Multithreading

Game Server Programming

Multithreading

1 | Programs & Processes
2 | Threads
3 | When do we need multithreaded programming?
4 | What is a thread?
5 | Things to consider when handing threads
6 | Critical sections and mutexes
7 | Deadlocks
8 | Rules for locking order
9 | Parallelism & serial bottlenecks
10 | Single-threaded game server
11 | Multi-threaded game server
12 | Atomic operation

source: Game Server Programming Textbook, Gilbut, 2019

1.1 | Programs & Processes

- Program: A chunk of data containing a set of instructions , executed on a computer
- Process: A state in which a program is active
- Loading: Loading the code and data in a program into process memory
- In Winedows, you can check the programs that are executed in this way with the Task Manager

Fig. 1-1 Programs are on disk, processes are in RAM

1.1 | Programs & Processes

- Multiprocessing: Multiple processes running

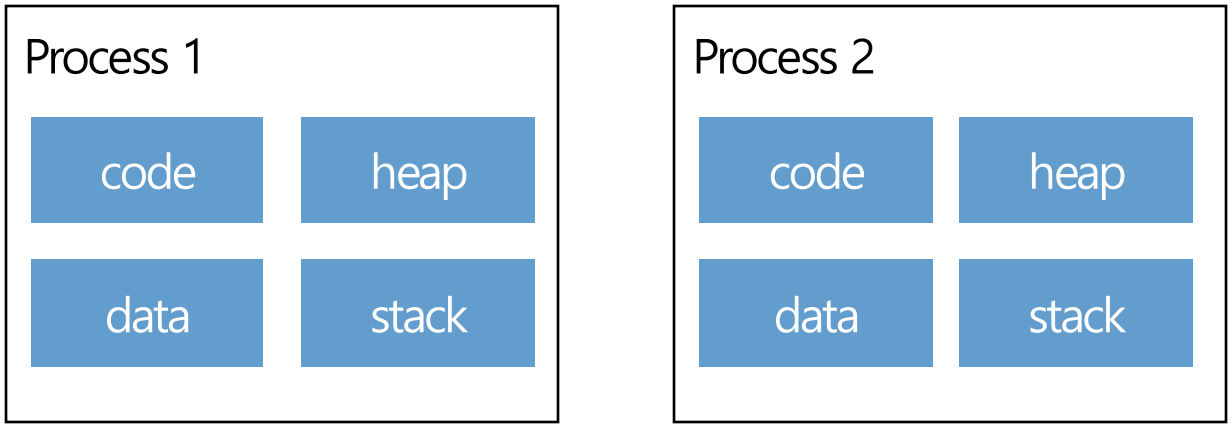


Fig. 1-3 multiprocessing

1.2 | Thread

- Most commonly used Oses provide a functionality called threads.
- Difference between threads and processes
  - There are multiple threads in a process
  - Threads are the flow of a program!
  - Threads in a process can share the memory space in the process
  - Each thread has a stack. This means that local variables of functions executed in each thread managed by each thread.

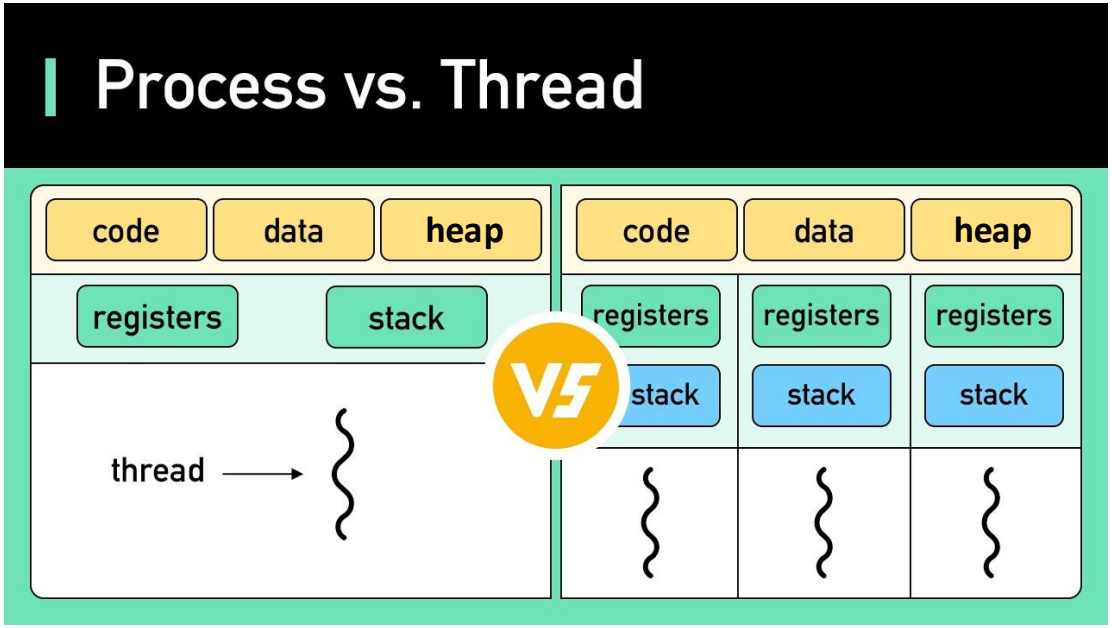


Fig. 1-4 Process and Threads

1.2 | Thread

"When you run a program, a process is created. Inside that process, there is a single thread, and the program runs inside that thread."

- Single-threaded program: A program that runs only one program at a time.
- Single-threaded model: A program designed and implemented to operate with only a single thread.

