System Level Programming

Software College of SCU

Week10

Unit 7: Performance Measurement

- 7.1 Rationale for this unit
- 7.2 Performance Principles
- 7.3 Performance Measurement
- 7.4 VC Profiler
- 7.5 Code::Blocks Profiler

7.1 Rationale for this unit(1/3)

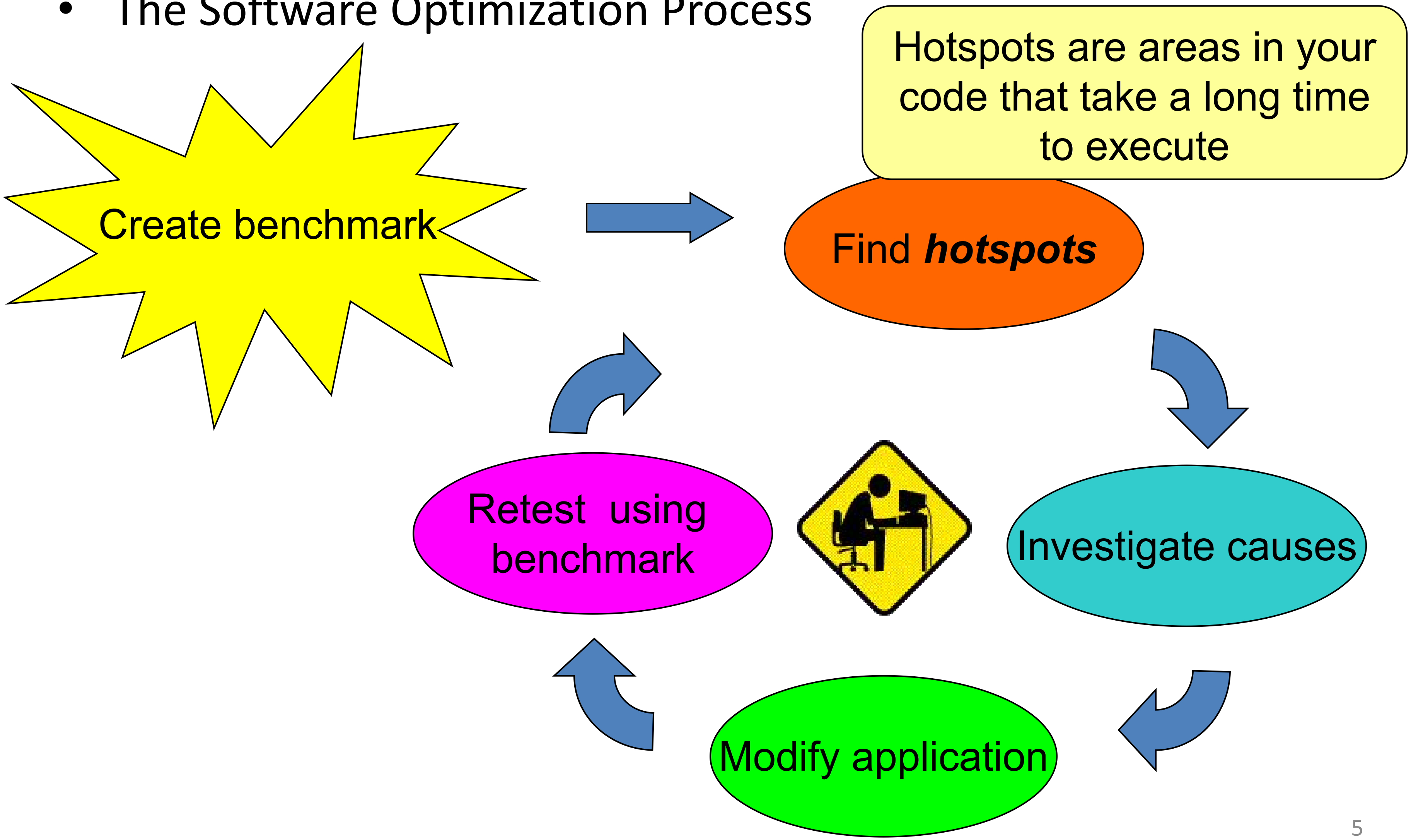
- This unit is about Performance Measurement
  - Next unit, we will focus on how to make programs run fast, that is optimizing program

7.1 Rationale for this unit(2/3)

- How to make the code faster
  - First, we must find what should be optimized
    - Bottlenecks/Hot spots

7.1 Rationale for this unit(3/3)

- The Software Optimization Process



Unit 7: Performance Measurement

- 7.1 Rationale for this unit
- <u>7.2 Performance Measurement</u>
- 7.3 Hot Spots
- 7.4 VC Profiler
- 7.5 Code::Blocks Profiler

7.2 Performance Principles(1/6)

- 80/20 Rule – It means 80% of the CPU time is spent in 20% of the program.

- In this case, you can have better performance by looking at this 20%.

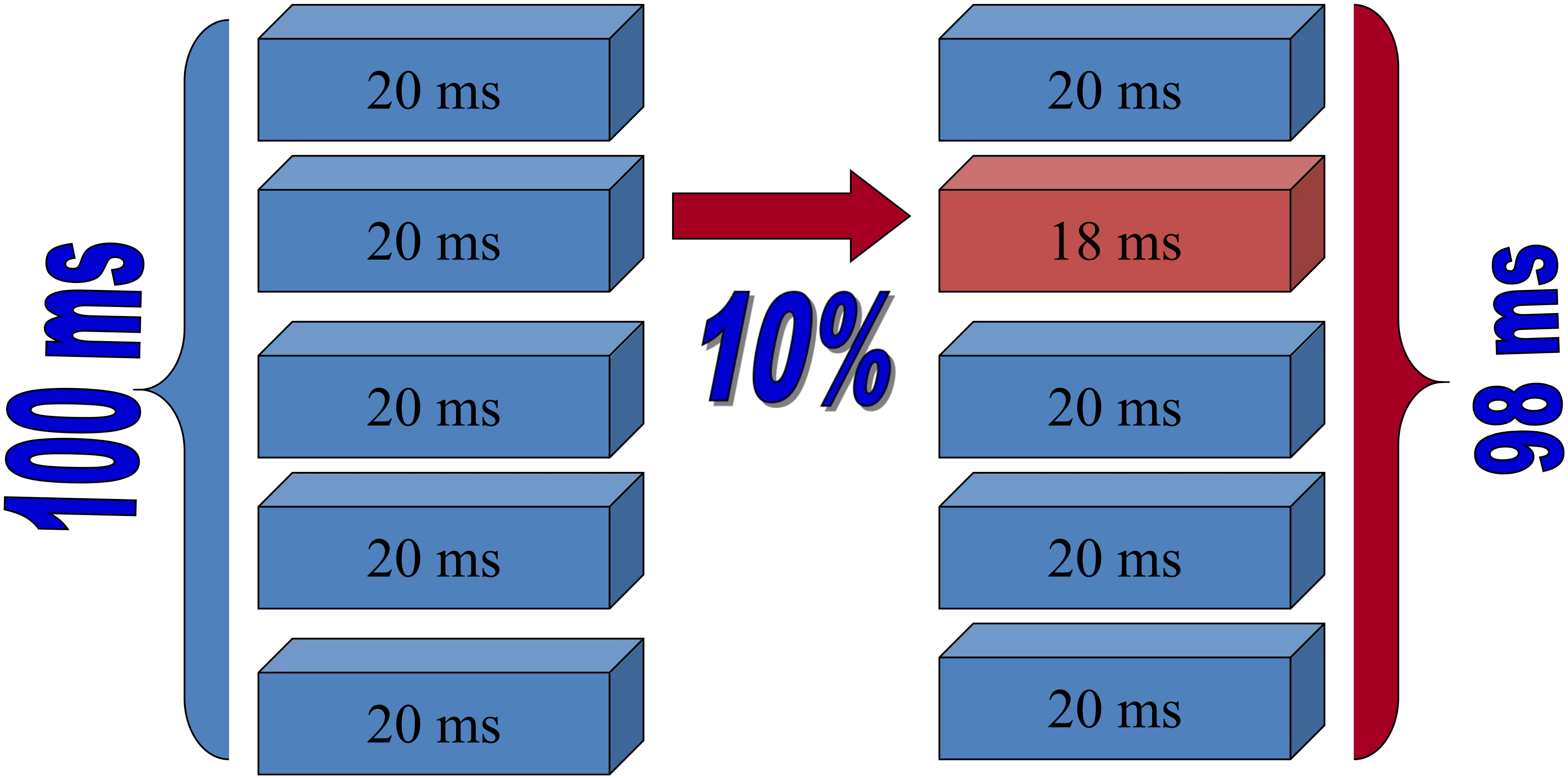
7.2 Performance Principles(2/6)

- Amdahl’s Law(阿姆达尔法则) , also known as Amdahl's argument,is named after computer architect Gene Amdahl, and is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

- 描述：系统优化某部件所获得的系统性能的改善程度，取决于该部件被使用的频率，或所占总执行时间的比例。

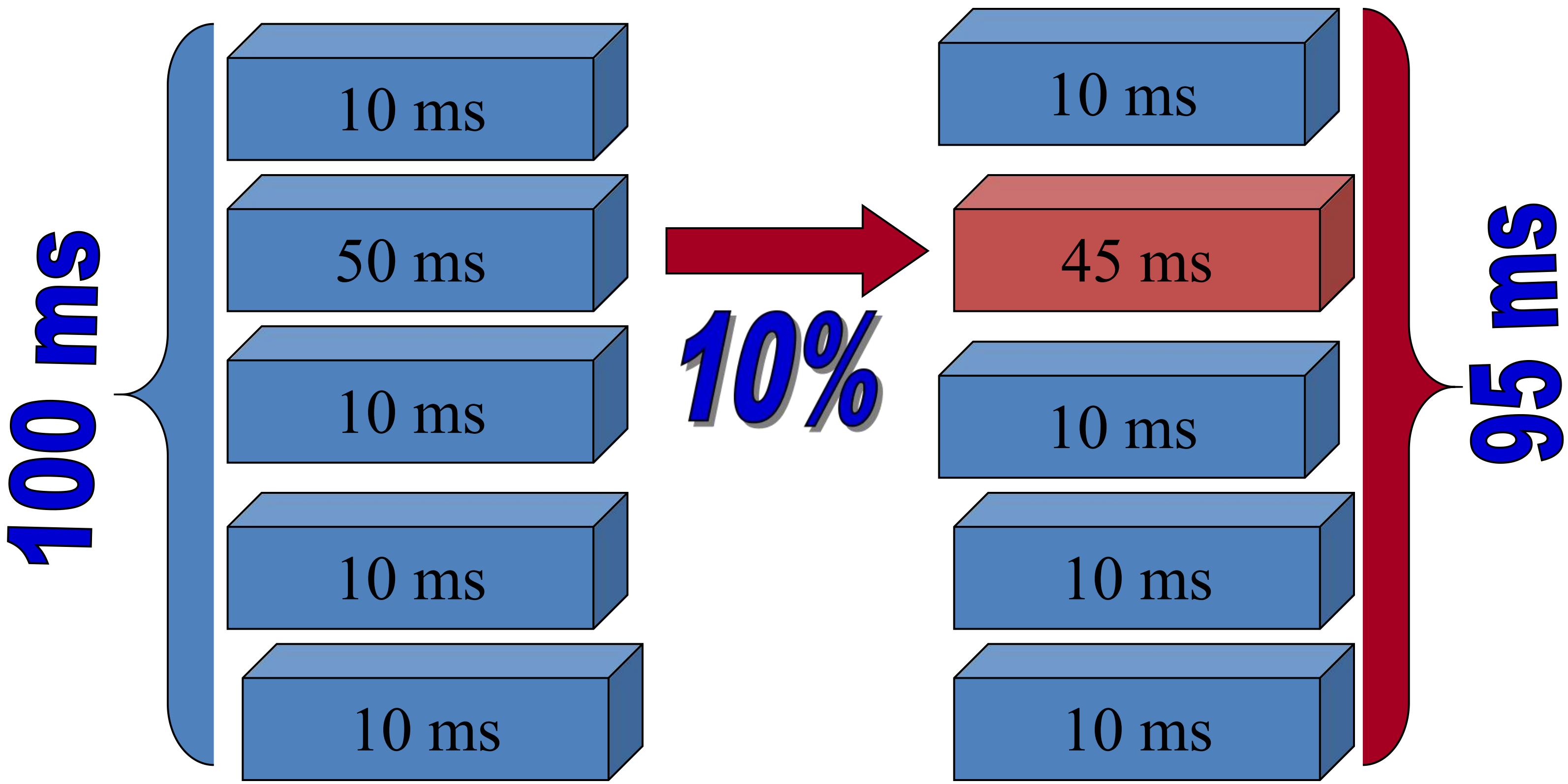
7.2 Performance Principles(3/6)

- 10% on one module means 2% as a whole



7.2 Performance Principles(4/6)

- 10% on one means 5% as a whole
- Conclusion: focus on module with more CPU time



7.2 Performance Principles(5/6)

- Speedup(due to EnhanceMent E)

  = ExcuteTime(without E) / ExcuteTime(With E)

  = Performance(With E) / Performanc(Without E)

- Suppose the enhancement E accelerates a fraction P of one task by a factor S and the remainder of the task unaffected then:

  ExcuteTime(With E)

  = {(1-P) + P/S}*ExcuteTime(Without E);

Speedup(E) = $$\frac{1}{(1-P)+\frac{P}{S}}$$

7.2 Performance Principles(6/6)

- Exercise：
  - It is found that in a typical execution of the Spice circuit simulation program 75% of all instructions do floating point operations. If we add the fastest available floating point co-processor to the system, what is the upper bound on the achievable speed up?

