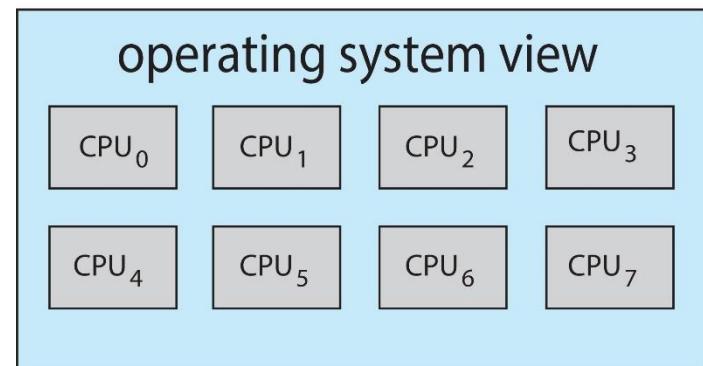
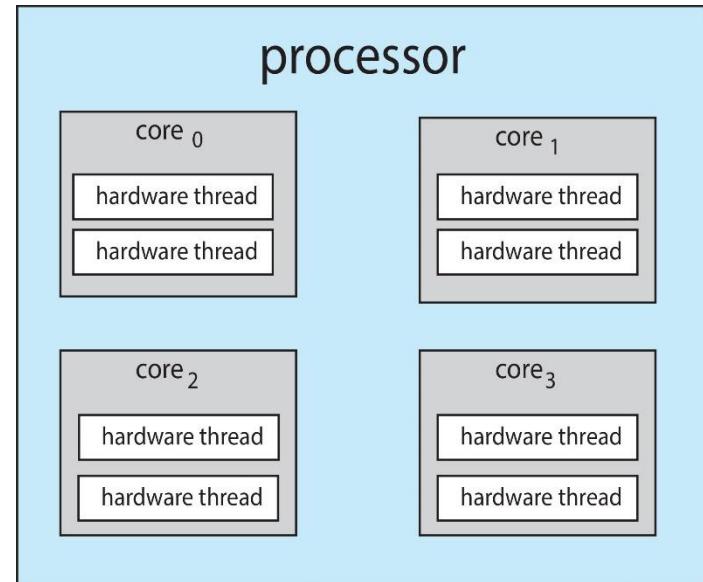
- 각 코어에는 1개 이상의 하드웨어 스레드가 있다.
- 한 스레드에 메모리 지연이 있으면 다른 스레드로 전환
- Figure



5. Multi-Processor Scheduling

Multithreaded Multicore System

- CMT(Chip-multithreading)는 각 코어에 여러 하드웨어 스레드를 할당. (Intel에서는 이것을 하이퍼스레딩 hyperthreading이라고 함.)
- 코어당 2개의 하드웨어 스레드가 있는 쿼드 코어 quad-core 시스템에서 운영 체제는 8개의 논리 프로세서를 인식.