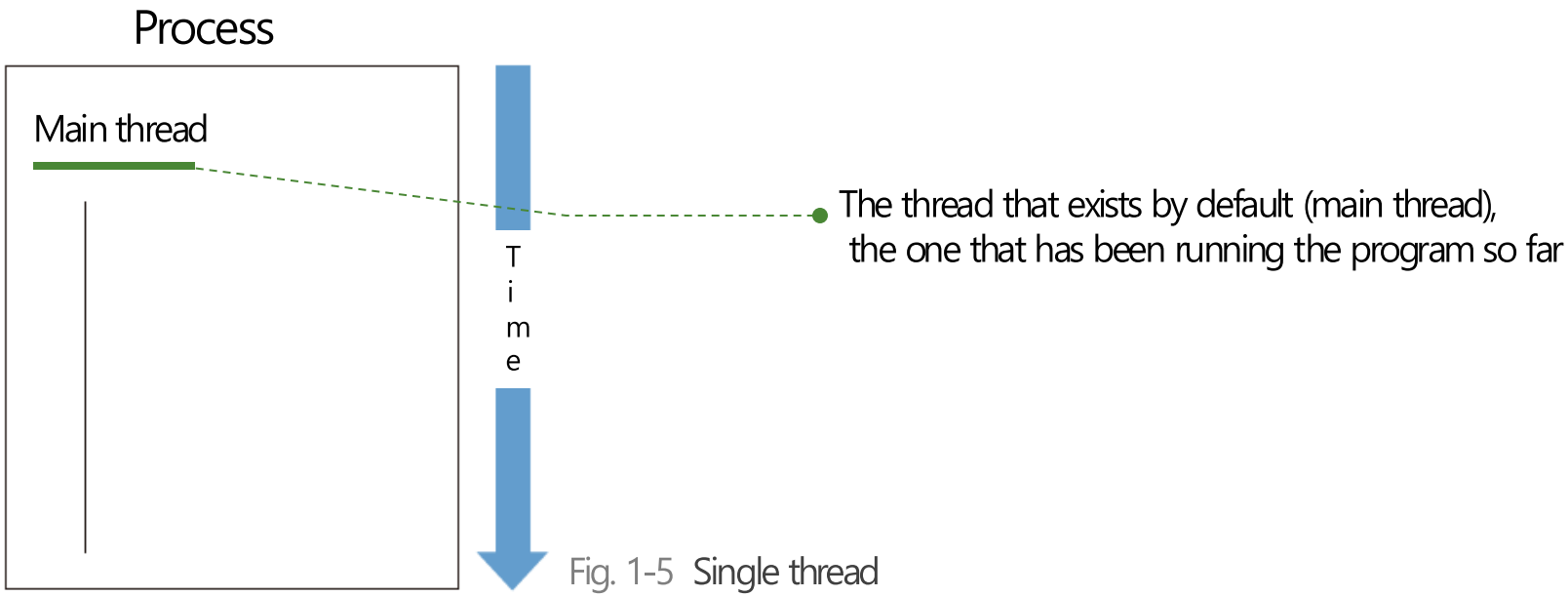
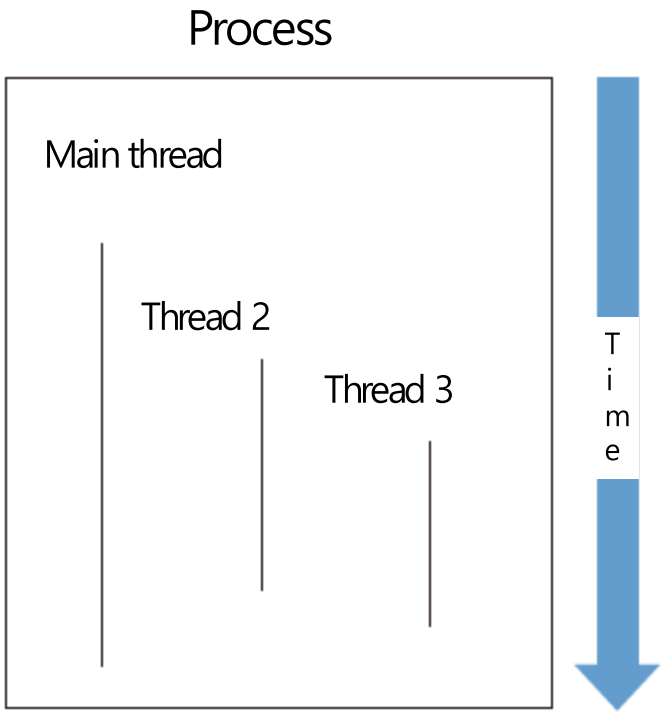


Fig. 1-5 Single thread

1.2 | Thread

- Multithreaded model or multithreading: Having multiple threads handle multiple tasks simultaneously




```
void main()
{
        t2 = CreateThread(Thread2Proc);
}
void Thread2Proc()
{
        t3 = CreateThread(Thread3Proc);
}
void Thread3Proc()
{
        ...
}
```

Fig. 1-6 Multithreading

1.2 | Thread

- Checking the call stack using the debugger (when using Visual Studio)

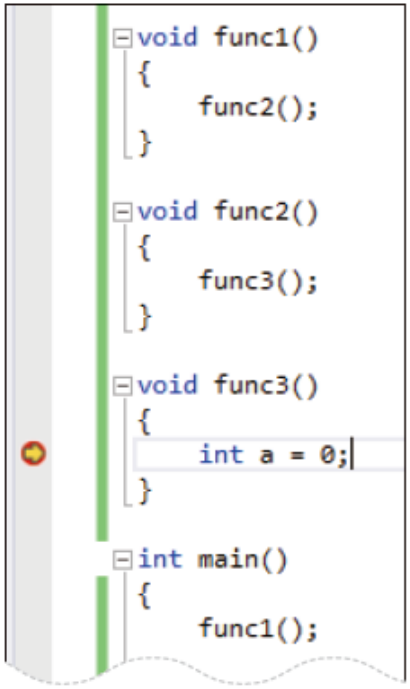


Fig. 1-7 Debugging in Visual Studio

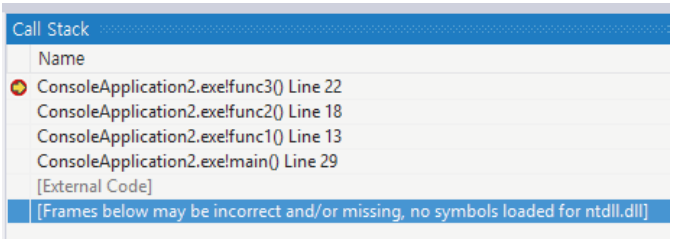


Fig. 1-8 Debugger call stack

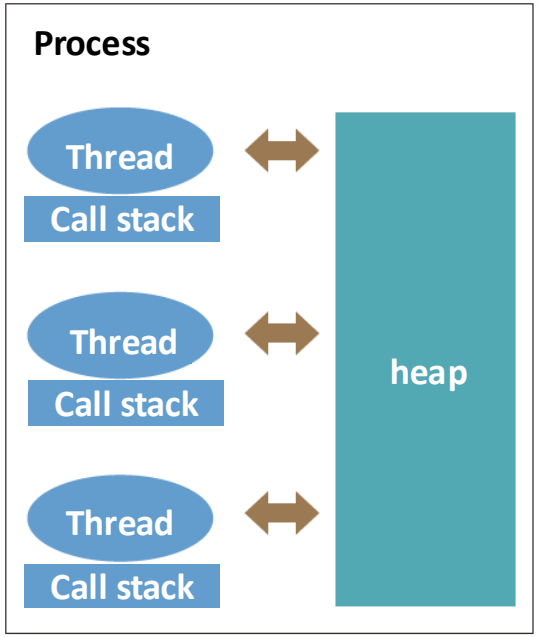


Fig. 1-9 Relationship among threads, call stack, and heap

*exCallStack

1.2 | Thread

- Example of main thread and another thread running concurrently

```
void main()
{
// ❶ start main thread
t1 = CreatThread(ThreadProc, 123);
// ❷ Dealing with long-term tasks
// ...
// ❸ Wait until t1 thread terminates
t1.Join();
// ❹ End of main()
}
ThreadProc(int)
{
// ❺ Dealing with long-term tasks
// ...
// ❻ End of ThreadProc()
}
```

No one knows which of ❷ and ❺ will be executed first!