Unit 7: Performance Measurement

- 7.1 Rationale for this unit
- 7.2 Performance Principles
- 7.3 Performance Measurement
- 7.4 VC Profiler
- 7.5 Code::Blocks Profiler

7.3 Performance Measurement

- <u>7.3.1 What to Measure</u>
- 7.3.2 Timing Mechanisms
- 7.3.3 Statistical Sampling (统计抽样) /Profiling

7.3.1 What to Measure (1/1)

- The most common thing to measure is time.

- 计算机性能评测可以与体系结构、系统软件和算法并称为计算科学的四大组成部分。

7.3 Performance Measurement

- 7.3.1 What to Measure
- 7.3.2 Timing Mechanisms
- 7.3.3 Statistical Sampling (统计抽样) /Profiling

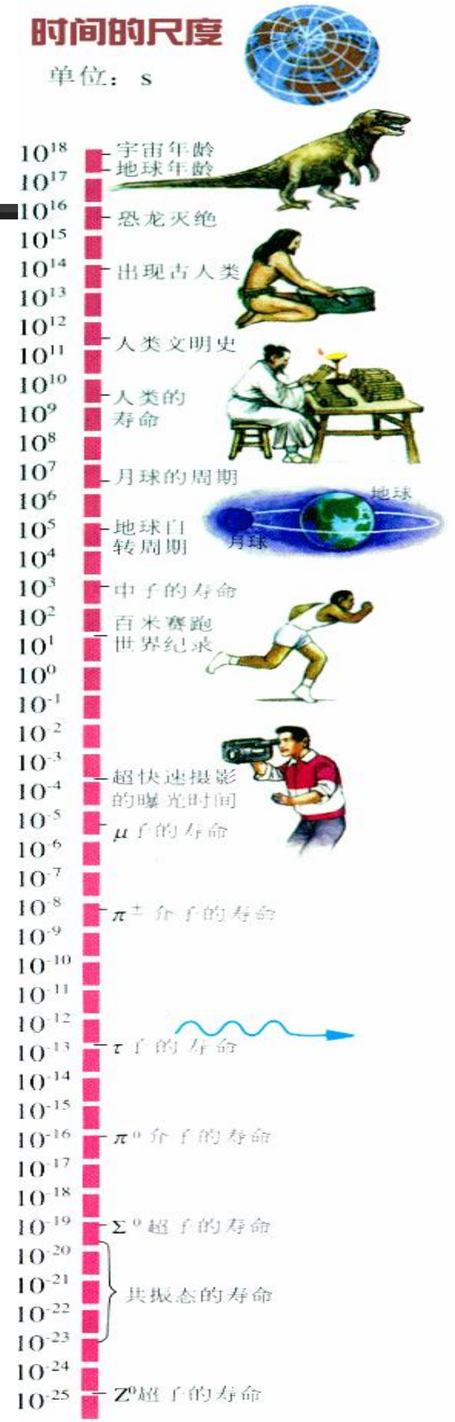
7.3.2 Timing Mechanisms

- 7.3.2.1 Introduction
- 7.3.2.2 Timer in Hardware
- 7.3.3.3 Timer in OS
- 7.3.3.4 Timer in C/C++

7.3.2.1 Introduction(1/7)

- What is time?
  - Time is a one-dimensional quantity used to sequence events, to quantify the durations of events and the intervals between them, and (used together with space) to quantify and measure the motions of objects.

- Benjamin Franklin: ***"Do you love life? Then don’t waste time, because life is made of time."***



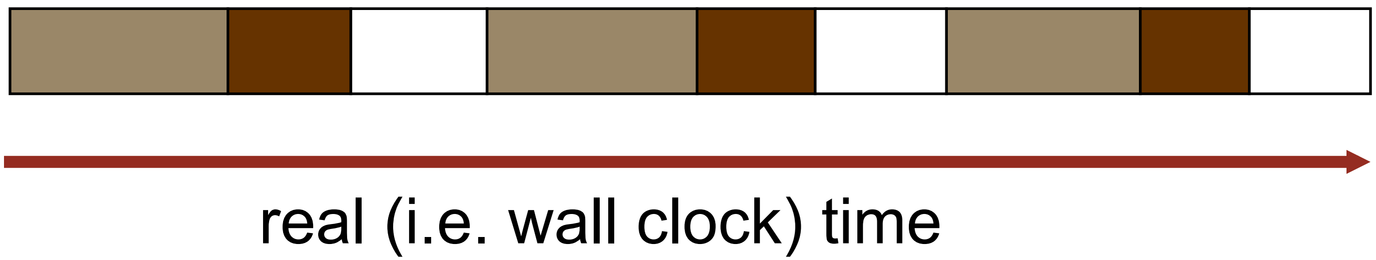
7.3.2.1 Introduction(2/7)

- Why need time in CS?
  - Hardware need time and timer
  - OS need time and timer
  - Measuring program performance, such as execution time.
    - One common question people ask is “How fast does Program run on Machine”

- Time
  - Wall Time
  - CPU Time

7.3.2.1 Introduction(3/7)

- Wall clock time is the time an ordinary clock on the wall or a wrist watch shows.



= **User Time**: time spent executing instructions in the user process

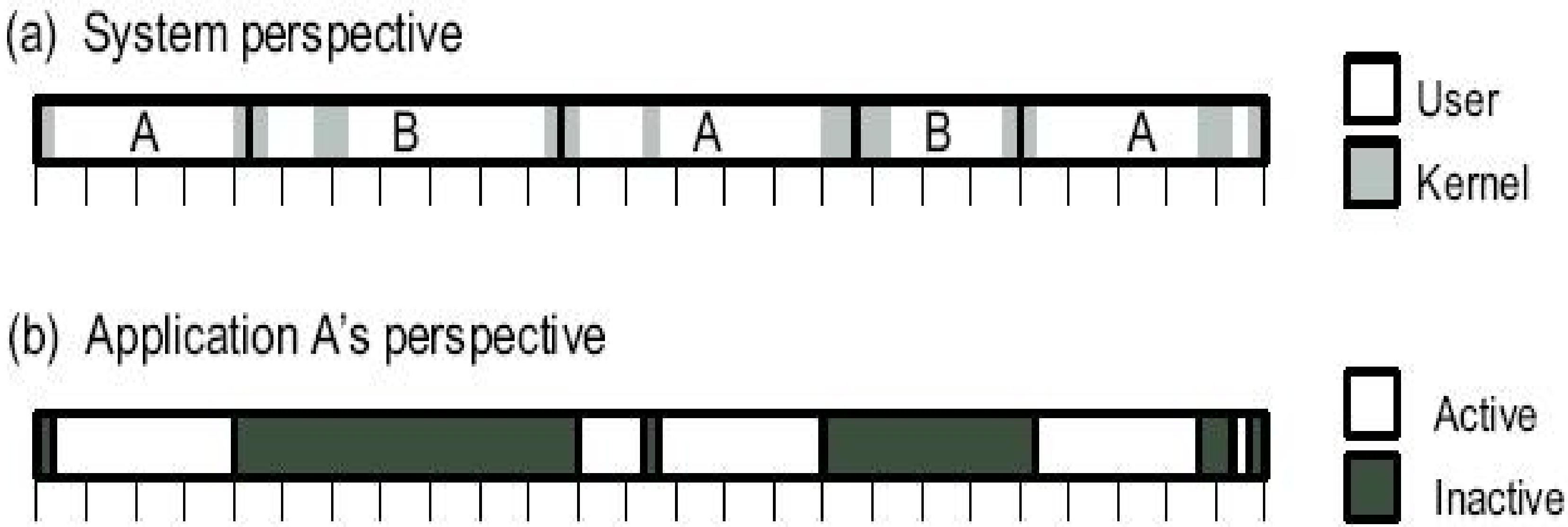
= **System Time:** time spent executing instructions in the ***kernel*** on behalf of the user process

= **all other time** (either idle or else executing instructions unrelated to the user process)

7.3.2.1 Introduction(4/7)

- CPU time
  - CPU time = user CPU time + system CPU time
  - CPU user time: the CPU time spent directly executing your program code,
  - CPU system time: the CPU time spent by the operating system on behalf of your program

7.3.2.1 Introduction(5/7)



7.3.2.1 Introduction(6/7)

- How to use time in CS？
  - By using many kinds of timer.
- What is timer？
  - A component in computer system/CS, as a hardware or software, which can provide the ability of measure time in some degree.

7.3.2.1 Introduction(7/7)

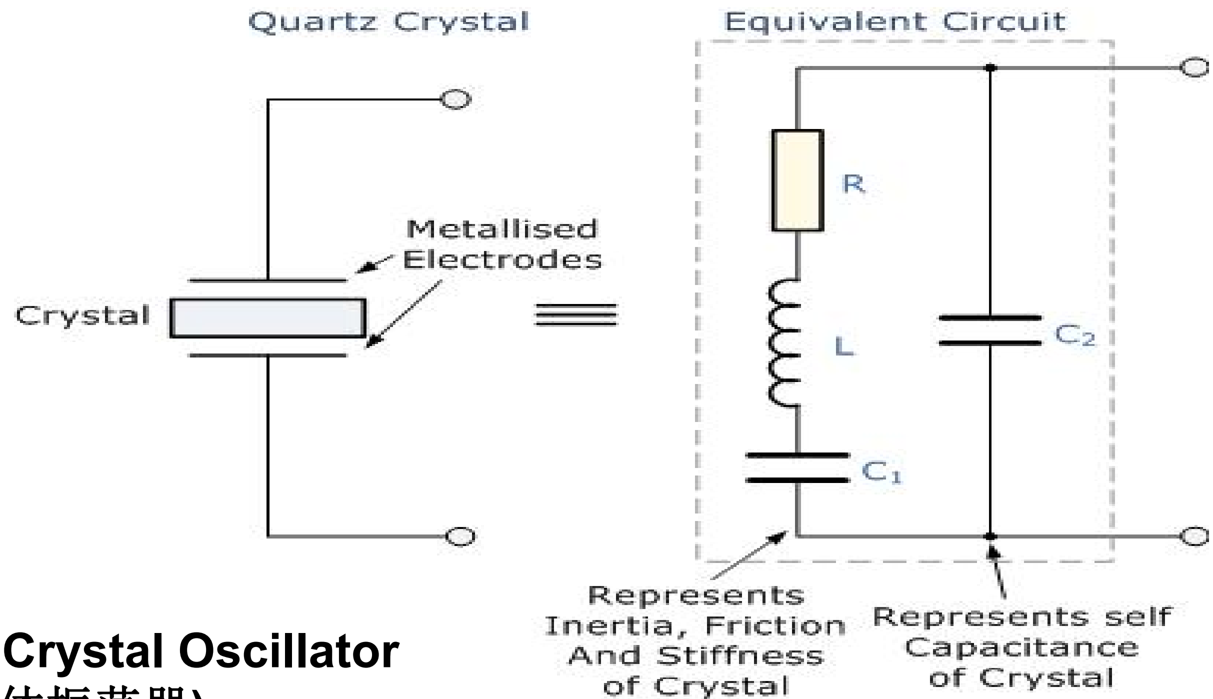
- Different kinds of timer：
  - Timer in Hardware
    - E.g.: in IA32/X86
  - Timer in OS
    - E.g.： in Windows
    - E.g.： in Linux
  - Timer in C/C++
    - E.g.: in <time.h>

7.3.2 Timing Mechanisms

- 7.3.2.1 Introduction
- <u>7.3.2.2 Timer in Hardware</u>
- 7.3.3.3 Timer in OS
- 7.3.3.4 Timer in C/C++

