

Global KPU!

글로벌 경쟁력을 갖춘 산업기술 명문대학
세계를 향해 더 큰 미래를 펼쳐갑니다

6장. 멀티스레드

네트워크 게임 프로그래밍

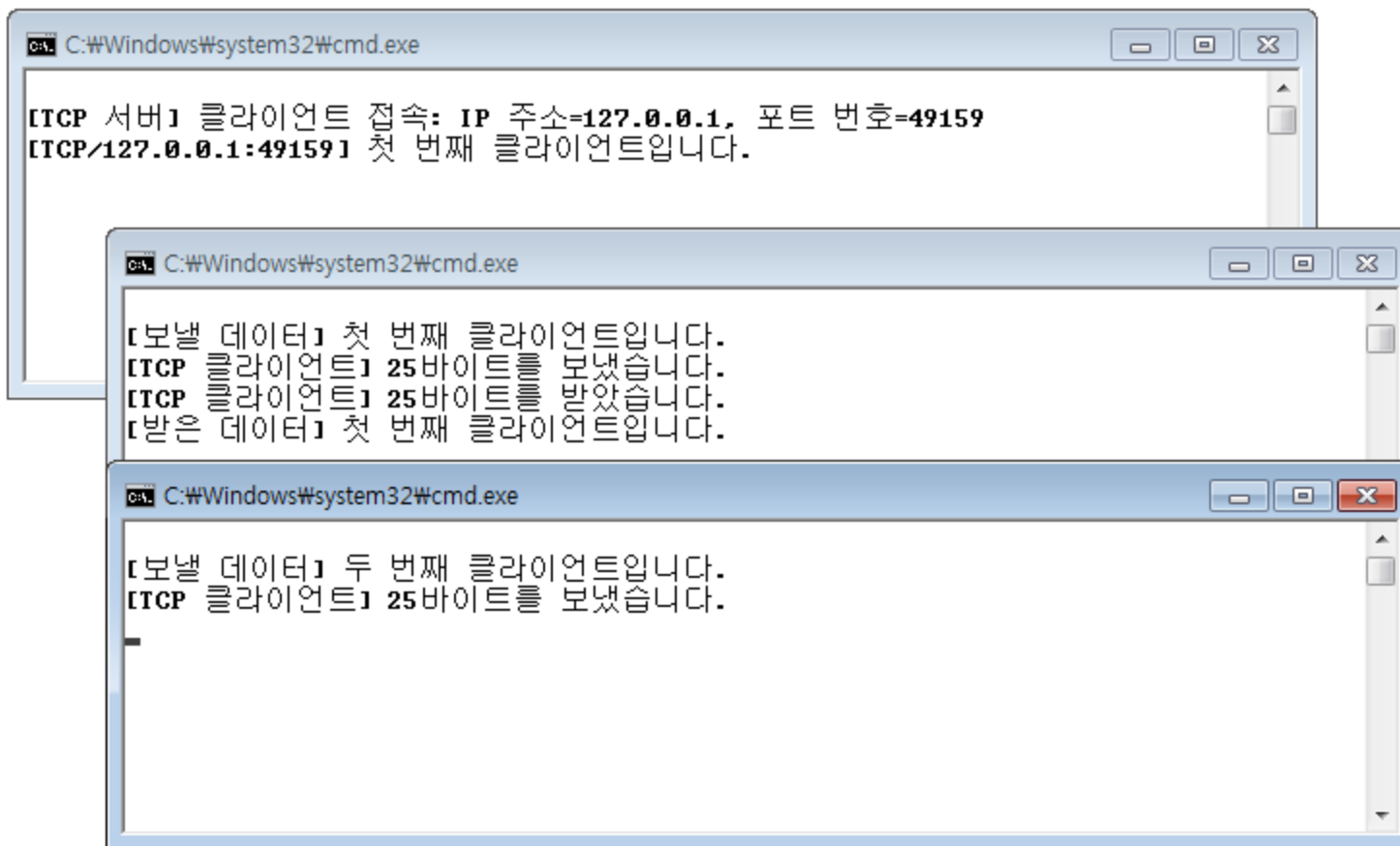
- ❖ 멀티스레드 프로그래밍의 필요성을 이해하고 기본 개념을 익힌다.
- ❖ 멀티스레드를 이용해 다중 클라이언트 처리가 가능한 서버를 작성할 수 있다.
- ❖ 스레드 동기화 기법을 이해하고 활용할 수 있다.



TCP 서버-클라이언트의 문제점 (1)

❖ 문제 ①: 동시에 둘 이상의 클라이언트 서비스 불가

- 첫번째로 접속한 클라이언트 이외의 다른 클라이언트 어플리케이션은 통신되지 않는다



The image displays three overlapping Windows command prompt windows, each titled "C:\Windows\system32\cmd.exe".

The top window shows the server's initial state:

```
[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=49159  
[TCP/127.0.0.1:49159] 첫 번째 클라이언트입니다.
```

The middle window shows the first client's interaction:

```
[보낼 데이터] 첫 번째 클라이언트입니다.  
[TCP 클라이언트] 25바이트를 보냈습니다.  
[TCP 클라이언트] 25바이트를 받았습니다.  
[받은 데이터] 첫 번째 클라이언트입니다.
```

The bottom window shows the second client's interaction:

```
[보낼 데이터] 두 번째 클라이언트입니다.  
[TCP 클라이언트] 25바이트를 보냈습니다.
```



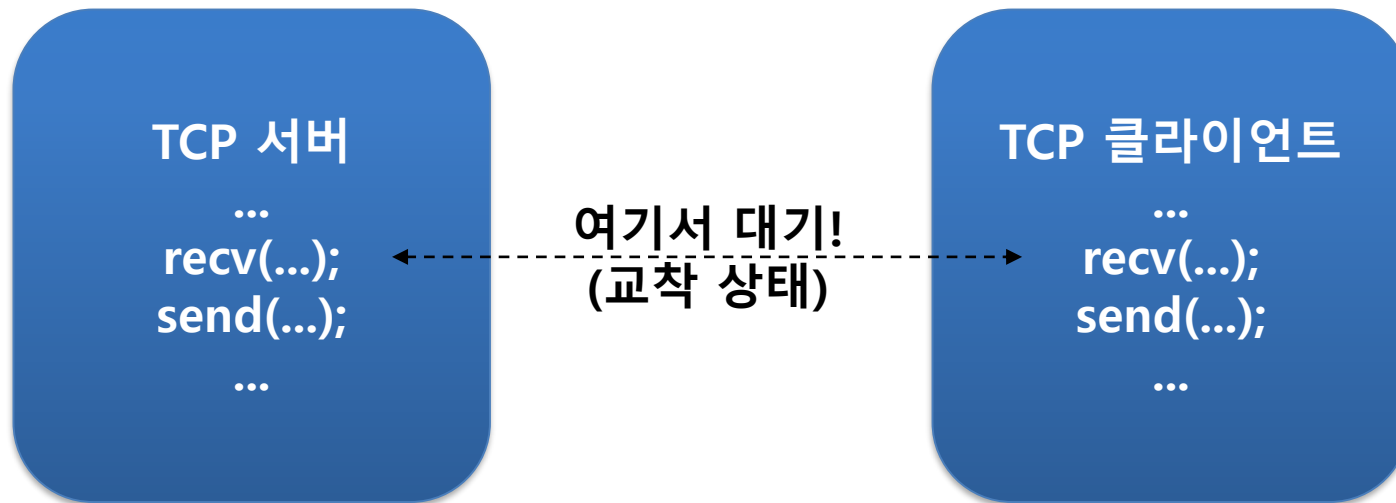
❖ 문제 ① 해결책

- 서버가 각 클라이언트와 통신하는 시간을 짧게 줄임
 - 장점: 구현이 쉬움, 가장 적은 시스템 자원 사용
 - 단점: 각 클라이언트의 처리 지연 시간이 길어질 수 있음. 대용량 데이터 전송에는 비합리적.
- 각 클라이언트를 **스레드**를 이용해 독립적으로 처리
 - 장점: 소켓 입출력 모델에 비해 구현이 쉬움
 - 단점: 가장 많은 시스템 자원 사용
- 소켓 입출력 모델 사용(10~11장)
 - 장점: 소수의 **스레드**를 이용해 다수의 클라이언트를 처리
⇒ 상대적으로 적은 시스템 자원 사용
 - 단점: 구현이 어려움



❖ 문제 ②: 교착 상태 발생 가능성

- 서버와 클라이언트의 `send()`, `recv()` 함수의 호출 순서가 서로 맞아야 한다. 데이터를 보내지 않은 상태에서 양쪽에서 동시에 `recv()` 함수를 호출하면 교착상태가 발생할 수 있다.
- 교착상태: 영원히 일어나지 않을 이벤트를 두개 이상의 프로세스가 기다리는 상황
- `recv()` 함수에서 빠져나가지 못하고 `send()` 함수를 호출할 수 없어 교착상태 발생



❖ 문제 ② 해결책

- 데이터 송수신 부분 잘 설계하기 => 멀티스레드로 해결
 - 장점: 특별한 기법 없이 곧바로 구현 가능
 - 단점: 모든 경우에 대한 해결책은 될 수 없음
- 소켓에 타임아웃 옵션 적용하기(8장)
 - 장점: 구현이 쉬움
 - 단점: 다른 방법에 비해 성능이 낮음
- 년블로킹 소켓 사용하기(10장)
 - 장점: 근본적으로 교착 상태 해결
 - 단점: 구현이 복잡, 시스템 자원(특히 CPU 시간) 낭비
- 소켓 입출력 모델 사용(10~11장)
 - 장점: 년블로킹 소켓의 단점을 보완 & 교착 상태 해결
 - 단점: 첫 번째나 두 번째 해결책보다 구현이 어려움



❖ 스레드?