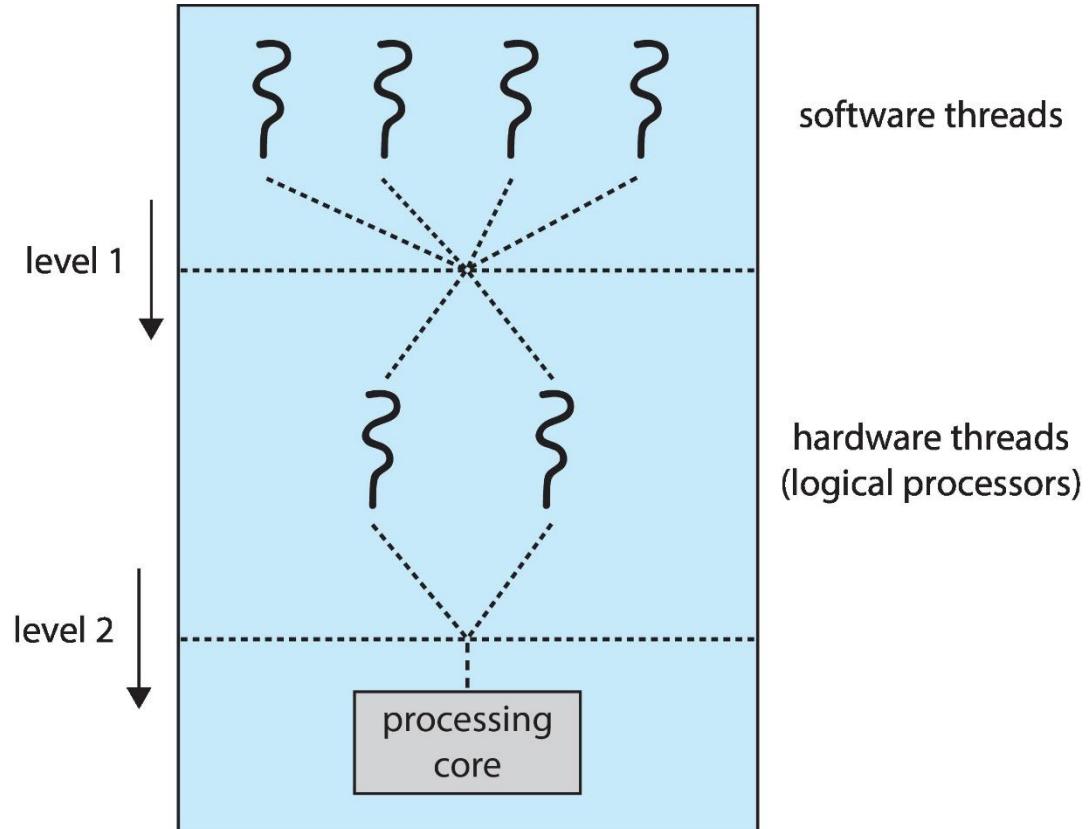




5. Multi-Processor Scheduling

Multithreaded Multicore System

- 두 가지 일정 수준:
 1. 논리 CPU에서 실행할 소프트웨어 스레드를 결정하는 운영 체제
 2. 각 코어가 물리적 코어에서 실행할 하드웨어 스레드를 결정하는 방법.





5. Multi-Processor Scheduling

다중 프로세서 스케줄링 – 로드 밸런싱

가

- SMP인 경우 효율성을 위해 모든 CPU를 로드된 상태로 유지해야 함
- 로드 밸런싱 Load balancing 은 워크로드를 고르게 분산시키려는 시도.
- 푸시 마이그레이션 Push migration - 주기적인 작업이 각 프로세서의 부하를 확인하고 발견되면 과부하된 CPU에서 다른 CPU로 작업을 푸시.
- 마이그레이션 가져오기 Pull migration – 유휴 프로세서가 바쁜 프로세서에서 대기 중인 작업을 가져온다.



