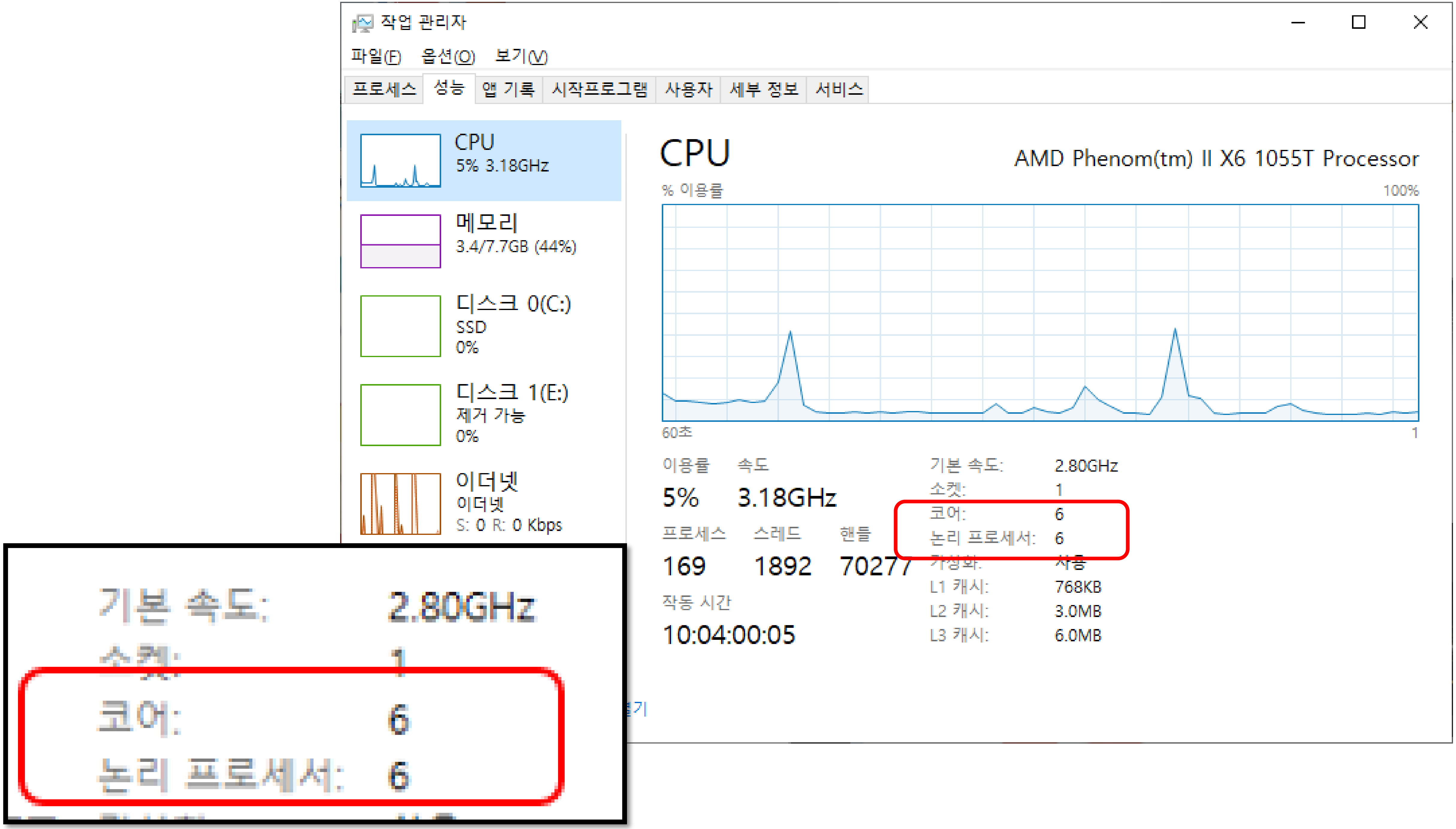
Multithreading

OODP, 2022

Concurrent Programming

What is a **thread**?