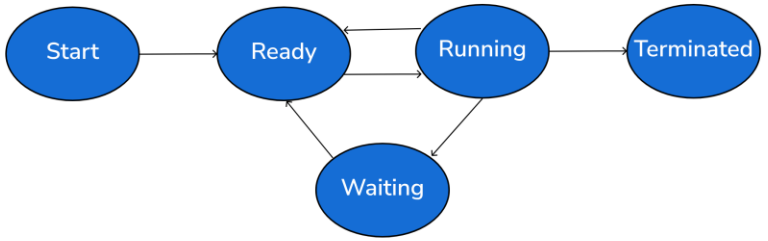
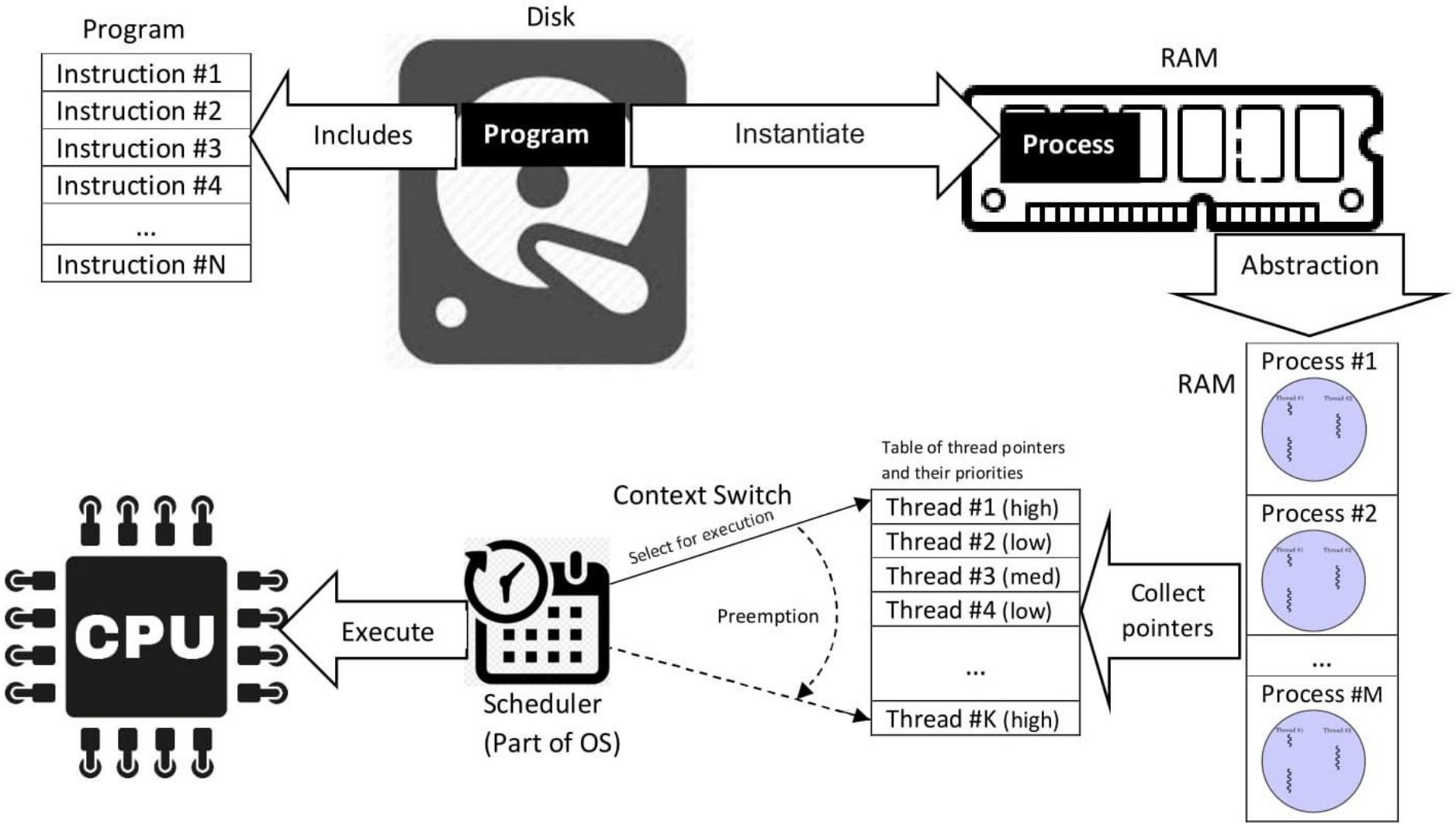


Fig. 1-10 Life of a thread

1.2 | Program journey to Execution



https://upload.wikimedia.org/wikipedia/commons/2/25/Concepts-_Program_vs._Process_vs._Thread.jpg

1.3 | When should we need multithreaded programming?

- Typical situations where multi-threaded programming is required

1. When you need to do one long-running task and several quick-running tasks at the same time
2. When you need to do other short tasks while performing a long process
3. When you need to utilize all the CPUs on the device

1.3 | When should we need multithreaded programming?

- 1.3.1 When you need to do one long-term task and several quick-to-finish tasks at the same time

  -When loading in a game program

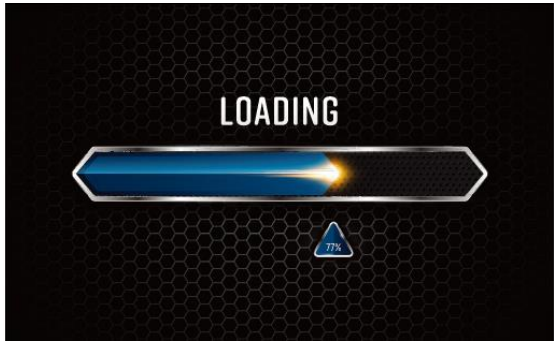


Fig. 1-11 Graphical representation of loading situation

Fig. 1-12 Animation to relieve the tedious loading time (<Street Fighter 5>)

Fig.1-13 Minigame to enjoy while loading

1.3 | When should we need multithreaded programming?

When not multithreading

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

- Messy code
- Temporary frame rate drops when loading large files
- Frame rate is uneven when loading and rendering files in parts, and the code becomes messier

Rendering: The process of visually representing data. Mainly used to display graphics, images, videos, etc. on the screen.

When multithreading

```
bool isStillLoading; // global variable

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

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

        isStillLoading = false;
}
```

- Perform rendering continuously
- Loading data required for the game from disk, and changing a specific variable when loading is complete. Thread1 repeatedly renders the loading screen until this variable is changed.

1.3 | When should we need multithreaded programming?

- 1.3.2 When you need to do other short tasks while doing some long processing.
  - When accessing a disk to read or write player information.

The time it takes to write player information to disk is about 1/10,000th of a second → A huge waste of time for commercial game servers.

Disks are mechanical devices made up of motors, so they work much slower than electronic circuits.