5. Multi-Processor Scheduling

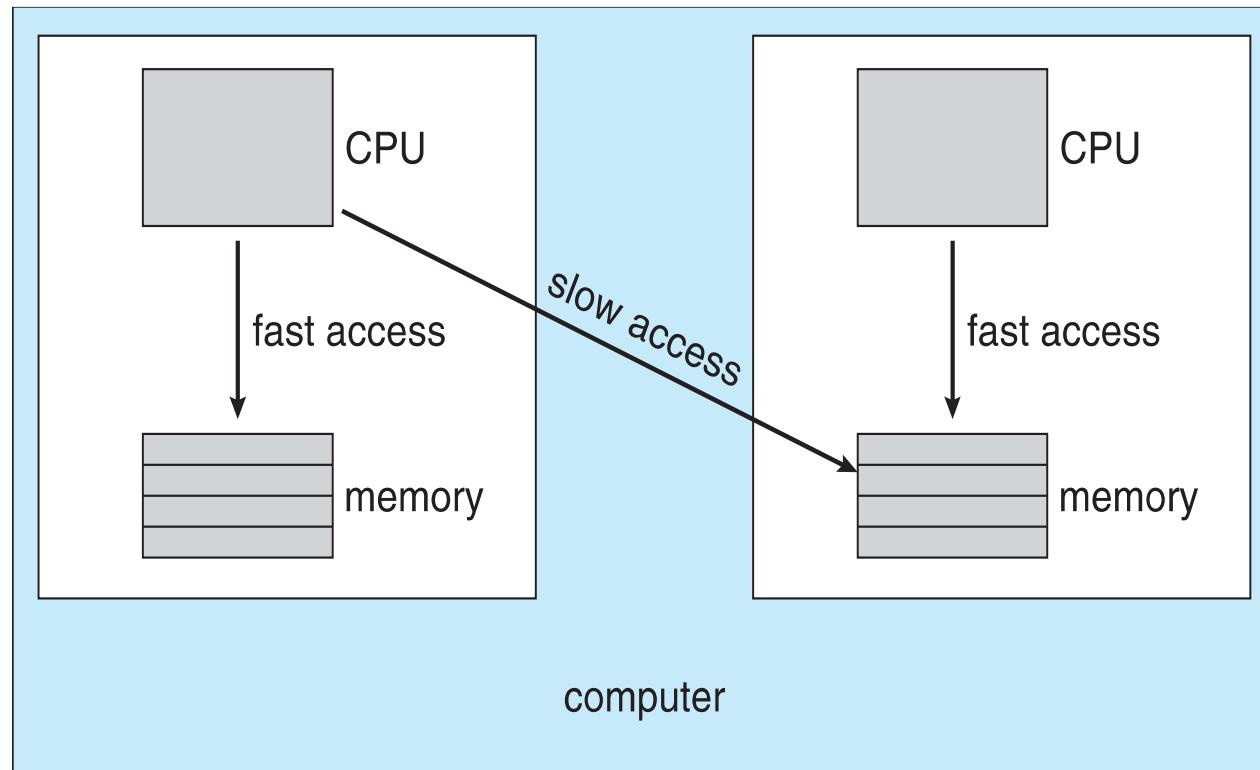
다중 프로세서 스케줄링 – 프로세서 선호도 Affinity

- 스레드가 하나의 프로세서에서 실행되면 해당 프로세서의 캐시 내용은 해당 스레드가 액세스하는 메모리를 저장.
- 우리는 이것을 프로세서에 대한 선호도(즉, "프로세서 선호도 processor affinity")가 있는 스레드라고 함.
- 로드 밸런싱은 스레드가 한 프로세서에서 다른 프로세서로 이동하여 부하를 분산할 수 있기 때문에 프로세서 선호도에 영향을 미칠 수 있지만 해당 스레드는 이전된 프로세서의 캐시에 있던 내용을 잊게 된다.
- 소프트 선호도 Soft affinity – 운영 체제는 스레드가 동일한 프로세서에서 계속 실행되도록 시도하지만 보장하지는 않는다.
- Hard Affinity – 프로세스가 실행할 수 있는 프로세서 집합을 지정 할 수 있다.

5. Multi-Processor Scheduling

NUMA 및 CPU 스케줄링

- 운영 체제가 NUMA NUMA-aware 를 인식하는 경우 스레드가 실행 중인 CPU에 가까운 메모리를 할당.





6. Real-Time CPU Scheduling

명백한 문제 obvious challenges

- 소프트 실시간 시스템 **Soft real-time systems** – 중요한 실시간 작업의 우선순위가 가장 높지만 작업 일정이 언제 잡힐지는 보장되지 않는다.
- 하드 실시간 시스템 **Hard real-time systems** – 작업은 마감일까지 서비스되어야 한다.