7.3.2.2 Timer in Hardware (1/11)

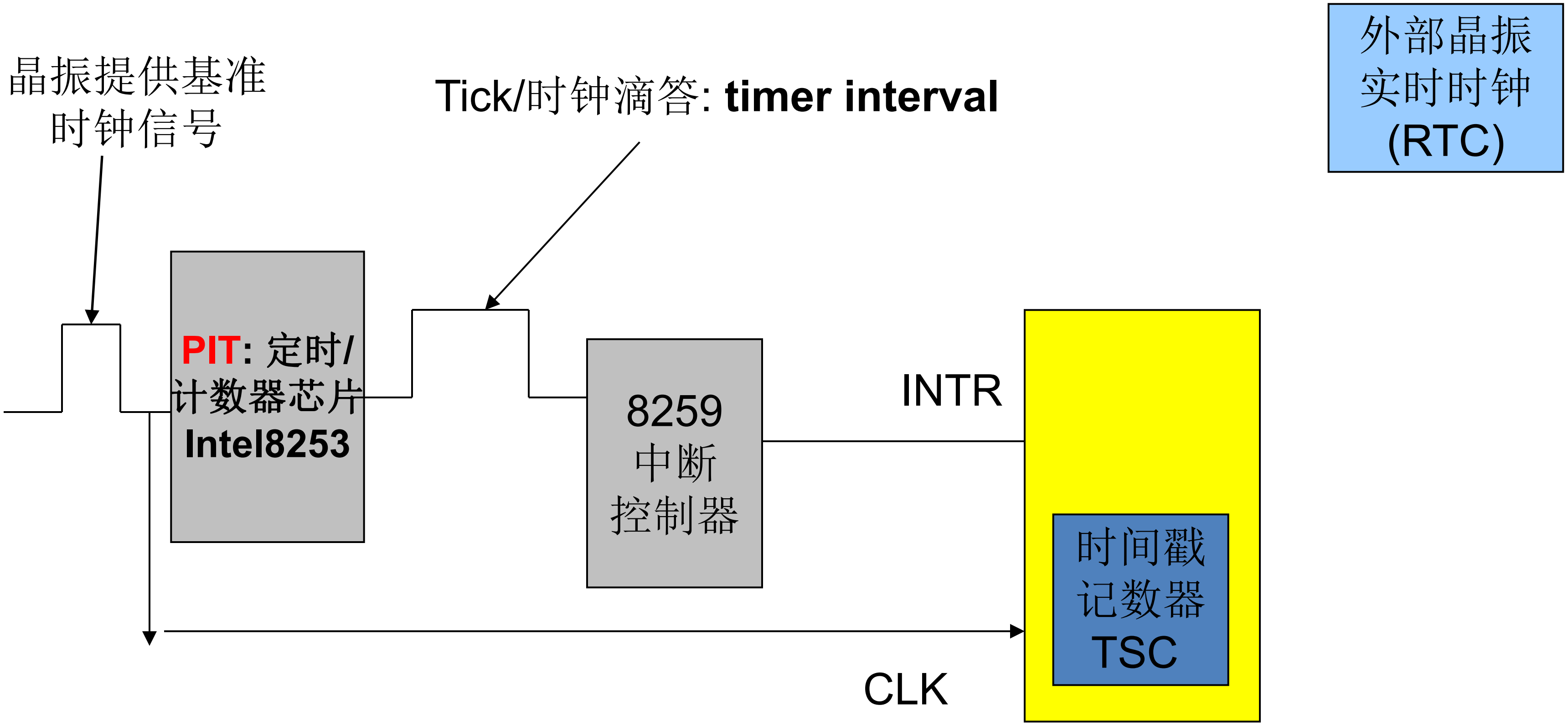
- Crystal oscillator(晶振) produces the original clock frequency
  - Clock Cycle (时钟周期/振荡周期) = seconds per cycle
  - Clock Frequency (时钟频率/脉冲) = cycles per second (1 Hz.=1 cycle/sec)



Quartz Crystal Oscillator (石英晶体振荡器)

7.3.2.2 Timer in Hardware (2/11)

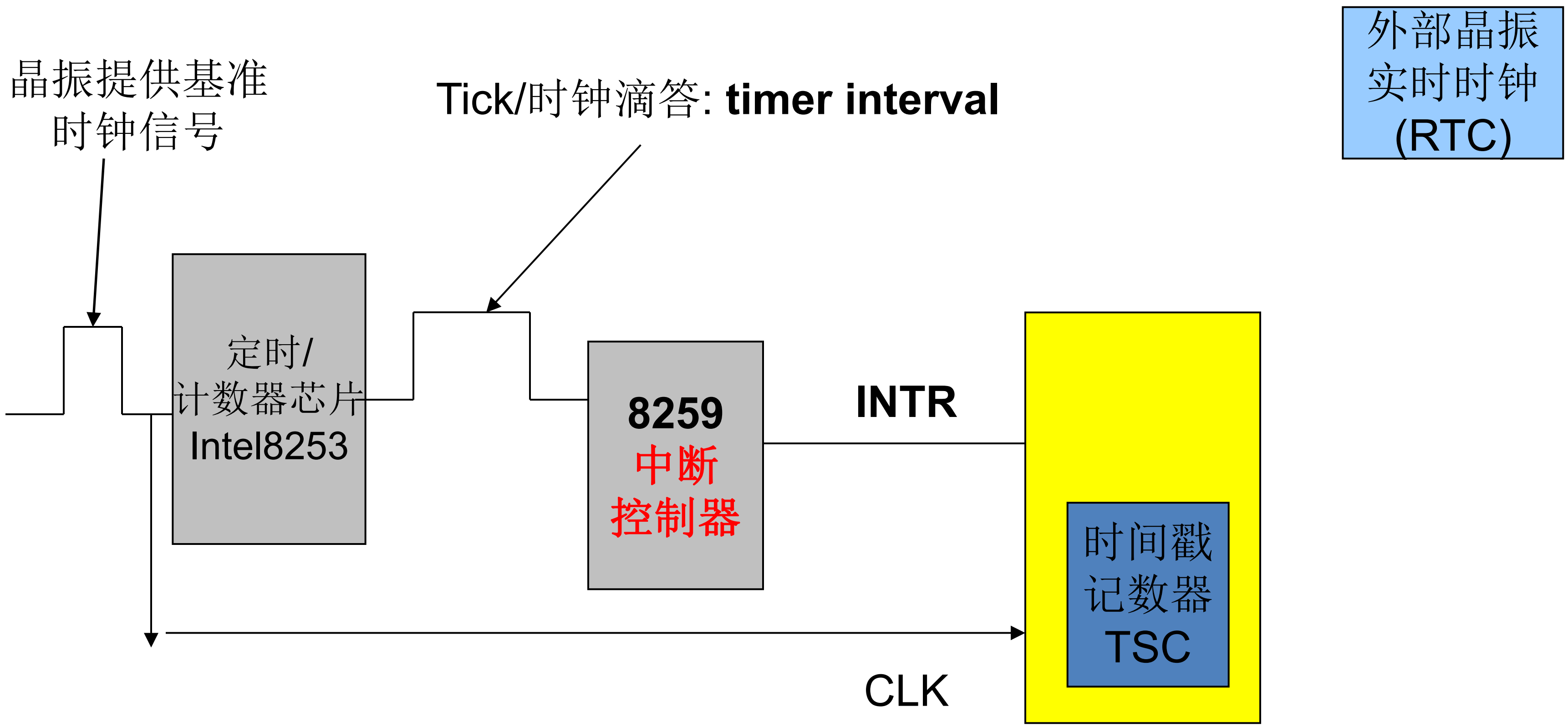
- E.g.: in IA32/ x86



8253具有3个独立的计数通道，采用减1计数方式。在门控信号有效时，每输入1个计数脉冲，通道作1次计数操作。当计数脉冲是已知周期的时钟信号时，计数就成为定时。时钟芯片再以这个时钟频率为参考，在芯片内部进行一系列的倍频动 作，为主板提供系统所需的各式类型信号频率。

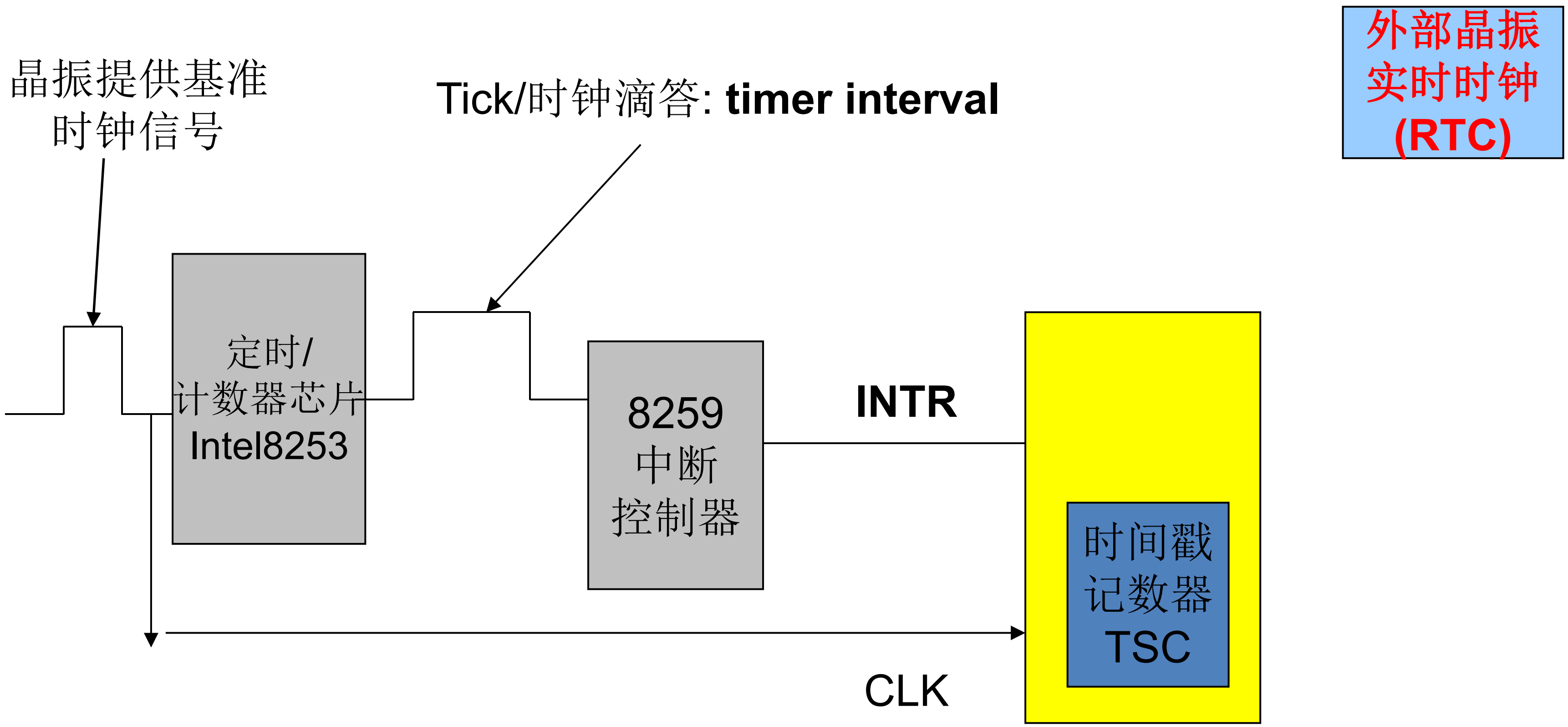
7.3.2.2 Timer in Hardware (3/11)

- E.g.: in IA32/ x86



8253计数器0的输出端OUT0与8259的中断请求输入IRQ0相连，设定8253的计时时间为20ms，计时满后，8259产生一次时钟中断，然后调用时钟中断处理程序(time_interrupt)。

7.3.2.2 Timer in Hardware (4/11)

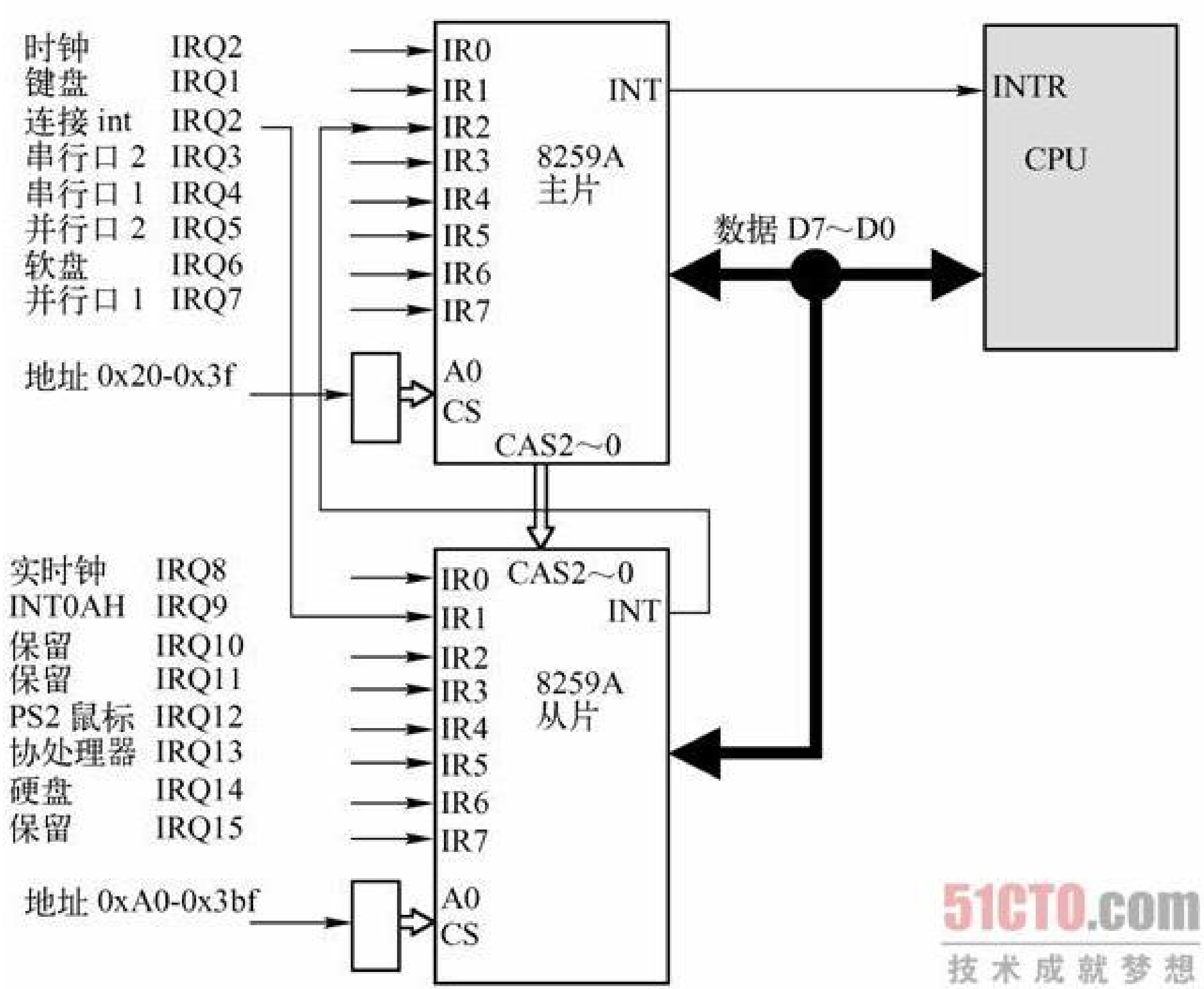


RTC: Real Time Clock. This is the battery-backed clock that keeps time even when the system is shut down. Within the chip is also the 64 bytes of CMOS RAM. Generate clock ticks on IRQ8

