

게임서버프로그래밍

2019년 2학기

한국산업기술대학교
게임공학부

정내훈

강사 소개



경력

1990년부터 온라인게임 개발

LPMUD, Archmage

2002년 3월 – 2008년 2월 NCSoft

MMORPG 개발 : Lineage forever, Alterlife, Blade & Soul

2015,2016,2017 모바일 게임 서버 개발 Netmarble

GSF, FinalShot, Lineage2Revolution

전공

Parallel Processing

관심분야

차세대 게임 서버 구조

연락처

nhjung@kpu.ac.kr 공학관 E동 314호

교재



“게임서버 프로그래밍 교과서” 배현직, 길벗출판사, 2019

성적 산출



중간 고사 : 20%

학기말 고사 : 25%

과제 : 45% (15% 숙제, 30% 팀프로젝트)

출석 : 10%

결석 1번에 1% 감점, 지각 3번에 1% 감점, ¼ 이상 결석 F)

강의 편성



숙제

강의 내용 실습

각 숙제는 연계되어 있음

Delay시 하루당 10%감점

팀 프로젝트

간단한 MMORPG 게임 만들기

숙제와 연계되어 있음

게임서버 프로그래밍 교과서



1장 멀티스레딩

- | | |
|-------------------------------|-----------------------------|
| 1 프로그램과 프로세스 | 10 싱글스레드 게임 서버 |
| 2 스레드 | 11 멀티스레드 게임 서버 |
| 3 멀티스레드 프로그래밍은
언제 해야 할까? | 12 스레드 풀링 |
| 4 스레드 정체 | 13 이벤트 |
| 5 스레드를 다룰 때 주의 사항 | 14 세마포어 |
| 6 임계 영역과 뮤텍스 | 15 원자 조작 |
| 7 교착 상태 | 16 멀티스레드 프로그래밍의
흔한 실수들 |
| 8 잠금 순서의 규칙 | 17 심화 내용 및 더 읽을 거리 |
| 9 병렬성과 시리얼 병목 | |

1.1 | 프로그램과 프로세스



- 프로그램 : 컴퓨터에서 실행되는 명령어와 데이터의 모음이 들어있는 파일
- 프로세스 : 프로그램이 활동을 하는 상태
- 로딩 : 프로그램에 있는 코드와 데이터를 프로세스 메모리로 불러들임
- 윈도(Windows) 운영체제에서는 이렇게 실행된 프로그램을 [작업 관리자](#)로 확인할 수 있음

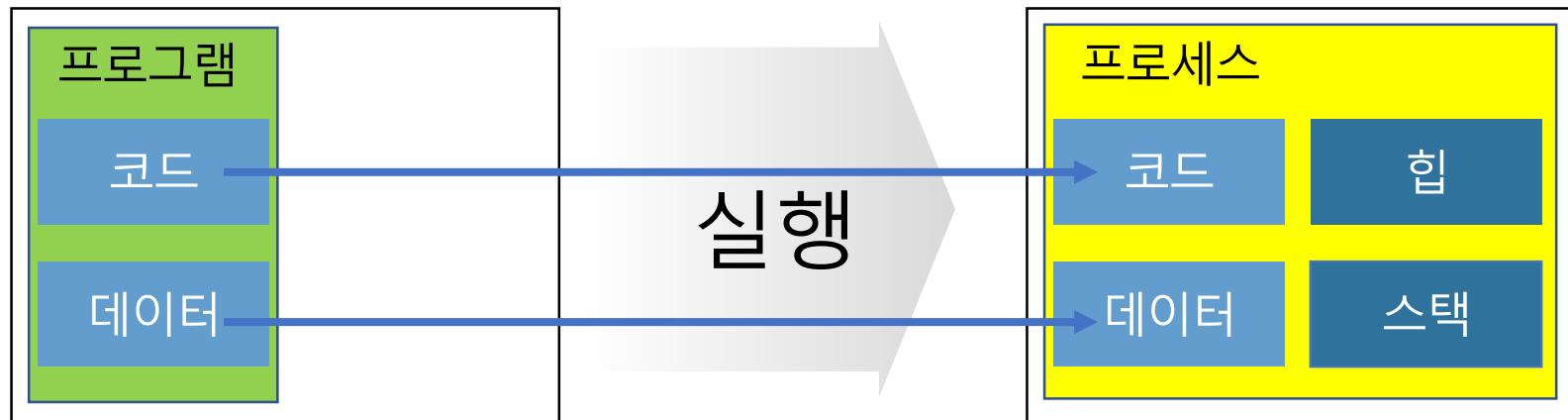


그림 1-1 프로그램은 디스크에 있고, 프로세스는 RAM에 있음

1.1 | 프로그램과 프로세스

- 구조
 - 코드 : 실행용 기계어, 데이터 : 전역 변수/상수, 힙 : 메모리 할당 공간(new/malloc), 스택 : 임시 저장소(함수 복귀 주소/매개 변수/ 지역변수)
- 멀티 프로세싱 : 프로세스가 여러 개 실행되고 있는 것



그림 1-3 멀티프로세싱

1.2 | 스레드

- 일반적으로 많이 쓰는 운영체제는 대부분 스레드(thread)라는 기능을 제공
- 스레드와 프로세스의 차이점
 - 스레드는 한 프로세스 안에 여러 개가 있다
 - 한 프로세스 안에 있는 스레드는 프로세스 안에 있는 메모리 공간을 같이 사용한다
 - 스레드마다 스택을 가진다. 이는 각 스레드에서 실행되는 함수의 로컬 변수들이 스레드마다 따로 있다는 의미.

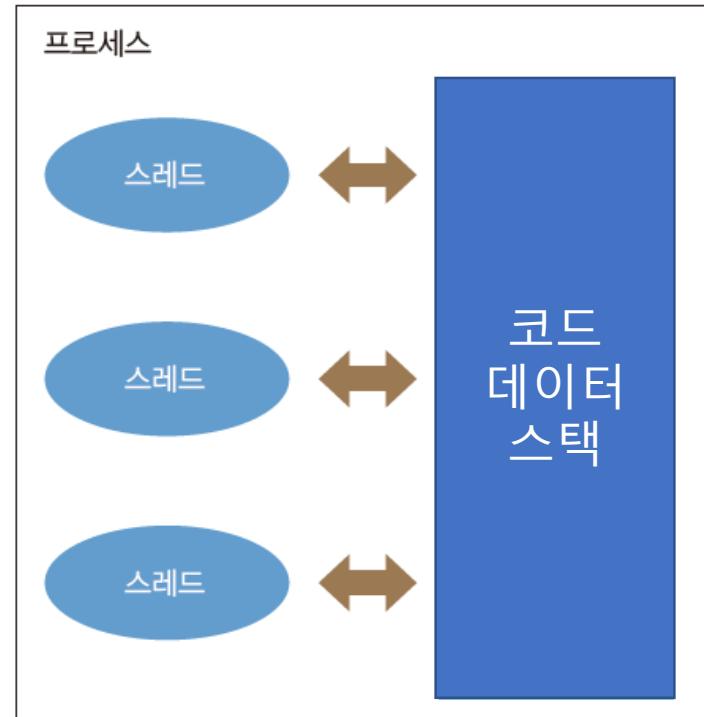
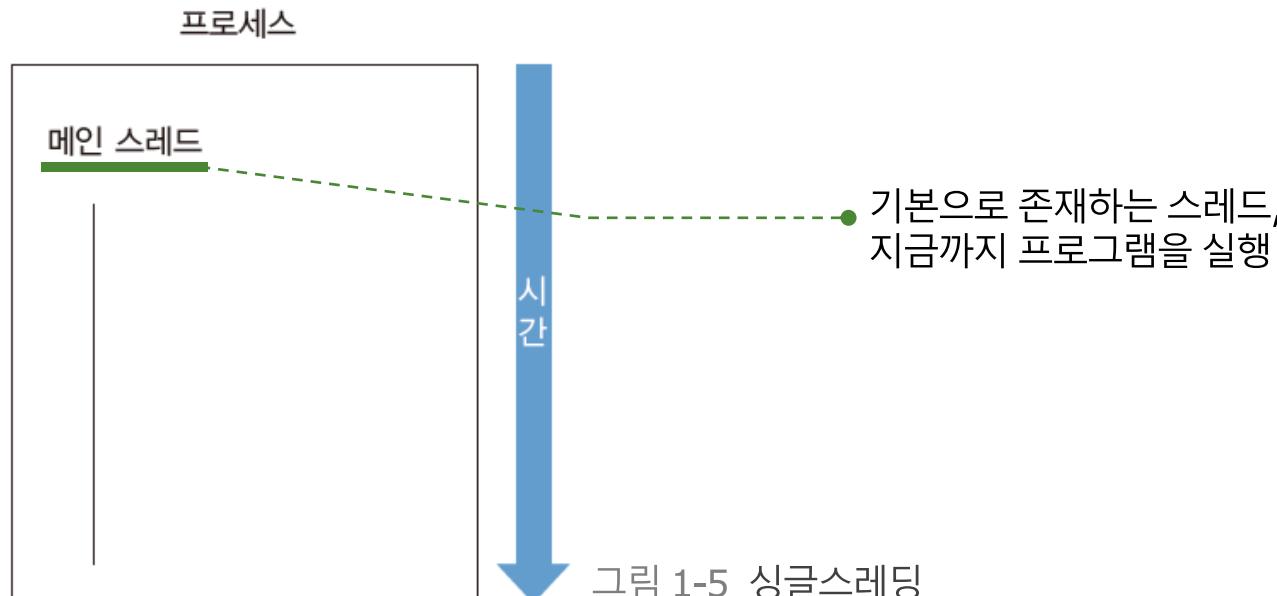


그림 1-4 프로세스와 스레드

1.2 | 스레드

“프로그램을 실행하면 프로세스가 생성됩니다. 프로세스 안에는 유일한 스레드가 있고 그 안에서 프로그램이 실행됩니다.”

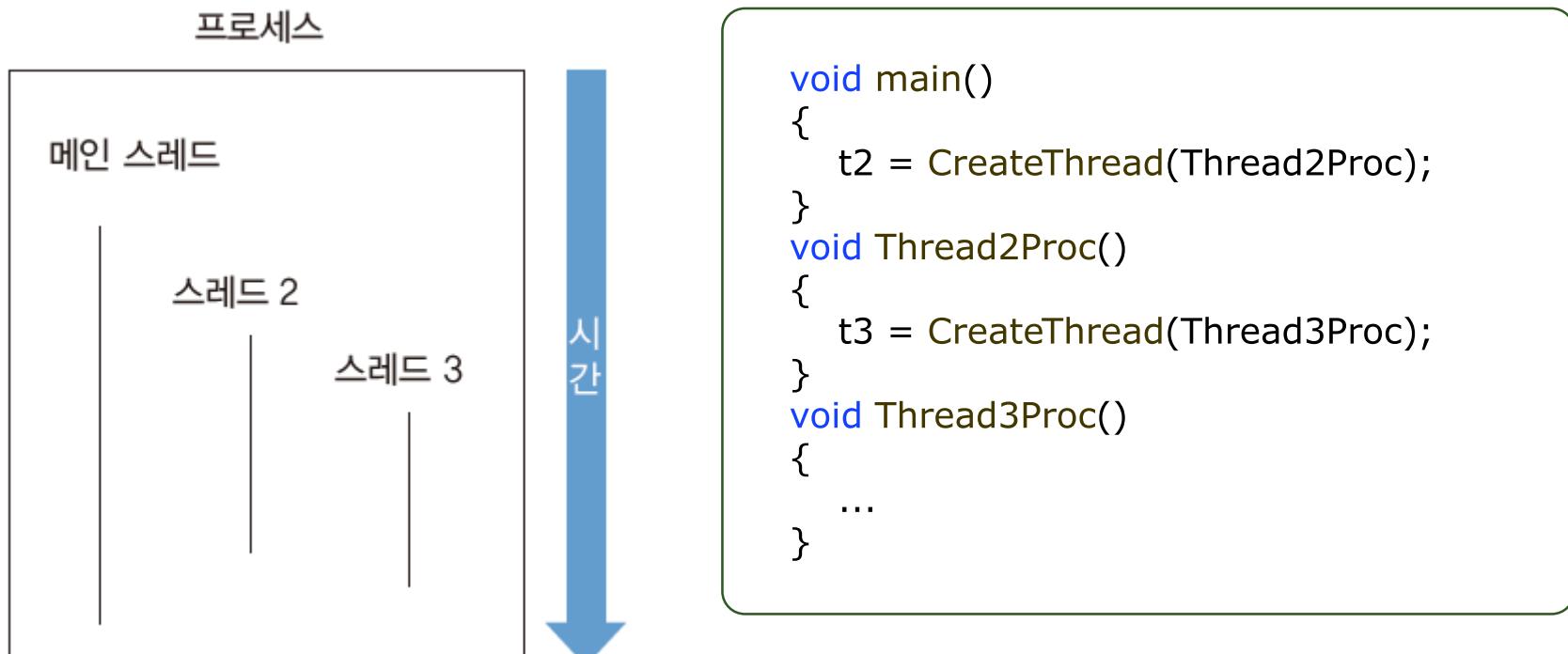
- 싱글 스레드 프로그램 : 동시에 하나만 실행되는 프로그램. 언제든 실행되고 있는 곳이 하나인 프로그램, 실행 궤적을 그리면 하나의 선(thread)이 나오는 프로그램
- 싱글스레드 모델(single threaded model) : 싱글스레드로만 작동하도록 프로그램을 설계하고 구현하는 것