First-person shooter (FPS) games process 30 requests per second for one client. If 10,000 of these clients connect to the server, 30 * 10,000 = 300,000 processings are required per second.

→ In the worst case, the server that should process 300,000 requests can only process 10,000 requests per second.

To utilize a very long time of 1/10,000 second

You need to do multithreading or asynchronous programming!

1.3 | When should we need multithreaded programming?

- 1.3.3 When you need to utilize all the CPUs on your device
  - Experiment: Program to find and print prime numbers (integers that are not divisible by any number other than 1 and itself, such as 2, 3, 5, and 7)
  - Demo: prime_number

- Program execution: Displays the time taken to calculate the prime number.

Took 3920 milliseconds.

- Run after changing MaxCount to a much larger number

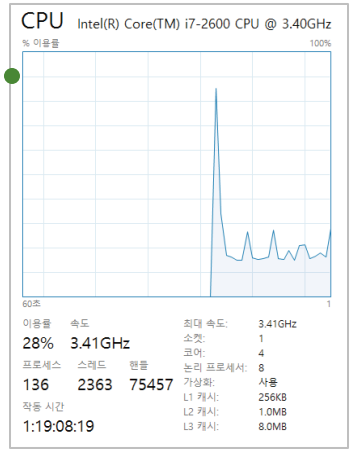


Fig. 1-14 the entire CPU cannot be used

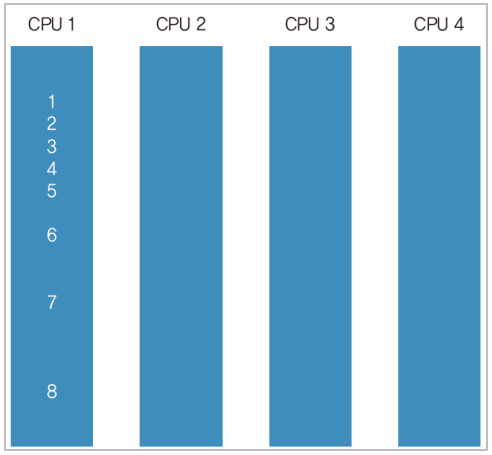


Fig. 1-15 The unfortunate situation where calculations are processed with only one CPU (core)

Task Manager (Ctr+shift+esc) > Performance

*prime_number

1.3 | When should we need multithreaded programming?

-solving the issue

- Create a global variable *num*.
- Each thread gets one value from *num*. Determine if the value is a prime number.
- If it is a prime number, put the found number in the array *primes*.
- Once all threads have finished their work, print out the prime numbers.

*Demo: prime_number_errorneous

- Expressing in code

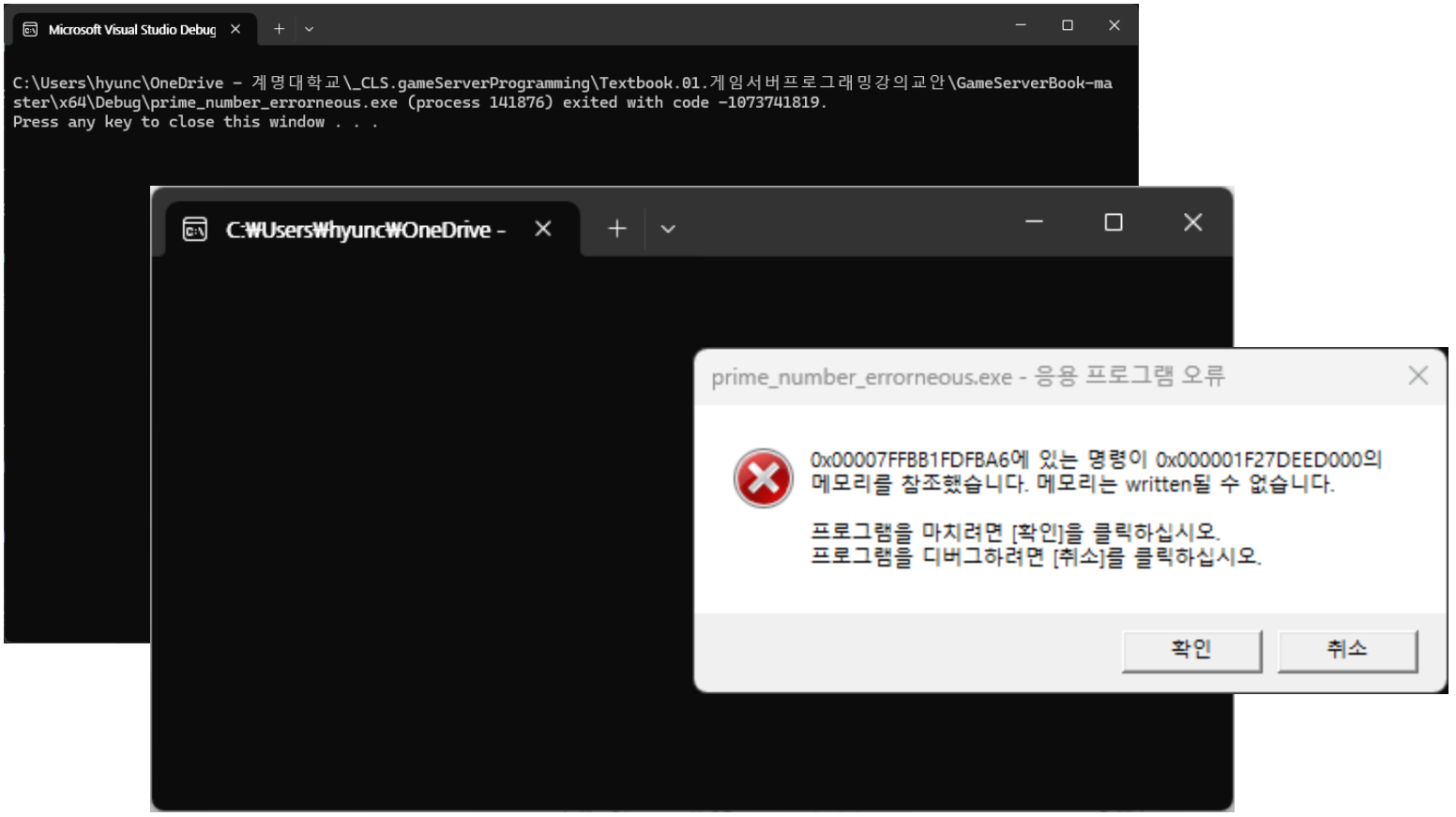
```
main()
{
        Array<Thread> threads;

        for (i = 0; i < 4; i++)
                threads.Add(BeginThread(ThreadProc));
        for (i = 0; i < 4; i++)
        {
                threads.waitForExit();
        }
        printNumbers(primes);
}
```

```
int num = 1;
Array<int> primes;
```

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

1.3 | When should we need multithreaded programming?