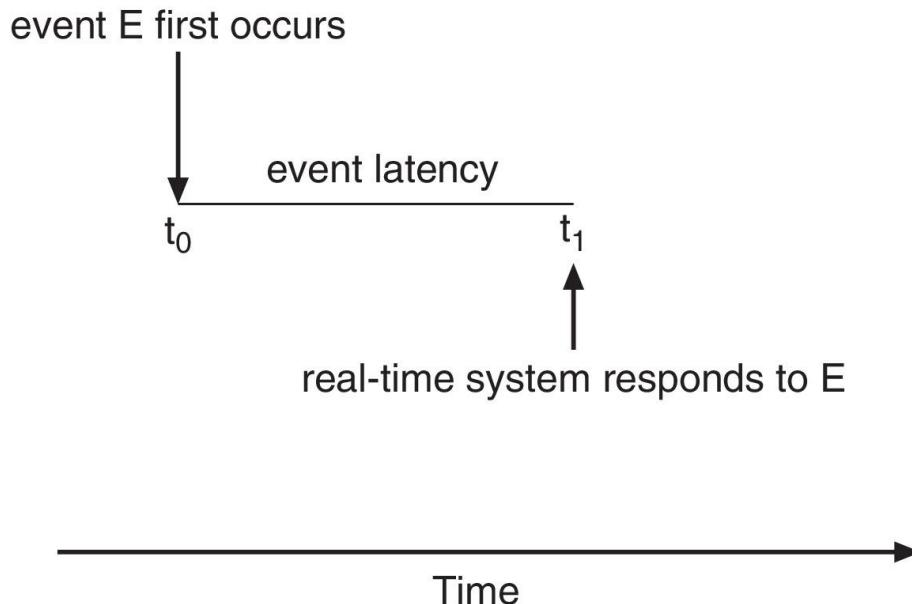




6. Real-Time CPU Scheduling

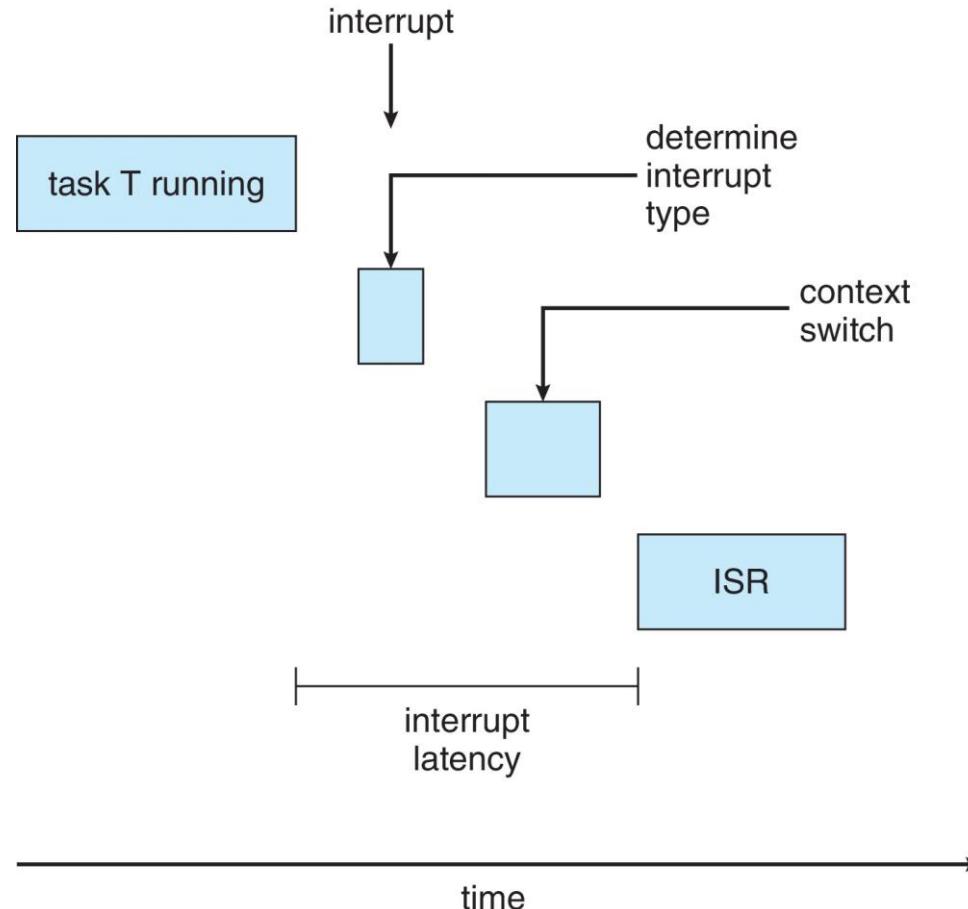
명백한 문제 obvious challenges

- 이벤트 대기 시간 Event latency – 이벤트가 발생한 시점부터 서비스될 때까지 경과된 시간.
- 성능에 영향을 미치는 두 가지 유형의 대기 시간
 1. 인터럽트 대기 시간 Interrupt latency – 인터럽트 도착부터 서비스가 인터럽트하는 루틴 시작까지의 시간
 2. 디스패치 대기 시간 Dispatch latency – 현재 프로세스를 CPU에서 제거하고 다른 프로세스로 전환하는 일정에 걸리는 시간