1.2 | 스레드



- 멀티스레드 모델 혹은 멀티스레딩 : 여러 스레드가 동시에 여러 가지 일을 처리하게 하는 것



1.2 | 스레드

- 디버거를 사용해서 호출 스택 확인해 보기(Visual Studio 사용시)

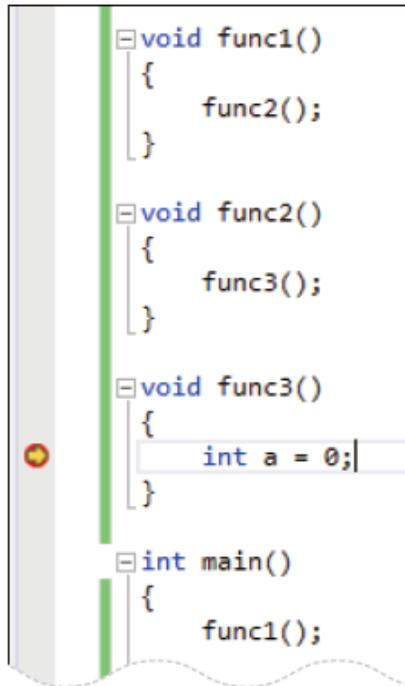


그림 1-7 Visual Studio에서 디버깅 중

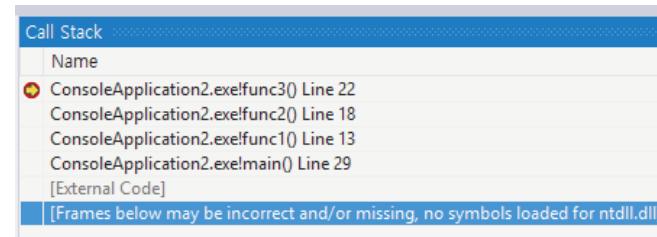


그림 1-8 디버거의 호출스택

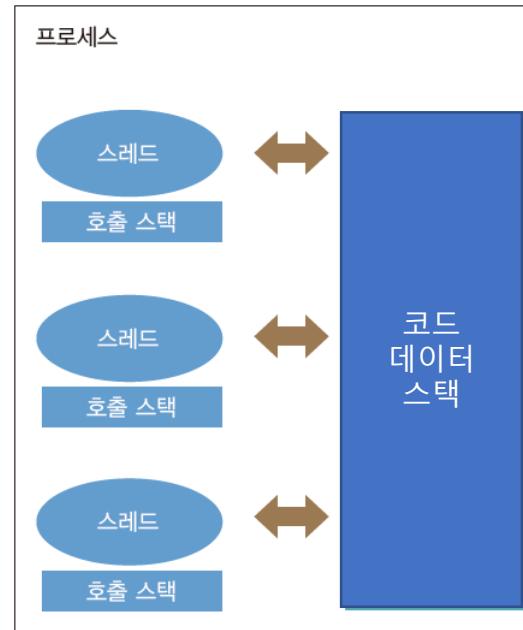


그림 1-9 스레드, 호출 스택, 힙의 관계

1.2 | 스레드



- 메인 스레드와 또 다른 스레드가 동시에 작동하는 예제 프로그램

```

void main()
{
    // ❶ 스레드 시작
    t1 = CreateThread(ThreadProc, 123);
    // ❷ 오랫동안 무슨 일을 처리
    // ...
    // ❸ 스레드가 종료될 때까지 대기
    t1.Join();
    // ❹ 프로그램 실행 끝
}

ThreadProc(int)
{
    // ❺ 오랫동안 무슨 일을 처리
    // ...
    // ❻ 쓰레드(=함수) 실행 끝
}

```

❷와 ❹ 중 어느 것이 먼저 실행될지는 아무도 모른다

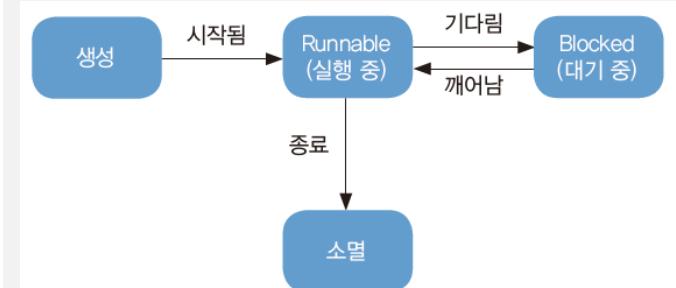


그림 1-10 스레드의 일생

1.2 | 스레드

- 스레드를 생성하는 함수

코드 윈도일 때

```
DWORD threadID;  
ThreadParam threadParam;  
threadParam.value = 123;  
CreateThread(...,  
    ThreadProc, &threadParam,  
    ..., &threadID);
```

코드 리눅스나 유닉스
계열일 때

```
pthread_t threadID;  
pthread_create(&threadID,  
...,  
    ThreadProc, &threadParam);
```

코드 모던 C++

```
std::thread t1(ThreadProc,  
123);
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

- 멀티스레드 프로그래밍을 해야 하는 대표적인 상황

1. 오래 걸리는 일 하나와 빨리 끝나는 일 여럿을 같이 해야 할 때 (x)

=> 이유 : 짧은 작업의 종료시간이 뒤로 밀리는 경우

=> 차라리 오래 걸리는 일을 쪼개서 처리해야 한다.

2. 어떤 긴 처리를 진행하는 동안 다른 짧은 일을 처리해야 할 때 (x)

=> 이유 : 짧은 작업의 반응 속도가 떨어지는 경우

=> 차라리 오래 걸리는 일을 쪼개서 처리해야 한다.

3. 기기에 있는 CPU(다른 말로 Core)를 모두 활용해야 할 때

4. 프로그램을 모듈화 하여 다른 쓰레드에서 독립적으로 실행 (x)

=> 이유 : Readability, Object Oriented

=> 부작용이 너무 크다

**Single Core의 사용량이 100%가 되지 않았다면
멀티쓰레드 프로그래밍을 절대로 하면 안된다.**

1,2,4 번은 이미 멀티쓰레드 프로그래밍에 발을 넣은 경우 적용할 수는 있으나
이것을 이유로 멀티쓰레드 프로그래밍에 발을 들여 넣는 것은 잘못된 일이다.

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

- 1.3.1 오래 걸리는 일 하나와 빨리 끝나는 일 여럿을 같이 해야 할 때
 - 게임 프로그램에서 로딩/loading)을 할 때



그림 1-11 로딩 상황을 그래프로 표현 그림 1-12 지루한 로딩 시간을 달랠 그림 1-13 로딩하는 동안 즐기는
애니메이션 연출 미니 게임
(<스트리트파이터5>)

이미 멀티쓰레딩을 하고 있으면 멀티쓰레드로 구현하는 것이 문제가 없으나, 오로지 이 이유만으로 멀티쓰레드를 도입하는 것은 득보다 실(위험성)이 더 많다

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

멀티스레딩 하지 않을 때

```
LoadScene()
{
    Render();
    LoadScene();
    Render();
    LoadModel();
    Render();
    LoadTexture();
    Render();
    LoadAnimation();
    Render();
    LoadSound();
}
```

- 코드가 지저분함
- 큰파일 로딩하는 경우 일시적으로 프레임률 끊김
- 파일을 부분 단위로 로딩하며 렌더링할 경우 프레임율은 균일하지 못함, 코드가 더 지저분해짐

멀티스레딩 할 때

```
bool isStillLoading; // 전역 변수
```

```
Thread1
{
    isStillLoading = true;
    while (isStillLoading)
    {
        FrameMove();
        Render();
    }
}

Thread2
{
    LoadScene();
    LoadModel();
    LoadTexture();
    LoadAnimation();
    LoadSound();

    isStillLoading = false;
}
```

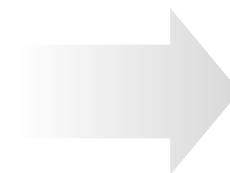
위험

- 렌더링을 지속적으로 수행
- 게임에 필요한 데이터를 디스크에서 로딩, 로딩이 끝나면 특정 변수를 변경
- Thread1에서는 이 변수가 변경될 때까지 로딩 화면을 반복해서 렌더링

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

- 1.3.2 어떤 긴 처리를 진행하는 동안 다른 짧은 일을 처리해야 할 때
 - 플레이어 정보를 읽거나 쓰려고 디스크를 액세스하는 경우

플레이어 정보를 디스크에 기록하는 시간은 1만분의 1초정도
→ 상용 게임 서버 입장에서는 엄청난 시간 낭비
디스크는 모터로 구성된 기계 장치이므로
전자회로보다 일을 훨씬 느리게 함



비동기 I/O를 사용해서
프로세스가 I/O의 완료를
기다리게 않게 한다.

비동기 I/O로 충분히 대처가 가능하고, 멀티쓰레드로
처리하는 것은 너무 위험하다.

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

- 1.3.3 기기에 있는 CPU를 모두 활용해야 할 때
 - 싱글 코어에서는 처리가 불가능 할 때. CPU의 클럭을 몇 배 높여야 해결될 작업 일 때.

- 일인칭 슈팅(First-Person Shooter, FPS)
게임은 클라이언트 하나에 초당 30번의 요청을 처리
- 이러한 클라이언트가 서버 기기에 1만 개 접속할 경우 초당 $30 * 10,000 = 30만$ 번의 처리 필요
→ 최악의 경우 30만 번을 처리해야 할 서버가 초당 1만 번밖에 처리하지 못하는 상황

이경우 처리에 CPU가 필요하기 때문에 멀티쓰레드 이외의 해결책이 없음

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

- 1.3.3 기기에 있는 CPU를 모두 활용해야 할 때
 - 실험 : 소수(2, 3, 5, 7처럼 1과 자신을 제외한 나머지 숫자로는 나누지 않는 정수)를 찾아내서 출력하는 프로그램

```
#include <vector>
#include <iostream>
#include <chrono>

using namespace std;
using namespace chrono;
const int MaxCount = 150000;

bool IsPrimeNumber(int number)
{
    if (number == 1) return false;
    if (number == 2 || number == 3) return true;
    for (int i = 2; i < number - 1; i++)
    {
        if ((number % i) == 0) return false;
    }
    return true;
}
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

```
void PrintNumbers(const vector<int>& primes)
{
    for (int v : primes)
    {
        cout << v << endl;
    }
}

void main()
{
    vector<int> primes;
    auto t0 = chrono::system_clock::now();
    for (int i = 1; i <= MaxCount; i++)
    {
        if (IsPrimeNumber(i))
        {
            primes.push_back(i);
        }
    }
}
```

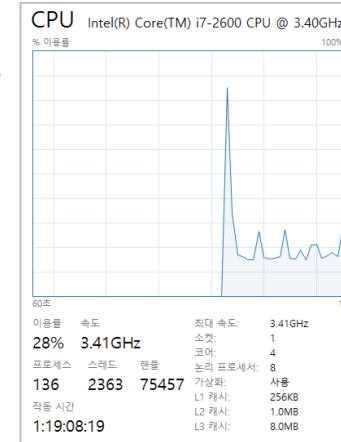
1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

```

auto t1 = system_clock::now();
auto duration = duration_cast<milliseconds>(t1 - t0).count();
cout << "Took " << duration << " milliseconds." << endl;
PrintNumbers(primes);
}
    
```

- 프로그램 실행 : 소수를 계산하는데 걸린 시간 표시

Took 3920 milliseconds.



- MaxCount를 훨씬 큰 수로 바꾼 후 실행하기

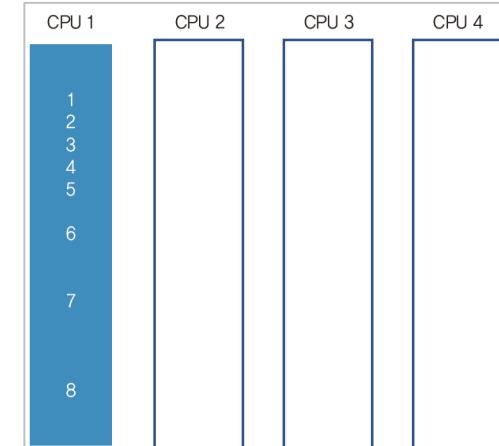


그림 1-14
소수 구하기 때문에
CPU 전체를 쓰지 못하는
상황

그림 1-15
CPU 하나로만 연산을
처리하는 안타까운 상황

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

-문제 개선하기

- 전역 변수 num을 만듭니다.
- 각 스레드는 num에서 값을 하나씩 가져옵니다.** 가져온 값이 소수인지 판별합니다.
- 소수면 배열 primes에 찾은 숫자를 넣습니다.
- 모든 스레드가 일을 마치고 나면 소수를 출력합니다.

