시스템 프로그래밍

- GDB & 함수의 이용 -

2014. 10. 16.

박시형
sihyeong@cnu.ac.kr

Embedded System Lab. Computer Engineering Dept.
Chungnam National University

개요

실습 명

- 함수의 이용
- GDB의 사용

목표

- 함수를 사용 할 수 있다.
- GDB를 사용하여 프로그램을 디버깅 할 수 있다.
- 디버깅 제어명령을 사용할 수 있다.

내용

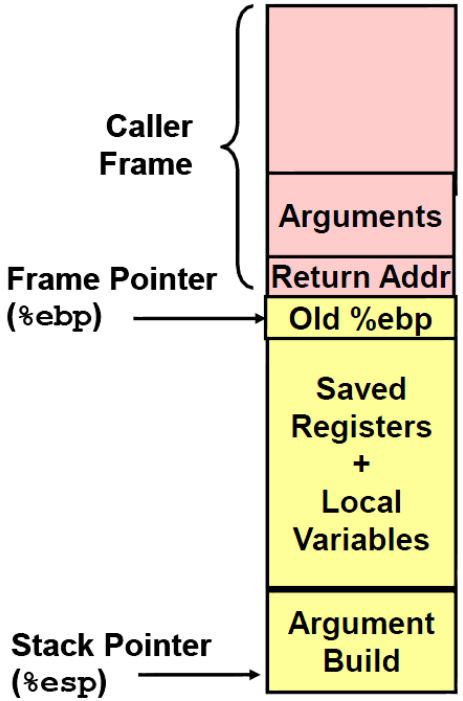
- 함수의 이용
- GDB

어셈블리어 함수

- 어셈블리어에서 함수는 아래와 같이 선언한다.
  - .type 함수 명, @function

- 함수의 호출은 아래와 같다.
  - call 함수 명

- 함수의 인자는 오른쪽의 stack 그림 처럼 ebp에서 8 byte 떨어진 곳부터 저장된다.
  - 8(%ebp), 12(%ebp), …



따라 하기 1. 함수 이용

```
.section .data
message :
        .string "%d + %d = %d\n"
val1 :
        .int 100
val2 :
        .int 200

.section .text
.global main
main :
        pushl   %ebp
        movl    %esp, %ebp

        pushl   val2
        pushl   val1
        call    add_func

        pushl   %eax
        pushl   val2
        pushl   val1
        pushl   $message
        call    printf
        movl    %ebp, %esp
        popl    %ebp
        ret

.type add_func, @function
add_func :
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    12(%ebp), %ebx
        addl    %ebx, %eax
        movl    %ebp, %esp
        pop     %ebp
        ret
```

- add_func 함수를 호출 후 결과 값 출력
- main 함수에서 다음과 같이 add_func 함수 호출

```
pushl   val2        # 두 번째 인자 값
pushl   val1        # 첫 번째 인자 값
call    add_func    # 함수 호출
```

- add_func의 내용은 다음 코드와 같다.

```
int add_func(int a, int b){
        return (a + b)
}
```

- 인자의 값은 %ebp를 이용해 얻는다.

```
%ebp + 8        -> val1
%ebp + 12       -> val2
```

- %eax에 결과 값을 넣으면 결과가 반환된다.

따라 하기 2. 함수 이용

```
.section .data
message :
        .string "%d %d\n"
val1 :
        .int 100
val2 :
        .int 200

.section .text
.global main
main :
        pushl   %ebp
        movl    %esp, %ebp
        pushl   val2
        pushl   val1
        pushl   $message
        call    printf

        pushl   val2
        pushl   val1
        call    swap

        pushl   val2
        pushl   val1
        pushl   $message
        call    printf

        movl    %ebp, %esp
        popl    %ebp
        ret

.type swap, @function
swap :
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        movl    12(%ebp), %ebx
        movl    %ebx, val1
        movl    %eax, val2
        movl    %ebp, %esp
        popl    %ebp
        ret
```

- swap 함수를 호출 후 결과를 출력
- 다음과 같은 명령어를 통해 값이 변경된다.

```
movl    8(%ebp), %eax
movl    12(%ebp), %ebx
movl    %ebx, val1
movl    %eax, val2
```

- 실행 결과

```
[b000000000@eslab 05_gdb]$ gcc -o swap.out swap.s
[b000000000@eslab 05_gdb]$ ./swap.out
100 200
200 100
```

GDB ?