7.3.2.2 Timer in Hardware (5/11)

外部晶振
实时时钟
(RTC)

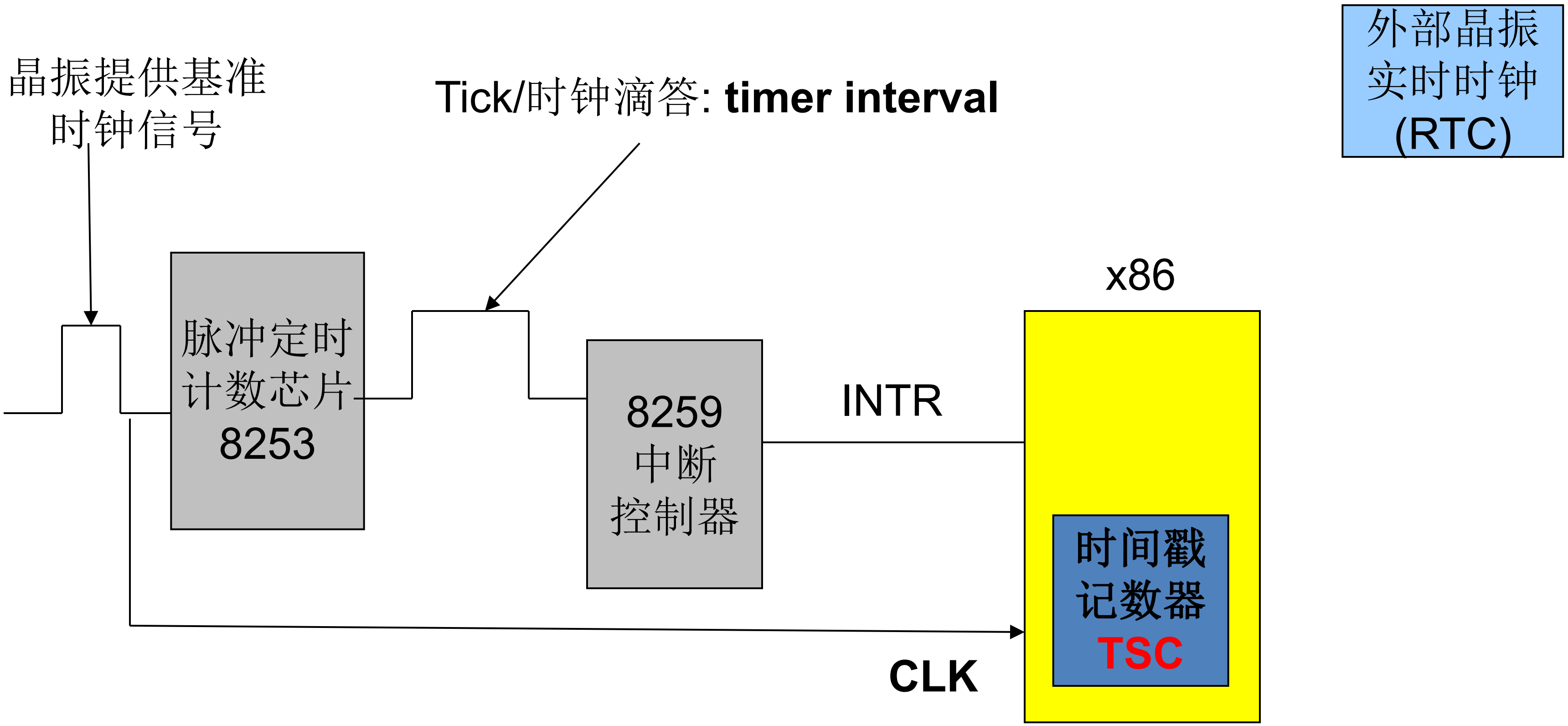
RT/CMOS RAM：
a battery-backed
one that is always
running,



Ref：http://book.csdn.net/bookfiles/824/10082424795.shtml

7.3.2.2 Timer in Hardware (6/11)

- E.g.: in IA32/ x86



在80x86微处理器中，有一个CLK输入引线接收外部振荡器的时钟信号。在每个时钟信号到来时,64位的时间戳记数器寄存器(TSC)加1。汇编指令rdtsc读这个寄存器。

7.3.2.2 Timer in Hardware (7/11)

- Intel Pentium processors (among others) have a very high-speed internal 64-bit counter that can be accessed by special instructions.
  - TSC(Time stamp counter): It counts the number of cycles since reset.
  - rdstc instruction: returns the TSC in EDX:EAX.

- Windows operating systems have an interface to access this high-precision timer.

7.3.2.2 Timer in Hardware (8/11)

- High-Resolution Timer Function
  - The QueryPerformanceFrequency() function retrieves the frequency of the high-resolution performance counter, if one exists.

```
BOOL QueryPerformanceFrequency (
        LARGE_INTEGER* lpFrequency
          // address of current frequency  );
```

  - The QueryPerformanceCounter() function retrieves the current value of the high-resolution performance counter, if one exists.

```
BOOL QueryPerformanceCounter(
          LARGE_INTEGER *lpPerformanceCount
           // pointer to counter value );
```

7.3.2.2 Timer in Hardware (9/11)

- LARGE_INTEGER
  - The LARGE_INTEGER structure is used to represent a 64-bit signed integer value.
  - If your compiler has built-in support for 64-bit integers, use the **QuadPart** member to store the 64-bit integer.
  - Otherwise, use the **LowPart** and **HighPart** members to store the 64-bit integer.

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER;
```

7.3.2.2 Timer in Hardware (10/11)

```
LARGE_INTEGER litmp;
LONGLONG QPart1,QPart2;
double dfFreq;
double dfMinus, dfTim;

QueryPerformanceFrequency(&litmp);
dfFreq = (double)litmp.QuadPart;// 获得计数器的时钟频率

QueryPerformanceCounter(&litmp);
QPart1 = litmp.QuadPart;// 获得初始值

Sleep(100);

QueryPerformanceCounter(&litmp);
QPart2 = litmp.QuadPart;//获得中止值

dfMinus = (double)(QPart2-QPart1);
dfTim = dfMinus / dfFreq;// 获得对应的时间值，单位为秒
```

Ref：http://www.vckbase.com/document/viewdoc/?id=1301 VC中基于 Windows 的精确定时

7.3.2.2 Timer in Hardware (11/11)

- Example code
  - using the high precision timer under Windows can be found in the .zip file precise.zip
    - void precise_start() :begin timing
    - double precise_stop() :to get the elapsed time in seconds

7.3.2 Timing Mechanisms

- 7.3.2.1 Introduction
- 7.3.2.2 Timer in Hardware
- <u>7.3.3.3 Timer in OS</u>
- 7.3.3.4 Timer in C/C++

7.3.3.3 Timer in OS (1/3)

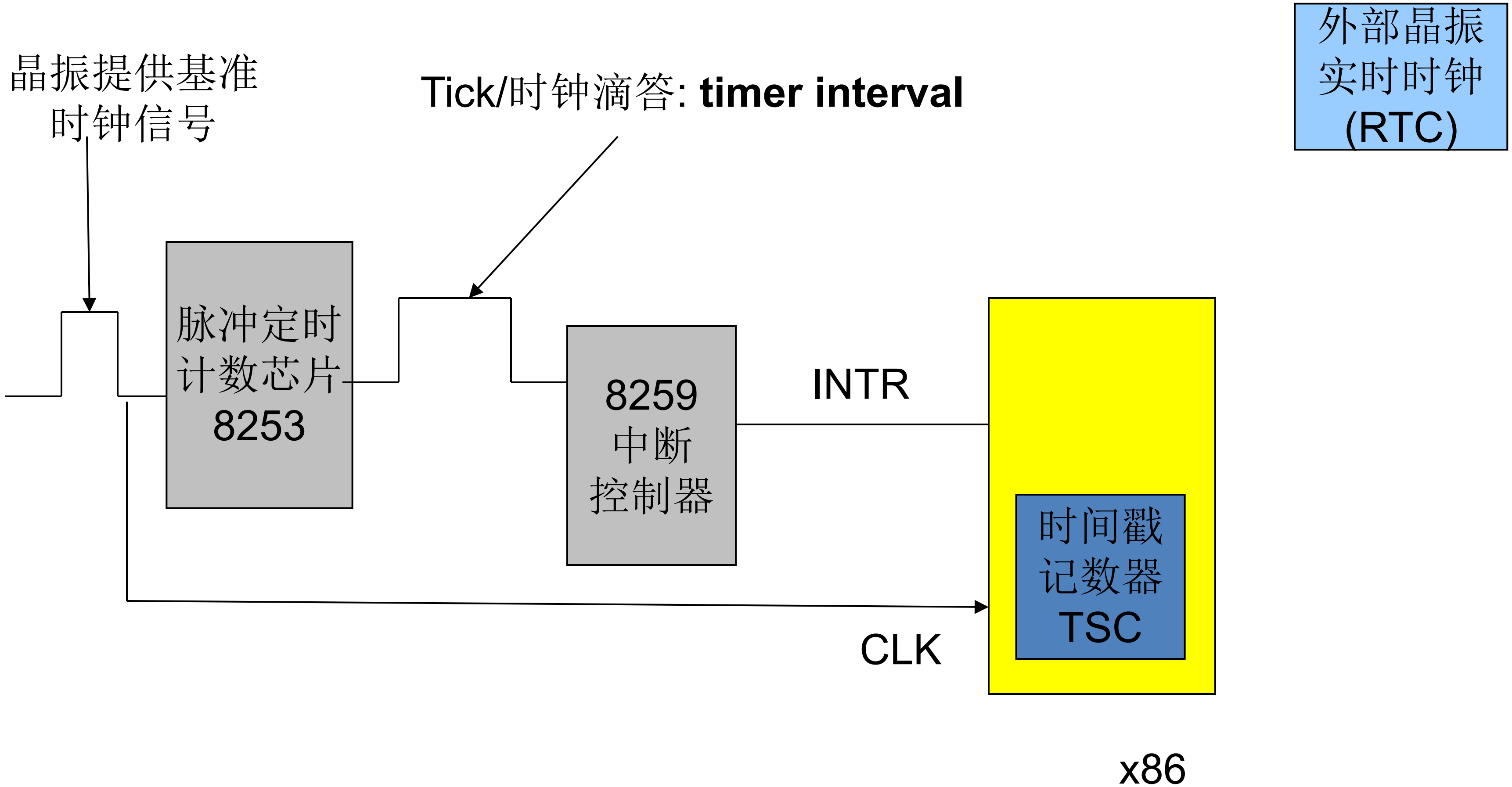
- Linux
  - 内部实现
    - 计时始于：by 启动的时候读取RTC后，进行转换
    - 1970年1月1日 午夜0点
  - 系统调用：times（）、结构tms
  - 系统调用：gettimeofday（）、结构timeval
  - 系统调用：SIGALRM信号和alarm（）函数

  - 时间片：OS Scheduler

Ref：教材CSAPP 9.2.2 and 9.3

7.3.3.3 Timer in OS (2/3)

- Time slice = A specific number of clock ticks before process gets moved to another state. (about 10-15ms)



7.3.3.3 Timer in OS (3/3)

- Windows
  - 内部实现
    - 计时始于：启动的时候读取RTC后，进行转换
    - 1970年1月1日 中午12:00点
  - 系统调用：GetLocalTime（）、GetSystemTime（）、结构SYSTEMTIME
  - 系统调用：WM_TIMER消息和SetTimer（）函数

  - 时间片：OS Scheduler

Ref：<Windows 程序设计> 5th chapter6 by Charles Petzold
Ref: http://hi.baidu.com/writer_wjr/blog/item/d7e4d84f8194d202b2de05ec.html
Windows 获取当前系统时间函数总结

7.3.2 Timing Mechanisms

- 7.3.2.1 Introduction
- 7.3.2.2 Timer in Hardware
- 7.3.3.3 Timer in OS
- <u>7.3.3.4 Timer in C/C++</u>

7.3.3.4 Timer in C/C++ (1/4)

- 数据类型： clock_t, time_t
- 宏： CLOCKS_PER_SEC： 一秒钟内CPU运行的时钟周期数
  - POSIX定义CLOCKS_PER_SEC为一百万，无关乎 clock 的实际精度
- 结构： struct tm
- 函数
  - Clock（）
  - Time（）
  - Difftime（）
  - Mktime（）
  - Asctime（）
  - Ctime（）
  - Gmtime（）
  - Localtime（）
  - Strftime（）

- Ref: c programming language
  - B.10

7.3.3.4 Timer in C/C++ (2/4)

```
time.h
#define CLOCKS_PER_SECOND((clock_t)1000)

#ifndef _CLOCK_T_DEFINED
typedef long clock_t;
#define _CLOCK_T_DEFINED
#endif
```

7.3.3.4 Timer in C/C++ (3/4)

- clock()
  - **clock_t clock(void);**
  - **Required header** <time.h>
  - **Return Value**: clock returns the number of processor timer ticks that have elapsed.
  - R**emarks**:
    - The clock function tells how much processor time the calling process has used.
    - The time in seconds is approximated by dividing the clock return value by the value of the **CLOCKS_PER_SEC** constant.
    - A timer tick is approximately equal to 1/**CLOCKS_PER_SEC** second.

7.3.3.4 Timer in C/C++ (4/4)

- clock() in C/C++

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
void my_subroutine(long n) {
        // timing a subroutine call:
        char s[16];
        for (long i = 0; i < n; i++) {
                _itoa(i, s, sizeof(s));
        }
}
```

```
int main(int argc, char* argv[]) {
        long n = 1000000;
        clock_t start = clock();
         my_subroutine(n);
        clock_t finish = clock();
        double duration = (double)(finish - start) / CLOCKS_PER_SEC;
        cout << "Time for " << n << " iterations: " << duration << "s,"
              << " precision is " << CLOCKS_PER_SEC
              << " clocks per second." << endl;
        return 0;
```

7.3 Performance Measurement

- 7.3.1 What to Measure
- 7.3.2 Timing Mechanisms
- 7.3.3 Statistical Sampling (统计抽样) /Profiling

7.3.3 Statistical Sampling/Profiling(1/3)

- In this approach, a timer periodically interrupts the program and records the program counter
- This approach can only estimate where time is spent in the program.
- Fortunately, if the program spends most of its time in a few places, these are almost certain to be identified and accurately measured.