-Multi-thread programming attempted without any plan…



1.4 | Thread and context switch

- When told to "do two things at the same time"
  - Computers work by going back and forth
  - Context switch: The process of executing another thread while stopping execution of each thread
  - If the context switch interval is too frequent: The time spent executing T1 and T2 is much less, so it is inefficient
  - If the context switch is as few as possible: If context switch takes 1 second, then the animation during loading can only be executed once per second!
  - If the number of threads is more than the number of CPUs: The context switch must occur within a CPU
  - If the number of threads that are actually runnable is less than the number of CPUs, there is no problem
  - The context switch is performed in machine language instruction units.

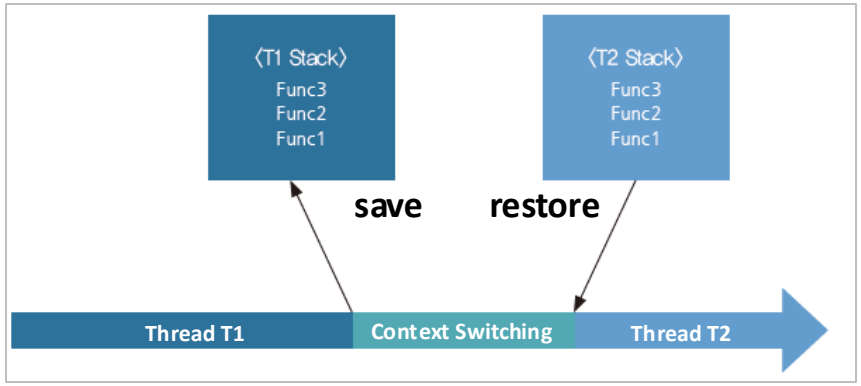


Fig. 1-19 Context Switching

1.5 | Things to keep in mind when handling threads

- When two threads access one value simultaneously

```
x += y;
```

```
int x = 2
# Thread 1
x += 3
# Thread 2
x += 4
# Expected value
x is 9
```

When compile the code into machine language,

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

What we want

```
x = 2
# Thread 1
t1 = x        // t1 = 2
t1 = t1 + 3   // t1 = 5
x = t1        // x = 5
# Thread 2
t2 = x        // t2 = 5
t2 = t2 + 4   // t2 = 9

x = t2        // x = 9
```

Wrong result!
- Context switches occur randomly

```
x = 2
# Thread 1
t1 = x        // t1 = 2
t1 = t1 + 3   // t1 = 5
# Thread 2
t2 = x        // t2 = 2
t2 = t2 + 4   // t2 = 6
x = t2        // x = 6
# Thread 1
x = t1        // x = 5 !!!
```

1.5 | Things to keep in mind when handling threads

- Find out what went wrong in the program that calculates prime numbers previously
  - Multiple threads access the Array<int> array object simultaneously (member variable: pointer, size of array).
  - When there is not enough space, reallocate memory (often change the value of the pointer variable).
  - The following properties are not satisfied
    - Atomicity: Either change both variables or neither
    - Consistency: Both variables always maintain a consistent state

```
main()
{
        Array<Thread> threads;

        for (i = 0; i < 4; i++)

        threads.Add(BeginThread(ThreadProc));
        for (i = 0; i < 4; i++)
        {
                threads.waitForExit();
        }
        printNumbers(primes);
}
```

```
int num = 1;
Array<int> primes;
```

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

1.5 | Things to keep in mind when handling threads

- Find out what went wrong in the program that calculates prime numbers
  - Only one 4-byte integer data space is required to store the information called -num, but more than one data space is required to store the information called Array<int> primes.
  - Array<int> has a pointer variable that points to the array object and the size of the array as a member variable. The array object would have been allocated from the memory heap, and the memory heap is handled by the runtime library of the currently running process.
  - If two threads call the Add() function of Array<int> at the same time, multiple threads will change the Array<int> variables, and then the other thread will access the array while only one of the two variables has changed.
  - During this process, the pointer variable pointing to the array may temporarily point to a wrong value, for example, memory that has already been freed from the heap: data race condition occurs.
  - Atomicity: While accessing the two member variables, other threads must not access them. –
  - Consistency: The two variables of Array<int> always maintain a consistent state.

Therefore, a synchronization technique is required:

- Special measures to maintain atomicity and consistency, representative examples are critical sections, mutexes (mutual exclusion), and locking techniques.

A **critical section** is a part of code area where shared resources (like variables, memory, or files) are accessed by multiple threads simultaneously, and might causing problems.

1.6 | Critical Sections and Mutexes

- One of several ways to resolve a race condition

"While a thread is using some information X, prevent other threads from accessing X!"

To be more precise…

"Other threads will wait until the current thread has finished using X before accessing it!"

1.6 | Critical Sections and Mutexes

- Mutex: Short for MUTual EXclusion.

How to use a mutex

1. Create a mutex MX that protects X, Y.
2. Before accessing X, Y, the thread asks MX to "gain the right to use it."
3. The thread accesses X, Y.
4. After accessing it, it asks MX to "release the right to use it."

- Mutex: expressing it in code

Explicitly call lock and unlock

```
std::mutex mx;      // ❶
mx.lock();          // ❷
read(x);            // ❸
write(y);           // ❹
sum(x);             // ❺
mx.unlock();        // ❻
```

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

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

1.6 | Critical Sections and Mutexes

Fig. 1-24 Multithreaded implementation of a program to find prime numbers

```
main()
{
        List<Thread> threads;
        for (i = 0; i < 4; i++)
        {
           threads.Add(
               new Thread(ThreadProc);
        }
        for (i = 0; i < 4; i++)
        {
           threads.Join();
        }
}
```

```
int num = 1;
CriticalSection num_critSec;
```

```
Array<int> primes;
CriticalSection primes_critSec;
```

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

*Demo: prime_number_multithreaded

- Running the program gives the following results – it runs much faster than the previous case.

Took 1358 milliseconds.

1.6 | Critical Sections and Mutexes

- Measure CPU usage after greatly increasing MaxCount

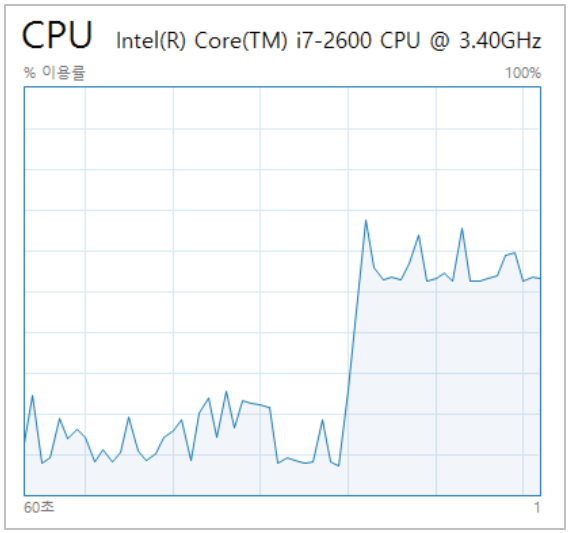


Fig. 1-25 Checking CPU usage

Execution time when there is one thread: 3900 milliseconds

Execution time when there are four threads: 1300 milliseconds

→ Why is it actually only 3 times faster?