- a sequential flow of **execution** ;
- a sequence of **control** steps, executed one at a time, through a program
- multiple threads may run at the same time in the same program (or process)



CPU AMD Phenom(tm) II X6 1055T Processor

이용률 5%

속도 3.18GHz

프로세스 169

스레드 1892

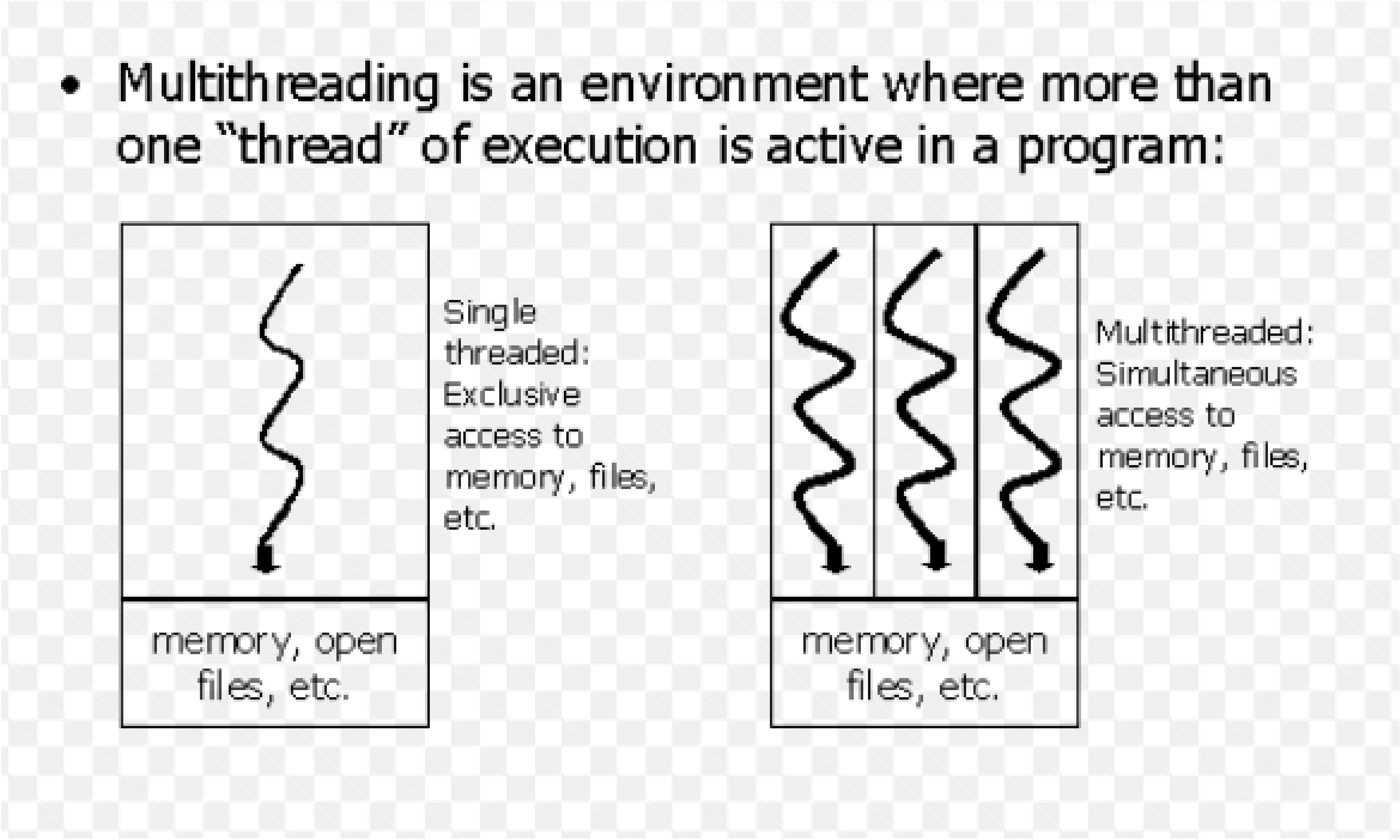
핸들 70277

작동 시간 10:04:00:05

기본 속도:	2.80GHz
소켓:	1
코어:	6
논리 프로세서:	6
가상화:	사용
L1 캐시:	768KB
L2 캐시:	3.0MB
L3 캐시:	6.0MB

기본 속도:	2.80GHz
소켓:	1
코어:	6
논리 프로세서:	6

https://docs.huihoo.com/aolserver/intro/tcl2k/html/sld017.htm



- Multithreading is an environment where more than one “thread” of execution is active in a program:

Single threaded: Exclusive access to memory, files, etc.