7.3.3 Statistical Sampling/Profiling(2/3)

- Software profiler - a program that benchmarks the execution of one or more pieces of procedural code to help the user understand where the time is being spent in terms of code execution.
  - Gprof
  - VTune
  - Visual C++ Profiler
  - IBM Quantify

7.3.3 Statistical Sampling/Profiling(3/3)

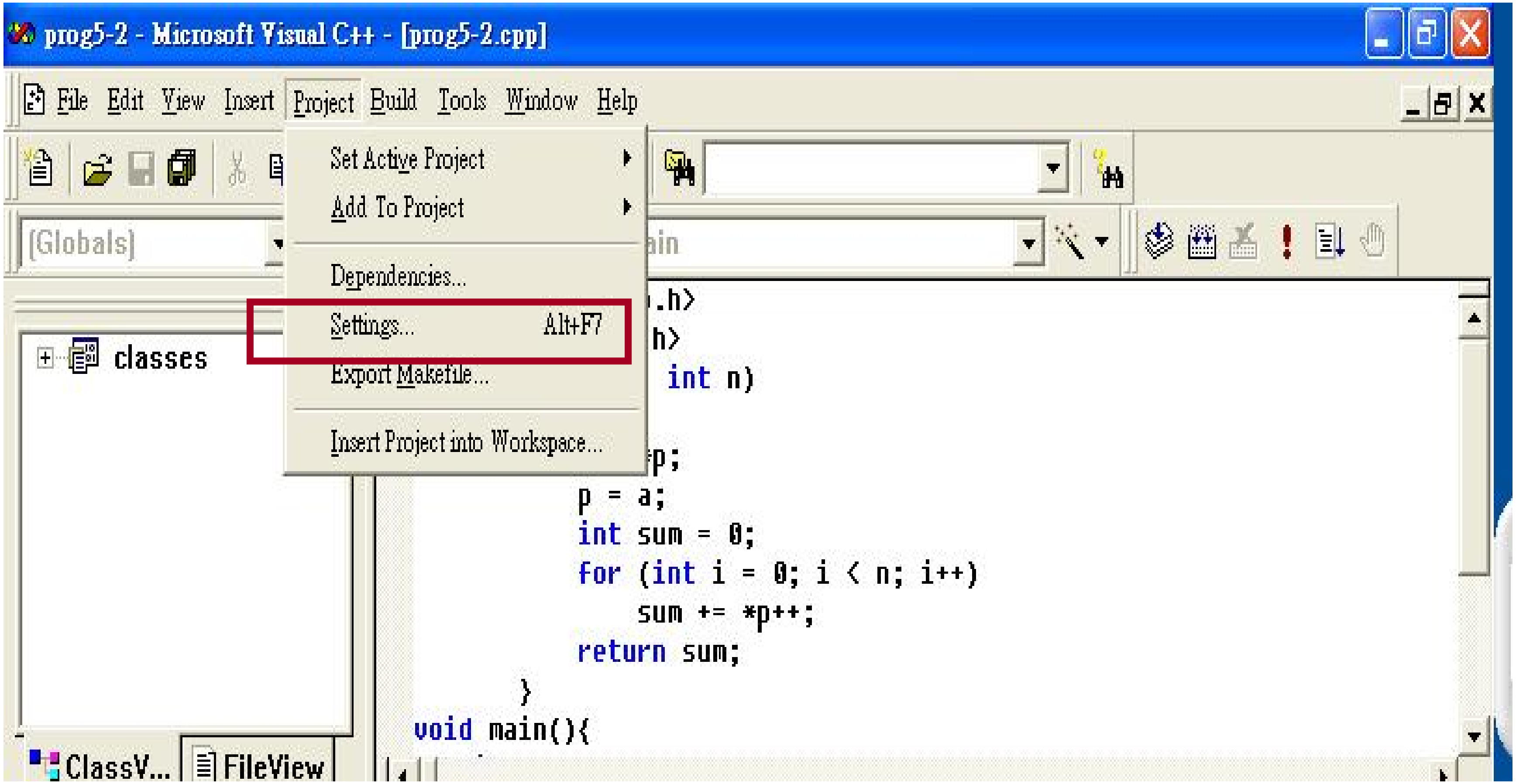
- VC Profiler
  - In Visual C++, the **Profile...** entry in the **Build** menu gives instructions on obtaining a profile. (Note: Profiling is only available in the Professional and Enterprise editions of Visual C++.)
  - We will discuss more later on

Unit 7: Performance Measurement

- 7.1 Rationale for this unit
- 7.2 Performance Principles
- 7.3 Performance Measurement
- 7.4 VC Profiler
- 7.5 Code::Blocks Profiler

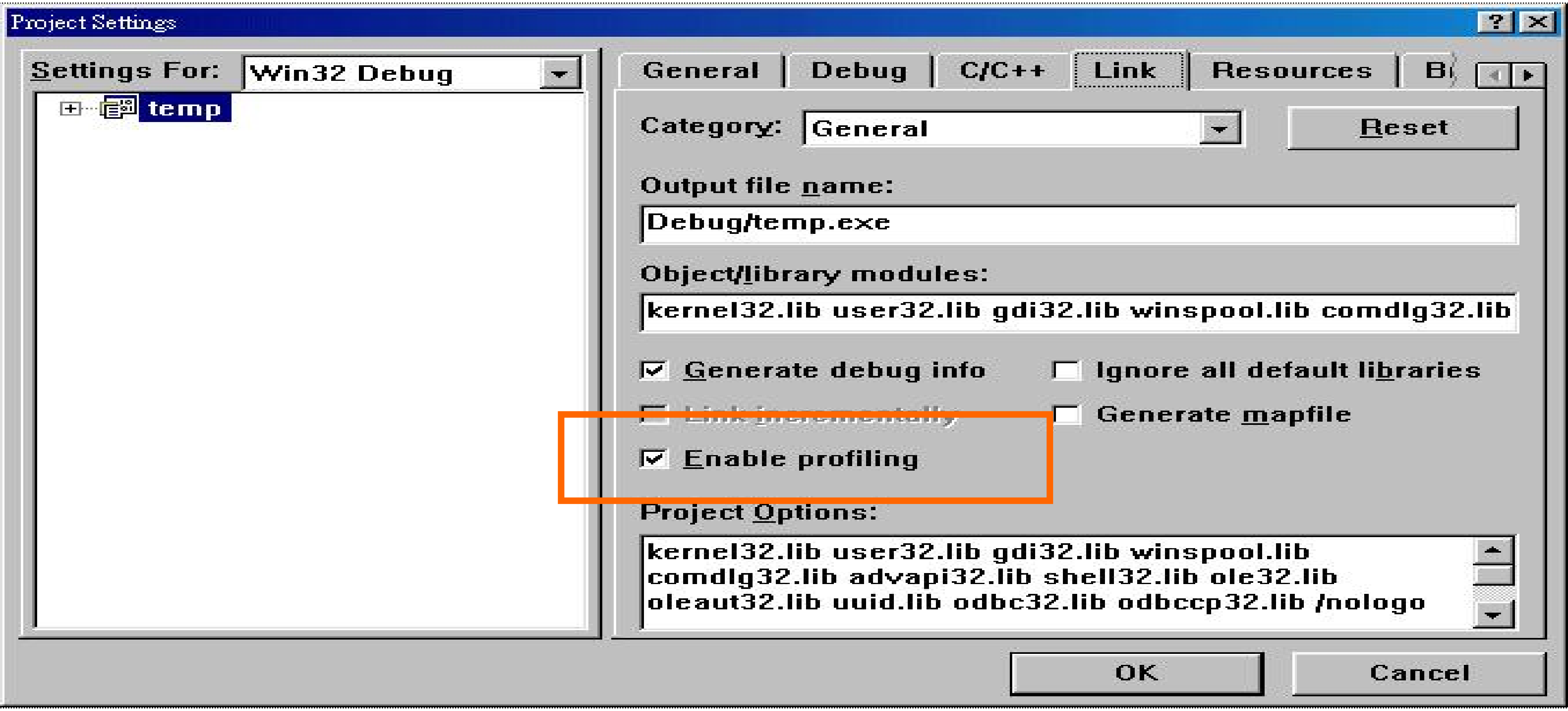
7.4 VC Profiler (1/18)

- Procedure (1) – setting



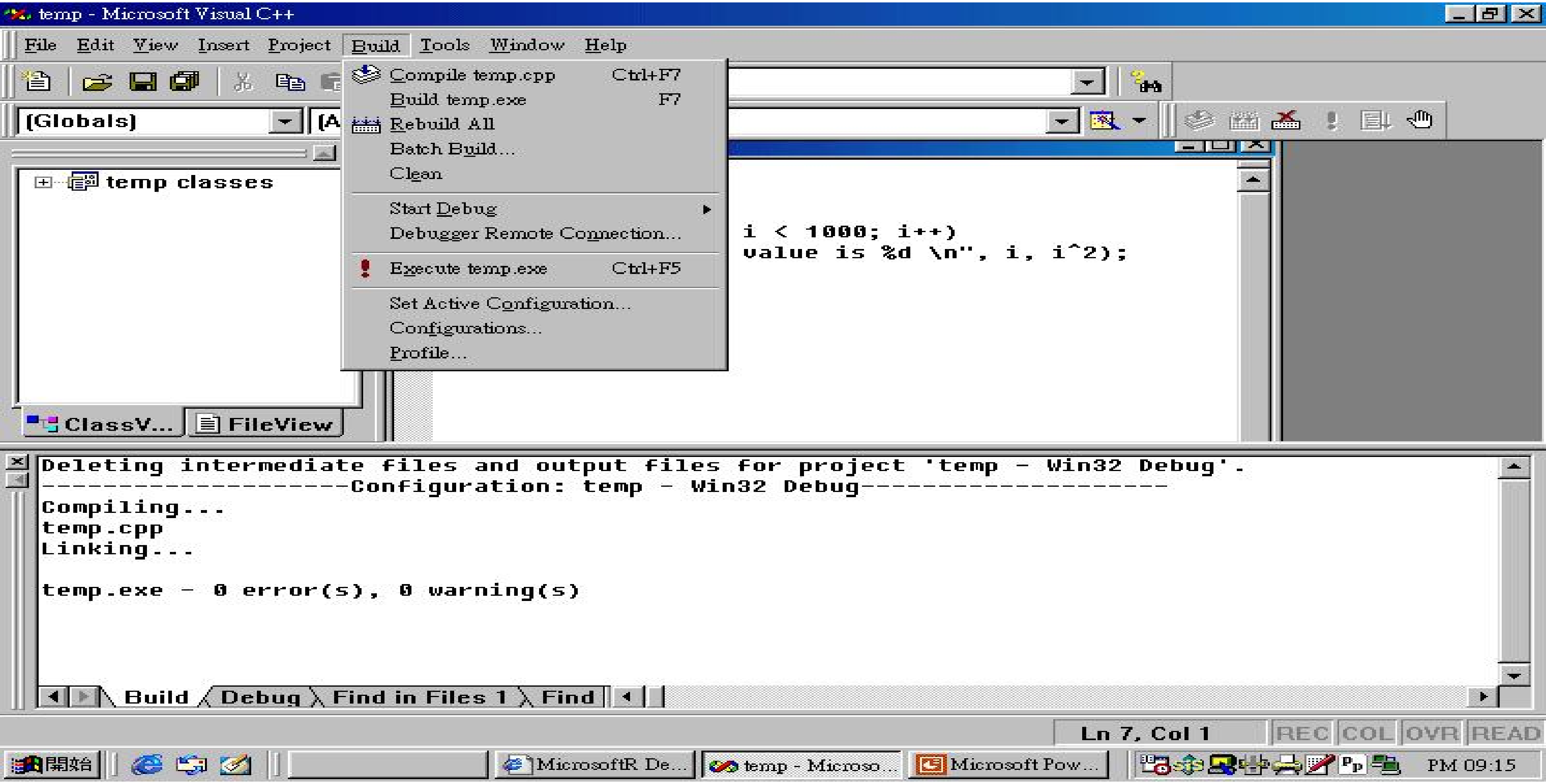
7.4 VC Profiler (2/18)