1.6 | Critical Sections and Mutexes

- Main loop of the thread

```
// Main function for each thread
    // Loop if we can get a value.
    while (true)
{
        int n;
        { // ❶ ----------
                lock_guard<recursive_mutex> num_lock(num_mutex);
                n = num;
                num++;
        }
        if (n >= MaxCount)
                break;

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

In the ❶, ❷ section, a lock is applied, and

a situation occurs where other threads

switch to a waiting state.

However, the operation for calculating

prime numbers requires much more

calculations than this section.

❶, ❷ can also be a reason for unsatisfactory performance,

but the proportion is not large.

Even if you have a lot of CPUs, we can't use 100% of their performance.

1.6 | Critical Sections and Mutexes

- If you divide the mutex too finely,
  1. Rather, the program performance is lowered → because the process of accessing the mutex itself is heavy.
  2. The program becomes very complex, especially deadlock problems easily occur.

Code having too small critical section

```
class Player
{
        CriticalSection m_positionCritSec;
        Vector3 m_position;     // ❶
        CriticalSection m_nameCritSec;
        string m_name;          // ❷
        CriticalSection m_hpCritSec;
        int m_hp;               // ❸
}
```

The above code must access in order to avoid deadlock. It is difficult to handle increasing rules.

It is recommended to divide the mutex range appropriately.
It is recommended to divide the lock units for parts that are advantageous when operated simultaneously (multiple CPUs in parallel).
And not to divide the lock units for parts that do not significantly affect performance .

1.7 | Deadlock

- **Deadlock**

A situation where two threads are waiting for each other

Thread 1 is waiting for Thread 2 to finish what it was doing,

but Thread 2 is actually waiting for Thread 1 to finish what it was doing

→ In this case, both threads are stuck forever

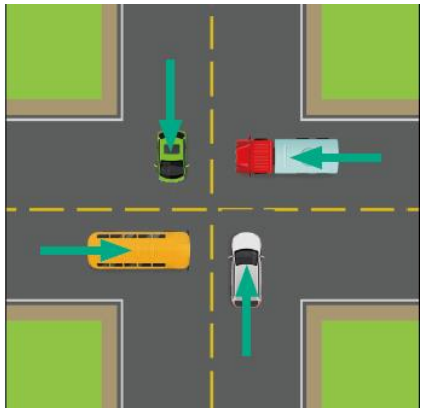


Fig. 1-26 Deadlock in reality

```
int a;
int b;

Thread1()
{
        lock(a)
        {
                a++;
                lock(b)
                {
                        b++;
                }
        }
}

Thread2()
{
        lock(b)
        {
                b++;
                lock(a)
                {
                        a++;
                }
        }
}
```

Wait forever

Fig. 1-27 If the program is executed in this order, a deadlock occurs.

1.7 | Deadlock

- What happens when a deadlock occurs on a game server?

  1. CPU usage is significantly low or 0%, regardless of the number of concurrent users
  2. Clients cannot use the server, for example, they cannot log in or send a request and do not receive a response.

- How to use CRITICAL_SECTION

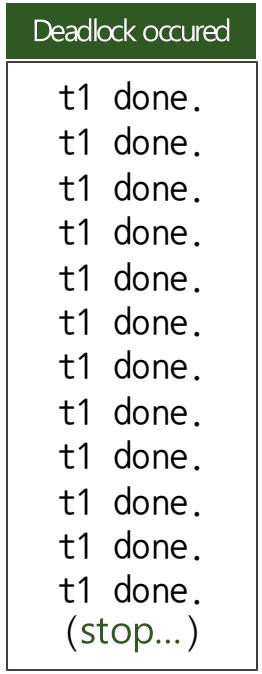
Critical section creation is done with InitializeCriticalSectionEx. At this time, a CRITICAL_SECTION object is created.

Critical section removal is done with DeleteCriticalSection.

Critical section locking is done with EnterCriticalSection.

Critical section unlocking is done with LeaveCriticalSection.

*Demo: deadlock-example



1.8 | Rules for locking order

- To prevent deadlock when using multiple mutexes:

  First, draw the locking order of each mutex as a graph.
  Then, when locking, you should check the locking order graph to see if there is anything locked in reverse.

- Locking order of mutexes A, B, and C
  A → B → C // ❶

-The unlock order does not affect deadlock.

When locking in the order of A → B → C (does not cause deadlock)

```
lock(A)
lock(B)
lock(C)
unlock(C)
unlock(B)
unlock(A)
```

When locking in the order of A → B (does not cause deadlock)

```
lock(A)
lock(B)
unlock(B)
unlock(A)
```

When only B and C are locked (does not cause deadlock)

```
lock(B)   // lock B
unlock(B) // unlock B
lock(C)   // lock C
          // Since B has already been unlocked, only C is locked.
unlock(C) // unlock C
```

When locking in the order of B → C (does not cause deadlock)

```
lock(B)
lock(C)
```

When locking in the order of A → C (does not cause

```
lock(A)
lock(C)
unlock(C)
unlock(A)
```

1.8 | Rules for locking order

A → B → C // ❶

When locking in the order of B → A (causing	When locking in the order of C→A (causing deadlock)
lock(B) lock(A) unlock(A) unlock(B)	lock(C) lock(A) unlock(A) unlock(C)

recursive mutex Handles a thread locking a mutex repeatedly multiple times smoothly.

```
lock(M)   // Acquired a lock.
lock(M)   // lock again.
unlock(M) // Unlock once. But there's still one more to go.
unlock(M) // Unlock twice. Only then is the unlocking actually done.
```

1.8 | Rules for locking order

- recursive mutex

A → B → C → B → A

```
lock(A)      // ❶
lock(B)      // ❷
lock(C)      // ❸
lock(B)      // ❹
lock(A)      // ❺
unlock(C)    // ❻
unlock(B)    // ❼
unlock(A)    // ❽
```

First lock, keep the lock order

Since locking that is already locked, we can ignore the lock order graph.

A → C → B → C → A