■ 프로세스의 장단점

- 프로세스는 최소 실행 단위
- 멀티프로세스를 사용할 경우 불필요한 함수(코드) accept()까지 fork해야 하는 비효율성이 발생
- 문맥 전환: 필요한 함수(코드)만 사용

■ 문맥이 스레드라고 보면 문맥전환은 멀티스레드로 말할 수 있음

❖ 멀티 스레드

■ 장점

- 하나의 프로세스일때에는 스레드 동기화가 필요없음
- 신속하게 원하는 코드를 만들수 있음
- 데이터 교환 및 관리가 용이
- CPU를 효율적으로 활용할 수 있음
- 오래된 시간과 기술발전으로 신뢰성이 높음

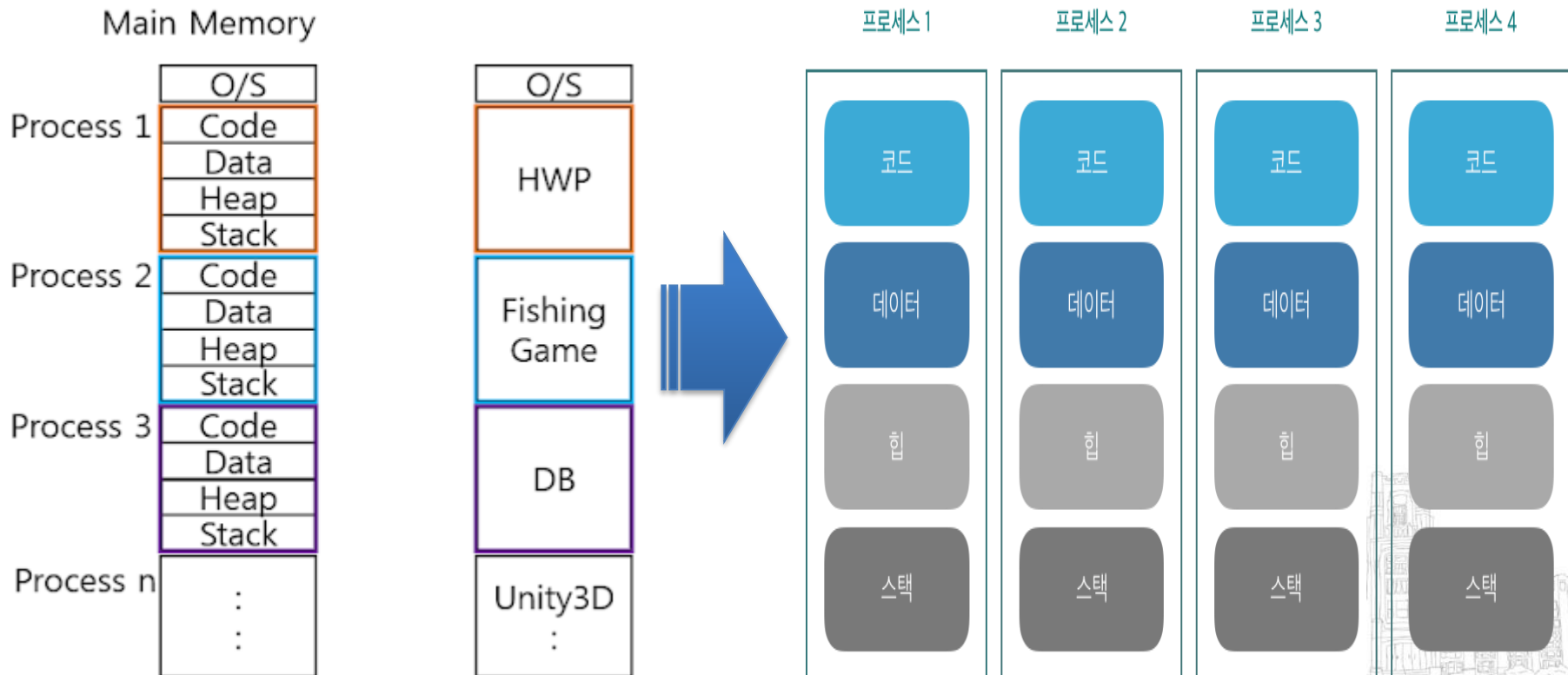
■ 단점

- 프로그래밍이 복잡
- 디버깅이 어려움

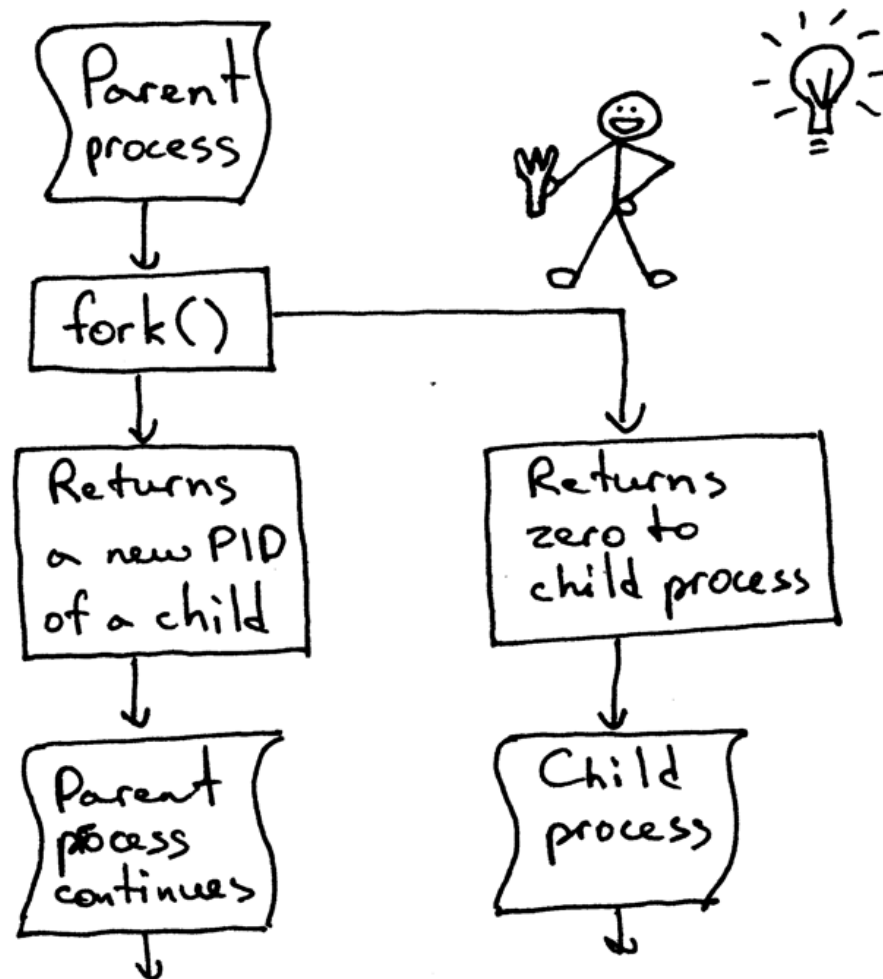


❖ 멀티프로세스

- 프로세스를 여러 개로 복사
- Linux : 프로세스 기반 (스레드는 linux, unix, mac등에서 공통으로 사용하기 위한 코드)
- Windows : 프로세스 + 스레드 기반



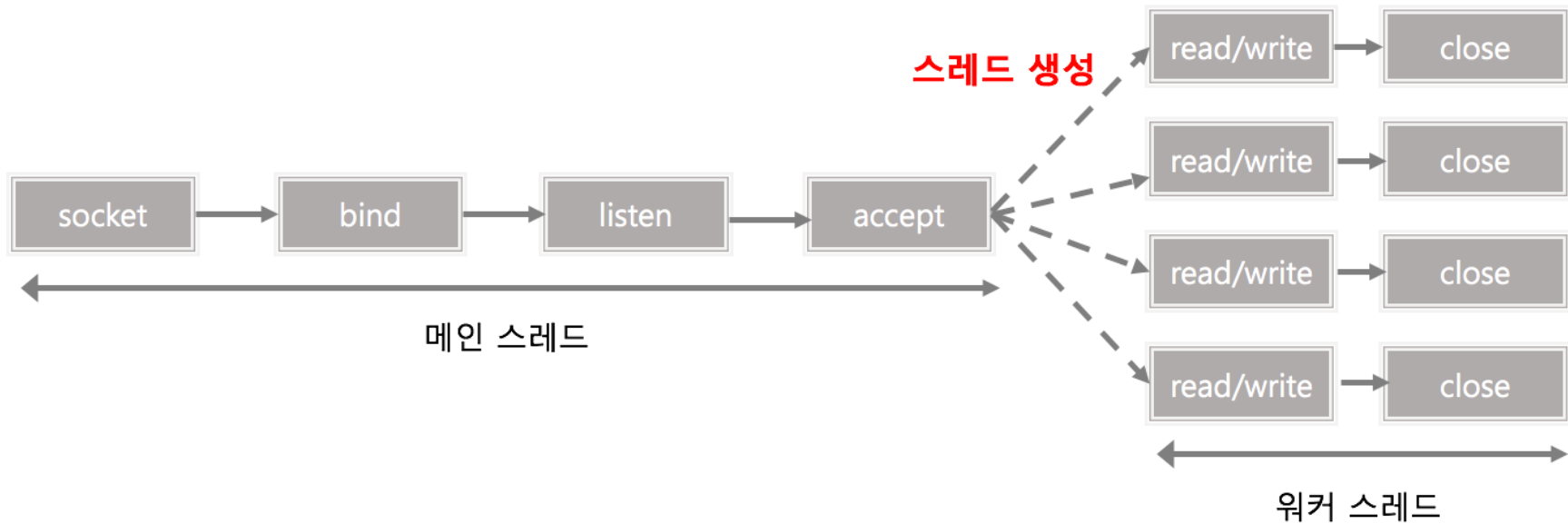
- ❖ Linux vs Windows ?
- ❖ fork() ?



❖ Linux vs Windows ?