6. Real-Time CPU Scheduling

인터럽트 대기 시간 Interrupt Latency



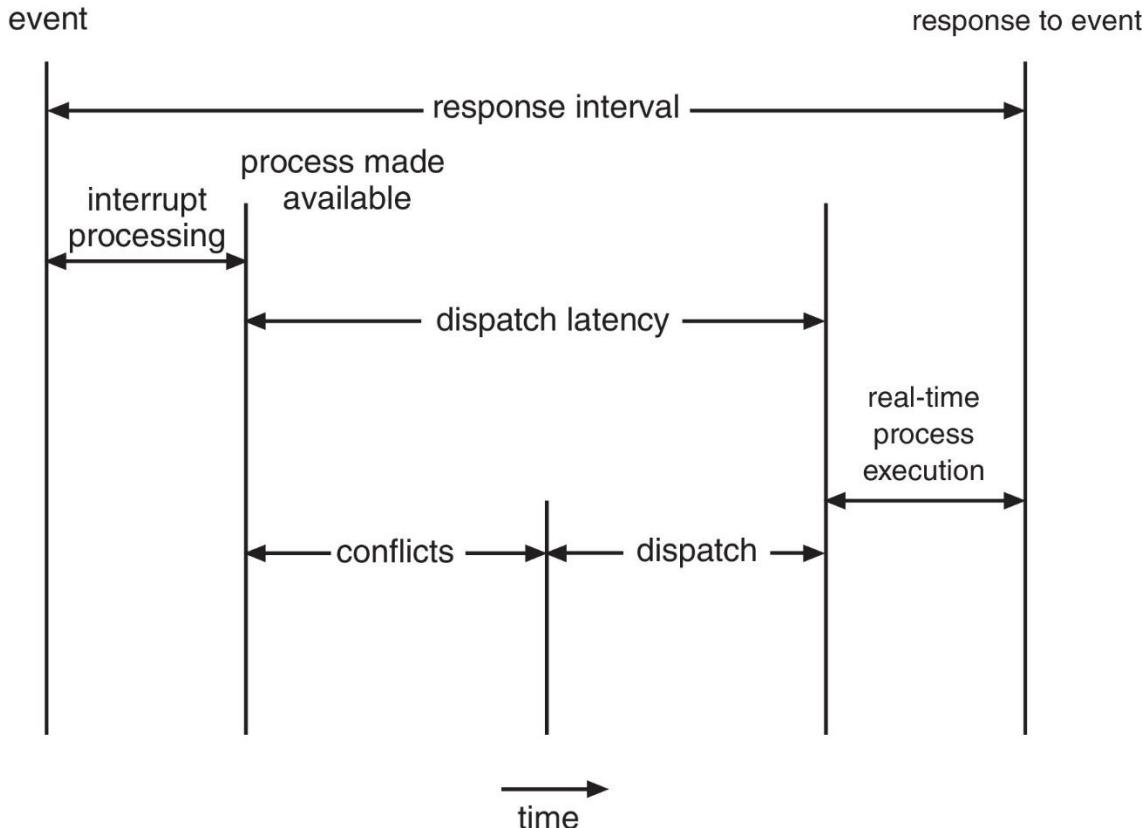


6. Real-Time CPU Scheduling

Dispatch Latency

- 디스패치 대기 시간의 충돌 단계:

- 커널 모드에서 실행 중인 모든 프로세스의 선점
- 우선순위가 높은 프로세스에 필요한 자원을 우선순위가 낮은 프로세스에 의해 해제



6. Real-Time CPU Scheduling

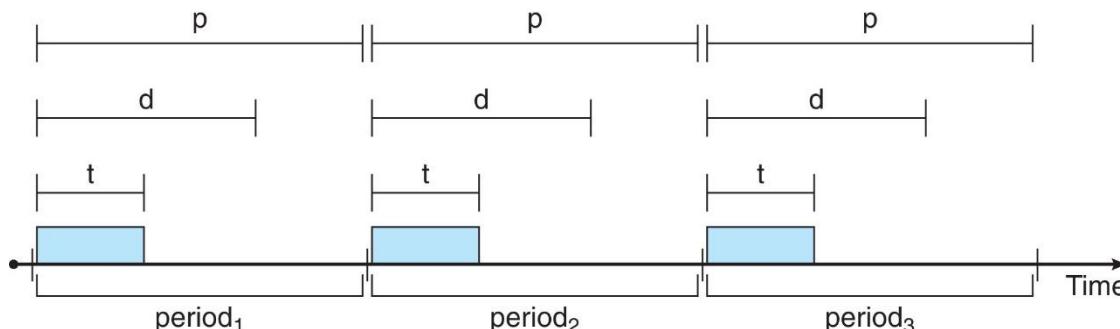
Priority-based Scheduling

- 실시간 스케줄링을 위해 스케줄러는 선제적 우선 순위 기반 스케줄링을 지원해야 한다.
 - 그러나 부드러운 실시간만 보장.
- 하드 실시간을 위해 마감 기한을 맞출 수 있는 기능도 제공해야 한다.
- 프로세스에는 새로운 특성이 있다. 주기적인 프로세스에는 일정한 간격으로 CPU가 필요.

처리 시간 t, 기한 d, 기간 p(processing time t, deadline d, period p)

$$0 \leq t \leq d \leq p$$

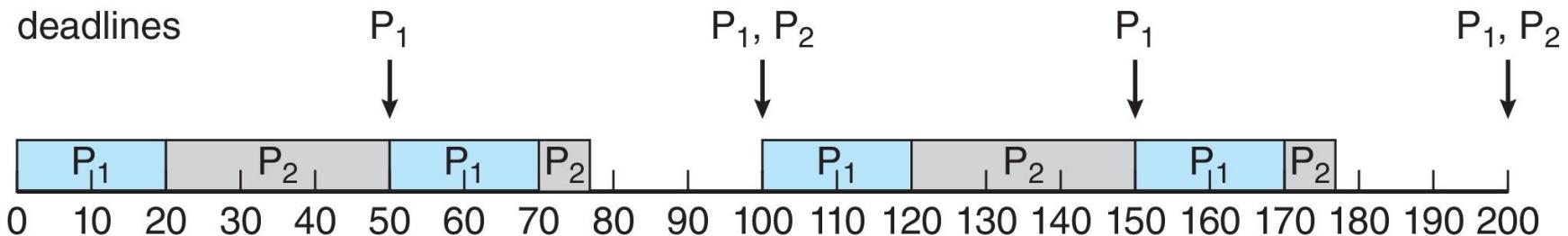
주기적 태스크 비율은 $1/p$ (Rate of periodic task is $1/p$)