```
lock(A)      // ❶
Lock(C)      // ❷
lock(B)      // ❸
lock(C)      // ❹
lock(A)      // ❺
unlock(C)    // ❻
unlock(B)    // ❼
unlock(A)    // ❽
```

❶ safe

❷ Safe since no backwards

❸ Deadlock occur (break B → C)

❹ ❺ Safe with recursive locking

All is safe, but deadlock occurs because of one line of code in ❸

"To avoid deadlock, locking order must be preserved (not reversed)."

1.9 | Parallelism and Serial Bottlenecks

- Parallelism and Serial Bottlenecks

  Parallelism: Multiple CPUs execute each thread's calculations to increase simultaneous processing.

  Serial bottleneck: A phenomenon in which a program is designed to run in parallel, but only one CPU actually does the calculation.

- What each thread of the prime number program does

  1. Lock num.
  2. Get a value from num.
  3. Unlock num.
  4. Determine if num is prime.
  5. If it is, lock primes.
  6. Insert a prime number into primes.
  7. Unlock primes

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

Fig. 1-35 What a thread does over time

1.9 | Parallelism and Serial Bottlenecks

Fig. 1-36 A situation where everyone is playing except one CPU.

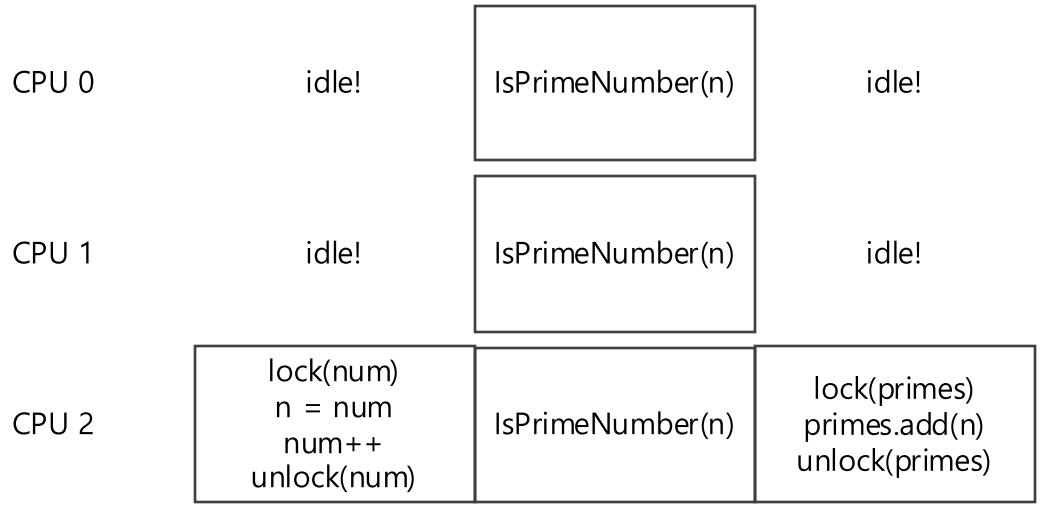
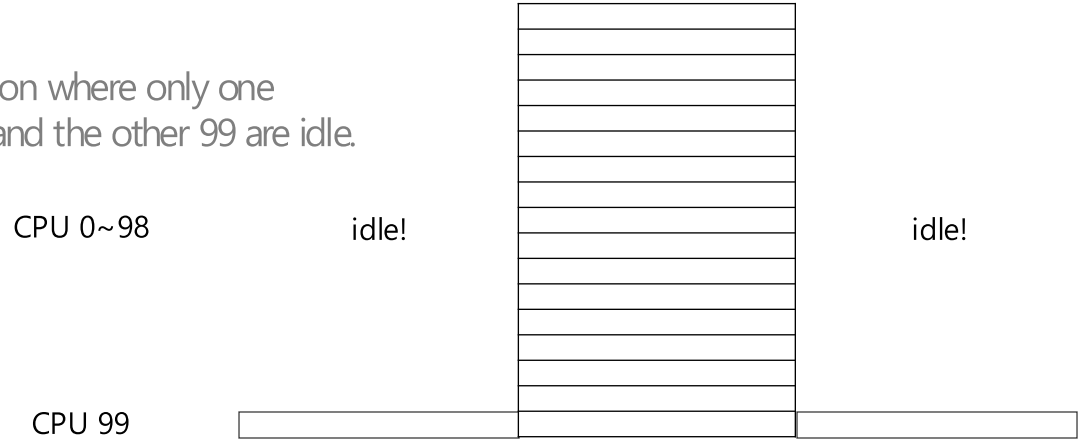


Fig. 1-36 A situation where only one CPU is working and the other 99 are idle.



1.9 | Parallelism and Serial Bottlenecks

- Reduce the curse of the Amdahl's Law

Amdahl's Law or Amdahl's Curse: The phenomenon where the more CPUs there are, the lower the overall processing efficiency.

To reduce the curse of the Amdahl's Law , the section where serial bottleneck occurs must be minimized.

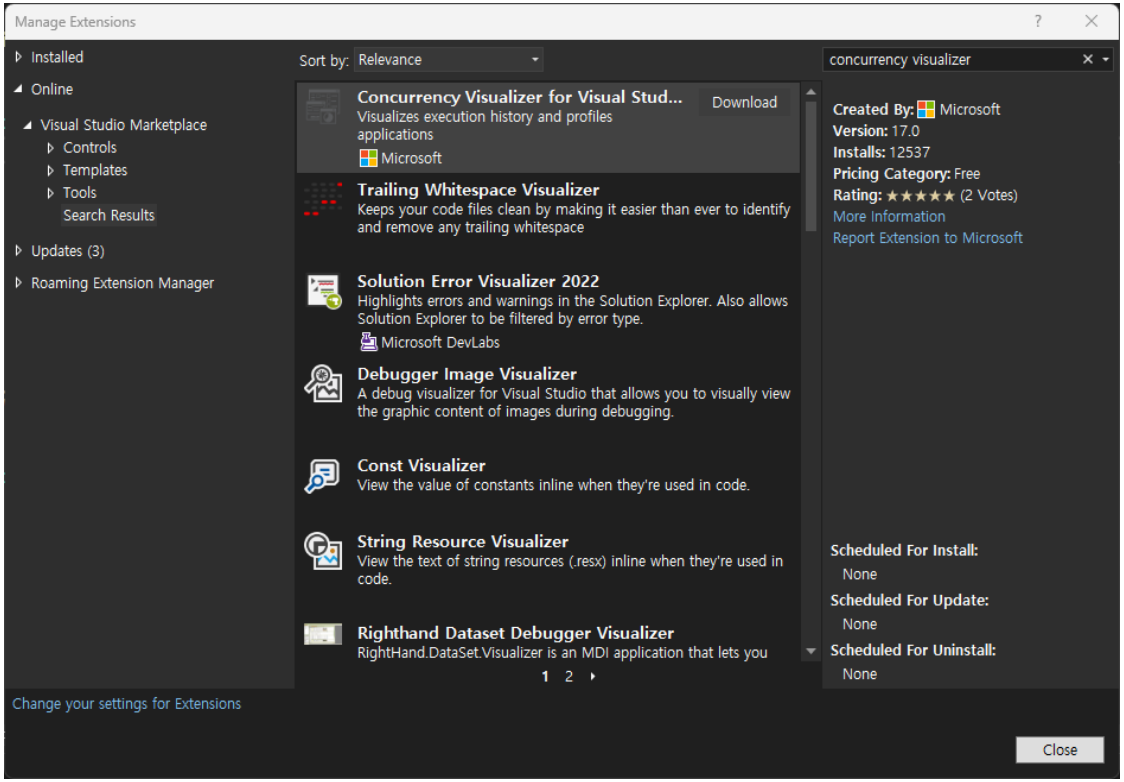


Fig. 1-38 Installing Concurrency Visualizer (Extensions on menu)

Analyze > Concurrency Visualizer > Launch new process … , demo…

1.10 | Single threaded game server

- When running a single-threaded server, it is common to launch as many processes as there are CPUs.

1. If there are as many threads or processes as the number of rooms, the number of context switches between threads or processes increases.
2. 2. Therefore, even if it is a server that handles the same number of concurrent users, the number of concurrent users that can actually be handled is greatly reduced.

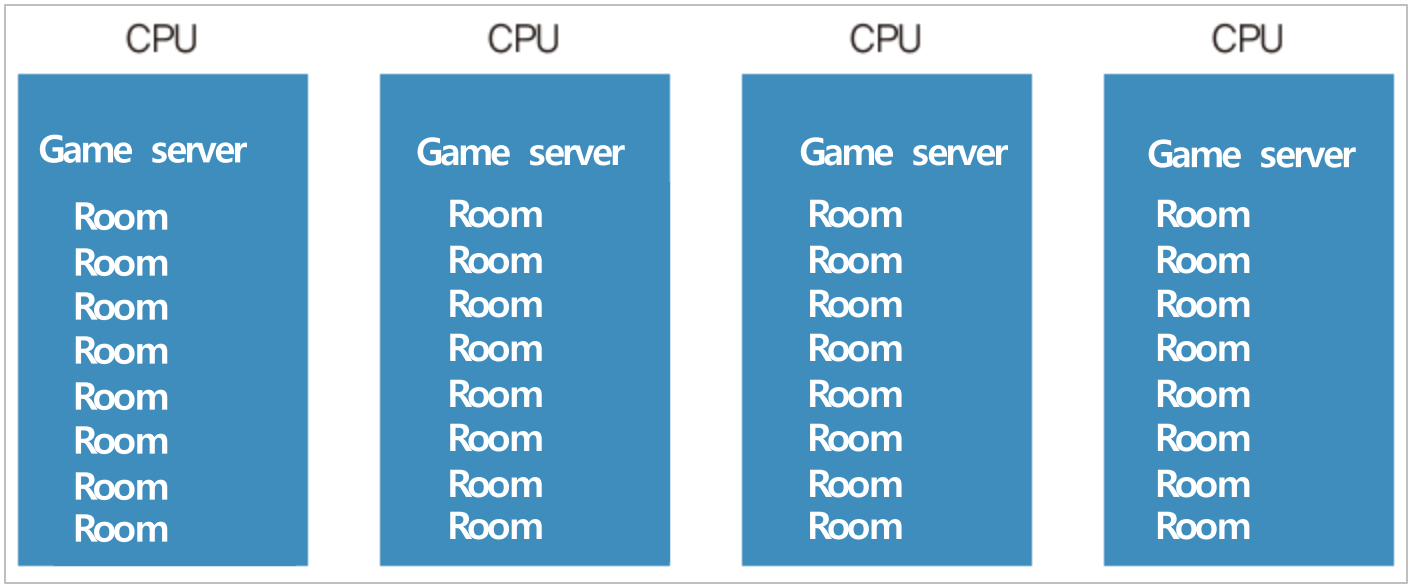