- GDB
  - 흔히 GDB라고 부르는 GNU 디버거(debugger)는 GNU 소프트웨어 시스템을 위한 기본 디버거 이다.
  - 디버거 : 프로그램의 오류를 찾아내기 위한 소프트웨어. 대상으로 하는 프로그램을 특정 단위로 실행해, 레지스터 값이나 프로그램 계수기, 플래그 등 중앙 처리 장치의 내부 상황을 나타냄.

- 디버깅을 위해 컴파일을 시 옵션을 주어야 한다.
  - GDB를 사용하기 위해서는 컴파일 시에 –g 플래그를 사용해야 한다.
  - 컴파일 시에 실행 파일에 여러 디버깅 정보를 삽입한다.
    - 내부에 사용된 심볼 문자열과 심볼의 주소, 컴파일에 사용된 소스파일, 컴파일 된 명령어가 어떤 소스 파일의 어떤 행에 해당되는 가와 같은 정보
    - 디버깅 레벨에 따라서 -g0, -g1, -g2, -g3 등과 같이 설정할 수 있다.
      - https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html

- 실행
  - gdb 실행파일

- 종료
  - q 입력 혹은 ctrl + d

따라 하기 3. GDB 실행

- 아래의 파일을 자신의 홈 디렉터리로 복사

```
[bOOOOOOOOO@eslab ~]$ cp /home/syspro/gdb_test.tar.gz ~
[bOOOOOOOOO@eslab ~]$ ls
gdb_test.tar.gz
```

- 압축 해제
  - tar xfvz gdb_test.tar.gz

```
[bOOOOOOOOO@eslab ~]$ tar xfvz gdb_test.tar.gz
gdb_test.c
```

- 디버깅 옵션(-g)를 주고 컴파일

```
[bOOOOOOOOO@eslab ~]$ gcc -g -o gdb_test.out gdb_test.c
[bOOOOOOOOO@eslab ~]$ ls
gdb_test.out       gdb_test.c
```

따라 하기 3. GDB 실행 – 따라 하기 2

❖ GDB 실행

- gdb 실행 파일
- 실행하면 GDB의 정보를 출력해준다.

```
[cOOOOOOOOO@eslab week5]$ gdb gdb_test.out
GNU gdb (GDB) Fedora 7.7.1-18.fc20
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from gdb_test.out...done.
(gdb)
```