- Procedure (2) – enable profiling



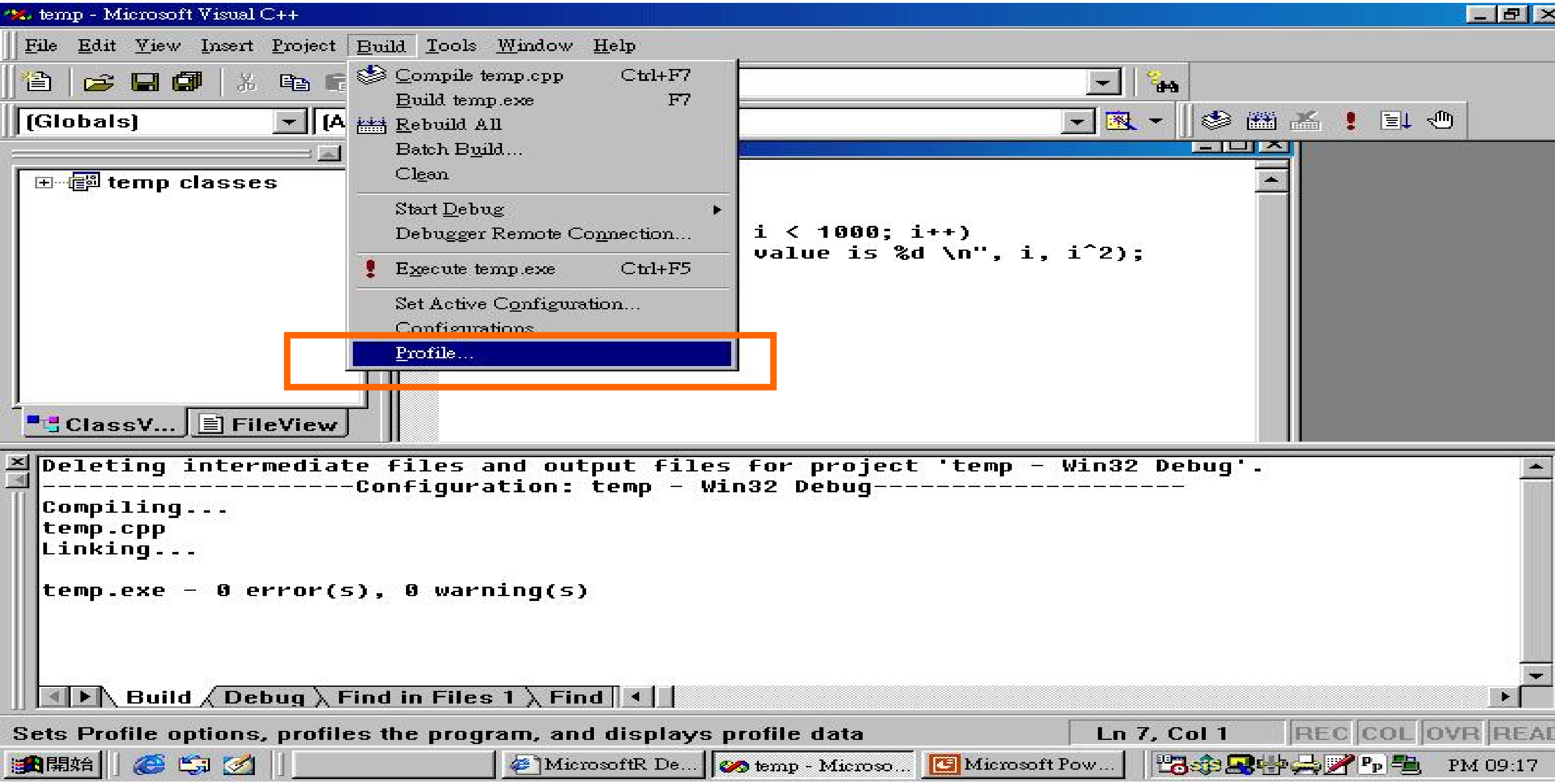
7.4 VC Profiler (3/18)

- Procedure (3) – rebuild



7.4 VC Profiler (4/18)

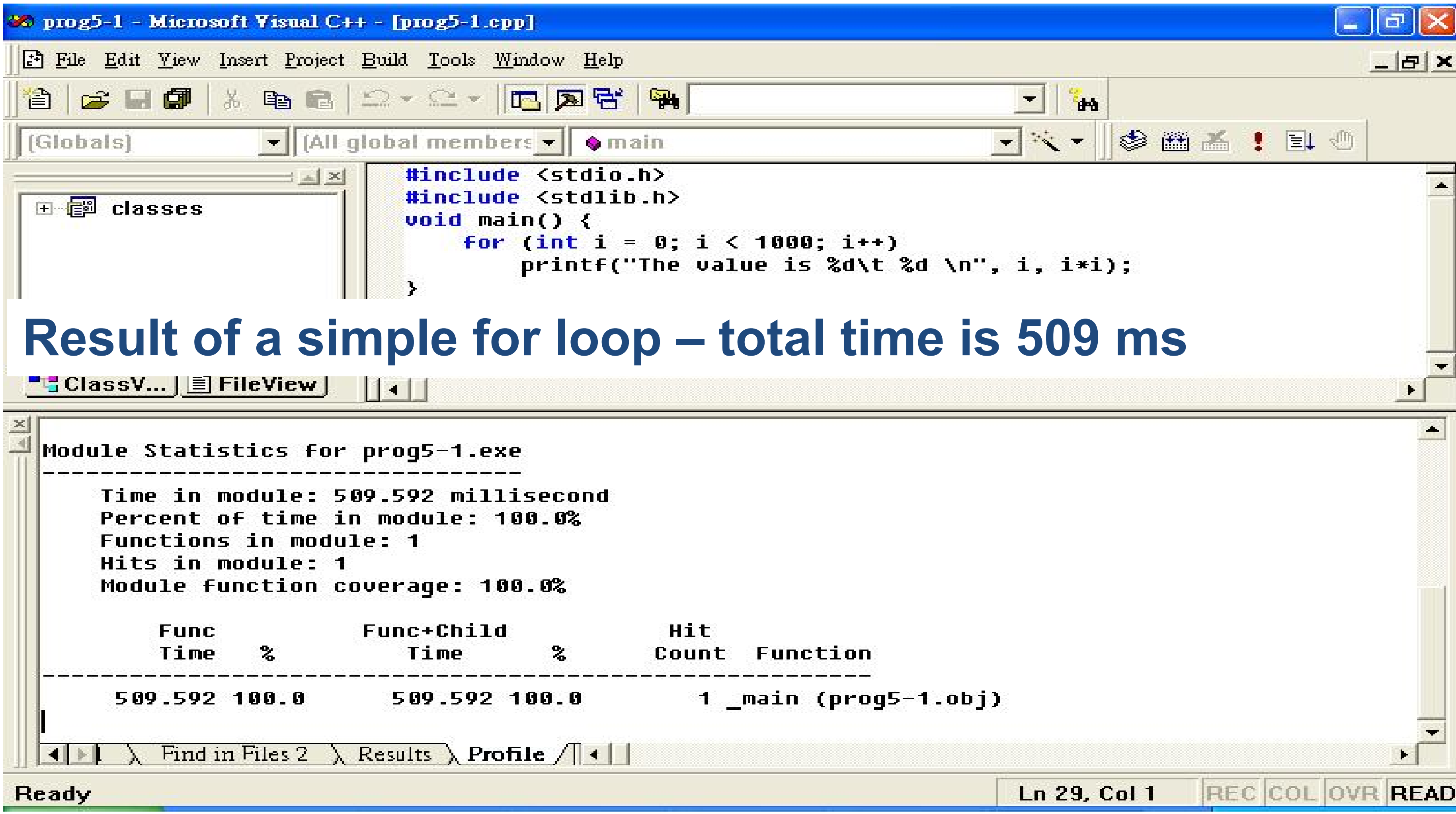
- Procedure (4) – run with profiling



7.4 VC Profiler (5/18)

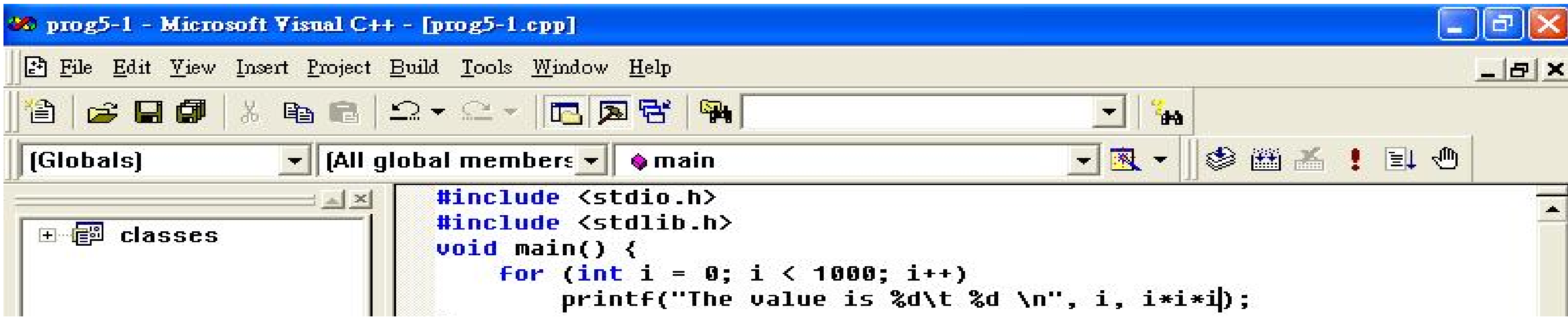
```
#include <stdio.h>
#include <stdlib.h>
void main()
{
   int I;
    for (i = 0; i < 1000; i++)
          printf("The value is %d\t %d \n", i, i*i);
}
```