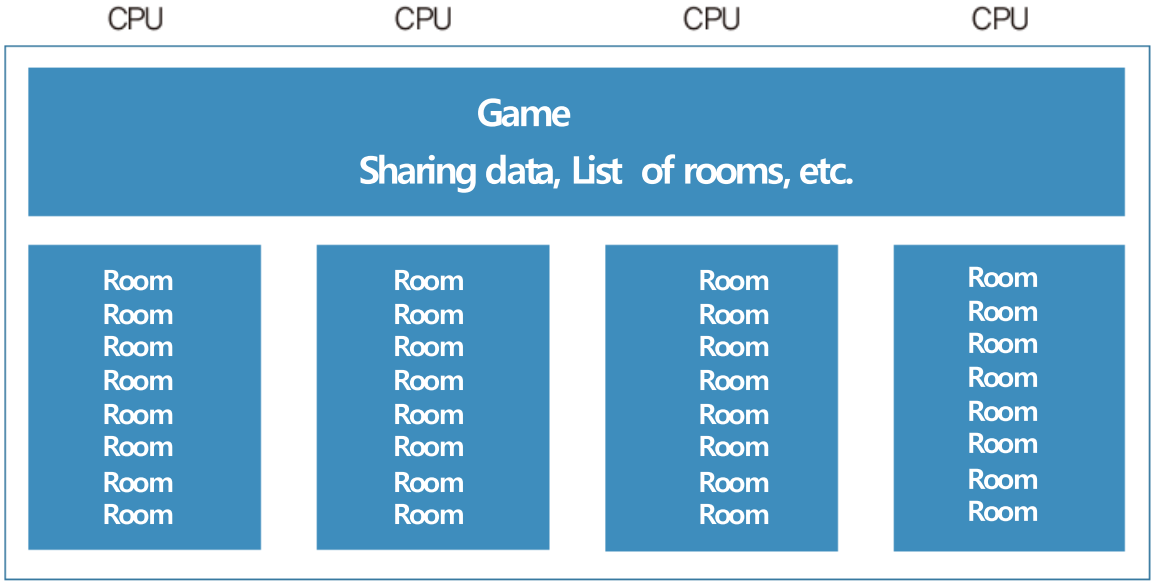
Fig. 1-42 Single-threaded, multi-process

*Room: A space where one or more players gather to play together. Also called a game room or multiplayer session.

1.11 | Multi-threaded game server

- When developing a server with multi-threading

1. When it is difficult to launch many server processes. For example, when the game information (map data, etc.) that needs to be loaded per process is very large (especially MMO game servers)
2. When a single server process performs so many operations that it requires the computational power of multiple CPUs
3. When co-routines or asynchronous functions cannot be used and device time is used
4. When only one server instance should be placed per server device
5. When different rooms need to access the same memory space



When creating a multi-threaded game server, the main thing to watch out for is **serial bottlenecks** and **deadlocks**. Especially when accessing files, performance degradation often occurs when accessing them with mutexes locked.

Fig. 1-43 Model of a game server that operates in multithreading in one process

1.12 | Atomic operation

- atomic operation

  It means that multiple threads can safely access it without a mutex or critical section lock.

  Atomic operations are a hardware feature, and most compilers provide atomic operations.

  Atomic operations ensure that when multiple threads access a 32-bit or 64-bit variable type, only one thread will process it.

  - Adding values with atomicity
  - Swapping values with atomicity
  - Conditional swapping values with atomicity

  We don't know what value the variable has, but we can add a specific value to that variable and get the result of that variable.

```
volatile int a = 0;
```

Declaring variable

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

"We don't know what's in a, but we're going to add 3 to it. What we want is for a to have exactly the value 3 added to it. And we want to know what a is, after it's been added."

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

The variable values and the values I want are swapped. The past value of a is filled into r, and 10 is entered into a.

```
int r = AtomicCompareExchange(&a, 10, 100);
```

"Change a to 100 only if a contains 10"

//EoF