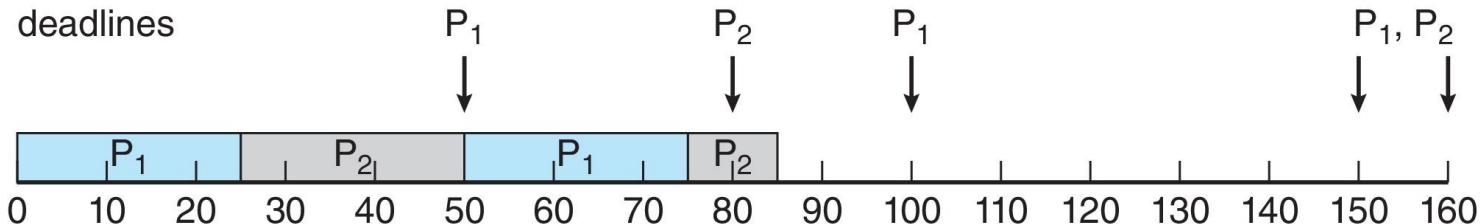
6. Real-Time CPU Scheduling

속도 단조 스케줄링 Rate Monotonic Scheduling

- 기간의 역수를 기준으로 우선 순위가 지정.
- 더 짧은 기간 = 더 높은 우선순위;
- 긴 기간 = 낮은 우선순위
- P_1 은 P_2 보다 높은 우선순위를 할당받는다.



- Rate Monotonic Scheduling으로 기한을 놓침
- 프로세스 P_2 가 시간 80에서 최종 기한을 완료하지 못함

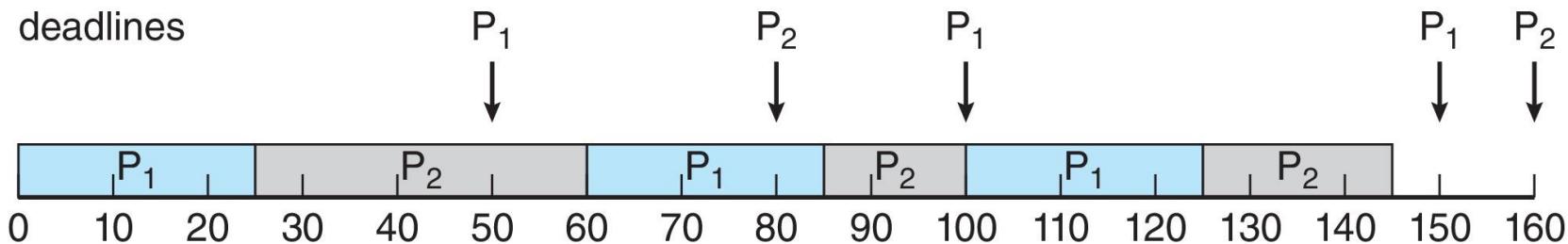




6. Real-Time CPU Scheduling

EDF(Earliest Deadline First Scheduling)

- 마감일에 따라 우선순위가 지정다.
- 기한이 빠를수록 우선순위가 높아진다.
- 기한이 늦을수록 우선순위가 낮아진다.



비례 공유 스케줄링 Proportional Share Scheduling

- T 공유는 시스템의 모든 프로세스에 할당.
- 애플리케이션은 N < T인 N 공유를 수신.
- 이렇게 하면 각 애플리케이션이 총 프로세서 시간의 N/T 를 수신하게 됨.



6. Real-Time CPU Scheduling

POSIX Real-Time Scheduling

- POSIX.1b 표준
- API는 실시간 스레드 관리를 위한 기능을 제공.
- 실시간 스레드에 대한 두 가지 스케줄링 클래스를 정의.
 - **SCHED_FIFO** - 스레드는 FIFO 대기열이 있는 FCFS 전략을 사용하여 예약됩니다. 우선 순위가 같은 스레드에 대한 시간 분할이 없다.
 - **SCHED_RR** - **SCHED_FIFO**와 유사하지만 우선 순위가 같은 스레드에 대해 시간 분할이 발생한다는 점을 제외하고
- 스케줄링 정책을 가져오고 설정하기 위한 두 가지 기능을 정의합니다.
 - `pthread_attr_getsched_policy(pthread_attr_t *attr, int *정책)`
 - `pthread_attr_setsched_policy(pthread_attr_t *attr, 정수 정책)`



6. Real-Time CPU Scheduling

POSIX Real-Time Scheduling API

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ cat > PthreadScheduling.c
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

void *runner(void *param);

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

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* get the current scheduling policy */
    if(pthread_attr_getschedpolicy(&attr, &policy) != 0)
    {
        fprintf(stderr, "Unable to get policy.\n");
        return 1;
    }
    else
    {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* set the scheduling policy - FIFO, RR, or OTHER */
    if(pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    {
        fprintf(stderr, "Unable to set policy.\n");
        return 1;
    }
}
```



6. Real-Time CPU Scheduling

POSIX Real-Time Scheduling API

```
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
{
    if (pthread_create(&tid[i], &attr, runner, NULL) != 0)
    {
        fprintf(stderr, "Unable to create thread.\n");
        return 1;
    }

