使用 git 將所有範例下載

```
git clone https://github.com/shiwulo/system-programming.git
```

使用 gdb 對 rdtsc.c 除錯

```
make
gdb ./rdtsc
```

in rdtsc.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <time.h>
#include <stdint.h>
#include <stdlib.h>
#include <sched.h>
#include <string.h>
#include <sched.h>
#include <assert.h>

/*
The RDTSCP instruction reads the same TSC as the RDTSC instruction, so if RDTSC
is invariant, then RDTSCP will be as well.
RDTSCP is slightly more ordered than RDTSC.  RDTSC is not ordered at all, which
means that it will execute some time in the out-of-order window of the processor,
which may be before or after the instruction(s) that you are interested in
timing.   RDTSCP will not execute until all prior instructions (in program order)
have executed.  So it can't execute "early", but there is no guarantee that the
execution won't be delayed until after some subsequent (in program order)
instructions have executed.
*/
extern __inline__ uint64_t rdtscp(void)
{
    uint32_t lo, hi;
    // take time stamp counter, rdtscp does serialize by itself, and is much
cheaper than using CPUID
    __asm__ __volatile__("rdtscp":"=a"(lo), "=d"(hi));
    return ((uint64_t) lo) | (((uint64_t) hi) << 32);
}

long ts_to_long(struct timespec t) {
    return t.tv_sec * 1000000000 + t.tv_nsec;
}

int main(int argc, char **argv)
{
    int tmp;
    uint64_t cycles1, cycles2;
    struct timespec ts1, ts2;
```

```
    printf("這個程式是量測一個指令執行的時間，但CPU可同時執行數十個指令\n");
    printf("因此這些量測方法比較適合量測大範圍的程式碼\n\n");

    cycles1 = rdtscp();
    tmp++;
    cycles2 = rdtscp();

    clock_gettime(CLOCK_MONOTONIC, &ts1);
    tmp++;
    clock_gettime(CLOCK_MONOTONIC, &ts2);

    printf("開始 %lu, 結束 %lu\n",cycles1, cycles2);
    printf("rdtscp: tmp++ consumes %lu cycles!\n", cycles2-cycles1);
    printf("開始 %lu, 結束 %lu\n",ts_to_long(ts1), ts_to_long(ts2));
    printf("clock_gettime: tmp++ consumes %lu nanoseconds!\n", ts_to_long(ts2)-
ts_to_long(ts1));
    assert(system("cat /proc/cpuinfo | grep 'cpu MHz' | head -1")>=0);
}
```

in GDB

```
Reading symbols from ./rdtsc...
(gdb) b main
Breakpoint 1 at 0x1100: file rdtsc.c, line 28.
(gdb) r
Starting program: /home/ubuntu2004/SP/GDB_test/system-programming/ch02/rdtsc

Breakpoint 1, main (argc=1, argv=0x7fffffffe3d8) at rdtsc.c:28
28      {
(gdb) b 36
Breakpoint 2 at 0x555555555132: file rdtsc.c, line 36.
(gdb) n
33          printf("這個程式是量測一個指令執行的時間，但CPU可同時執行數十個指令\n");
(gdb) c
Continuing.
這個程式是量測一個指令執行的時間，但CPU可同時執行數十個指令
因此這些量測方法比較適合量測大範圍的程式碼


Breakpoint 2, rdtscp () at rdtsc.c:36
36          cycles1 = rdtscp();
(gdb) n
20          return ((uint64_t) lo) | (((uint64_t) hi) << 32);
(gdb) b 38
Breakpoint 3 at 0x555555555141: file rdtsc.c, line 38.
(gdb) c
Continuing.

Breakpoint 3, rdtscp () at rdtsc.c:38
38          cycles2 = rdtscp();
(gdb) s
main (argc=<optimized out>, argv=<optimized out>)
    at rdtsc.c:20
20          return ((uint64_t) lo) | (((uint64_t) hi) << 32);
(gdb) p tmp
$1 = <optimized out>
```

這是因為編譯時加了編譯參數 -O3 所導致顯示 optimized out，所以我們用 vim 將 -O3 拿掉

```
make clean
vim makefile
```

```
SHELL = /bin/bash
CC = gcc
CFLAGS = -g
SRC = $(wildcard *.c)
EXE = $(patsubst %.c, %, $(SRC))

all: ${EXE}

%:  %.c
    ${CC} ${CFLAGS} $@.c -o $@

clean:
    rm ${EXE}
```

```
make
gdb ./rdtsc
```

```
Reading symbols from ./rdtsc...
(gdb) b 38
Breakpoint 1 at 0x123a: file rdtsc.c, line 38.
(gdb) r
Starting program: /home/ubuntu2004/SP/system-programming/ch02/rdtsc
這個程式是量測一個指令執行的時間，但CPU可同時執行數十個指令
因此這些量測方法比較適合量測大範圍的程式碼


Breakpoint 1, main (argc=1, argv=0x7fffffffe428) at rdtsc.c:38
38          cycles2 = rdtscp();
(gdb) p tmp
$1 = 1
(gdb) s
rdtscp () at rdtsc.c:19
19          __asm__ __volatile__("rdtscp":"=a"(lo), "=d"(hi));
(gdb) n
20          return ((uint64_t) lo) | (((uint64_t) hi) << 32);
(gdb) p lo
$2 = 3684310030
(gdb) backtrace
#0  rdtscp () at rdtsc.c:20
#1  0x000055555555523f in main (argc=1, argv=0x7fffffffe428) at rdtsc.c:38
(gdb) d
Delete all breakpoints? (y or n) n
(gdb) down
Bottom (innermost) frame selected; you cannot go down.
(gdb) up
#1  0x000055555555523f in main (argc=1, argv=0x7fffffffe428) at rdtsc.c:38
38          cycles2 = rdtscp();
(gdb) p lo
No symbol "lo" in current context.
(gdb) p tmp
```

```
$3 = 1
(gdb) b main
Breakpoint 2 at 0x5555555551ff: file rdtsc.c, line 28.
(gdb) awatch tmp
Hardware access (read/write) watchpoint 3: tmp
(gdb) c
Continuing.

Hardware access (read/write) watchpoint 3: tmp

Old value = 1
New value = 2
main (argc=1, argv=0x7fffffffe428) at rdtsc.c:42
42          clock_gettime(CLOCK_MONOTONIC, &ts2);
(gdb) info local
tmp = 2
cycles1 = 5140167280418
cycles2 = 5359508528142
ts1 = {tv_sec = 1481, tv_nsec = 948251738}
ts2 = {tv_sec = 140737353773000, tv_nsec = 93824992236416}
__PRETTY_FUNCTION__ = "main"
```

接著，我們在程式裡新增一個指標，不 malloc 也不指定任何變數的位址給它

```
int main(int argc, char **argv) {
    //...
    int *ptr;  // line 32

    printf("%d\n", *ptr);  // segmentation fault.
    //...
    return 0;
}
```

```
Reading symbols from ./rdtsc...
(gdb) r
Starting program: /home/ubuntu2004/SP/system-programming/ch02/rdtsc

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555219 in main (argc=1, argv=0x7fffffffe428) at rdtsc.c:34
34              printf("%d\n", *ptr); // segmentation fault.
```