-코드로 표현하기

```
main()
{
    vector <thread> threads;

    for (i = 0; I < 4; i++)
        threads.emplace_back(ThreadProc);
    for (auto &th : threads)
        th.join();
    printNumbers(primes);
}
```

```
int num = 1;
vector<int> primes;
TreadProc()
{
    while (num <= 1000000)
    {
        if (Isprime(num))
            primes.push_back(num);
        num++;
    }
}
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

-실제 구동되는 코드 만들기

```
#include <vector>
#include <iostream>
#include <chrono>
#include <thread>
#include <memory>

using namespace std;

const int MaxCount = 150000;
const int ThreadCount = 4;
bool IsPrimeNumber(int number)
{
    if (number == 1) return false;
    if (number == 2 || number == 3) return true;
    for (int i = 2; i < number - 1; i++) {
        if ((number % i) == 0)
            return false;
    }
    return true;
}
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

```
}

void PrintNumbers(const vector<int>& primes)
{
    for (int v : primes) {
        cout << v << endl;
    }
}

int main()
{
    // 각 스레드는 여기서 값을 꺼내 온다.
    int num = 1;

    vector<int> primes;

    auto t0 = chrono::system_clock::now();

    // 작동할 워커 스레드
    vector<thread> threads;

    for (int i = 0; i < ThreadCount; i++)
    {
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

```
threads.emplace_back([&]() {
    // 각 스레드의 메인 함수.
    // 값을 가져올 수 있으면 루프를 돈다.
    while (true)
    {
        int n;
        n = num;
        num++;

        if (n >= MaxCount) break;

        if (IsPrimeNumber(n))
            primes.push_back(n);
    }
}) ;

}

// 모든 스레드가 일을 마칠 때까지 기다린다.
for (auto &thread : threads)
{
```

1.3 | 멀티스레드 프로그래밍은 언제 해야할까?

```

        thread.join();
    }
    // 끝

    auto t1 = chrono::system_clock::now();
    auto duration = chrono::duration_cast<chrono::milliseconds>(t1 - t0).count();
    cout << "Took " << duration << " milliseconds." << endl;

    // PrintNumbers(primes);
}

```

-프로그램을 실행하면 오류가 발생한다

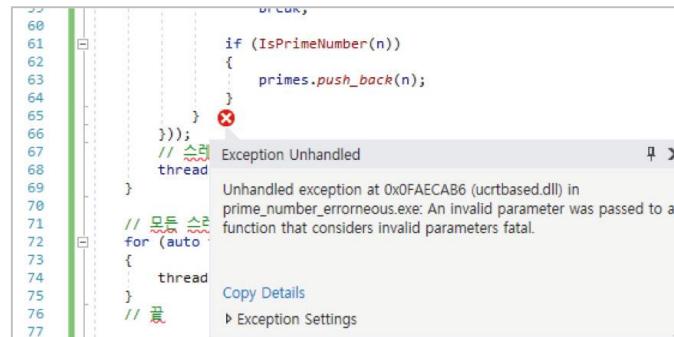


그림 1-17 무작정 시도한 멀티스레드 프로그래밍의 오류

1.4 | 스레드 정체

- “두 가지 일을 동시에 하라”라고 시켰을 때
 - 운영체제는 두개의 작업을 각각 다른 Core에서 실행한다.
 - 코어에 개수가 부족하면 컨텍스트 스위칭을 하면서 시분할 처리를 한다.

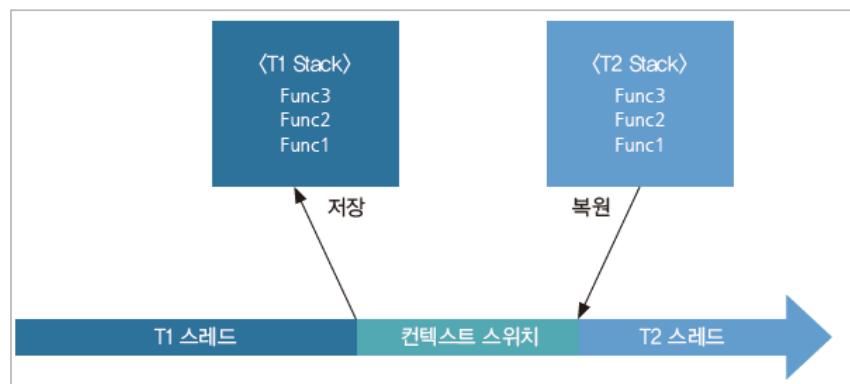
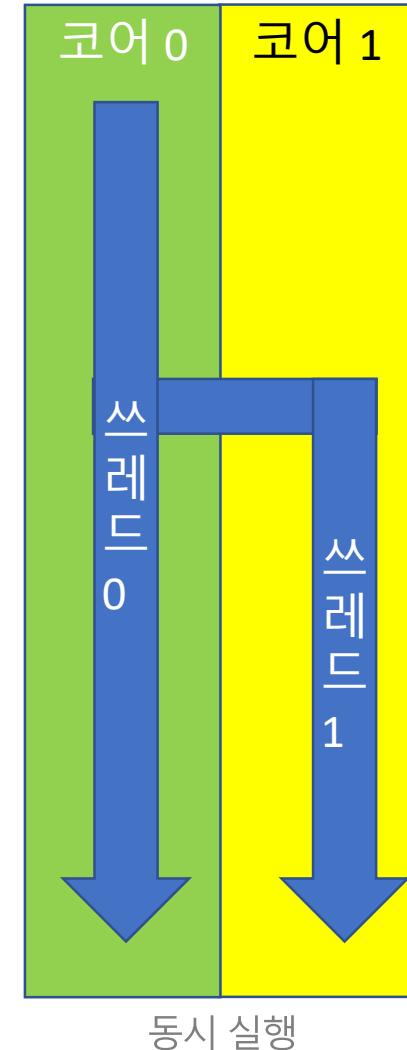


그림 1-19 컨텍스트 스위치



1.5 | 스레드를 다룰 때 주의 사항

- 스레드 2개가 값 하나에 동시에 접근하는 경우

```
x += y;
```

```
x = 2
# 스레드 1
x += 3

# 스레드 2
x += 4
# 기대하는 결과
x = 9
```

코드를
기계어로
컴파일하면

```
t1 = x
t1 = t1 + 3
x = t1
```

원하는 결과

```
x = 2
# 스레드 1
t1 = x      // t1 = 2
t1 = t1 + 3 // t1 = 5
x = t1      // x = 5
```

```
# 스레드 2
t2 = x      // t2 = 5
t2 = t2 + 4 // t2 = 9

x = t2      // x = 9
```

다른 결과
-컨텍스트 스위치가 무작위로(random)
발생

```
x = 2
# 스레드 2 생성
t1 = x      // t1 = 2

t1 = t1 + 3 // t1 =
x = t1      // x = 5

# 스레드 2
t2 = x      // t2 = 2

t2 = t2 + 4 // t2 = 6

x = t2      // x = 6
```

1.5 | 스레드를 다룰 때 주의 사항

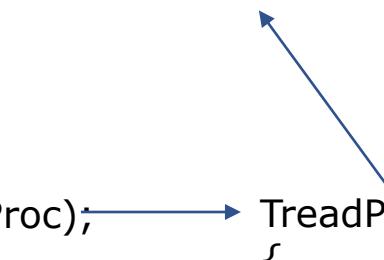
- 소수 구하는 프로그램에서 어느 부분이 잘못되었는지 알아보기

```
main()
{
    vector <thread> threads;

    for (i = 0; i < 4; i++)
        threads.emplace_back(ThreadProc);
    for (i = 0; i < 4; i++)
    {
        threads.join();
    }
    printNumbers(primes);
}

int num = 1;
vector <int> primes;

TreadProc()
{
    while (num <= 1000000)
    {
        if (Isprime(num))
            primes.push_back(num);
        num++;
    }
}
```



1.5 | 스레드를 다룰 때 주의 사항



- 소수 구하는 프로그램에서 어느 부분이 잘못되었는지 알아보기

-num이라는 정보를 보관하는 데 필요한 것은 정수형 4바이트 데이터 공간 하나뿐이지만 vector<int> primes 정보를 보관하는 데 필요한 데이터는 하나 이상

-vector<int>는 배열 객체를 가리키는 포인터 변수와 배열의 크기를 멤버 변수로 가짐. 배열 객체는 메모리 힙에서 할당되었을 것이고, 메모리 힙은 현재 실행 중인 프로세스의 런타임 라이브러리로 다뤄짐

-두 스레드가 동시에 vector<int>의 push_back() 함수를 호출하면 여러 스레드가 vector<int>의 멤버 변수들을 변경, 그러면 두 변수 중 하나만 변경된 상태에서 다른 스레드가 그대로 배열을 액세스함

- 이 과정에서 배열을 가리키는 포인터 변수는 엉뚱한 값, 예를 들어 이미 힙에서 해제된 메모리를 잠시 가리킬 수도 있다 : 충돌이 발생하는 이유(**데이터 레이스** 현상 중 하나)

* 원자성 : 두 멤버 변수를 건드리는 동안에는 다른 스레드가 절대 건드리지 못하게 해야 함

* 일관성 : vector<int>의 두 변수는 항상 일관성 있는 상태를 유지함

- 동기화가 필요 : 원자성과 일관성을 유지하는 특수한 조치, 대표적인 것이 임계 영역, 뮤텍스(상호 배제), 잠금(lock, 락) 기법

데이터 레이스 때문

1.6 | 임계 영역과 뮤텍스

- 경쟁상태를 해결하는 여러 방법 중 하나

“스레드에서 어떤 정보 X를 사용하고 있는 동안 다른 스레드는 X를 건드리지 못하게 한다!”

좀 더 정확히
말하자면

“내가 X를 사용하고 있으면 다른 스레드는 X를 건드리지 않고 기다린다.
현재 스레드가 X를 다 사용할 때까지!”

1.6 | 임계 영역과 뮤텍스

- 뮤텍스 : 상호배제(mutual exclusion)의 줄임말.
 - 뮤텍스를 사용하는 방법
 1. X, Y를 보호하는 뮤텍스 MX를 만듭니다.
 2. 스레드는 X, Y를 건드리기 전에 MX에 “사용권을 얻겠다.”라고 요청합니다.
 3. 스레드는 X, Y를 액세스합니다.
 4. 액세스가 끝나면 MX에 “사용권을 놓겠다.”라고 요청합니다.
 - 뮤텍스 : 코드로 표현하기

뮤텍스의 C++ 표준 객체 std::mutex 사용 예

std::mutex mx; // 1
mx.lock(); // 2
read(x); // 3
write(y); // 3
sum(x, y); // 3
mx.unlock(); // 4

1.6 | 임계 영역과 뮤텍스



- 뮤텍스 : 코드로 표현하기

lock_guard 사용 예

```
std::mutex mx;
{
    lock_guard<mutex> lock(mx);
    read(x);
    write(y);
    sum(x, y);
}
```

이렇게 하면 로컬 변수 lock 객체가 사라질 때
자동으로 mx.unlock()이 실행

- C# 언어에서는 lock과 unlock을 할 때 보호하려는
변수 자체를 뮤텍스처럼 사용하여 잠그거나 변수를
가리키는 임의의 객체를 따로 만들어 그 객체를
잠금(lock).
- 다 쓰고 나면 잠금을 해제 (unlock).

```
// C# 코드
object mx = new object();
lock(mx)
{
    read(x);
    write(y);
    sum(x);
}
```

- C++에서는 보호하려는 변수들을 위한 뮤텍스용 mutex
클래스의 객체를 생성, std::mutex를 사용
- mutex 객체를 lock() 함수로 잠금
- 다 쓰고 나면 unlock() 함수로 잠금 해제
- 또는, lock_guard 템플릿을 사용

```
// C++ 코드
std::mutex mx;
{
    std::lock_guard<std::mutex> lock(mx);
    read(x);
    write(y);
    sum(x, y);
}
```

1.6 | 임계 영역과 뮤텍스



그림 1-24 소수 구하는 프로그램을 멀티스레드로 구현

```
main()
{
    vector <thread> threads;
    for (int i = 0; i < 4; i++)
    {
        threads.emplace_back(ThreadProc);
    }
    for (auto &th : threads)
    {
        th.join();
    }
}

int num = 1;
mutex num_critSec;

vector<int> primes;
mutex primes_critSec;

ThreadProc()
{
    while (1)
    {
        int n;
        lock_guard(num_critSec)
        {
            n = num;
            num++;
            if (num > 1000000)
                break;
        }
        if (IsPrime(n))
        {
            lock_guard(primes_critSec)
            { primes.push_back(n); }
        }
    }
}
```

1.6 | 임계 영역과 뮤텍스



- 실제로 작동하는 코드로 만들기

```
#include <vector>
#include <iostream>
#include <chrono>
#include <thread>
#include <memory>
#include <mutex>

using namespace std;

const int MaxCount = 150000;
const int ThreadCount = 4;

bool IsPrimeNumber(int number)
{
    if (number == 1)
        return false;
    if (number == 2 || number == 3)
        return true;
    for (int i = 2; i < number - 1; i++)
    {
        if ((number % i) == 0)
            return false;
```

1.6 | 임계 영역과 뮤텍스



```
void PrintNumbers(const vector<int>& primes)
{
    for (int v : primes)
    {
        cout << v << endl;
    }
}

void main()
{
    // 각 스레드는 여기서 값을 꺼내 온다.
    int num = 1;
    mutex num_mutex;
    vector<int> primes;
    mutex primes_mutex;

    auto t0 = chrono::system_clock::now();