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
    {
        if (pthread_join(tid[i], NULL) != 0)
        {
            fprintf(stderr, "Unable to join thread.\n");
            return 1;
        }
    }

    return 0;
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(NULL);
}
```

```
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ gcc PthreadScheduling.c
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$ ./a.out
SCHED_OTHER
k8s@DESKTOP-RoEQ2U6:~/c_workspaces$
```



7. Operating System Examples

Linux Scheduling Through Version 2.5

- 커널 버전 2.5 이전에는 표준 UNIX 스케줄링 알고리즘의 변형을 실행했다.
- 버전 2.5는 일정 순서 $O(1)$ 스케줄링 시간으로 이동했다.
 - 선제적, 우선순위 기반
 - 두 가지 우선 순위 범위: 시분할 및 실시간
 - 0에서 99까지의 실시간 범위와 100에서 140까지의 좋은 값
 - 더 높은 우선 순위를 나타내는 낮은 숫자 값을 사용하여 전역 우선 순위에 매핑
 - 우선 순위가 높을수록 커짐 q
 - 타임 슬라이스에 남은 시간 동안 작업 실행 가능(활성)
 - 남은 시간이 없으면(만료됨) 다른 모든 작업이 슬라이스를 사용할 때까지 실행할 수 없다.
 - CPU별 실행 대기열 데이터 구조에서 추적되는 실행 가능한 모든 작업
 - ✓ 2개의 우선 순위 어레이(활성, 만료)
 - ✓ 우선순위에 따라 인덱싱된 작업
 - ✓ 더 이상 활성화되지 않으면 어레이가 교환.
 - 잘 작동했지만 대화식 프로세스에 대한 응답 시간이 좋지 않음



7. Operating System Examples

Linux Scheduling in Version 2.6.23 +

- 완전히 공정한 스케줄러(CFS)
- 스케줄링 클래스Scheduling classes
- 각각은 특정한 우선순위를 가진다
- 스케줄러는 가장 높은 스케줄링 클래스에서 가장 높은 우선 순위 작업을 선택한다.
- 고정된 시간 할당량을 기반으로 하는 양자 대신 CPU 시간의 비율을 기반으로 함
- 2개의 스케줄링 클래스 포함, 다른 클래스 추가 가능
 - ✓ 기본default
 - ✓ 실시간real-time



7. Operating System Examples

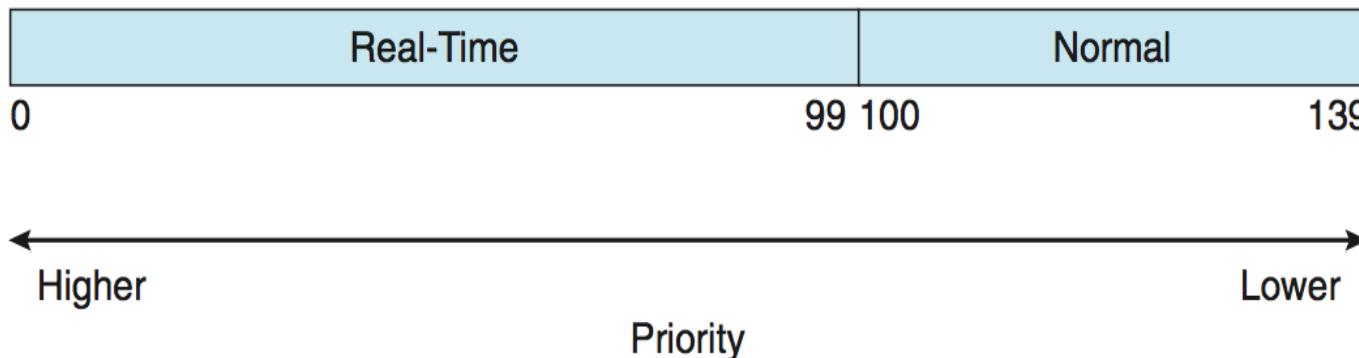
Linux Scheduling in Version 2.6.23 +

- -20에서 +19까지의 nice 값을 기준으로 계산된 Quantum
- 낮은 값이 높은 우선순위
- 목표 대기 시간 target latency 계산 – 작업이 적어도 한 번 실행되어야 하는 시간 간격
- 활성 작업 수가 증가하면 대상 대기 시간이 증가할 수 있다.
- CFS 스케줄러는 작업당 가상 런타임 virtual run time 을 변수 vruntime으로 유지.
- 작업 우선순위에 따른 감쇠 요인과 연관됨 – 우선순위가 낮을수록 감쇠율이 높아짐