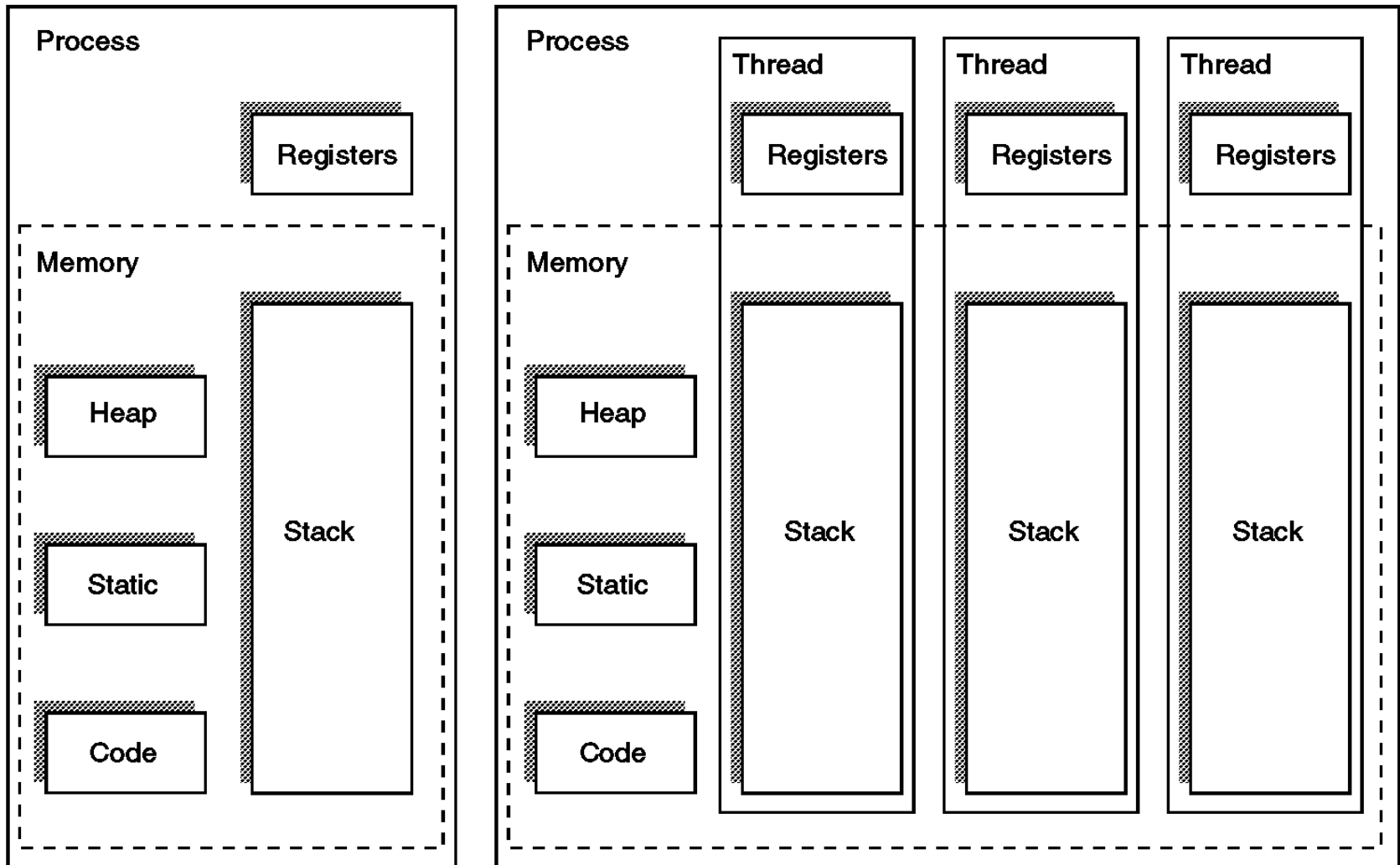
❖ pthread ?

- POSIX 표준을 따름
- 호환성이 높음



❖ 멀티 스레드

- 메일 스레드(부모 스레드) + 자식 스레드(워크 스레드)



❖ Linux vs Windows ?

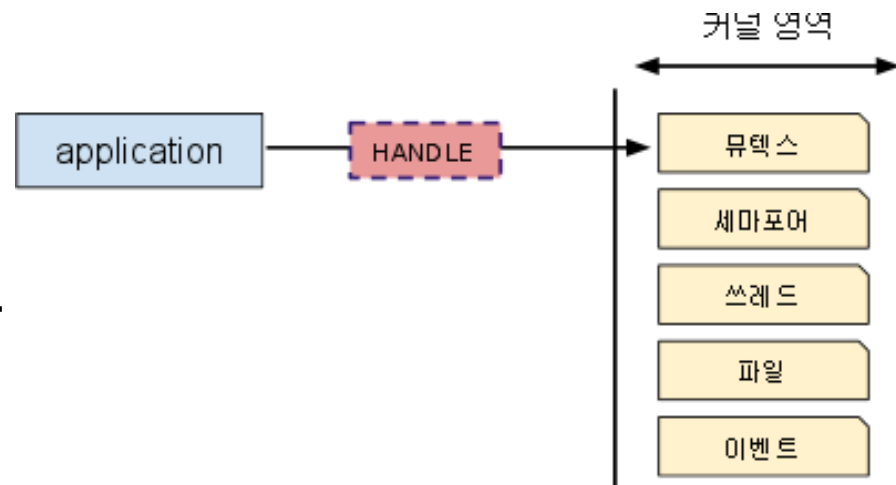
- FILE vs 객체

❖ Pthread

- 윈도우는 pthread를 사용하지 않음.
- 개발시 사용할 수는 있지만 추천하진 않음

❖ 윈도우는 독자적인 전용 함수를 사용

- 윈도우는 자원을 객체로 다룸
- 핸들로 자원을 다룸
- Create로 시작하고 HANDLE 를 반환
- Create로 시작하고 CloseHandle로 닫음
- Wait*()함수를 스레드 동기화의 범용 함수로 사용
 - 리눅스에서는 세마포어, 뮤텍스등에 따라 스레드 동기화나 종료를 달리 지정을 해줘야 함



Linux <u>pthread API</u>	Windows SDK API
<u>pthread create</u>	<u>CreateThread</u>
<u>pthread exit</u>	<u>ThreadExit</u>
<u>pthread join</u>	<u>WaitForSingleObject</u>
<u>pthread mutex init</u>	<u>CreateMutex</u>
<u>pthread mutex destroy</u>	<u>CloseHandle</u>
<u>pthread mutex lock</u>	<u>WaitForSingleObject</u>
<u>pthread mutex unlock</u>	<u>ReleaseMutex</u>
<u>pthread cond init</u>	<u>CreateEvent</u>
<u>pthread cond destroy</u>	<u>CloseHandle</u>
<u>pthread cond signal</u>	<u>SetEvent</u>
<u>pthread cond broadcast</u>	<u>SetEvent/ResetEvent</u>
<u>pthread cond wait/pthread cond timedwait</u>	<u>SingleObjectAndWait</u>



❖ 용어

■ 프로세스

- 코드, 데이터, 리소스를 파일에서 읽어들이 윈도우 운영체제가 할당해놓은 메모리 영역에 담고 있는 일종의 컨테이너로 정적인 개념

■ 스레드

- CPU 시간을 할당받아 프로세스 메모리 영역에 있는 코드를 수행하고 데이터를 사용하는 동적인 개념

■ 주 스레드 or 메인 스레드

- 응용 프로그램 실행 시 최초로 생성되는 스레드
- WinMain() 또는 main() 함수에서 실행 시작

■ 멀티스레드 응용 프로그램

- 응용 프로그램에서 주 스레드와 별도로 동시에 수행하고자 스레드를 추가로 생성해 이 스레드가 해당 작업을 수행하도록 구현

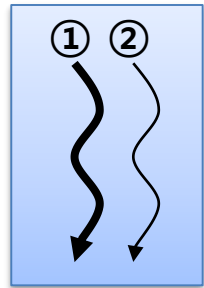
■ 컨텍스트 전환(Context switch)

- 하드웨어(CPU)와 소프트웨어(OS)의 협력으로 이루어지는 스레드 실행 상태의 저장과 복원 작업

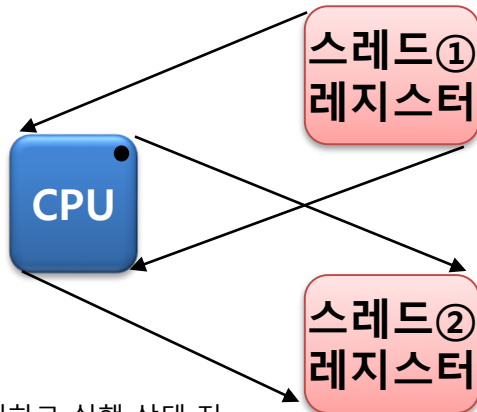
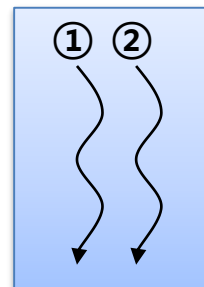
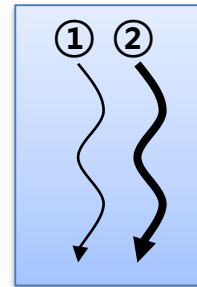
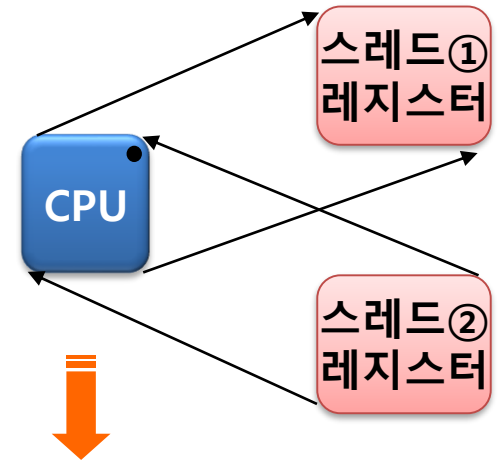
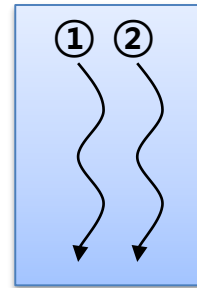