    // 작동할 워커 스레드
    vector<thread> threads;

    for (int i = 0; i < ThreadCount; i++)
    {
```

1.6 | 임계 영역과 뮤텍스



```
threads.emplace_back( [&]() {
    // 각 스레드의 메인 함수
    // 값을 가져올 수 있으면 루프를 돈다.
    while (true)
    {
        int n;
        {
            lock_guard<mutex> num_lock(num_mutex);
            n = num;
            num++;
        }
        if (n >= MaxCount)
            break;

        if (IsPrimeNumber(n))
        {
            lock_guard<mutex> primes_lock(primes_mutex);
            primes.push_back(n);
        }
    }
});
```

1.6 | 임계 영역과 뮤텍스



```
// 모든 스레드가 일을 마칠 때까지 기다린다.  
for (auto &thread : threads)  
{  
    thread.join();  
}  
// 끝  
  
auto t1 = chrono::system_clock::now();  
auto duration = chrono::duration_cast<chrono::milliseconds>(t1 - t0).count();  
cout << "Took " << duration << " milliseconds." << endl;  
  
// PrintNumbers(primes);  
}
```

-프로그램을 실행하면 다음 결과가 나온다 – 앞 경우보다 훨씬 빨리 실행이 끝남

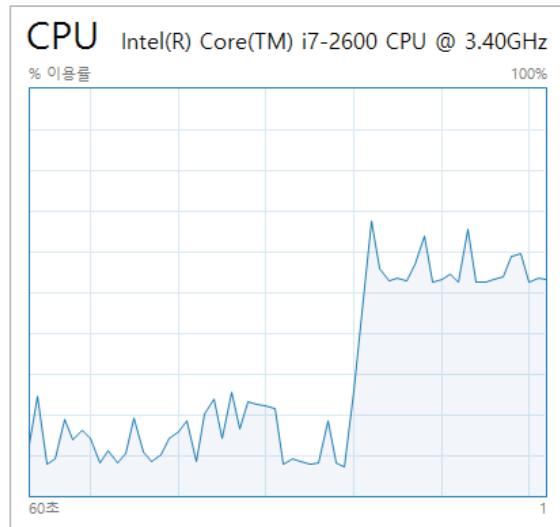
Took 1358 milliseconds.

교재는 mutex가 아니라 recursive_mutex이던데??

⇒ lock이 중첩되지 않는 코드이므로 recursive_mutex를 사용할 필요가 없다.

1.6 | 임계 영역과 뮤텍스

- MaxCount를 크게 올린 후에 CPU 사용량 측정하기



스레드가 하나일 경우 실행시간 : 3900밀리초
스레드가 4개일 경우 실행시간 : 1300밀리초
→ 실제로는 3배만 빠른 이유는?

그림 1-25 CPU 사용량 확인

1.6 | 임계 영역과 뮤텍스



- 스레드의 메인 루프

```
// 각 스레드의 메인 함수
// 값을 가져올 수 있으면 루프를 돈다.
while (true)
{
    int n;
    { //❶
        lock_guard<mutex> num_lock(num_mutex);
        n = num;
        num++;
    }
    if (n >= MaxCount)
        break;

    if (IsPrimeNumber(n))
    { //❷
        lock_guard<mutex> primes_lock(primes_mutex);
        primes.push_back(n);
    }
}
```

❶, ❷ 구간에서 락(lock)을 하고 있어 다른 스레드가 대기 상태로 전환하는 상황이 발생함

그런데 소수를 계산하는 연산은 이구간보다 훨씬 많은 양을 연산.

❶, ❷도 성능 불만족의 이유가 될 수 있지만, 그 비중이 크다고 할 수 없다

CPU 개수가 많더라도 100% 성능을 다 사용하지는 못한다

1.6 | 임계 영역과 뮤텍스



- 뮤텍스를 너무 잘게 나누면
 1. 오히려 프로그램 성능이 떨어짐→뮤텍스를 액세스하는 과정 자체가 무겁기 때문
 2. 프로그램이 매우 복잡해짐, 특히 교착 상태(dead lock) 문제가 쉽게 발생

코드 지나치게 자잘한 임계 영역

```
class Player
{
    mutex m_positionCritSec;
    Vector3 m_position; // ①
    mutex m_nameCritSec;
    string m_name;      // ②
    mutex m_hpCritSec;
    int m_hp;           // ③
}
```

뮤텍스 범위는 적당히 넓게 나누는 것이 좋다.

동시에(여러 CPU가 병렬로) 연산하면 유리한 부분은
잠금 단위를 나누고, 병렬로 하지 않아도 별로 성능에
영향을 주지 않는 부분들은 잠금 단위를
나누지 않는 것이 좋다.

1.7 | 교착 상태

- 교착 상태

두 스레드가 서로를 기다리는 상황

스레드 1은 스레드 2가 하던 일이 끝날 때까지 기다리는데,
 정작 스레드 2는 스레드 1이 하던 일이 끝날 때까지 기다리는데
 → 이러한 경우 두 스레드는 영원히 멈추거 있다

```
int a, b;
mutex m_a, m_b;
```

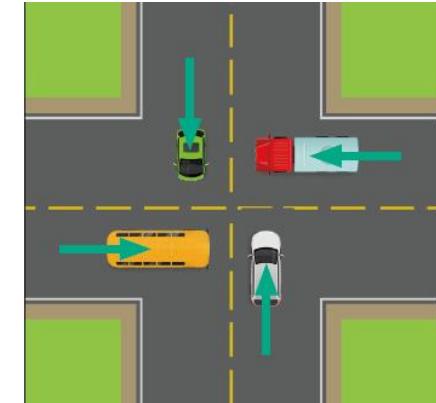
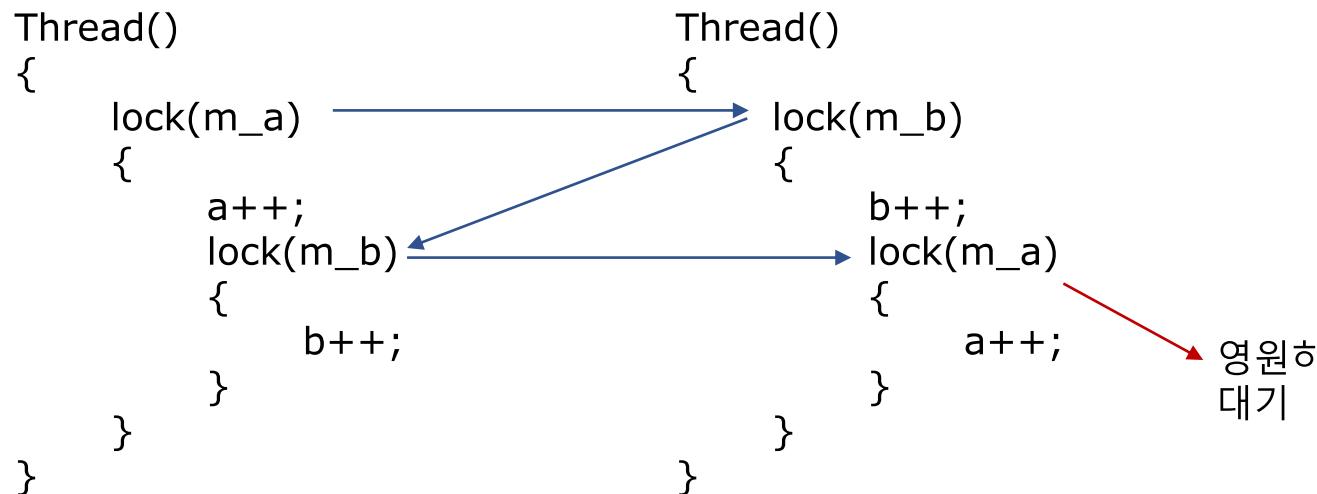


그림 1-26 현실의 교착 상태

그림 1-27 프로그램이 이러한 순서로 실행되면 교착 상태 발생

1.7 | 교착 상태



- 게임 서버에서 교착 상태가 되면 발생하는 현상
 1. CPU 사용량이 현저히 낮거나 0%, 동시접속자 수와 상관없음
 2. 클라이언트가 서버를 이용할 수 없음, 예를 들어 로그인을 못하거나 뭔가 요청을 보냈는데 응답이 오지 않는다.
- CRITICAL_SECTION을 쓰는 방법

CRITICAL_SECTION은 <windows.h>에서 제공하는 mutex 변수의 타입이다. 지금은 C++11의 mutex를 사용하면 된다.

1.7 | 교착 상태

- 교착 상태를 일으켰을 때 디버거로 원인을 찾는 방법 – 예시 코드
- 두 스레드가 그림 1-27의 일을 실제로 하게 만들기

```
int a, b;
mutex a_mutex, b_mutex;

int main()
{
    // t1 스레드를 시작한다.
    thread t1([]()
    {
        while (1)
        {
            lock_guard<mutex> al(a_mutex);
            a++;
            lock_guard<mutex> bl(b_mutex);
            b++;
            cout << "t1 done. \n";
        }
    });
}
```

1.7 | 교착 상태



```
// t2 스레드를 시작한다.  
thread t2([]()  
{  
    while (1)  
    {  
        lock_guard<mutex> bl(b_mutex);  
        b++;  
        lock_guard<mutex> al(a_mutex);  
        a++;  
        cout << "t2 done. \n";  
    }  
});  
  
// 스레드들의 일이 끝날 때까지 기다린다.  
// 사실상 무한 대기상태이므로 끝나지 않는다.  
t1.join();  
t2.join();  
  