- 일반 기본 우선 순위는 가상 실행 시간 = 실제 실행 시간을 산출.
- 실행할 다음 작업을 결정하기 위해 스케줄러는 가상 실행 시간이 가장 낮은 작업을 선택.

7. Operating System Examples

Linux Scheduling

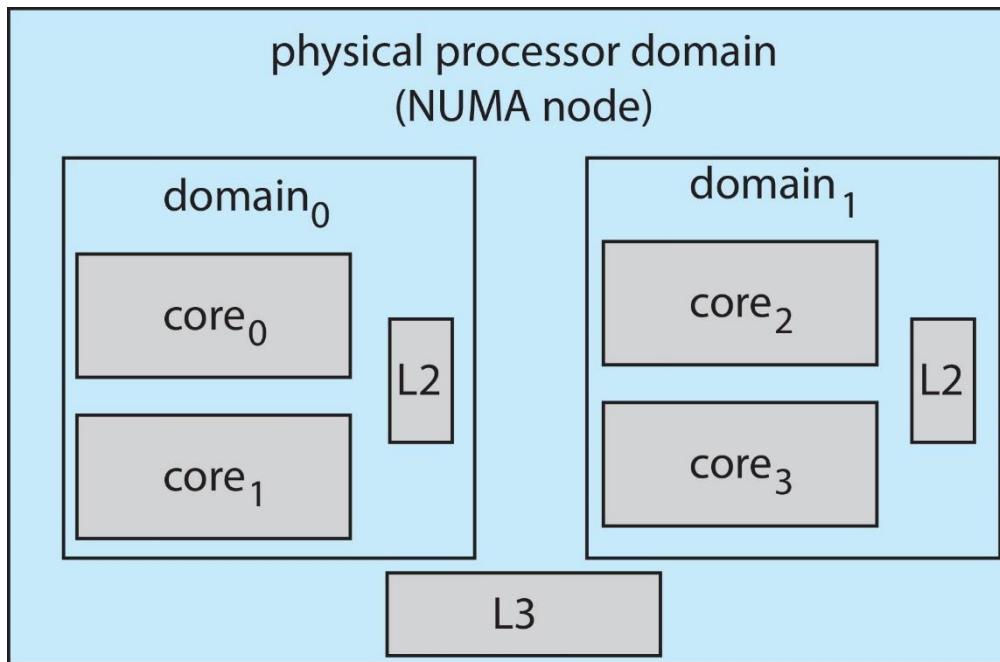
- POSIX.1b에 따른 실시간 스케줄링
- 실시간 작업에는 정적 우선 순위가 있다.
- 글로벌 우선 순위 체계에 대한 실시간 플러스 노멀 맵
- 좋은 값 -20은 전역 우선 순위 100에 매핑.
- +19의 좋은 값은 우선 순위 139에 매핑.



7. Operating System Examples

Linux Scheduling

- Linux는 로드 밸런싱을 지원하지만 NUMA-aware.
- 스케줄링 도메인은 서로 균형을 이룰 수 있는 CPU 코어 세트.
- 도메인은 공유하는 것(즉, 캐시 메모리)에 따라 구성. 목표는 스레드가 도메인 간에 마이그레이션되지 않도록 하는 것.





7. Operating System Examples

Windows Scheduling

- Windows는 우선 순위 기반 선제적 스케줄링을 사용.
- 우선 순위가 가장 높은 스레드가 다음에 실행.
- 디스패처 Dispatcher 는 스케줄러.
- 스레드는 (1) 블록, (2) 타임 슬라이스 사용, (3) 우선순위가 더 높은 스레드에 의해 선점될 때까지 실행.
- 실시간 스레드는 비실시간 스레드를 선점할 수 있다.
- 32레벨 우선순위 체계
- 가변 등급 Variable class 은 1-15, 실시간 등급 real-time class 은 16-31.
- 우선 순위 0은 메모리 관리 스레드.
- 우선 순위별 대기열
- 실행 가능한 스레드가 없으면 유휴 스레드idle thread 를 실행.



7. Operating System Examples

Windows 우선 순위 클래스

- Win32 API는 프로세스가 속할 수 있는 여러 우선 순위 클래스를 식별.

REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS,
BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS

- REALTIME을 제외하고 모두 가변적.
- 주어진 우선 순위 클래스 내의 스레드는 상대적 우선 순위를 갖는다.
- TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- 우선 순위 클래스와 상대적 우선 순위가 결합되어 숫자 우선 순위 부여
- 기본 우선 순위는 클래스 내에서 NORMAL입니다.
- 퀸텀이 만료되면 우선 순위가 낮아지지만 기본 이하가 되지 않습니다.

7. Operating System Examples



Windows 우선 순위 클래스

- 대기가 발생하면 대기한 항목에 따라 우선순위가 높아짐
- 3배의 우선 순위 부스트가 주어진 **Foreground window**
- Windows 7에는 UMS(user-mode scheduling) 사용자 모드 스케줄링)가 추가되었다.
- 응용 프로그램은 커널과 독립적으로 스레드를 만들고 관리.
- 많은 스레드의 경우 훨씬 더 효율적.
- UMS 스케줄러는 C++ Concurrent Runtime(ConcRT) 프레임워크와 같은 프로그래밍 언어 라이브러리에서 제공.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1