❖ 멀티스레드 동작 원리

b. 스레드1이 실행 중지하고 실행 상태 저장. 이전에 저장해둔 스레드2의 상태 복원



a. 스레드1이 실행중. 명령을 하나씩 수행할 때마다 CPU 레지스터 값과 메모리의 스택 내용이 변경



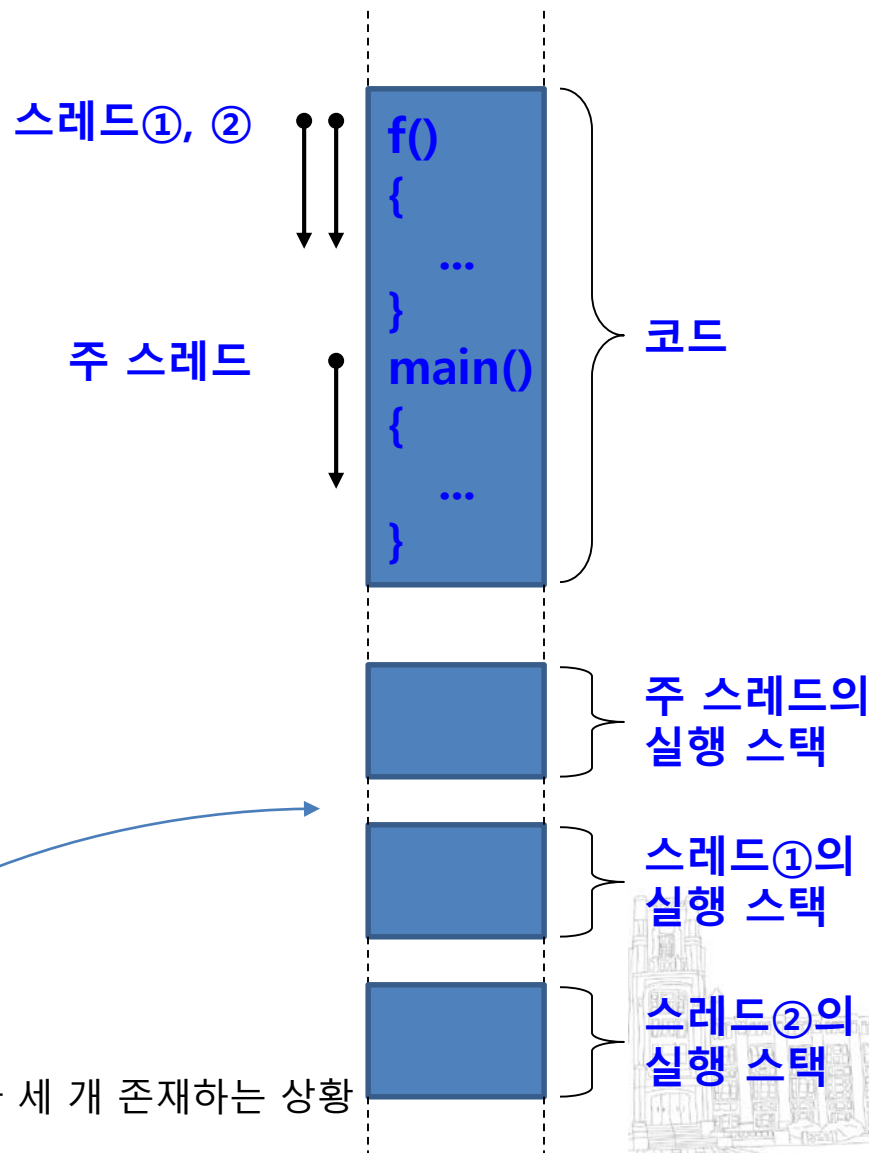
d. 스레드2이 실행 중지하고 실행 상태 저장. 이전에 저장해둔 스레드1의 상태 복원

c. 스레드2를 실행중. 명령을 하나씩 수행할 때마다 CPU 레지스터 값과 메모리의 스택 내용이 변경



❖ 스레드 생성에 필요한 요소

- 스레드 함수의 시작 주소
 - 운영체제는 f() 함수의 시작 주소를 알아야 한다
 - f() 함수와 같이 스레드 실행 시작점이 되는 함수를 스레드 함수라 칭함
- 스레드 함수 실행 시 사용할 스택의 크기
 - 모든 함수는 실행 중 인자 전달과 변수 할당을 위해 스택이 필요
 - 스레드 실행에 필요한 스택 생성은 운영체제가 자동으로 설정. 스택 크기만 정의하면 됨.



* 함수 두 개로 구성된 응용 프로그램에 스레드가 세 개 존재하는 상황

❖ CreateThread() 함수

- 스레드 생성 후 스레드 핸들을 리턴

```
HANDLE CreateThread (  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // NULL  
    SIZE_T dwStackSize, // 0, 스레드에 할당되는 스택 크기  
    LPTHREAD_START_ROUTINE lpStartAddress, // 스레드 함수 시작 주소  
    LPVOID lpParameter, // 스레드 함수에 전달할 인자, 전달할 인자가 없으면 NULL  
    DWORD dwCreationFlags, // 스레드 생성을 제어하는 값. 0 또는 CREATE_SUSPENDED*  
    LPDWORD lpThreadId // 스레드 ID, 필요없으면 NULL  
);
```

성공: 스레드 핸들, 실패: NULL

*CREATE_SUSPENDED를 사용하면 스레드가 생성은 되지만 ResumeThread() 함수를 호출하기 전까
지 실행되지 않음

ex) HANDLE hThread1 = CreateThread(NULL, 0, f, NULL, 0, NULL);

❖ 스레드 함수 형태

- LPTHREAD_START_ROUTINE lpStartAddress, // 스레드 함수 시작 주소

```
DWORD WINAPI ThreadProc(LPVOID lpParameter)
{
    ...
}
```

Q. 한 프로세스에서 최대 몇 개까지 스레드를 생성할 수 있을까?

CreateThread() 함수 호출시 기본 스택 크기는 1mb 이고 32비트 컴퓨터의 경우 스레드를 하나 만들때 마다 2GB(2028개)의 스택을 위한 공간을 할당 받는다.



❖ 스레드 종료 방법

- ① 스레드 함수가 리턴
- ② 스레드 함수 내에서 `ExitThread()` 함수를 호출
- ③ 다른 스레드가 `TerminateThread()` 함수를 호출
- ④ 주 스레드가 종료하면 모든 스레드가 종료

1번과 2번의 경우를 주로 사용

3번은 반드시 필요한 경우에만 사용



❖ 스레드 종료 함수

- ExitThread() 와 TerminateThread() 함수 원형

```
void ExitThread (  
    DWORD dwExitCode    // 종료 코드  
);
```

```
BOOL TerminateThread (  
    HANDLE hThread,      // 종료할 스레드를 가리키는 핸들  
    DWORD dwExitCode     // 종료 코드  
);
```

성공: 0이 아닌 값, 실패: 0



❖ 스레드 생성과 종료 예

- f() 함수를 실행 시작점으로 하는 스레드 두개를 만드는 코드 예

```
DWORD WINAPI f(LPVOID arg)
{
    ....
}

int main()
{
    ....
    // 첫번째 스레드 생성
    HANDLE hThread1 = CreateThread(NULL, 0, f, NULL, 0, NULL);
    If(hThread1 == NULL) 오류 처리;

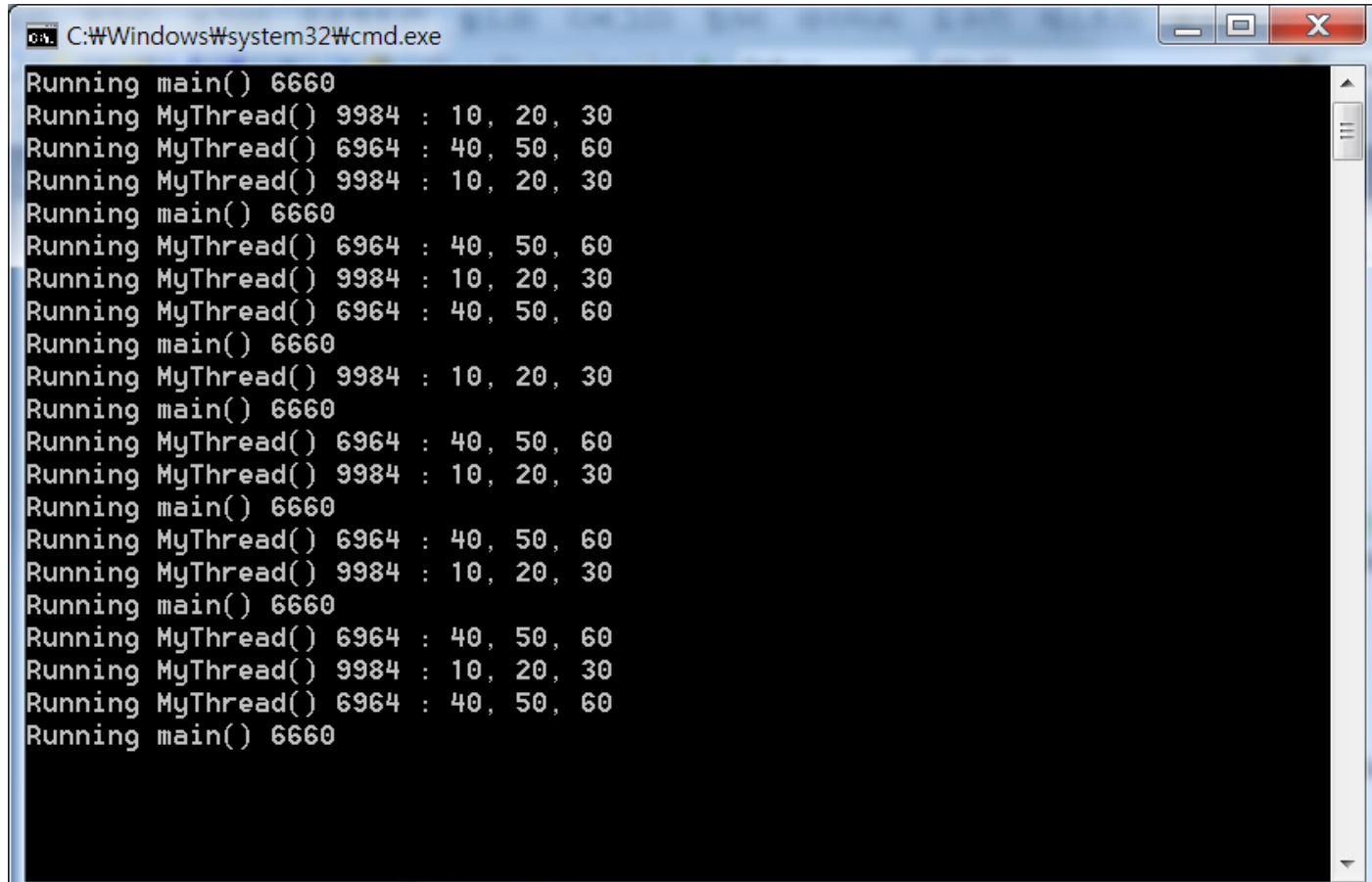
    // 두번째 스레드 생성
    HANDLE hThread2 = CreateThread(NULL, 0, f, NULL, 0, NULL);
    If(hThread2 == NULL) 오류 처리;

    ....
}
```