7.4 VC Profiler (6/18)

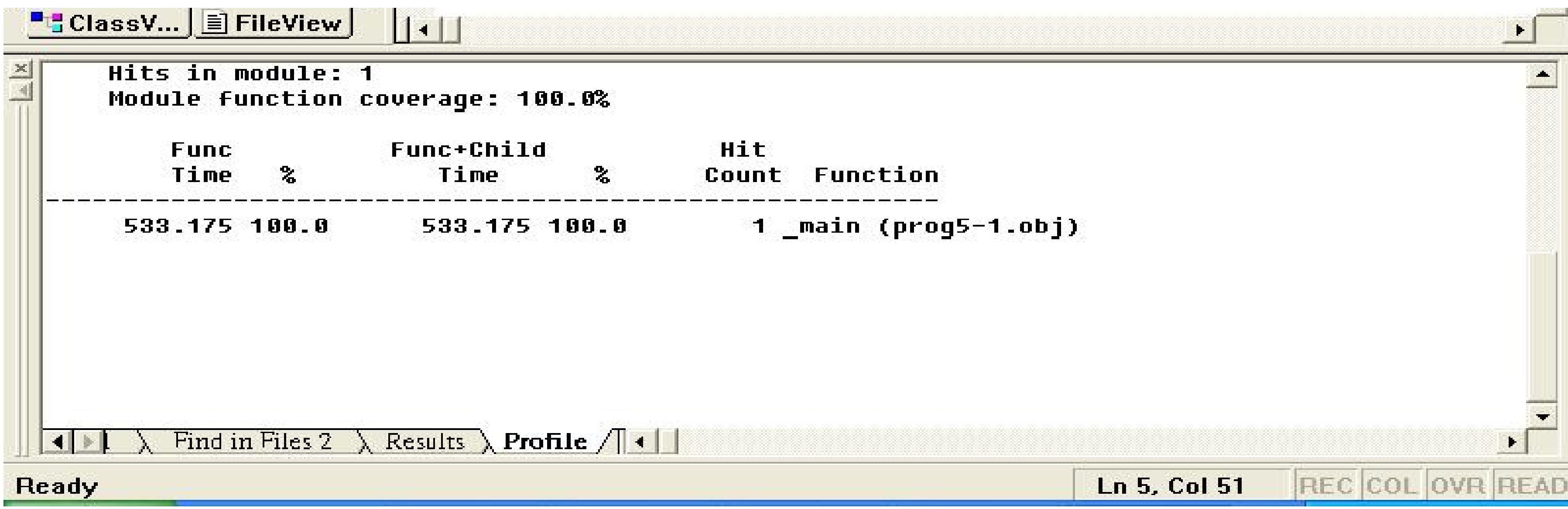


Result of a simple for loop – total time is 509 ms

7.4 VC Profiler (7/18)



Result of a simple for loop – total time is 533 ms



7.4 VC Profiler (8/18)

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int i = 0;
    while (i < 1000) {
        printf("The value is %d\t %d \n", i, i*i);
        i++;
    }
}
```

7.4 VC Profiler (9/18)

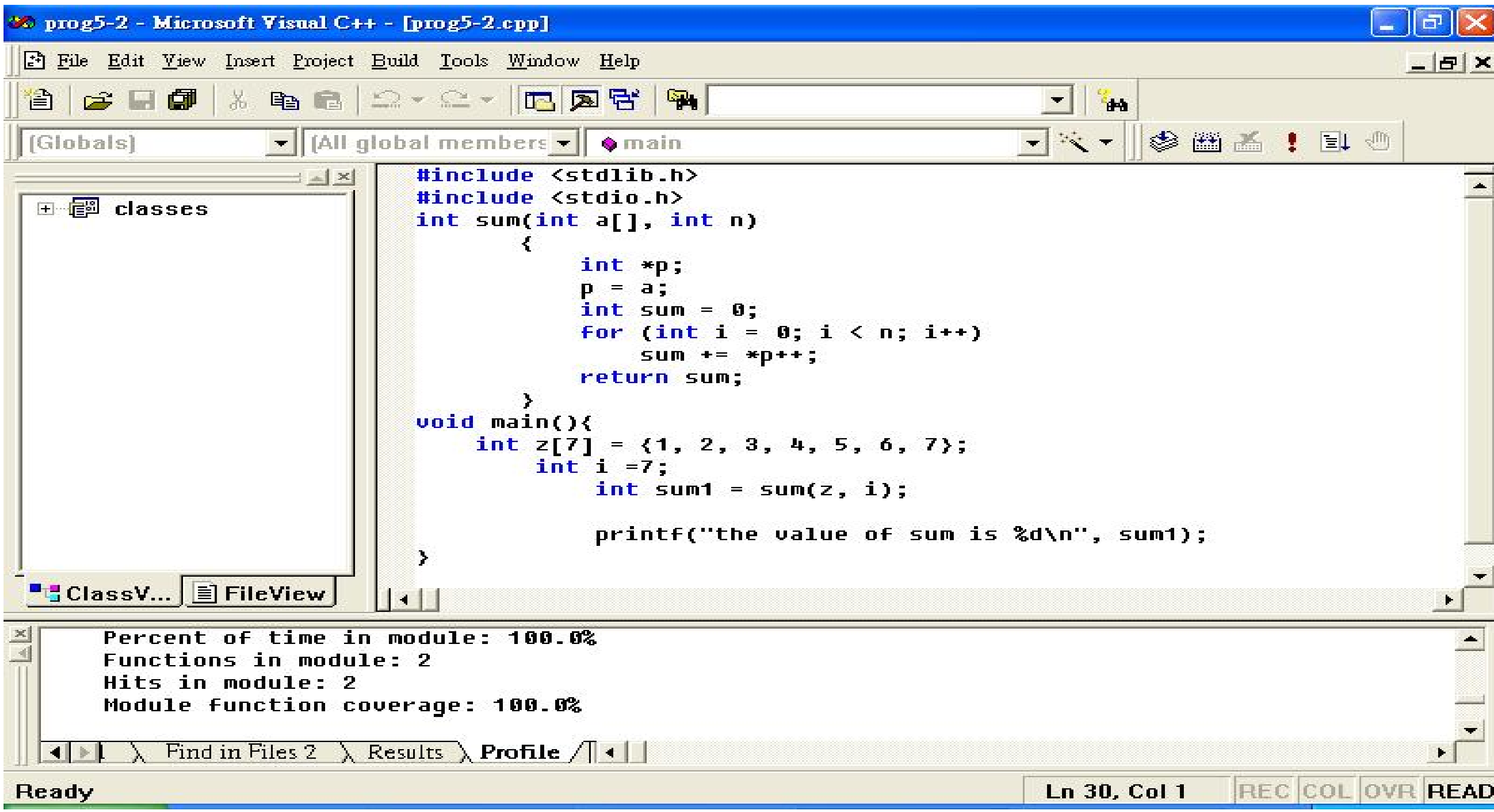
- Example – result in millisecond second

```
Percent of time in module: 100.0%
Functions in module: 1
Hits in module: 1
Module function coverage: 100.0%

      Func          Func+Child           Hit
      Time     %       Time        %    Count  Function
---------------------------------------------------------
   470.315 100.0     470.315 100.0        1 _main (temp.obj)
```

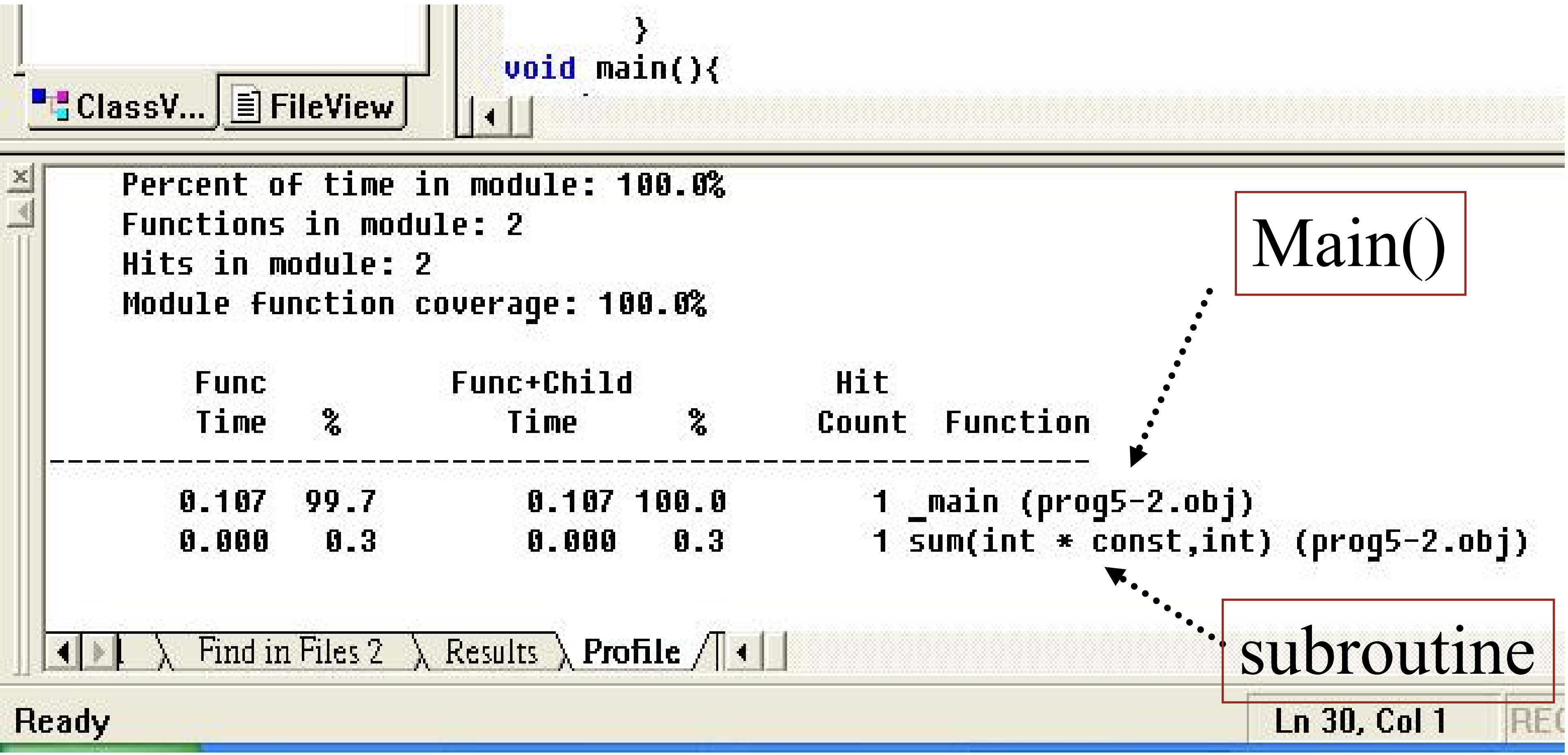
7.4 VC Profiler (10/18)

- Example with a sub-routine



7.4 VC Profiler (11/18)

- Example with a sub-routine



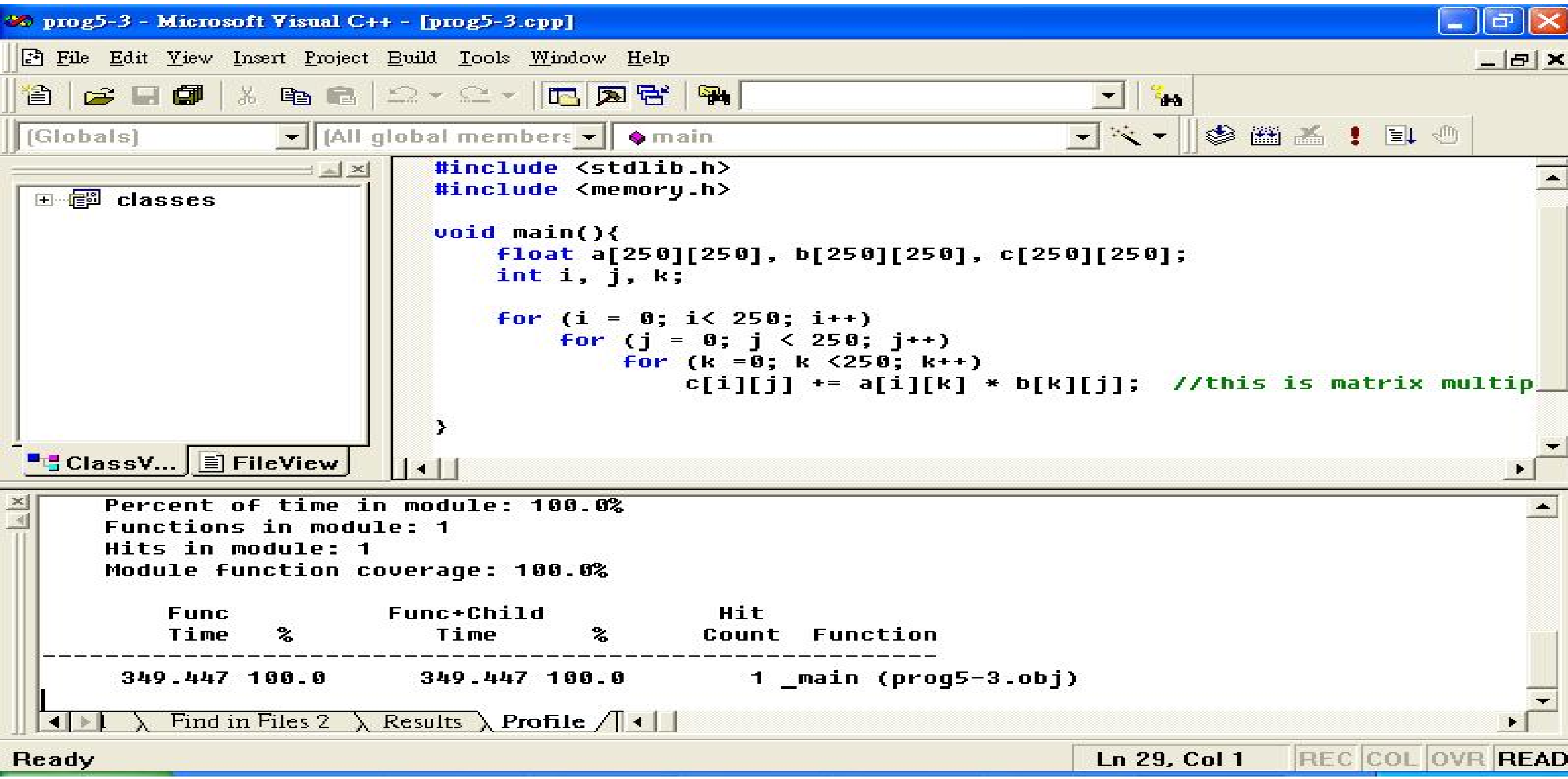
7.4 VC Profiler (12/18)

- A program that can be used to determine MFLOP