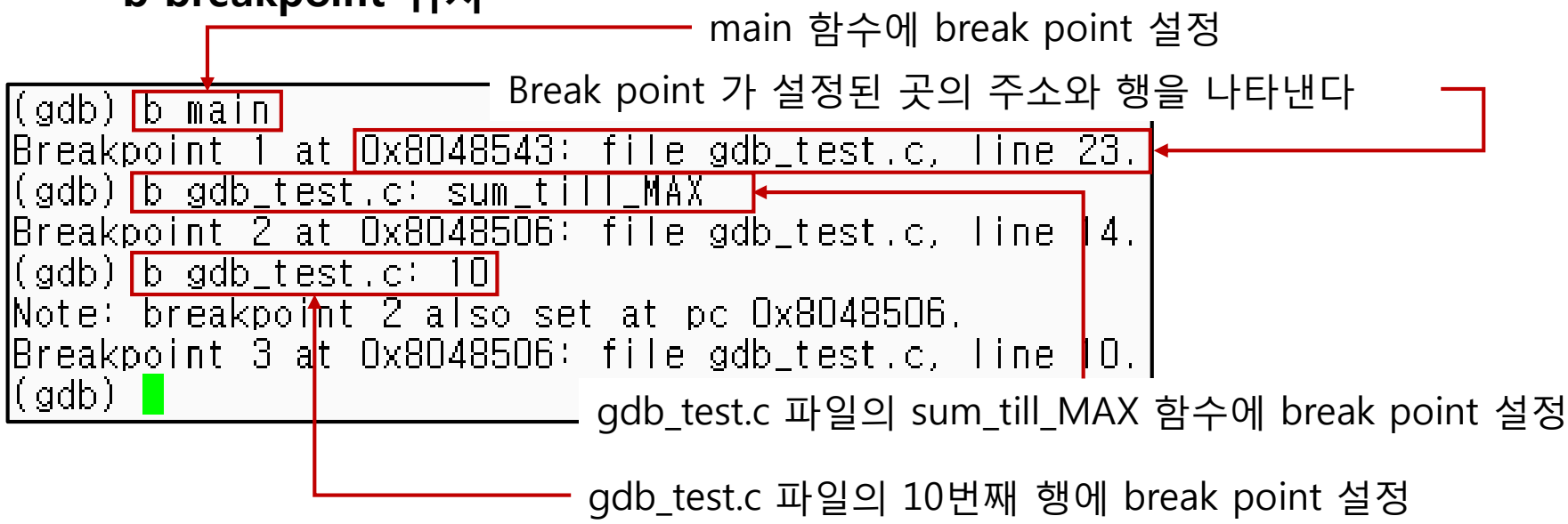
따라 하기 3. GDB Break point

Break Point

- 프로그램의 디버그 등에서 검사를 행하기 위하여 실행중인 처리를 일시적으로 정지하도록 설정된 점
- 정지된 상태에서 디버깅을 위한 기능을 사용할 수 있다.
  - 변수 값 및 스택 프레임 출력, break point 재설정, 재실행

Break Point 지정

- b breakpoint 위치



따라 하기 3. GDB Break point - 설정

❖ Break Point 설정

- 디버깅 시 중단점을 아래와 같이 설정 할 수 있다.

Break Point 위치	설 명
함수 명	해당 함수에 break point 설정한다.
행 번호	행에 break point 설정한다.
파일 명 : 함수 명	해당 파일의 함수에 break point 설정한다.
파일 명 : 행 번호	해당 파일의 행에 break point 설정한다.
+ 오프셋	현재 행을 기준으로 오프셋 만큼 + 행에 break point 설정한다.
- 오프셋	현재 행을 기준으로 오프셋만큼 – 행에 break point 설정한다.
*주소	특정 주소에 break point 설정 (어셈블리 디버깅) 한다.
행 if 조건	조건이 만족할 경우 행에 break point 설정한다.

따라 하기 3. GDB Break point – 제거

❖ Break Point 제거

▪ cl break point 위치

```
(gdb) cl main
Deleted breakpoint 1
(gdb)
```

main 함수에 설정된 break point를 제거한다.

Break Point 위치	설 명
함수 명	해당 함수의 break point 제거한다.
행 번호	해당 행의 break point 제거한다.
파일 명 : 함수 명	해당 파일의 함수의 break point 제거한다.
파일 명 : 행 번호	해당 파일의 행의 break point 제거한다.

관련 명령	설 명
delete	모든 break point 제거한다.

따라 하기 3. GDB Break Point - 확인

- **Break Point 확인**
  - **info break 명령어 사용**
  - **Break point의 주소와 설정된 위치를 확인할 수 있다.**

주소 설정위치

```
(gdb) info b
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:14
3       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:10
(gdb) info break
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:14
3       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:10
(gdb) info breakpoint
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:14
3       breakpoint     keep y   0x08048506 in sum_till_MAX at gdb_test.c:10
```

따라 하기 4. GDB 디버깅 – 프로그램 실행

- GDB에서의 프로그램을 실행
  - run 명령어 사용
  - 처음부터 프로그램을 시작 시킨다.
  - 따라 하기 3에서 설정한 break point에서 프로그램이 정지된다.

```
(gdb) b main
Breakpoint 4 at 0x8048543: file gdb_test.c, line 23.
(gdb) run
Starting program: /home/sys03/c000000000/week5/gdb_test.out

Breakpoint 4, main (argc=1, argv=0xbffff6b4) at gdb_test.c:23
23              u64 sum=0;
```

따라 하기 4. GDB 디버깅 – 디버깅 명령어

❖ **gdb에서 프로그램을 다루는데 사용하는 명령어다.**

실행 명령	설 명
run (r)	gdb에서 프로그램을 시작시킨다.
step (s)	현재 행을 수행하고 정지한다. 함수 호출 시 함수 내부로 들어간다.
next (n)	현재 행을 수행하고 멈춘다. 함수 호출 시 함수 수행 후 다음 행에서 멈춘다.
continue (c)	다음 break point 까지 진행한다.
kill (k)	디버깅 중인 프로그램의 실행을 취소한다.
u	현재 루프를 빠져나간다.
finish	현재 함수를 수행하고 빠져 나간다.
return	현재 함수를 수행하지 않고 빠져 나간다.
return123	현재 함수를 수행하지 않고 나간다. 이때 리턴 값을 123으로 설정한다.

따라 하기 5. Stack 디버깅

- 디버깅 시 현재 스택의 상태를 확인 할 수 있다.
  - bt (Back Trace)
  - Back Trace는 Stack Trace라는 의미이다. 프로그램의 실행 중에 현재 동작 중인 스택 프레임을 보고한다.