실습 6-1. 스레드 생성과 종료, 인자 전달 연습

P182 ~

스레드 두개 생성, 스레드 함수 인자를 통해 값을 전달하는 응용 프로그램



```
C:\Windows\system32\cmd.exe

Running main() 6660
Running MyThread() 9984 : 10, 20, 30
Running MyThread() 6964 : 40, 50, 60
Running MyThread() 9984 : 10, 20, 30
Running main() 6660
Running MyThread() 6964 : 40, 50, 60
Running MyThread() 9984 : 10, 20, 30
Running MyThread() 6964 : 40, 50, 60
Running main() 6660
Running MyThread() 9984 : 10, 20, 30
Running main() 6660
Running MyThread() 6964 : 40, 50, 60
Running MyThread() 9984 : 10, 20, 30
Running main() 6660
Running MyThread() 6964 : 40, 50, 60
Running MyThread() 9984 : 10, 20, 30
Running main() 6660
Running MyThread() 6964 : 40, 50, 60
Running MyThread() 9984 : 10, 20, 30
Running MyThread() 6964 : 40, 50, 60
Running main() 6660
```



❖ 스레드는 윈도우 운영체제의 실행 단위므로, 우선순위를 변경하거나 실행을 중지하고 재시작하는 등의 제어 기능을 윈도우 API 수준에서 지원

❖ 용어

■ 스레드 스케줄링 or CPU 스케줄링

- 윈도우 운영체제가 각 스레드에 CPU 시간을 적절히 분배하기 위한 정책

■ 우선순위 클래스

- 프로세스 속성으로, 같은 프로세스가 생성한 스레드는 우선순위 클래스가 모두 같음

■ 우선순위 레벨

- 스레드 속성으로, 같은 프로세스에 속한 스레드 간 상대적인 우선순위를 결정할 때 사용

■ 기본 우선순위

- 우선순위 클래스와 우선순위 레벨을 결합한 값으로, 스레드 스케줄링에 사용

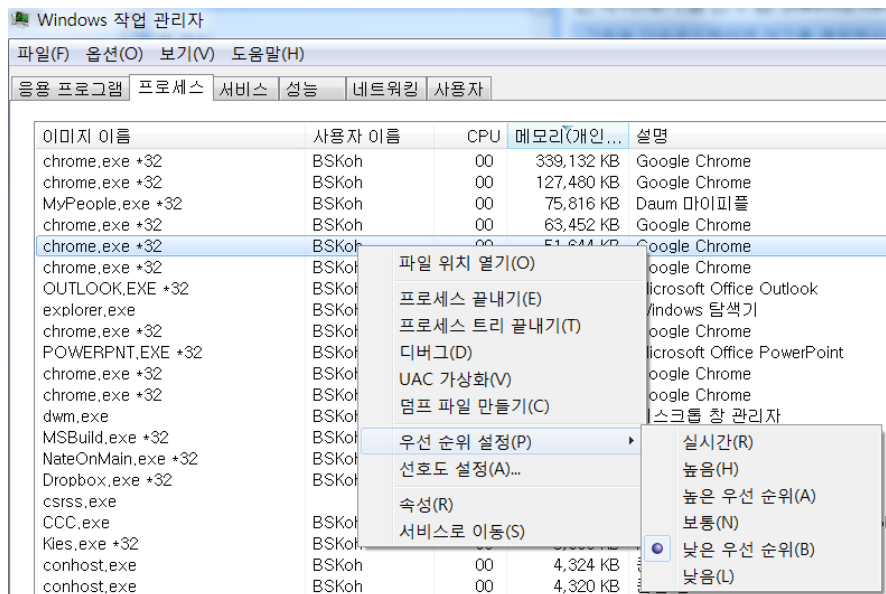


스레드 제어 – 우선순위 변경 (2)

❖ 윈도우 운영체제에서 제공하는 우선순위 클래스

- REALTIME_PRIORITY_CLASS(실시간)
- HIGH_PRIORITY_CLASS(높음)
- ABOVE_NORMAL_PRIORITY_CLASS(높은 우선순위; 윈도우2000 이상)
- NORMAL_PRIORITY_CLASS(보통)
- BELOW_NORMAL_PRIORITY_CLASS(낮은 우선순위; 윈도우2000 이상)
- IDLE_PRIORITY_CLASS(낮음)

❖ 윈도우 작업관리자를 통하여 프로세스의 우선순위 클래스를 변경할 수 있음



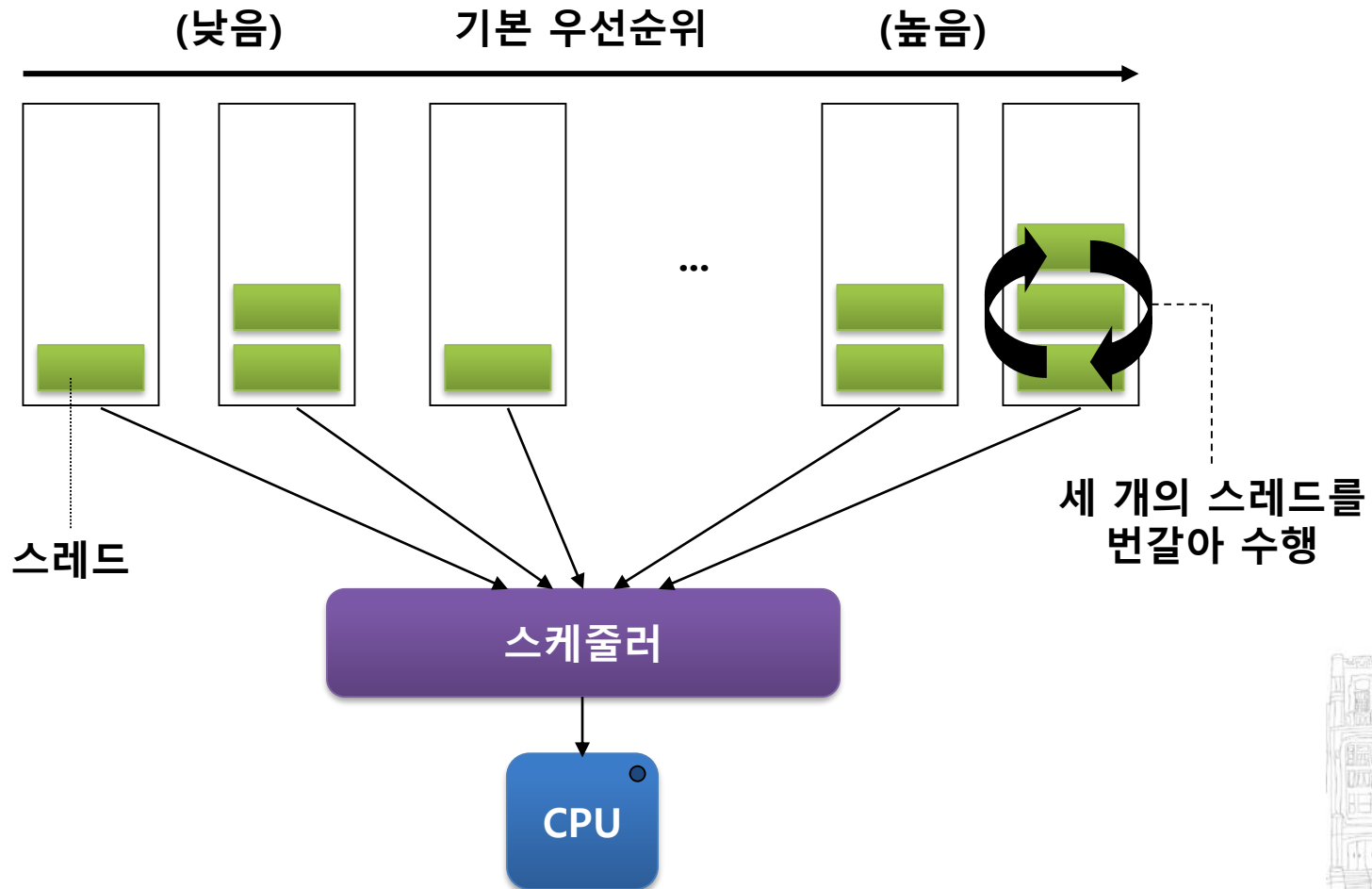
❖ 윈도우 운영체제에서 제공하는 우선순위 레벨

- 우선순위 레벨은 스레드 속성으로, 같은 프로세스에 속한 스레드 간 상대적인 우선순위를 정할 때 사용.
- THREAD_PRIORITY_TIME_CRITICAL
- THREAD_PRIORITY_HIGHEST
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_IDLE



❖ 윈도우의 스레드 스케줄링 방식

- 우선순위가 가장 높은 스레드에 CPU 시간을 할당하되, 우선순위가 같은 스레드가 여러 개 있을 때는 CPU 시간을 번갈아 가며 할당



❖ 우선순위 레벨 조작(변경) 함수

- SetThreadPriority() 함수는 우선순위 레벨을 변경할 때 사용
- GetThreadPriority() 함수는 우선순위 레벨을 얻을 때 사용

```
BOOL SetThreadPriority (  
    HANDLE hThread,      // 스레드 핸들  
    int nPriority         // 우선순위 레벨  
);
```