```
// This is matrix multiplication
#include <stdio.h>
#include <stdlib.h>
void main(){
        float a[250][250], b[250][250], c[250][250];
        int i, j, k;
        for (i = 0; i< 250; i++)
       for (j = 0; j < 250; j++)
           for (k =0; k <250; k++)
               // matrix multiplication
                       c[i][j] += a[i][k] * b[k][j];
}
```

7.4 VC Profiler (13/18)

- Performance is 349ms



7.4 VC Profiler (14/18)

- Determination of Mega Flop
  - The time it takes for my machine is 349ms.
  - This program involves 250^3 steps including two floating point operations, an add and a multiply 250 x 250 x 250 = 15625000.
  - The performance for this loop is 15625000/349ms = 15.625 x 10^6 /0.349 s = 44 MFLOPs (mega floating point operation).
  - Note that for super computer, the value is about 1000 MFLOPs.
  - You can try your computer at lab to determine your machine's performance.

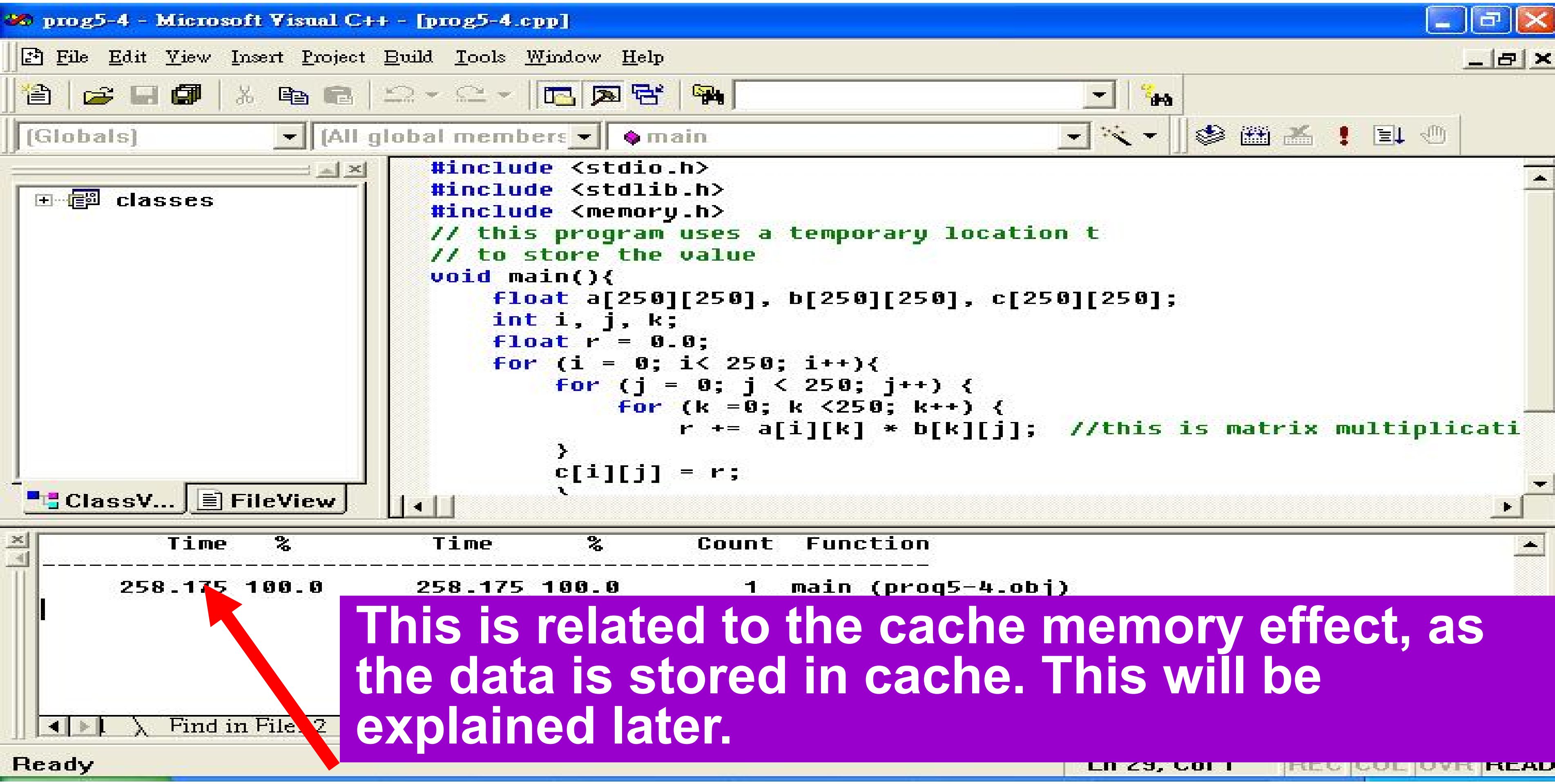
7.4 VC Profiler (15/18)

- Same output but change the program

```
#include <stdio.h>
#include <stdlib.h>
// this program uses a temporary location t to store the value
void main() {
    float a[250][250], b[250][250], c[250][250];
    int i, j, k;
    float r = 0.0;
    for (i = 0; i< 250; i++) {
        for (j = 0; j < 250; j++) {
            for (k =0; k <250; k++)
                r += a[i][k] * b[k][j];  //this is matrix multiplication
             c[i][j] = r;
        }
    } }
```

7.4 VC Profiler (16/18)

- Same machine – 258ms, why?



This is related to the cache memory effect, as the data is stored in cache. This will be explained later.

7.4 VC Profiler (17/18)

- A profiler can:
  - measure the elapsed time taken to run source code
  - Provide coverage analysis
  - Show program’s execution history
  - How many times a piece of code has been exec.
  - Time spent by all exec. of that piece of code

- It is a great way to find where bottlenecks occur, so we can make our code more efficient

7.4 VC Profiler (18/18)

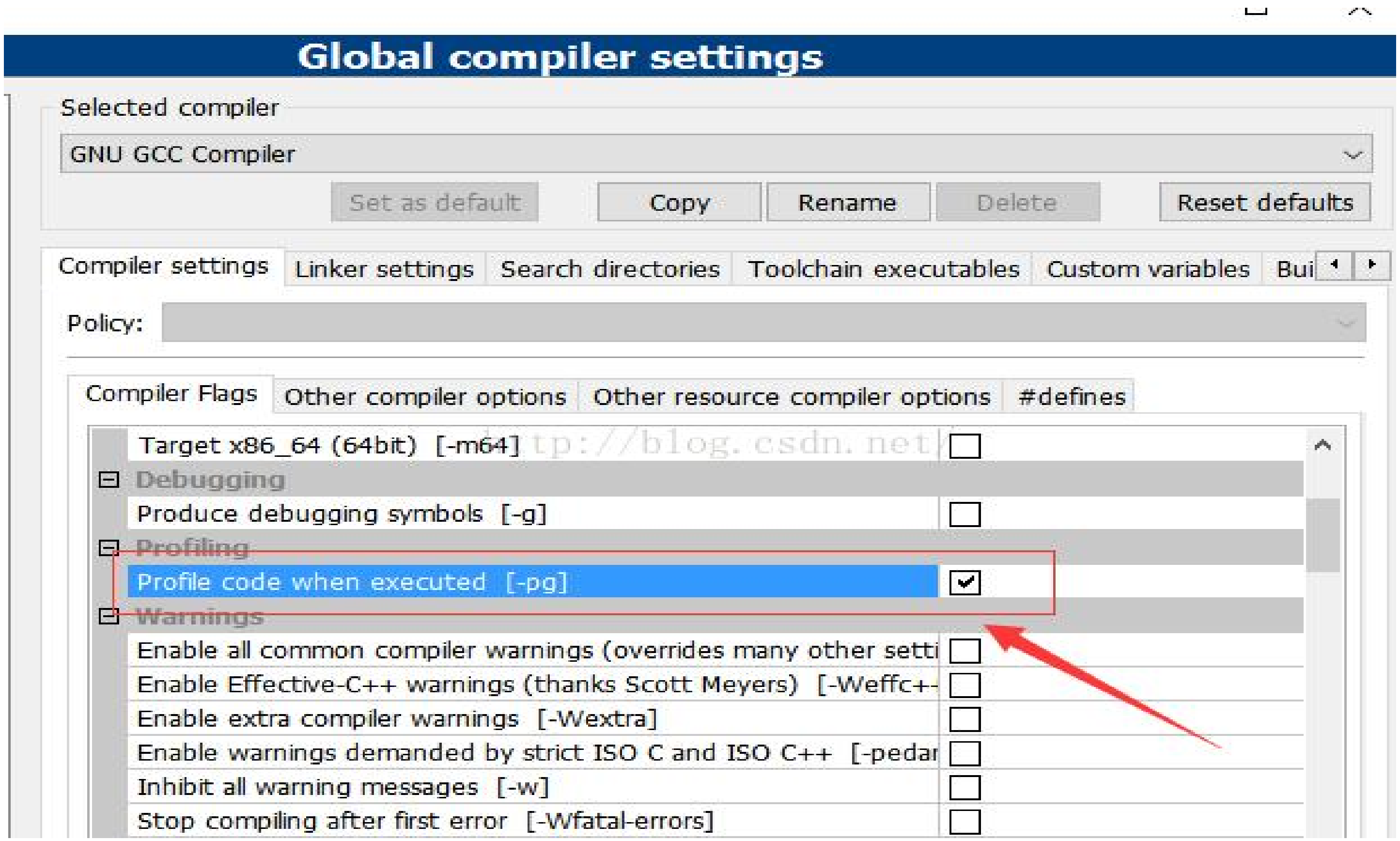
- Profiling is used to help programmers to identify
  - which areas of the program are causing sluggish bottlenecks.
  - which parts of the code are being called the most often.

Unit 7: Performance Measurement

- 7.1 Rationale for this unit
- 7.2 Performance Principles
- 7.3 Performance Measurement
- 7.4 VC Profiler
- <u>7.5 Code::Blocks Profiler</u>

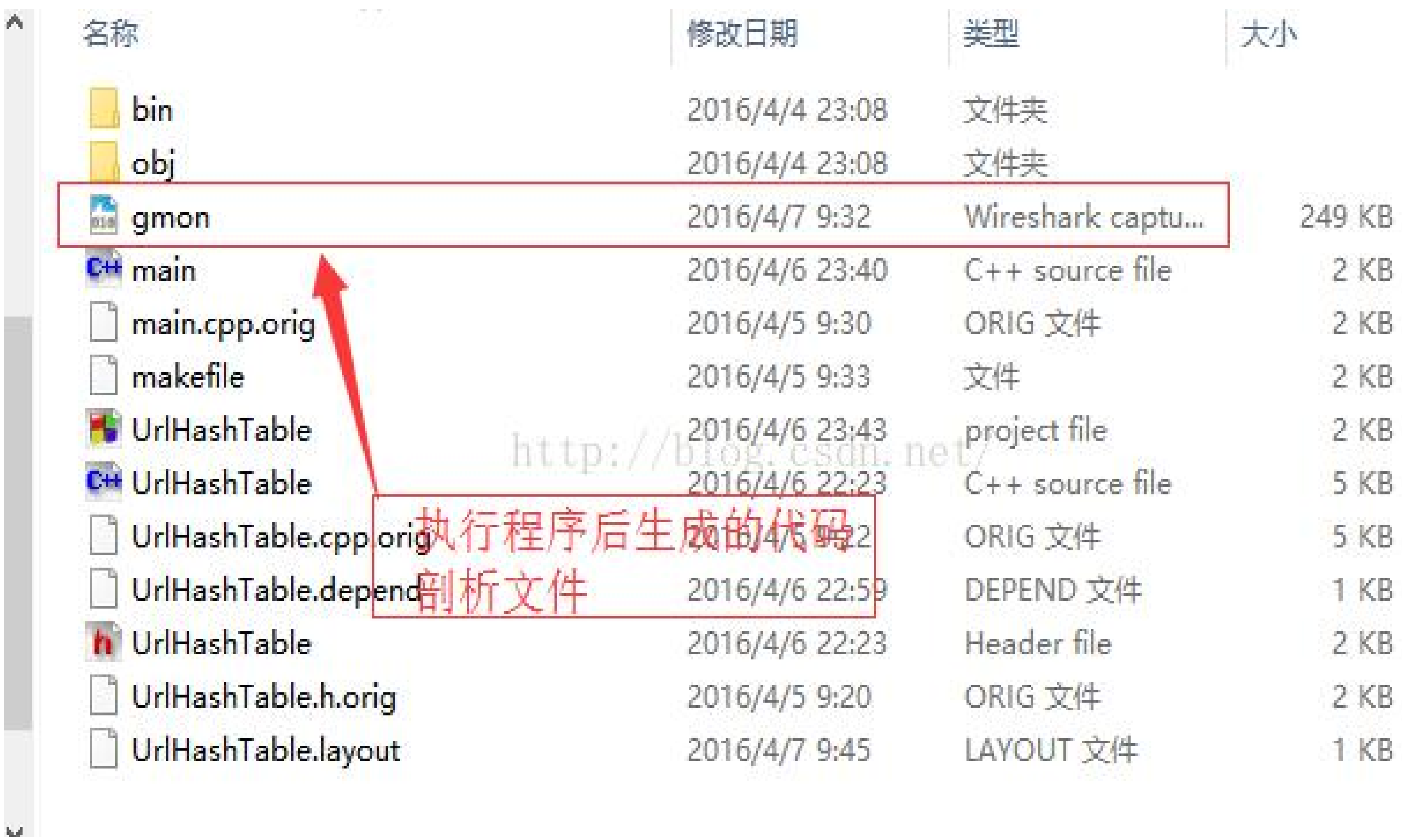
7.5 Code::Blocks Profiler(1/3)

- Ref: https://blog.csdn.net/qianghaohao/article/details/51082930
- 1. Enable profiler in code::blocks



7.5 Code::Blocks Profiler(2/3)

- 2. create profile -- compile and run



7.5 Code::Blocks Profiler(3/3)

- 3. Plugins-->Code profiler

C::B Profiler Results

Gprof's Output

Flat Profile | Call Graph | Misc

% time	cum. sec.	self sec.	calls	self ms/call	total ms/call	name
39.13	0.09	0.09				HashString(char const*)
30.43	0.16	0.07	2400000	0.00	0.00	UrlHashTable::AddUrl(UrlHashTable*, char const*)
17.39	0.20	0.04	2400004	0.00	0.00	main
8.70	0.22	0.02				std::string::operator+=(char)
4.35	0.23	0.01	1	10.00	10.00	UrlHashTable::DeleteHashTable(UrlHashTable*)
0.00	0.23	0.00	1	0.00	0.00	UrlHashTable::CreateHash(unsigned int)
0.00	0.23	0.00	1	0.00	0.00	UrlHashTable::DeleteUrl(UrlHashTable*, char const*)
0.00	0.23	0.00	1	0.00	0.00	UrlHashTable::SearchUrl(UrlHashTable*, char const*)
0.00	0.23	0.00	1	0.00	0.00	UrlHashTable::UrlHashTable()

http://blog.csdn.net/

```
%         the percentage of the total running time of the
```