Multithreaded: Simultaneous access to memory, files, etc.

```java
public class Main1 extends Thread {
  public static void main(String[] args) {
    Main1 thread = new Main1();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Run the code!

```
public class Main2 implements Runnable {
  public static void main(String[] args) {
    Main2 obj = new Main2();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Run the code!

multi-threading:

- each task performs ***independently*** of others,
- can **share an access** to objects with a get-modify-set sequence,
- potentially developing **race** hazard

race hazard

- two threads modify the same piece of data in *an **interleaved** way*
- the state of the object may be ***corrupted***

multitasking

- the ability to have more than one program working at what seems like the same time
- CPU and memory overhead,
- real-time and timesharing

multithreaded program

- individual program appear to do multiple tasks (threads) at the same time
- each task usually called a **thread** – short for thread of control

Synchronization

- forces execution of the two threads to be mutually **exclusive** in time especially when accessing to shared data (or critical section)
- synchronize the access to **critical section** using **lock**
- prevent interleaved processes from **corrupting** the data

wait and notify

- **synchronized locking** mechanism keeps threads from interfering with each other
- **wait / notify**_methods gives a way to communicate from one thread to another
- wait – enable a thread to wait until some condition (or event) occurs
- notify – *tell the waiting thread that something(event) being waited has occurred*

```
synchronized  void doWhenCondition() {
        while (!condition)
        wait ();
      ... Do actions;
}
synchronized void changeCondition() {
    ... change some value used in a condition test...
 notify();
}
```

Synchronization
Lock / Unlock
Race hazard
Data corruption

```
class Mediator {
private boolean slotFull = false;
private int number;
public synchronized void storeMessage( int num ) {
   while (slotFull == true) {
     try {
         wait();
       }
      catch (InterruptedException e ) { }
    }
    slotFull = true;
    number = num;
    notifyAll();
 }
```

```
      public synchronized int retrieveMessage() {
        while (slotFull == false)
          try {
            wait();
          }
          catch (InterruptedException e ) { }
        slotFull = false;
        notifyAll();
        return number;
      }
    }
```

```
public class Producer extends Thread {
        private Mediator med;
        private int     id;
        private static int num = 1;
        public Producer( Mediator m ) {
          med = m;
          id = num++;
        }
        public void run() {
          int num;
          for (int i =0; i<=5; i++) {
            med.storeMessage( num = (int)(Math.random()*100) );
            System.out.print( "p" + id + "-" + num + "   " );
          }
        }
```

```
public class Consumer extends Thread {
        private Mediator med;
        private int    id;
        private static int num = 1;
        public Consumer( Mediator m ) {
          med = m;
          id = num++;
        }
        public void run() {
          while (true) {
            System.out.print("c" + id + "-" +
                                med.retrieveMessage() + "  ");
          }
        }
      }
```

```
public class MediatorDemo {
public static void main( String[] args ) {
    Mediator mb = new Mediator();
    new Producer( mb ).start();
    new Producer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
    new Consumer( mb ).start();
  }
}
```

p1-27 c1-27 p2-44 p2-33 p2-21
p2-32 c1-44 c2-98 p1-98 c2-80
p2-80 p2-28 c1-32 c4-21 c3-33
c1-28 c2-15 p1-15 p1-94 c2-94
c4-34 p1-34 p1-23 c4-23

yield()

Yields the currently executing thread so that *any other runnable threads can run*. The thread scheduler chooses the highest-priority runnable thread to run.

```
class Babble extends Thread {
  static boolean doYield;
  static int howOften;
  String word;
  Babble(String whatToSay) {
     word = whatToSay;
     }
```

The Java Programming Language
by **James Gosling**

```
public void run() {
for (int i= 0; i< howOften; i++) {
    System.out.println(word);
      if (doYield)    yield();
    }
  }
  public static void main(String[] args) {
    howOften = Integer.parseInt(args[1]);
    doYield =
      new Boolean(args[0]).booleanValue();
      for (int i = 2; i< args.length; i++)
      new Babble(args[i]).start();
   }
  }
```

```
java Babble false 4 did didnot
```

```
>java Babble false 4 did didnot
```

Please write down your own running result.

```
>java Babble true 4 did didnot
```

Please write down your own running result.

Run several times and try to explain the output variations.