return 0;  
}
```

1.7 | 교착 상태

교착상태 발생

```
t1 done.  
(멈춤)
```

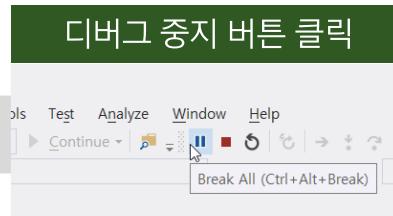
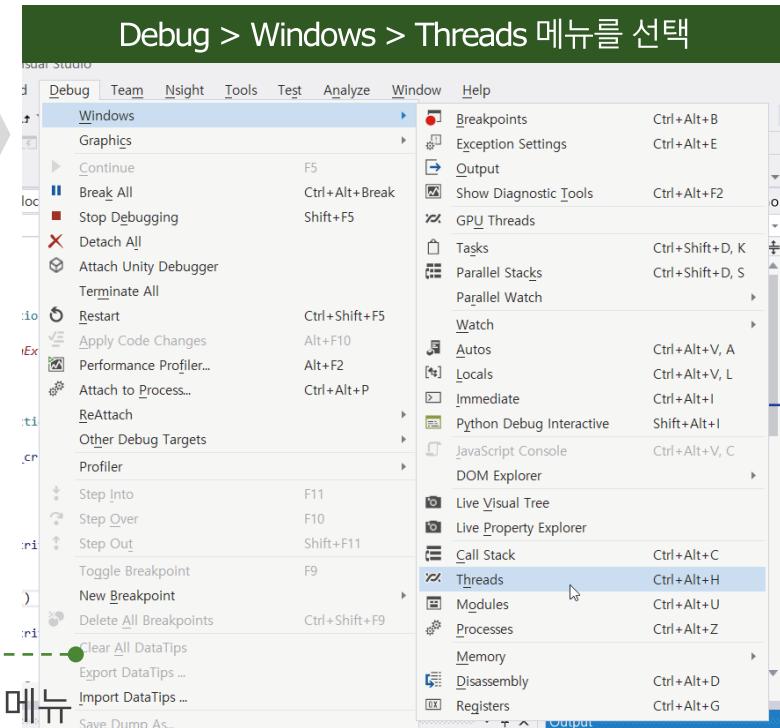


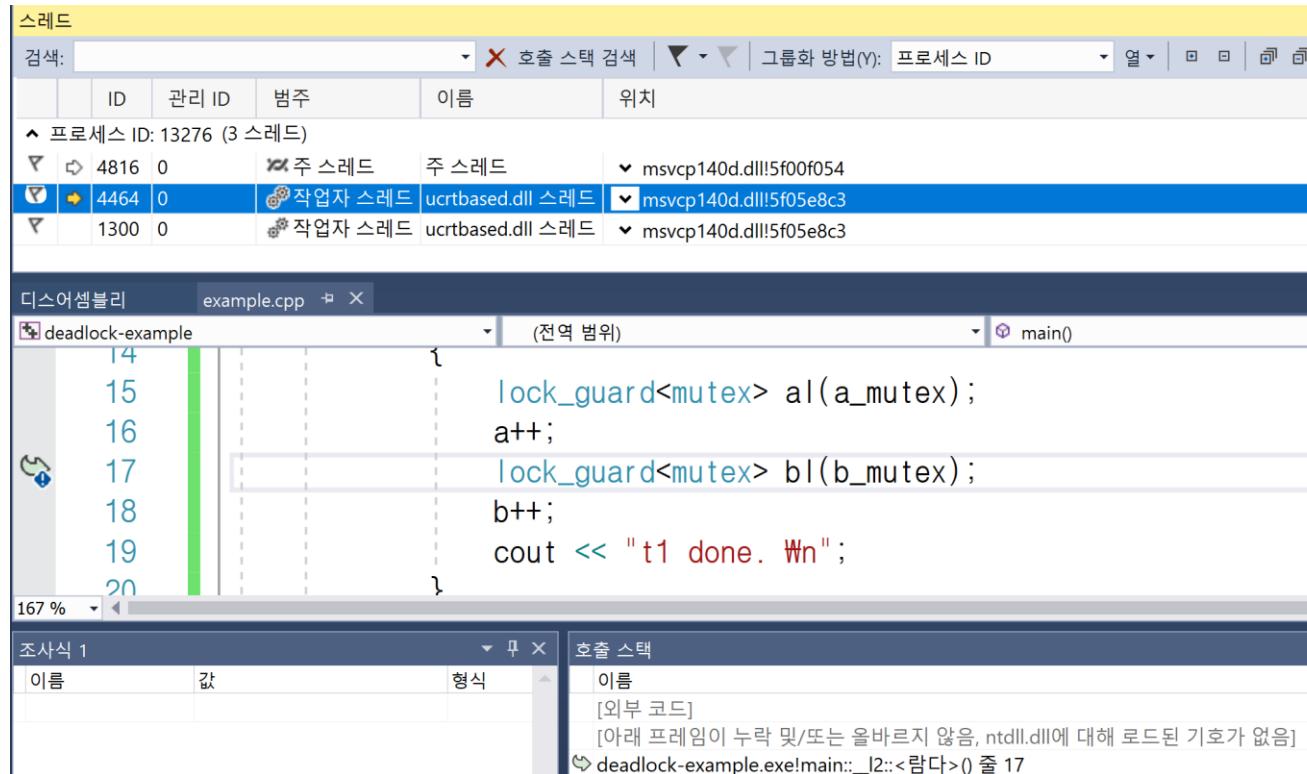
그림 1-28 디버그 중지 버튼

그림 1-29
Debug > Windows > Threads 메뉴

Threads				
	ID	Managed ID	Category	Name
Process ID: 27676 (3 threads)				
▼	11188	0	Main Thread	Main Thread
▼	7156	0	Worker Thread	ucrtbased.dll!thread_start<unsigned int (__cdecl*)(void * _ptr64)>
▼	21048	0	Worker Thread	ucrtbased.dll!thread_start<unsigned int (__cdecl*)(void * _ptr64)>
				▼ msvcp140d.dll!_Thrd_join
				▼ deadlock-example.exe!CriticalSection::Lock
				▼ deadlock-example.exe!CriticalSection::Lock

그림 1-30
디버거가 보여 주는 스레드 2개

1.7 | 교착 상태

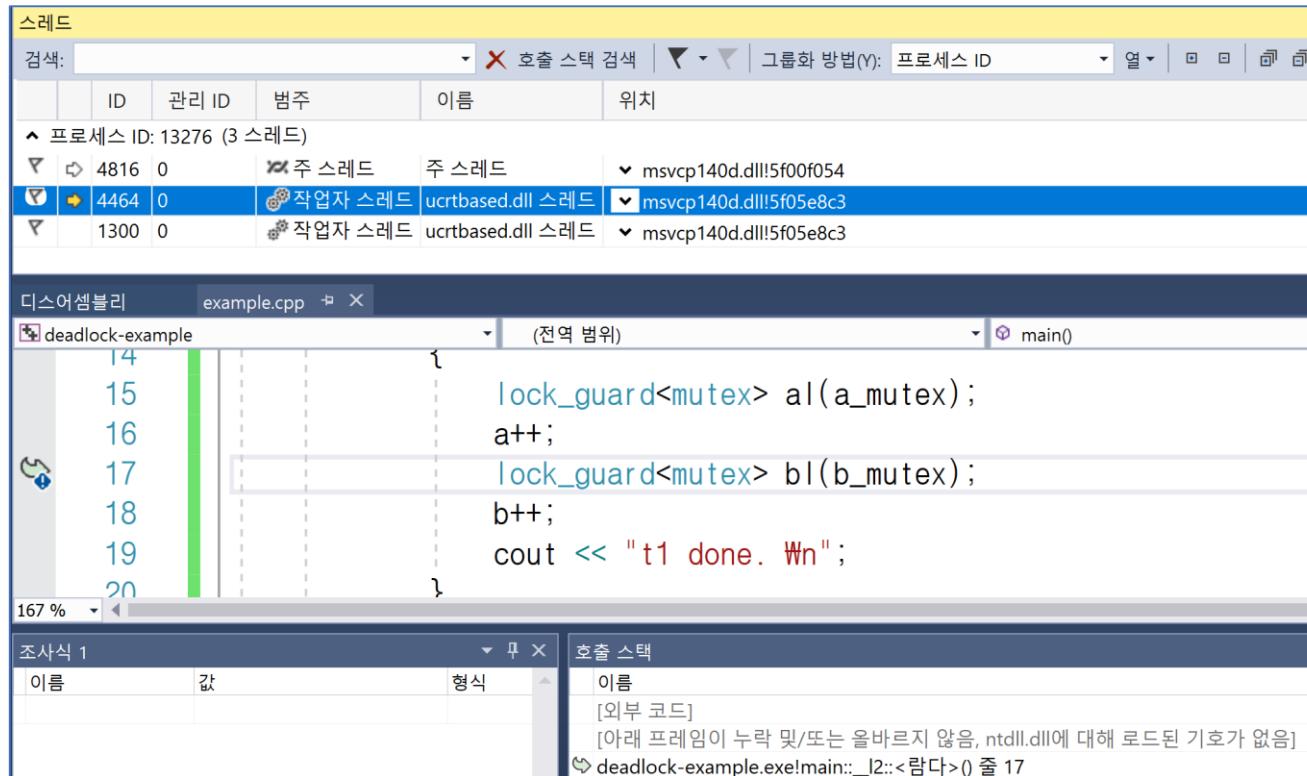


The screenshot shows the WinDbg debugger interface with several windows open:

- 스레드 (Threads) window:** Displays a list of threads for process ID 13276. Thread 4464 is selected, showing it is a worker thread from ucrtbased.dll.
- 디스어셈블리 (Disassembly) window:** Shows assembly code for the main() function of deadlock-example. Lines 14-19 are visible, involving mutex locks and increments.
- 조사식 1 (Registers) window:** Shows register values for the current thread.
- 호출 스택 (Call Stack) window:** Shows the call stack, indicating the current state of the thread.

T1 쓰레드가 b_mutex를 얻기 위해 멈춰있음 (a_mutex를 얻은 상태)

1.7 | 교착 상태



스레드

ID	관리 ID	범주	이름	위치
4816	0	주 스레드	주 스레드	msvcp140d.dll!5f00f054
4464	0	작업자 스레드	ucrtbased.dll 스레드	msvcp140d.dll!5f05e8c3
1300	0	작업자 스레드	ucrtbased.dll 스레드	msvcp140d.dll!5f05e8c3

디스어셈블리 example.cpp

```
deadlock-example
14
15     lock_guard<mutex> a1(a_mutex);
16     a++;
17     lock_guard<mutex> b1(b_mutex);
18     b++;
19     cout << "t1 done. \n";
20 }
```

조사식 1

이름	값	형식

호출 스택

이름
[외부 코드]
[아래 프레임이 누락 및/또는 올바르지 않음, ntdll.dll에 대해 로드된 기호가 없음]
deadlock-example.exe!main::_l2::<람다>()

t1 쓰레드가 b_mutex를 얻기 위해 멈춰있음 (a_mutex를 얻은 상태)

1.7 | 교착 상태

스레드

	ID	관리 ID	범주	이름	위치
▶	4816	0	主线程	主线程	msvcp140d.dll!5f00f054
▶	4464	0	线程池线程	线程池线程	msvcp140d.dll!5f05e8c3
▶	1300	0	线程池线程	线程池线程	msvcp140d.dll!5f05e8c3

디스어셈블리 example.cpp

```

deadlock-example
26
27     lock_guard<mutex> b(b_mutex);
28     b++;
29     lock_guard<mutex> a(a_mutex);
30     a++;
31     cout << "t2 done. \n";
32 }
```

조사식 1

이름	값	형식

호출 스택

이름
[외부 코드]
[아래 프레임이 누락 및/또는 올바르지 않음, ntdll.dll에 대해 로드된 기호가 없음]
deadlock-example.exe!main::_12::<람다>() 줄 29

t2 쓰레드가 a_mutex를 얻기 위해 멈춰있음 (b_mutex를 얻은 상태)

=> 서로 상대방이 갖고 있는 mutex를 얻으려고 함 => 교착 상태

1.8 | 잠금 순서의 규칙



- 여러 뮤텍스를 사용할 때 교착 상태를 예방하려면
각 뮤텍스의 잠금 순서를 먼저 그래프로 그려둔다.
그리고 잠금을 할 때는 잠금 순서 그래프를 보면서 거꾸로 잠근 것이 없는지 체크 해야 한다.

- 뮤텍스 A·B·C의 잠금 순서

A → B → C // ❶

A → B → C 순서로 잠글 때
(교착상태 일으키지 않음)

A.lock()
B.lock()
C.lock()
C.unlock()
B.unlock()
A.unlock()

A → B 순서로 잠글 때
(교착상태 일으키지 않음)

A.lock()
B.lock()
B.unlock()
A.unlock()

B → C 순서로 잠글 때
(교착상태 일으키지 않음)

B.lock()
C.lock()

1.8 | 잠금 순서의 규칙



- 여러 뮤텍스를 사용할 때 교착 상태를 예방하려면

각 뮤텍스의 잠금 순서를 먼저 그래프로 그려둔다.

그리고 잠금을 할 때는 잠금 순서 그래프를 보면서 거꾸로 잠근 것이 없는지 체크 해야 한다.

- 뮤텍스 A·B·C의 잠금 순서

$A \rightarrow B \rightarrow C // ①$

$A \rightarrow B \rightarrow C$ 순서로 잠글 때
(교착상태 일으키지 않음)

A.lock()
B.lock()
C.lock()
C.unlock()
B.unlock()
A.unlock()

$A \rightarrow B$ 순서로 잠글 때
(교착상태 일으키지 않음)

A.lock()
B.lock()
B.unlock()
A.unlock()

$B \rightarrow C$ 순서로 잠글 때
(교착상태 일으키지 않음)

B.lock()
C.lock()

B, C만 잠글 때 (교착상태 일으키지 않음)

B.lock()	// B를 잠갔다.
B.unlock()	// B가 해제되었다.
C.lock()	// C를 잠갔다. B가 이미 해제되었으므로 C만 잠근 셈이다.
C.unlock()	// C가 해제되었다.

$A \rightarrow B$ 순서로 잠글 때 (교착상태 일으키지 않음)

A.lock()
C.lock()
C.unlock()
A.unlock()

1.8 | 잠금 순서의 규칙

B → A 순서로 잠글 때
(교착상태 일으킴)

B.lock()
A.lock()
A.unlock()
B.unlock()

C → A 순서로 잠글 때
(교착상태 일으킴)

C.lock()
A.lock()
A.unlock()
C.unlock()

재귀 뮤텍스(recursive mutex) 한 스레드가 뮤텍스를 여러 번 반복해서 잠그는 것을 원활하게 처리

```
M.lock() // 잠금을 획득했다.  
M.lock() // 잠근 것을 또 잠갔다.  
M.unlock() // 잠금이 해제되었다. 그러나 아직 한 번 더 남았다.  
M.unlock() // 잠금이 해제되었다. 비로소 잠금 해제가 실질적으로 된다.
```

1.8 | 잠금 순서의 규칙



- 재귀 잠금

A → B → C → B → A

```

A.lock()    // ❶
B.lock()    // ❷ 첫 잠금, 잠금 순서 그래프 지킴
C.lock()    // ❸
B.lock()    // ❹
A.lock()    // ❺ 이미 잠근 것을 또 잠그는 것이므로
C.unlock()  // ❻ 잠금 순서 그래프 무시 가능
B.unlock()  // ❼
A.unlock()  // ❽

```

A → C → B → C → A