성공: 0이 아닌 값, 실패: 0

```
int GetThreadPriority (  
    HANDLE hThread      // 스레드 핸들  
);
```

성공: 우선순위 레벨

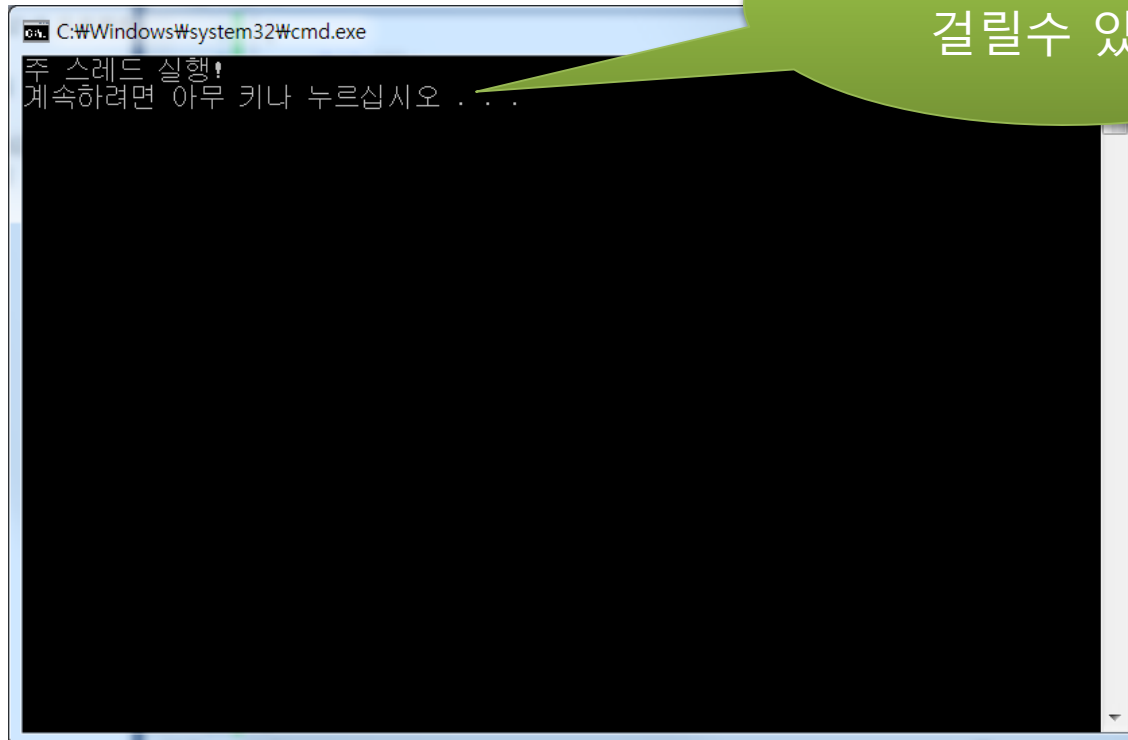
실패: THREAD_PRIORITY_ERROR_RETURN



실습 6-2. 스레드 우선순위 변경 연습

주 스레드 외에 새로운 스레드를 CPU 개수만큼 생성하고 우선순위 레벨을 변경하는 응용프로그램

P188 ~



❖ WaitForSingleObject() 함수

- 특정 스레드가 종료할 때까지 기다리기

```
DWORD WaitForSingleObject (  
    HANDLE hHandle,           // 종료를 기다릴 대상 스레드  
    DWORD dwMilliseconds      // 대기 시간, 밀리초 단위  
);
```

성공: WAIT_OBJECT_0 또는 WAIT_TIMEOUT, 실패: WAIT_FAILED

- dwMilliseconds: 이 시간안에 스레드가 종료되지 않으면 WaitForSingleObject() 함수는 리턴하고, 이때 리턴 값은 WAIT_TIMEOUT이 됨. 스레드가 종료한 경우에는 WAIT_OBJECT_0를 리턴. 대기 시간으로 INFINITE 값을 사용하면 스레드가 종료될때까지 무한히 기다림.

❖ WaitForSingleObject() 함수 사용 예

```
HANDLE hThread = CreateThread(...);  
DWORD retval = WaitForSingleObject(hThread, 1000);  
if(retval == WAIT_OBJECT_0) { ... }           // 스레드 종료  
else if(retval == WAIT_TIMEOUT) { ... } // 타임아웃(스레드는 아직 종료 안 함)  
else { ... }                                 // 에러 발생
```

❖ WaitForMultipleObjects() 함수

- 여러 스레드가 종료하기를 기다리려면 WaitForSingleObject() 함수를 스레드 개수만큼 호출해야 함. 하여, WaitForMultipleObjects() 함수를 사용하면 호출 한번으로 끝낼 수 있음. MAXIMUM_WAIT_OBJECTS 는 최대 60개까지 가능.
- 둘 이상의 스레드가 종료할 때까지 기다리기

DWORD WaitForMultipleObjects (

DWORD nCount, // 배열 원소 개수, 최대값은 MAXIMUM_WAIT_OBJECTS로 정의

const HANDLE* lpHandles, //배열의 시작 주소

BOOL bWaitAll, //TRUE면 모든 스레드가 종료할 때까지 대기, FALSE면 한 스레드가 종료하는 즉시 리턴

DWORD dwMilliseconds // 대기시간으로 밀리초 단위 사용

);

성공: WAIT_OBJECT_0 ~ WAIT_OBJECT_0 + nCount-1

또는 WAIT_TIMEOUT

실패: WAIT_FAILED

❖ WaitForMultipleObjects() 함수 사용 예 ①

// 모든 스레드의 종료를 기다린다.

```
HANDLE hThread[2];
```

```
HANDLE hThread[0] = CreateThread(...);
```

```
HANDLE hThread[1] = CreateThread(...);
```

```
WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
```

- ❖ 참고로, Wait*() 함수는 스레드 종료를 기다리는 전용 함수가 아니라, 스레드 동기화에 사용되는 범용 함수이다.



❖ WaitForMultipleObjects() 함수 사용 예 ②

// 스레드 하나의 종료를 기다린다.

```
HANDLE hThread[2];
```

```
HANDLE hThread[0] = CreateThread(...);
```

```
HANDLE hThread[1] = CreateThread(...);
```

```
DWORD retval = WaitForMultipleObjects(2, hThread, FALSE, INFINITE);
```

```
switch(retval){
```

```
case WAIT_OBJECT_0:           // hThread[0] 종료
```

```
    break;
```

```
case WAIT_OBJECT_0 + 1:       // hThread[1] 종료
```

```
    break;
```

```
case WAIT_FAILED:             // 오류 발생
```

```
    break;
```

```
}
```



❖ 스레드 실행 중지 함수 ①

```
DWORD SuspendThread (  
    HANDLE hThread  
);
```

// 스레드 핸들

성공: 중지 횟수, 실패: -1

❖ 스레드 재실행 함수

```
DWORD ResumeThread (  
    HANDLE hThread  
);
```

// 스레드 핸들

성공: 중지 횟수, 실패: -1

- 윈도우 운영체제는 스레드의 중지 횟수(suspend count)를 관리



❖ 실행 중지 함수 ②

- SuspendThread() 함수를 호출한 경우에는 반드시 ResumeThread() 함수를 사용하여 스레드가 재시작
- Sleep() 함수를 호출하면 dwMilliseconds로 지정한 시간이 지나면 자동으로 재시작

```
void Sleep (  
    DWORD dwMilliseconds    // 밀리초(ms)  
);
```



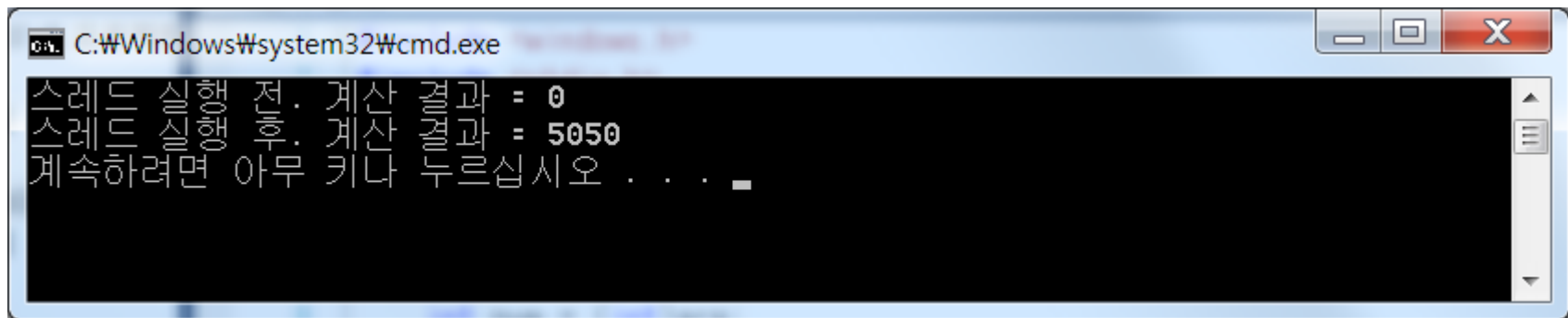
실습 6-3. 스레드 실행 제어와 종료 기다리기 연습

P192 ~

스레드를 이용해 1부터 100까지 합을 구한다.

ResumeThread() 함수를 사용해 재시작.