```

A.lock()    // ❶ ● 안전함
C.lock()    // ❷ ● 거꾸로 가지 않았으므로 안전
B.lock()    // ❸ ● 교착 상태 발생(B → C 어김)
C.lock()    // ❹ ● 재귀 잠금으로 안전
A.lock()    // ❺
C.unlock()  // ❻
B.unlock()  // ❼
A.unlock()  // ❽

```

모두 안전하지만 ❸의 코드 한 줄 때문에 교착상태

“교착 상태를 예방하려면 첫 번째 잠금 순서를 지켜야 한다(거꾸로 가지 말아야 한다).”

1.9 | 병렬성과 시리얼 병목



- **병렬성과 시리얼 병목**

병렬성(parallelism) : 여러 CPU가 각 스레드의 연산을 실행하여 동시 처리량을 올리는 것

시리얼 병목(serial bottleneck) : 병렬로 실행되게 프로그램을 만들었는데 정작 한 CPU만 연산을 수행하는 현상

- 소수 구하는 프로그램의 각 스레드가 하는 일(의사 코드)

1. num을 잠근다.
2. num에서 값을 하나 가져온다.
3. num을 잠금 해제한다.
4. num이 소수인지 판별한다.
5. 소수면 primes를 잠근다.
6. primes에 소수를 넣는다.
7. primes를 잠금 해제한다.

lock(num) n = num num++ unlock(num)	IsPrimeNumber (n)	lock(primes) primes.add(n) unlock(primes)
--	----------------------	---

그림 1-35 시간이 지나면서 스레드가 하는 일

1.9 | 병렬성과 시리얼 병목

그림 1-36 한 CPU를 제외하고는 모두가 노는 상황

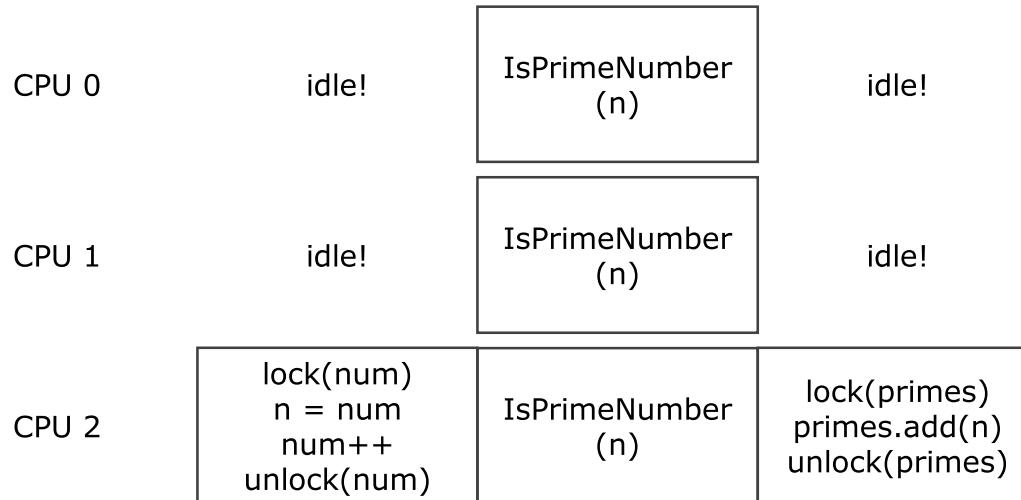


그림 1-36 한 CPU만 일하고
나머지 99개는 노는 상황

CPU 0~98 idle!

idle!

CPU 99

1.9 | 병렬성과 시리얼 병목

- 암달의 저주 줄이기

암달의 법칙 (Amdahl's Law) 혹은 암달의 저주 : CPU 개수가 많을수록 총 처리 효율성이 떨어지는 현상
암달의 저주를 줄이려면 시리얼 병목이 발생하는 구간을 최소로 줄여야 한다

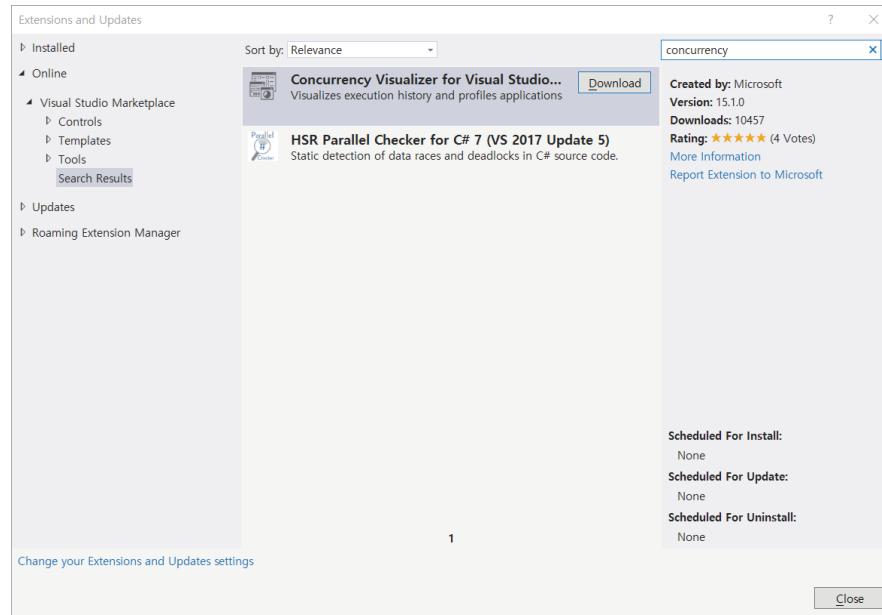


그림 1-38 Concurrency Visualizer 설치

1.9 | 병렬성과 시리얼 병목

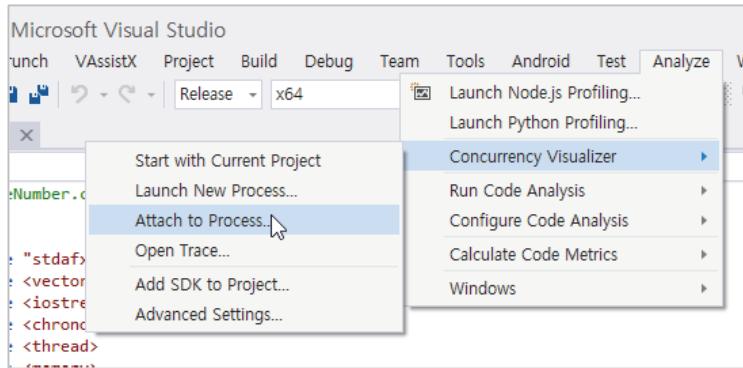


그림 1-39

Analyze > Concurrency Visualizer > Attach to Process 선택

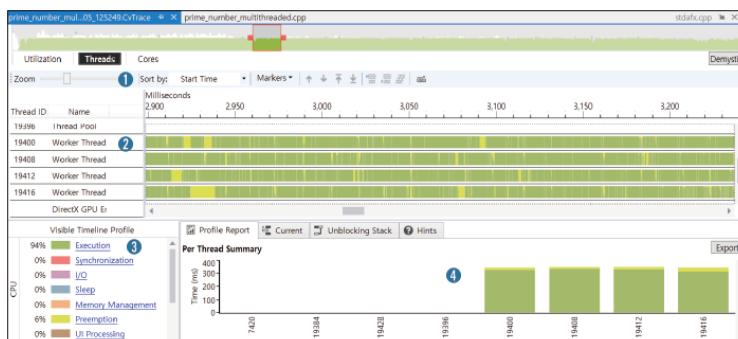


그림 1-40

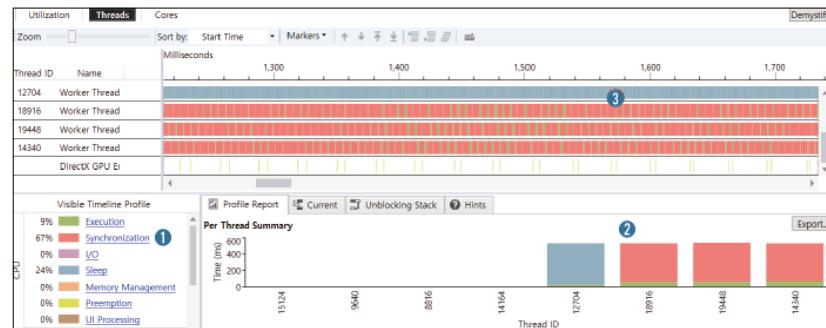
Concurrency Visualizer 화면

- ➊ 새로운 창이 뜨면 누름
- ➋ 각 스레드가 무슨 일을 했는지 그래프로 보여 줌
- ➌ 해당 구간 동안 각 스레드가 한 일의 통계를 보여 줌
- ➍ 스레드 4개가 각각 무엇을 했는지 보여 줌

1.9 | 병렬성과 시리얼 병목

- 암달의 저주

```
lock_guard<recursive_mutex>
primes_lock(primes_mutex);
Sleep(1);
primes.push_back(n);
```



- 연산을 하는 데 사용한 시간은 9%
- 을 클릭하고 빨간색(Synchronization)이나 회색(Sleep)을 클릭하면 어떤 스레드가 무엇 때문에 잠금이 발생하고 잠을 자는지 등을 확인할 수 있다

그림 1-41 많은 시간이 Lock 대기로 낭비되고 있는 상황 표시

1.9 | 병렬성과 시리얼 병목

```
A.lock();
ReadFromDisk(X)
A.unlock();
```

이코드는 엄청난 병목을 야기한다. DISK I/O는 시간이 많이 걸리므로 락A를 기다리는 스레드들이 너무 많이 기다리게 된다.



```
Object Local_X;
ReadFromDisk(Local_X)
A.lock();
X = Local_X;
A.unlock();
```

이렇게 수정하면 I/O로 인한 바틀넥을 해소할 수 있다. 하지만 추가 버퍼 메모리와 메모리 복사 시간을 요구한다.
우리는 나중에 이것을 비동기 I/O로 해결할 수 있음을 알 수 있을것이다.

1.9 | 병렬성과 시리얼 병목

```
A.lock();
X = A.GetBufferPtr();
A.unlock()          // ①
ReadFromDisk(X)    // ②
A.lock()
X.Something();     // ②
A.unlock()
```

이러한 방식의 프로그램은 위험하다. ②에서 X를 액세스한다. 하지만 X는 이미 사라진 객체일 가능성이 있다. ①의 unlock(A)가 실행된 후에 다른 스레드가 A를 변화시킬 수 있기 때문.

1.10 | 싱글 스레드 게임 서버

- 싱글스레드 서버를 구동하는 경우 CPU 개수만큼 프로세스를 띄우는 것이 일반적
 - 방 개수만큼 스레드나 프로세스가 있으면 스레드나 프로세스 간 컨텍스트 스위치의 횟수가 증가
 - 따라서 같은 동시접속자를 처리하는 서버라고 하더라도 실제로 처리할 수 있는 동시접속자 수를 크게 떨어뜨림.

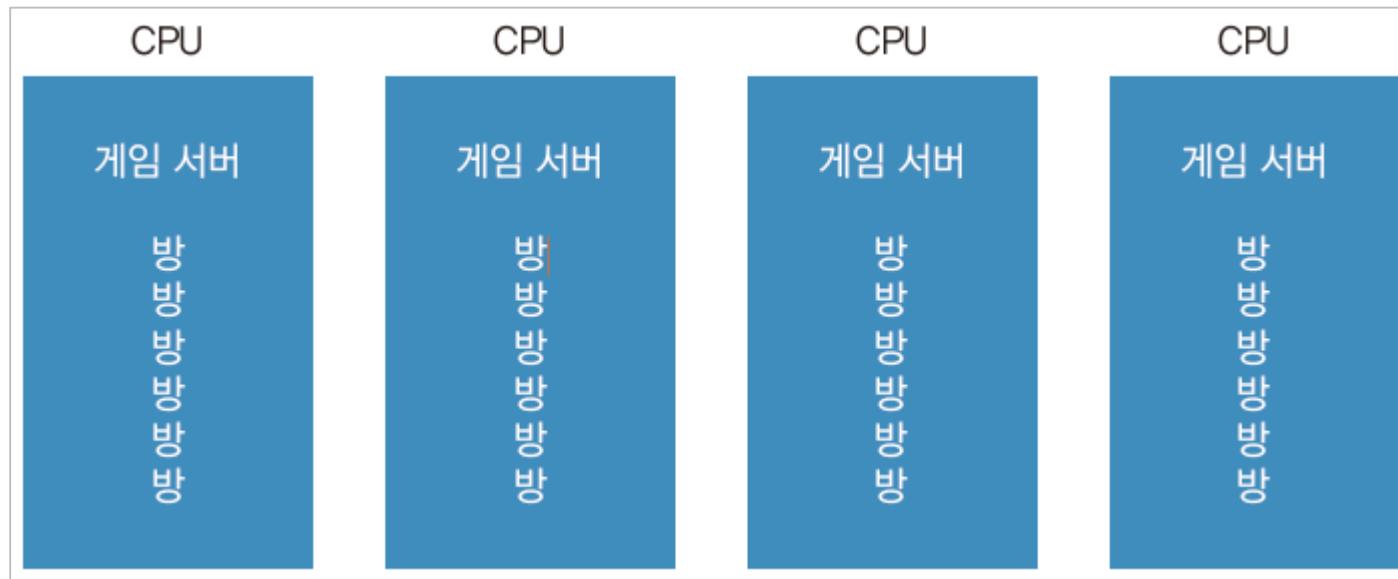


그림 1-42 싱글스레드, 멀티프로세스

1.11 | 멀티 스레드 게임 서버

- 멀티스레드로 서버를 개발하는 경우 (1,2,3,4는 해결가능하나, 5는 멀티스레드가 아니면 해결 불가능 => 5의 경우가 아니면 멀티스레드로 가면 안됨)
 1. 서버 프로세스를 많이 띄우기 곤란할 때. 예를 들어 프로세스당 로딩해야 하는 게임 정보(맵 데이터 등)의 용량이 매우 클 때
(특히 MMO 게임 서버)
 2. 서버 한 대의 프로세스가 여러 CPU의 연산량을 동원해야 할 만큼 많은 연산을 할 때
 3. 코루틴이나 비동기 I/O 함수를 쓸 수 없고 디바이스 타임이 발생할 때
 4. 서버 인스턴스를 서버 기기당 하나만 두어야 할 때
- 5. 서로 다른 방이 같은 메모리 공간을 액세스해야 할 때

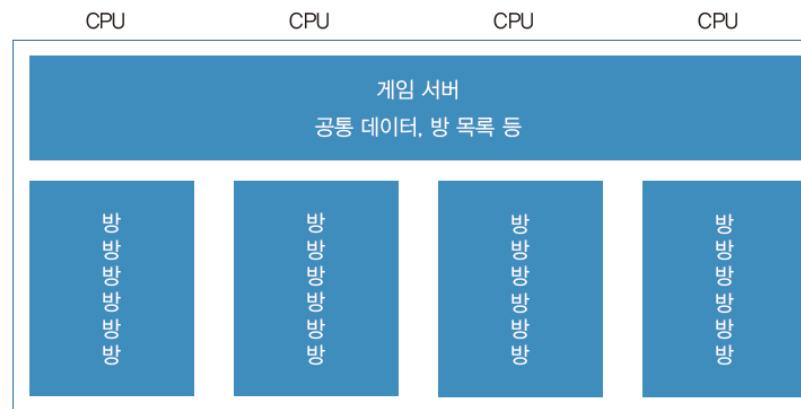


그림 1-43 한 프로세스에 멀티스레드로 작동하는 게임 서버의 모델

1.11 | 멀티 스레드 게임 서버

그림 1-44 게임 서버 메인, 게임방, 뮤텍스 선언

싱글스레드 서버

```
class MyGameServer
{
    class Room
    {
        String m_roomName;
        List<Player> m_players;
        List<Character> m_characters;
    }
    map<PlayerID, shared_ptr<Room>>
        m_roomList;
    String m_serverName;
}
```

멀티 스레드 서버

```
class MyGameServer
{
    class Room
    {
        // 각 게임방 안의 데이터들을 보호한다.
        mutex m_critSec;
        String m_roomName;
        List<Player> m_players;
        List<Character> m_characters;
    }
    map<PlayerID, shared_ptr<Room>>
        m_roomList;
    String m_serverName;
    // 서버 메인과 방 목록을 보호한다.
    // 단 방 안의 데이터는 보호하지 않는다.
    mutex m_critSec;
}
```

1.11 | 멀티 스레드 게임 서버

- 게임 서버에서 플레이어 A에 대한 처리를 할 때의 작동법
 1. 공통 데이터(방 목록 등)를 잠금
 2. 플레이어 A가 들어 있는 방을 방 목록에서 찾는다.
 3. 공통 데이터를 잠금 해제
 4. 찾은 방을 잠금
 5. 플레이어 A의 방 안에서 처리
 6. 방을 잠금 해제

```
MyGameServer.DoSomething(playerID)
{
    m_critSec.lock();
    room = m_roomList.find(player);
    m_critSec.unlock();
    room.m_critSec.lock();
    room.DoSomething(playerID);
    room.m_critSec.unlock();
}
```

멀티스레드 게임 서버를 만들 때 크게 주의할 점은
시리얼 병목과 교착 상태
특히 파일을 액세스할 때
자주 잠그는 뮤텍스를 잠근 채로 액세스하는 경우
성능 저하가 자주 발생

1.12 | 스레드 풀링

- 멀티스레드 모델의 게임 서버를 개발할 때 스레드는 몇 개 만들고, 각 스레드는 무엇을 위해 일을 하게 만들면 좋을까?

- 어떤 서버의 주 역할이 CPU 연산만 하는 스레드라면(즉, 디바이스 타임이 없다면) 스레드 풀의 스레드 개수는 서버의 CPU개수(= 모든 코어의 개수)와 동일하게 잡아도 충분함
- 서버에서 데이터베이스나 파일 등 다른 것에 액세스하면서 디바이스 타임이 발생할 때 스레드 개수는 CPU 개수보다 많아야 함
- 작업이 없으면 스레드들은 스레드 풀에서 대기하고 있어야 하고(block상태), 작업이 들어오면 깨어나서 작업을 실행해야 한다. (대기의 구현??? => 이벤트)



그림 1-45 화장실 문(스레드)과
기다리는 사람들(이벤트)

```
// 서버에서 사용자 인증 요청을 처리하는 함수
void RequestLogin(userName, password)
{
    // 사용자 정보를 파일에서 불러온다.
    // 파일 액세스가 끝날 때까지는
    // 하드디스크가 바쁘고, CPU는 논다.
    string prof = File.GetUserProfile(userName);
    if (password == prof.password)
    {
        // 사용자 인증 처리
    }
}
```

1.13 | 이벤트



- 이벤트 : 잠자는 스레드를 깨우는 도구
 - Reset: 이벤트가 없음, 정수 값으로 표현하자면 0
 - Set: 이벤트가 있음, 정수 값으로 표현하자면 1

```
Event event1;

void Thread1()
{
    // 이벤트가 신호를 일으킬 때까지
    // 기다린다.
    event1.Wait();
}

void Thread2()
{
    // 이벤트에 신호를 준다.
    event1.SetEvent();
}
```

윈도우의 이벤트 관련 함수

- CreateEvent: 이벤트를 생성
- CloseHandle: 이벤트를 파괴
- WaitForSingleObject: 이벤트를 기다림
- SetEvent: 이벤트에 신호를 줌

1.13 | 이벤트

- 앞 페이지 이벤트의 문제점 : C++11 표준이 아님.
- C++11에서의 이벤트는? `future/promise`, `condition_variable` 같은 것으로 구현
- 이벤트는 할 작업이 없는 스레드를 재워서 CPU의 낭비를 막을 수 있으나 (아니면 작업이 생길 때 까지 루프를 실행하면서 기다려야 한다 <= Busy Waiting)
- 스레드의 실행 상태(`running`, `blocked`)를 바꿔야 하기 때문에 운영체제 레벨에서 구현이 되어야 하고, 운영체제 호출 오버헤드를 갖는다.
- 따라서, 자주 실행되는 코드에서는 성능 문제로 사용하면 안된다.

1.13 | 이벤트

- C++11의 이벤트 : future/promise
 - 대기하고 있는 쓰레드를 깨울 때 원하는 데이터를 전달할 수 있다.
 - 주의 : 재사용 불가, 반드시 1:1 통신

```
#include <future>

promise<int> event1;

void Thread1()
{
    future<int> data = event1.get_future();
    data.wait();
    cout << "이벤트 값 : " << data.get() << endl;
}

void Thread2()
{
    // 이벤트에 신호를 준다.
    event1.set_value(999);
}
```

future/promise관련 함수

- get_future: 이벤트를 생성
- get : 데이터를 받음
- wait: 이벤트를 기다림
- set_value: 이벤트에 신호와 데이터를 줌

1.13 | 이벤트



- C++11의 이벤트 : condition_variable
 - 다대다 통신 가능, 재사용 가능, data race에서 변수 보호 가능

```
#include <condition_variable>
condition_variable event;
void Thread1()
{
    mutex mx;
    unique_lock<mutex> ul(mx);
    event.wait(ul);
    cout << "이벤트 접수\n";
}
void Thread2()
{
    // 이벤트에 신호를 준다.
    event.notify_one();
}
```

condition_variable 관련 함수

- wait: 이벤트를 기다림
- notify_all : 모든 기다리는 이벤트를 깨움
- notify_one: 기다리는 이벤트들 중 하나를 깨움

mutex가 존재하는 이유

- wait가 종료했을 때 Data Race에서 데이터를 보호하기 위해

1.13 | 이벤트

- 스레드 둘 이상이 이벤트를 기다리는 경우

```
#include <condition_variable>
condition_variable event;
void Thread1()
{
    mutex mx;
    unique_lock<mutex> ul(mx);
    event.wait(ul);
    cout << "thread1 이벤트 접수\n";
}
void Thread2()
{
    mutex mx;
    unique_lock<mutex> ul(mx);
    event.wait(ul);
    cout << "thread2 이벤트 접수\n";
}
void Thread3()
{
    // 이벤트에 신호를 준다.
    event.notify_one();
}
```

결과를
특정 할 수
없다

1.13 | 이벤트



- `notify_all` 기능을 사용

```
#include <condition_variable>
condition_variable event;
void Thread1()
{
    mutex mx;
    unique_lock<mutex> ul(mx);
    event.wait(ul);
    cout << "thread1 이벤트 접수\n";
}
void Thread2()
{
    mutex mx;
    unique_lock<mutex> ul(mx);
    event.wait(ul);
    cout << "thread2 이벤트 접수\n";
}
void Thread3()
{
    // 이벤트에 신호를 준다.
    event.notify_all();
}
```

모든 스레드가 성공적으로
깨어남

주의 :
`notify_all()`이 실행된 이후의
`wait()`는 계속 기다린다.
`notify_one()`도 마찬가지이다.

1.14 | 세마포어

- 세마포어는 원하는 개수의 스레드가 자원을 액세스할 수 있게 함

```
Semaphore sema1;

void Main()
{
    // 스레드 2개만 자원을 액세스할 수 있게 제한한다.
    sema1 = new Semaphore(2);
}

void Thread1()
{
    // 리소스를 액세스할 수 있을 때까지 기다린다.
    sema1.Wait();

    // 리소스 액세스가 다 끝났음을 세마포어에 알린다.
    sema1.Release();
}

void Thread2()
{
    // 리소스를 액세스할 수 있을 때까지 기다린다.
    sema1.Wait();
```

이 코드는 다음과 같이 실행됨

- 세마포어가 있으며 스레드 2개만 액세스를 허락
- 그리고 스레드 3개가 세마포어에 액세스를 요청.
- 그러면 스레드 2개만 액세스를 허가받고 나머지를 실행.
- 일을 마친 스레드는 세마포어에 액세스가 끝났음을 통보.
- 먼저 액세스를 허가받던 스레드 중 하나가 액세스 끝을 통보하는 순간 대기하고 있던 나머지 스레드가 액세스를 허가받음.

1.14 | 세마포어

```
// 리소스 액세스가 다 끝났음을 세마포어에 알린다.  
sema1.Release();  
}  
  
void Thread3()  
{  
    // 리소스를 액세스할 수 있을 때까지 기다린다.  
    sema1.Wait();  
  
    // 리소스 액세스가 다 끝났음을 세마포어에 알린다.  
    sema1.Release();  
}
```

세마포어 관련 함수(윈도)

CreateSemaphore : 세마포어를 생성
자원을 몇 개 허락하는지도
이때 설정

WaitForSingleObject: 세마포어가 자원 액세스를
요청하고, 허락할 때까지 기다림

ReleaseSemaphore: 세마포어에 자원 액세스가
끝났음을 통보

CloseHandle: 세마포어를 파괴.

1.14 | 세마포어



1.14.1 세마포어의 또 다른 용도

세마포어는 이벤트와 여러모로 비슷함.

다만 이벤트는 상태 값이 0과 1로 제한되지만, 세마포어는 0 이상의 아무 값이나 가질 수 있다는 차이가 있다.

```
Queue queue;  
Event queueIsEmpty;  
  
void Thread1()  
{  
    while (true)  
    {  
        queueIsEmpty.Wait();  
        queue.PopFront();  
    }  
}  
  
void Thread2()  
{  
    while (true)  
    {  
        queue.PushBack();  
        queueIsEmpty.SetEvent();  
    }  
}
```

- 이벤트를 하나 준비, 이 이벤트는 '큐에 뭔가가 있다'를 알리는 역할
- 스레드 1은 이벤트를 기다리고 큐에서 항목을 꺼냄.
- 스레드 2는 큐에 항목을 넣고 이벤트에 신호를 줌.
- 스레드 1·2는 이 일을 계속 반복.

1.14 | 세마포어



- 1.14.1 세마포어의 또 다른 용도_이 코드의 문제점
 - 스레드 1에서 이벤트를 기다림. 이벤트 상태 값은 0, 큐는 비어 있음.
 - 스레드 2에서 이벤트에 신호를 줌. 이벤트 상태 값은 1 이 됨. 큐에 항목이 1개 있음.
 - 스레드 1이 깨어남. 이벤트 상태 값은 0이됨, 큐에서 항목을 꺼내면 큐에 항목이 0개가 됨.
 - 나머지

표 1-1 정상적인 처리일 때

처리	이벤트 상태	큐 항목 개수
스레드 1에서 이벤트 대기 후 큐에서 꺼낸다	0	0
스레드 2에서 큐에 넣고 이벤트에 신호를 준다.	1	1
스레드 1에서 이벤트 대기 후 큐에서 꺼낸다.	0	0
스레드 2에서 큐에 넣고 이벤트에 신호를 준다.	1	1
스레드 1에서 이벤트 대기 후 큐에서 꺼낸다.	0	0

표 1-2 비정상적인 처리일 때

처리	이벤트 상태	큐 항목 개수
스레드 1에서 이벤트 대기 후 큐에서 꺼낸다.	0	0
스레드 2에서 큐에 넣고 이벤트에 신호를 준다.	1	1
스레드 2에서 큐에 넣고 이벤트에 신호를 준다	1	2
스레드 1에서 이벤트 대기 후 큐에서 꺼낸다.	0	1

1.14 | 세마포어

1.14.1 세마포어의 또 다른 용도_문제해결

이벤트 대신 세마포어로 대체하고, 세마포어의 초기값을 0으로 설정하기.

이벤트에 신호를 주는 대신 세마포어에 “자원 액세스가 다 끝났다.”라고 통보하기.