스레드 계산 완료 확인을 위해 WaitForSingleObject() 함수를 사용



```
C:\Windows\system32\cmd.exe
스레드 실행 전. 계산 결과 = 0
스레드 실행 후. 계산 결과 = 5050
계속하려면 아무 키나 누르십시오 . . .
```

스레드 실행전에는 결과가 틀리지만 실행 후 올바른 결과 출력



❖ 기본 구조

- 스레드 생성과 종료 함수, 스레드 제어 함수 중 스레드 생성함수 사용

```
DWORD WINAPI ProcessClient(LPVOID arg)
{
    // ③ 전달된 소켓 저장, 소켓 타입으로 형변환
    SOCKET client_sock = (SOCKET)arg;

    // ④ 클라이언트 정보 얻기
    addrlen = sizeof(clientaddr);
    getpeername(client_sock, (SOCKADDR *)&clientaddr, &addrlen);

    // ⑤ 클라이언트와 데이터 통신
    while(1){
        ...
    }
    ...
}
```

3. 스레드 함수는 인자로 전달된 소켓을 SOCKET 타입으로 형변환하여 저장
4. getpeername() 함수를 호출해 클라이언트 IP 주소와 포트 번호 획득. IP 주소를 얻어 출력하는데만 사용
5. 클라이언트와 데이터 송/수신



❖ 기본 구조(계속)

```
int main(int argc, char *argv[])
{
    ...
    while(1){
        // ① 클라이언트 접속 수용 (소켓 리턴)
        client_sock = accept(listen_sock, ...);

        ...
        // ② 스레드 생성, 스레드 함수에 소켓을 넘겨줌
        CreateThread(NULL, 0, ProcessClient, (LPVOID)client_sock, 0, NULL);
    }
    ...
}
```

1. 클라이언트가 접속하면 accept() 함수는 클라이언트와 통신할 수 있는 소켓을 리턴
2. 클라이언트와 통신을 담당할 스레드를 생성. 이때 스레드 함수에 소켓을 넘겨줌



❖ 소켓과 연관된 주소 정보 얻기

- 스레드 함수에 소켓만 전달한 경우에는 별도의 주소 정보가 없으므로, 소켓을 통해 주소 정보를 얻는 기능 구현이 필요
- `getpeername()`은 원격 IP/Port 리턴, `getsockname()`은 지역 IP/port 리턴

```
int getpeername (  
    SOCKET s,  
    struct sockaddr *name,  
    int *namelen  
);
```

성공: 0, 실패: `SOCKET_ERROR`

```
int getsockname (  
    SOCKET s,  
    struct sockaddr *name,  
    int *namelen  
);
```

성공: 0, 실패: `SOCKET_ERROR`



실습 6-4. 멀티스레드 TCP 서버 작성과 테스트 P195 ~

* 기존 TCPClient 는 변경할 필요 없음

```

C:\Windows\system32\cmd.exe

[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=4010
[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=4015
[TCP 서버] 클라이언트 접속: IP 주소=127.0.0.1, 포트 번호=4017
[TCP/127.0.0.1:4010] 첫번째 클라이언트
[TCP/127.0.0.1:4015] 두번째 클라이언트
[TCP/127.0.0.1:4017] 세번째 클라이언트 접속

[보낼 데이터] 첫번째 클라이언트
[TCP 클라이언트] 17바이트를 보냈습니다.
[TCP 클라이언트] 17바이트를 받았습니다.
[받은 데이터] 첫번째 클라이언트

[보낼 데이터] 두번째 클라이언트
[TCP 클라이언트] 17바이트를 보냈습니다.
[TCP 클라이언트] 17바이트를 받았습니다.
[받은 데이터] 두번째 클라이언트

[보낼 데이터]

[보낼 데이터] 세번째 클라이언트 접속
[TCP 클라이언트] 20바이트를 보냈습니다.
[TCP 클라이언트] 20바이트를 받았습니다.
[받은 데이터] 세번째 클라이언트 접속

[보낼 데이터]
    
```

❖ 스레드 동기화 필요성

- 멀티스레드를 이용하는 프로그램에서 스레드 두 개 이상이 공유 데이터에 접근하면 다양한 문제가 발생
- 스레드 동기화: 멀티스레드 환경에서 발생하는 문제를 해결하기 위한 일련의 작업

공유 변수

`int money = 1000`

스레드 1

...

① read money into ECX

② ECX = ECX + 2000

③ write ECX into money

...

스레드 2

...

① read money into ECX

② ECX = ECX + 4000

③ write ECX into money

...



❖ 스레드 동기화 기법

- 윈도우에서는 프로그래머가 상황에 따라 적절한 동기화 기법을 선택하여 사용할 수 있도록 API를 제공
- 대표적인 스레드 동기화 기법은 다음과 같다

종류	기능
임계 영역 (critical section)	공유 자원에 대해 오직 한 스레드의 접근만 허용 (한 프로세스에 속한 스레드 간에만 사용 가능)
뮤텍스(mutex)	공유 자원에 대해 오직 한 스레드의 접근만 허용 (서로 다른 프로세스에 속한 스레드 간에도 사용 가능)
이벤트 (event)	사건 발생을 알려 대기 중인 스레드를 깨움
세마포어 (semaphore)	한정된 개수의 자원에 여러 스레드가 접근할 때, 자원을 사용할 수 있는 스레드 개수를 제한
대기 가능 타이머 (waitable timer)	정해진 시간이 되면 대기 중인 스레드를 깨움

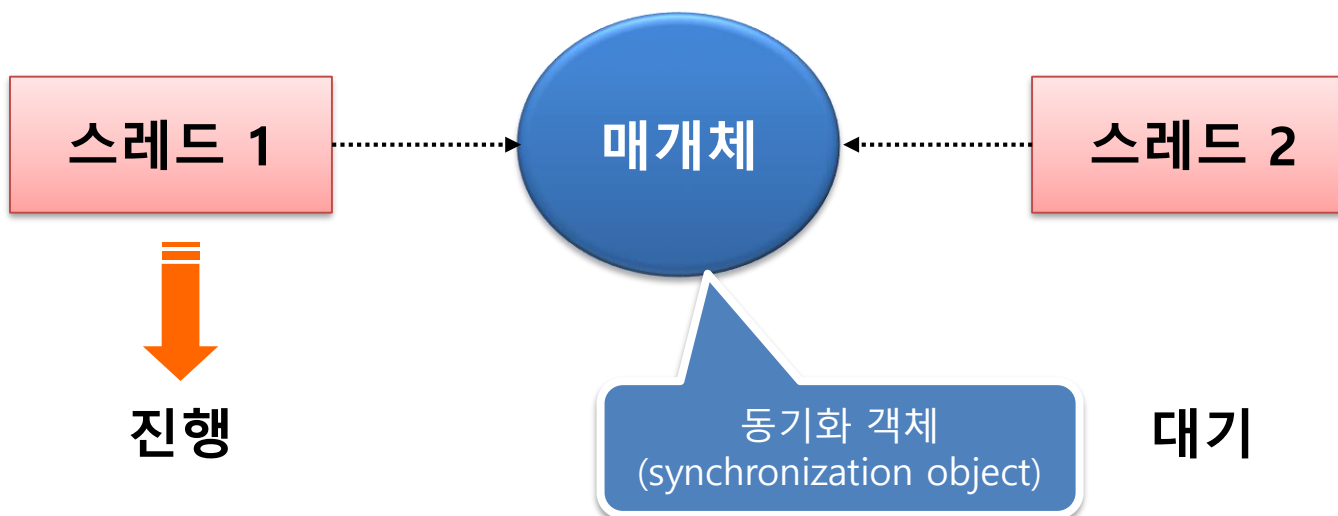


❖ 스레드 동기화가 필요한 상황

- ① 둘 이상의 스레드가 공유 자원에 접근
- ② 한 스레드가 작업을 완료한 후, 기다리고 있는 다른 스레드에 알림

❖ 스레드 동기화 원리

- 스레드를 동기화 하려면 스레드가 상호 작용해야 하므로 중간에 매개체 역할을 하는 부분이 필요
- 두 스레드가 동시에 진행하면 안되는 상황에서 두 스레드는 매개체를 통해 진행 여부를 판단하고 이에 근거해 자신의 실행을 계속할지를 결정



❖ (스레드) 동기화 객체의 특징

- Create*() 함수를 호출하면 커널 메모리 영역에 동기화 객체가 생성되고, 이에 접근할 수 있는 핸들이 리턴됨
- 평소에는 비신호 상태로 있다가 특정 조건이 만족되면 신호 상태가 됨. 비신호 상태에서 신호 상태로 변화 여부는 Wait*() 함수를 사용해 감지(아래 그림 참조)
- 사용이 끝나면 CloseHandle() 함수를 호출



- Wait*() 함수는 스레드 동기화를 위한 필수 함수.
- 동기화 객체 활용시 비신호->신호, 신호->비신호 상태 변화 조건 이해 필요하며, 상황에 맞게 Wait*() 함수를 사용해야 함.



❖ 임계 영역

- 둘 이상의 스레드가 공유 자원에 접근할 때, 오직 한 스레드만 접근을 허용해야 하는 경우에 사용
- 대표적인 스레드 동기화 기법이지만, 생성과 사용법이 달라서 동기화 객체로 분류하지는 않음

❖ 특징

- 프로세스의 유저 메모리 영역에 존재하는 단순한 구조체이므로 한 프로세스에 속한 스레드 간 동기화에만 사용
- 일반 동기화 객체보다 빠르고 효율적



❖ 임계 영역 사용 예

```
#include <windows.h>

CRITICAL_SECTION cs;                                ①

DWORD WINAPI MyThread1(LPVOID arg)
{
    ...
    EnterCriticalSection(&cs);                      ②
    // 공유 자원 접근
    LeaveCriticalSection(&cs);                      ③
    ...
}

DWORD WINAPI MyThread2(LPVOID arg)
{
    ...
```