```
Queue queue;  
Semaphore queueIsEmpty;  
  
void Main()  
{  
    // 초기값이 0인 세마포어를 만든다.  
    queueIsEmpty = new Semaphore(0);  
}  
  
void Thread1()  
{  
    while (true)  
    {  
        queueIsEmpty.Wait();  
        queue.PopFront();  
    }  
}  
  
void Thread2()
```

```
    }  
    while (true)  
    {  
        queue.PushBack();  
        queueIsEmpty.Release();  
    }  
}
```

세마포어는 원래 용도와 다른
곳에서도 활용할 수 있다

1.14 | 세마포어

- 1.14 세마포어의 문제

- 너무 simple하고 파워풀 하다. => C 언어의 "goto"와 같다. 쓰기 편하지만 조금이라도 복잡한 것을 구현하려 한다면 여러 개의 세마포어를 복잡하게 연결시켜서 구현해야 하고, 이는 프로그램의 readability를 심각하게 해손한다.

=> 같은 이유로 C++11 표준에 존재하지 않는다.

- Simple하지만 다른 event제어 연산과 마찬가지로 운영체제를 호출해야 해서 실행 오버헤드가 크다.

1.15 | 원자 조작

- 원자 조작(atomic operation)

뮤텍스나 임계 영역 잠금 없이도 여러 스레드가 안전하게 접근할 수 있는 것을 의미

원자 조작은 하드웨어 기능이며, 대부분의 컴파일러는 원자 조작 기능을 쓸 수 있게 함

원자 조작은 32비트나 64비트의 변수 타입에 여러 스레드가 접근할 때 한 스레드씩만 처리됨을 보장

- 원자성을 가진 값 더하기
- 원자성을 가진 값 맞바꾸기
- 원자성을 가진 값 조건부 맞바꾸기
- 변수에 무슨 값이 있는지는 모르지만, 그 값에 특정 값을 더하고 그 값의 결과를 얻어 온다

`atomic <int> a = 0;`

변수 선언

`int r = atomic_fetch_add(&a, 3);`

“a에 뭐가 들었는지는 모르지만 여하튼 우리는 거기에 3을 더할 것이다.
우리가 원하는 것은 a에 정확히 3이라는 값이 추가되는 것이다. 그리고
추가되기 전 a가 뭔지 알고 싶다.”

`int r = atomic_exchange(&a, 10);`

변수 값과 내가 원하는 값을 서로 맞바꿈
a의 과거 값이 r에 채워지고, a에는 10이 들어감

`bool r = atomic_compare_exchange_strong(&a, &b, 100);`

“a와 b가 같을 때만 a를 100으로 바꾸고 true를 리턴해라. 같지 않으면 a의 값을 b에 넣고 false를 리턴해라.”

1.16 | 멀티스레드 프로그래밍의 혼한 실수들



• 1.16.1 읽기와 쓰기 모두에 잠금하지 않기

메모리에 값을 쓰고 있는데 다른 스레드가 그것을 읽으면 망가질 수 있다는 것은 인지하지만,
그냥 값을 읽고만 있으면 잠금을 하지 않더라도 안전하다는 막연한 생각 때문에 발생

```
int a;
bool flag = false;

void func1()
{
    while (false == flag);
    cout << a;
}

void func2()
{
    a = 100;
    flag = true;
}
```

func1()에서 읽어 들이는 flag 값이
정상적이지 않다는 버그가 있다

1.16 | 멀티스레드 프로그래밍의 혼한 실수들

- 1.16.1 읽기와 쓰기 모두에 잠금하지 않기

해결책 : mutex를 사용하던가 flag을 volatile로 선언

```
int a;  
volatile bool flag = false;  
  
void func1()  
{  
    while (false == flag);  
    cout << a;  
}  
  
void func2()  
{  
    a = 100;  
    flag = true;  
}
```

그래도 func1()에서 읽어 들이는 a 값이 매우 낮은 확률로 정상적이지 않다는 버그가 있다



atomic <int> a;
와 같이 atomic한 변수를 사용해야 한다.

1.16 | 멀티스레드 프로그래밍의 혼한 실수들

- 1.16.1 읽기와 쓰기 모두에 잠금하지 않기

메모리에 값을 쓰고 있는데 다른 스레드가 그것을 읽으면 망가질 수 있다는 것은 인지하지만,
그냥 값을 읽고만 있으면 잠금을 하지 않더라도 안전하다는 막연한 생각 때문에 발생

```
volatile bool done = false;
volatile int *bound;

void func1()
{
    for (int j = 0; j <= 25000000; ++j) *bound = -(1 + *bound);
    done = true;
}
void func2()
{
    while (!done) {
        int v = *bound;
        if ((v != 0) && (v != -1)) cout << "error" ;
    }
}
```

func2()에서 읽어 들이는 *bound 값이
가끔 정상적이지 않다는 버그가 있다

1.16 | 멀티스레드 프로그래밍의 혼란 실수들

- 1.16.2 잠금 순서 꼬임

프로그램 규모가 커지면 1.8절의 규칙을 준수하기가 생각보다 어려움, 제일 좋은 것은 잠금 순서 규칙을 최대한 적게 유지하는 것

```
int a;
mutex a_mutex;

int b;
mutex b_mutex;

void func1()
{
    a_mutex.lock();
    a...
    b_mutex.lock();
    b...
}

void func2()
{
    b_mutex.lock();
    b...
    a_mutex.lock();
    a...
}
```

1.16 | 멀티스레드 프로그래밍의 혼란 실수들



• 1.16.3 너무 좁은 잠금 범위

잠금 객체 범위가 너무 넓으면 컨텍스트 스위치가 발생할 때 운영체제가 해야 할 일이 매우 많아지고 처리 병렬성이 떨어지기 때문에 멀티스레드 프로그래밍의 이유가 퇴색되기도 함. (X)

=> 잠금 객체와 운영체제는 관계 없다.

잠금 범위를 좁히면 컨텍스트 스위치의 확률이 떨어지기는 하나 임계 영역 잠금이 컨텍스트 스위치보다는 훨씬 적더라도 단순한 산술 연산보다는 더 많은 처리 시간을 차지하므로 임계 영역을 적당한 수준에서 나누면 좋다. (X)

=> 잠금 범위와 컨텍스트 스위치는 관계가 없다. 잟은 잠금 연산으로 인한 성능 저하가 단점이다.

```
class A
{
    int a;
    mutex a_mutex;

    int b;
    mutex b_mutex;
};
```

클래스 멤버 변수 각각에 잠금을 하고 있음
→지나치게 잘게 자른 케이스

```
class A
{
    int a;
    int b;
    mutex mutex;
};
```

잠금 하나로 모든 멤버 변수를 보호

1.16 | 멀티스레드 프로그래밍의 혼한 실수들



• 1.16.4 디바이스 타임이 섞인 잠금

디바이스 타임(1.9절 참고)이 있을 때는 다른 스레드가 자주 접근하는 리소스에 대한 잠금을 하지 말아야 하는데도, 디바이스 타임이 섞인 잠금을 하는 실수를 함-그중 특히 자주 하는 실수는 로그 출력이나 콘솔 출력.

동시접속자는 많은데 게임 서버에서 CPU 사용량이 적게 나오는 경우, 이 때문에 문제가 생김

```
void func()
{
    guard_lock<mutex> gl(my_mutex);

    a...
    b...

    cout << a << b; // 콘솔 출력
}
```

1.16 | 멀티스레드 프로그래밍의 혼란 실수들



- 1.16.5 잠금의 전염성으로 발생한 실수

잠금의 전염성 : 잠금으로 보호되는 리소스(변수 값 등)에서 얻어 온 값이나 포인터 주소 값 등이 로컬 변수로 있는 경우에도 잠금 상태를 계속 유지해야 할 때

```
class A
```

```
{
    int x;
    int y;
};
```

```
mutex list_mutex;
List<A> list;
```

```
void func()
```

```
{
    list_mutex.lock();
    A* a = list.GetFirst();
    list_mutex.unlock();
    a->x++; // 문제가 되는 부분
}
```

목록 자체를 액세스하는
것이 아니더라도
로컬 변수가 목록의
무언가를 가리키고
있기 때문에
데이터 레이스(버그)
발생

```
class Player
```

```
{
    int x;
    int y;
};
```

```
mutex p_mutex;
List<Player*> playerList;
```

```
void func()
```

```
{
    p_mutex.lock();
    Player* p = playerList.GetPlayer("John");
    p_mutex.unlock(); // ❶
    p->x...
    p->y...
}
```

❶이라고 표기된 곳에서는 잠금
해제를 하지 말아야 함

(변수 y와 변수 x를 액세스하려면
다른 스레드가 playerList에
접근해서 "john"을 지우는 것을
막아야 함)

1.16 | 멀티스레드 프로그래밍의 혼한 실수들



- 1.16.6 잠금된 뮤텍스나 임계 영역 삭제

```
class A
{
    mutex a_mutex;
    int data;
};

void func()
{
    A* a = new A();
    a->a_mutex.lock();
    delete a;
}
```

고의로 언락을 하지 않고 삭제하는 경우가 있고, 아무 문제도 없음.

하지만, 문맥상 그러한 경우가 오류라면, 클래스 A의 파괴자 함수 안에 “이미 잠금하고 있으면 오류를 내라.”라는 기능을 추가하면 됨

1.16 | 멀티스레드 프로그래밍의 혼란 실수들



1.16.7 일관성 규칙 깨기

잠금 범위가 여럿일 때 여러분이 정의하는 일관성 규칙을 깨는 실수를 범하기도 함.

```
class Node
{
    Node* next;
};

Node* list = null;
int listCount = 0;

mutex listMutex;
mutex listCountMutex;

void func()
{
    listMutex.lock();
    Node* newNode = new Node();
    newNode->next = list;
    list = newNode;
    listMutex.unlock();

    listCountMutex.lock();
    listCount++;
    listCountMutex.unlock();
}
```

list와 listMutex는 같이 맞물려 돌아가야 하는 변수, 즉 일관성이 유지되어야 하는데 잠금 객체 2개 때문에 일관성이 유지되지 못함.

→ 해결 방법은 list와 listMutex를 한 뮤텍스나 임계 영역으로 보호하는 것

1.16 | 멀티스레드 프로그래밍의 혼란 실수들



```
ParallelQueue<int> queue;    // 병렬 자료 구조
atomic<int> item;           // 원자 조작 변수
```

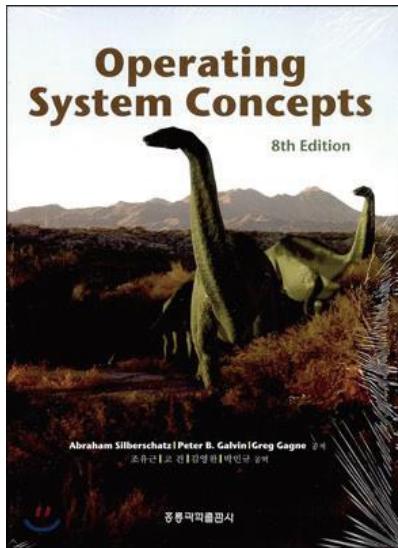
```
void func()
{
    int i = queue.pop();      // 큐에서 꺼낸다.
    item = i;                // 꺼낸 것을 여기다 넣는다.
}
```

```
void func2()
{
    int i = item.exchange(0); // 꺼냈던 항목을 가져와서
    사용한다.
    if (i != 0)
    {
        ...
    }
}
```

크래시를 일으키지 않으나
큐와 큐에서 꺼낸 항목의 상태가
항상 일관성을 가진다고 보장할 수
없음

예를 들어 항목이 5개 있는 큐에서
항목을 꺼내 원자 조작 변수에 넣는
과정을 시행하는 아주 짧은 찰나에,
'큐에서 꺼낸 항목이 아직 원자 조작
변수에 들어가 있지 않음'이라는
상태를 다른 스레드가 만나 버릴
수도 있다.

1.17 | 심화 내용 및 더 읽을거리



《Operating System Concepts 8판》(홍릉과학출판사, 2013)*의 Part 2 프로세스 관리

원자 조작을 잘 활용하면 잠금 없이도 여러 스레드가 안전하게 접근할 수 있는 프로그램을 만들 수 있다. (**non-blocking** 자료 구조)

멀티스레드 프로그래밍을 편리하게 해 주는 도구나 API, 언어들이 있다.
(Visual Studio Concurrency Visualizer 말고 Intel Parallel Studio, Windows Performance Toolkit도 멀티스레드 프로그래밍을 도와줌)

C++11에서 제공하는 std::async 등을 이용하여 비동기 프로그래밍을 할 수 있다.

OpenMP는 프로그래밍은 편리하나, C++11에 비해 성능 향상이나 프로그램 편의성이 떨어져서 게임에서는 사용하지 않는다.

그림 1-46 운영체제론 도서

1 숙제 (#1)

게임 클라이언트 프로그램 작성

- 내용

- 체스 판을 화면에 그린다. (8x8)

- 체스 말 하나를 화면에 그린다.

- 커서 키로 말을 상하좌우로 이동한다.

- 목적

- 앞으로 작성하게 될 서버 프로그램의 동작을 확인할 수 있는 프로그램

- 제약

- Windows에서 Visual Studio 2017로 작성 할 것

- 그래픽의 우수성을 보는 것이 아님

- 제출

- 제목에 “2019 게임서버프로그래밍 학번 이름 숙제 1번”

- 9월 24일 오후 1시까지 제출 (1일 딜레이 당 10% 감점)

- 꼭 필요한 파일들만 Zip으로 소스를 둙어서 e-mail로 제출

- 소스만(sdf, obj, log, manifest 같은거 제외!)

- E-mail주소는 nhjung@kpu.ac.kr