1. CRITICAL_SECTION 구조체 변수를 전역 변수로 선언.
2. 임계 영역을 사용하기 전에 InitializeCriticalSection() 함수 호출 (초기화)

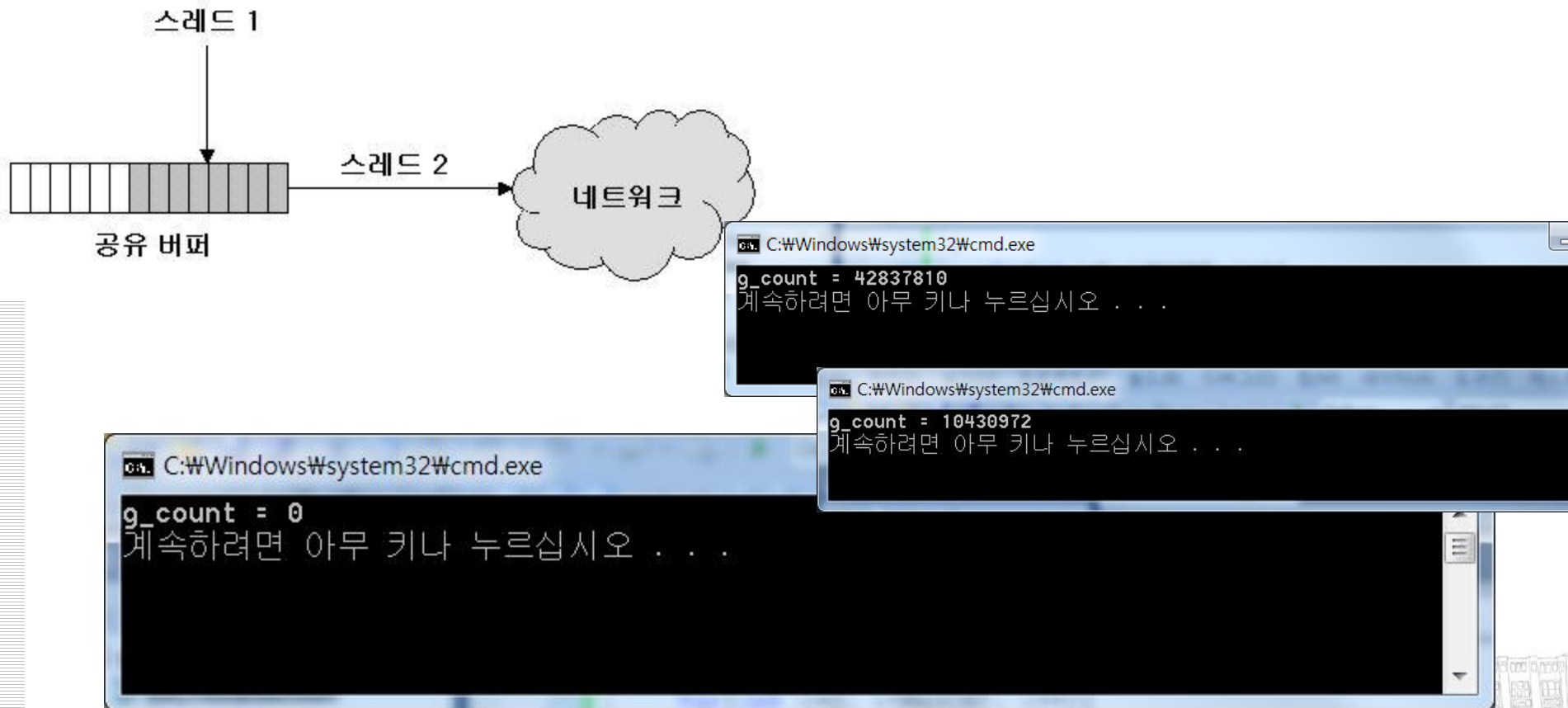
❖ 임계 영역 사용 예(계속)

```
EnterCriticalSection(&cs);  
// 공유 자원 접근  
LeaveCriticalSection(&cs);  
...  
}  
  
int main(int argc, char *argv[])  
{  
    ...  
    InitializeCriticalSection(&cs);  
    // 둘 이상의 스레드를 생성하여 작업을 진행한다.  
    // 생성한 모든 스레드가 종료할 때까지 기다린다.  
    DeleteCriticalSection(&cs);  
    ...  
}
```

3. 공유 자원에 접근하기 전에 EnterCriticalSection() 함수를 호출. 공유자원을 사용하고 있는 스레드가 없다면 EnterCriticalSection() 함수는 곧바로 리턴. 공유자원을 사용하고 있는 스레드가 있다면 EnterCriticalSection() 함수는 리턴하지 못하고 스레드는 대기상태가 됨.
4. 공유 자원 사용을 마치면 LeaveCriticalSection() 함수를 호출. 이때 EnterCriticalSection() 함수에서 대기 중인 스레드가 있다면 하나만 선택되어 깨어남.
5. 임계 영역을 사용하는 모든 스레드가 종료하면 DeleteCriticalSection() 함수를 호출하여 삭제

실습 6-5.임계 영역 연습 P205 ~

* 임계 영역만으로는 어느 스레드가 먼저 리소스를 사용할지 결정할 수 없음.



❖ 이벤트

- 사건 발생을 다른 스레드에 알릴 때 사용
- 한 스레드가 작업을 완료한 후 기다리고 있는 다른 스레드에 알릴 때 사용

❖ 이벤트를 사용하는 전형적인 절차

- ❶ 이벤트를 비신호 상태로 생성
- ❷ 한 스레드가 작업을 진행하고, 나머지 스레드는 이벤트에 대해 `Wait*()` 함수를 호출해 이벤트가 신호 상태가 될 때까지 대기 (sleep)
- ❸ 스레드가 작업을 완료하면 이벤트를 신호 상태로 바꿈
- ❹ 기다리고 있던 스레드 중 하나 혹은 전부가 깨어남 (wakeup)



❖ 이벤트 상태 변경

```
BOOL SetEvent(HANDLE hEvent);    // 비신호 상태 ⇨ 신호 상태  
BOOL ResetEvent(HANDLE hEvent);  // 신호 상태   ⇨ 비신호 상태
```

❖ 이벤트의 종류

■ 자동 리셋 이벤트

- 이벤트를 신호 상태로 바꾸면, 기다리는 스레드 중 하나만 깨운 후 자동으로 비신호 상태가 됨

■ 수동 리셋 이벤트

- 이벤트를 신호 상태로 바꾸면, 기다리는 스레드를 모두 깨운 후 계속 신호 상태를 유지함
- 비신호로 변경이 필요할 시 ResetEvent() 함수를 호출



❖ 이벤트 생성

```
HANDLE CreateEvent (  
    LPSECURITY_ATTRIBUTES lpEventAttributes,  
    BOOL bManualReset,      //TRUE면 수동 리셋, FALSE면 자동 리셋  
    BOOL bInitialState,    // TRUE면 신호, FALSE면 비신호 상태로 시작  
    LPCTSTR lpName         // 이벤트에 부여할 이름  
);
```

성공: 이벤트 핸들, 실패: NULL

1. lpEventAttributes: 핸들 상속과 보안 디스트리뷰터 관련 구조체로 기본값인 NULL 사용
2. bManualReset: TRUE면 수동 리셋, FALSE면 자동 리셋 이벤트
3. bInitialState: TRUE면 신호, FALSE 면 비신호 상태로 시작
4. lpName: 이벤트에 부여할 이름. NULL을 사용하면 이름없는(anonymous) 이벤트가 생성되므로 같은 프로세스에 속한 스레드 간 동기화에만 사용 가능. 서로 다른 프로세스에 속한 스레드간 동기화를 하려면 같은 이름으로 생성해야 함.

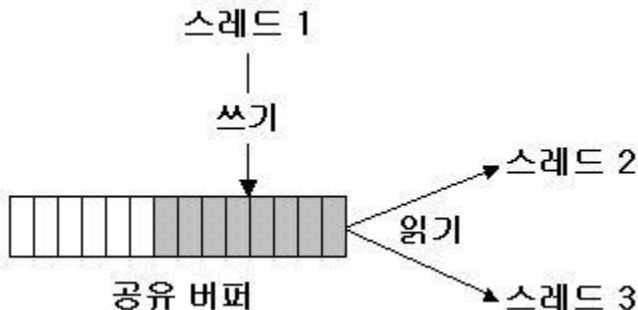


실습 6-6.이벤트 연습

P210 ~

전제 조건:

1. 스레드 1이 쓰기를 완료한 후 스레드 2나 스레드3이 읽을 수 있다. 이때 스레드 2와 스레드 3중 한 개만 버퍼 데이터를 읽을 수 있으며, 일단 한 스레드가 읽기 시작하면 다른 스레드는 읽을 수 없다.
2. 스레드 2나 스레드 3이 읽기를 완료하면 스레드 1이 다시 쓰기를 할 수 있다.



```
C:\Windows\system32\cmd.exe
Thread 7556: 477 477 477 477 477 477 477 477 477 477
Thread 10160: 478 478 478 478 478 478 478 478 478 478
Thread 7556: 479 479 479 479 479 479 479 479 479 479
Thread 10160: 480 480 480 480 480 480 480 480 480 480
Thread 7556: 481 481 481 481 481 481 481 481 481 481
Thread 10160: 482 482 482 482 482 482 482 482 482 482
Thread 7556: 483 483 483 483 483 483 483 483 483 483
Thread 10160: 484 484 484 484 484 484 484 484 484 484
Thread 7556: 485 485 485 485 485 485 485 485 485 485
Thread 10160: 486 486 486 486 486 486 486 486 486 486
Thread 7556: 487 487 487 487 487 487 487 487 487 487
Thread 10160: 488 488 488 488 488 488 488 488 488 488
Thread 7556: 489 489 489 489 489 489 489 489 489 489
Thread 10160: 490 490 490 490 490 490 490 490 490 490
Thread 7556: 491 491 491 491 491 491 491 491 491 491
Thread 10160: 492 492 492 492 492 492 492 492 492 492
Thread 7556: 493 493 493 493 493 493 493 493 493 493
Thread 10160: 494 494 494 494 494 494 494 494 494 494
Thread 7556: 495 495 495 495 495 495 495 495 495 495
Thread 10160: 496 496 496 496 496 496 496 496 496 496
Thread 7556: 497 497 497 497 497 497 497 497 497 497
Thread 10160: 498 498 498 498 498 498 498 498 498 498
Thread 7556: 499 499 499 499 499 499 499 499 499 499
Thread 10160: 500 500 500 500 500 500 500 500 500 500
```



Thank You !

oasis01@gmail.com / rhqudtn75@nate.com