실행 중인 함수

```
(gdb) bt
#0  main (argc=1, argv=0xbffff6b4) at gdb_test.c:23
```

스택 프레임 번호

함수 소스코드 위치

따라 하기 5. Stack 디버깅

- 다음 break point 까지 이동 한 후 스택을 확인한다.
  - c (continue) 명령을 통해서 다음 break point 까지 이동.
  - bt 명령으로 현재 스택의 상태를 확인한다.

```
(gdb) c
Continuing.

Breakpoint 6, sum_till_MAX (n=0) at gdb_test.c:14
14                n++;
(gdb) bt
#0  sum_till_MAX (n=0) at gdb_test.c:14
#1  0x080485f3 in main (argc=1, argv=0xbffff6b4) at gdb_test.c:31
```

스택 프레임이 증가 한 것을 볼 수 있다

  - 스택 프레임에 대한 자세한 설명은 이론 자료 참고.

따라 하기 5. Stack 디버깅

❖ 스택 프레임 내의 지역 변수 확인

- **bt full**

```
(gdb) bt full
#0  sum_till_MAX (n=0) at gdb_test.c:14
        sum = 5777311251110205388
#1  0x080485f3 in main (argc=1, argv=0xbffff6b4) at gdb_test.c:31
        sum = 0
```

명령	설 명
bt full N	최초 N개의 프레임의 bt와 로컬 변수 출력한다.
bt full –N	마지막 N개의 프레임의 bt와 로컬 변수 출력한다.
frame N	N번 스택 프레임으로 변경한다.
up	상위 스택 프레임으로 변경한다.
up N	N번 상위 프레임으로 변경한다.
down	하위 스택 프레임으로 변경한다.
down N	N번 하위 스택 프레임으로 변경한다.

따라 하기 5. Stack 디버깅

❖ 스택 프레임 내의 정보 확인

- **info [인자]**
- **help info로 info 관련 명령어 확인 가능**

명령	설 명
info frame	스택 프레임의 정보를 출력한다.
info args	함수가 호출 될 때 인자를 출력한다.
info locals	함수의 지역 변수를 출력한다.
info handle	함수의 예외 핸들러를 출력한다.
info display	현재 display 정보를 출력한다.

- **스택에는 지역 변수가 저장된다.**
- **bt full 명령을 통해서 각 스택 프레임에 저장된 지역 변수를 확인 할 수 있다.**

따라 하기 6. 소스 코드 출력

❖ 소스 코드 출력

- 디버깅 중에 현재 실행하고 있는 소스 코드를 볼 수 있다.
- l (List lines of source code)
- 현재 14번 행에서 break point를 만나 프로그램이 정지된 상태
- 소스 코드는 break point를 기준으로 10행을 출력한다.

```
(gdb) l
9       u32 max_addend= MAX;
10
11      u64 sum_till_MAX(u32 n)
12      {
13              u64 sum;
14              n++;
15              sum=n;
16              if(n<max_addend)
17                      sum+= sum_till_MAX(n);
18              return sum;
```

따라 하기 6. 소스 코드 출력

❖ 소스 코드 출력 명령어

명령	설 명
l	main 함수를 기점으로 소스코드 출력한다.
l 10	10행을 기준으로 출력한다.
l func	func 함수의 소스를 출력한다.
l -	출력된 행의 이전 행을 출력한다.
l file.c:func	file.c 파일의 func 함수를 기준으로 출력한다.
l file.c:1	file.c 파일의 1행을 기준으로 출력한다.
set listsize 20	list 명령의 기본 출력 행의 수를 20행으로 설정한다.

따라 하기 7. 변수 및 레지스터 확인

❖ 변수 값 및 레지스터 확인

- 전체 지역 변수와 레지스터의 값을 확인 할 수 있다.
- info locals
- info registers

```
(gdb) info locals
sum = 0
(gdb) info registers
eax            0x213    531
ecx            0x925225db       -1840110117
edx            0xbffff644       -1073744316
ebx            0xb7fc4000       -1208205312
esp            0xbfff9230       0xbfff9230
ebp            0xbfff9258       0xbfff9258
esi            0x0      0
edi            0x0      0
eip            0x8048506        0x8048506 <sum_till_MAX+6>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

지역 변수 확인

레지스터 확인

따라 하기 7. 변수 및 레지스터 확인

❖ 변수 값 및 레지스터의 값을 확인 할 수 있는 명령어는 아래와 같다.

명령	설 명
watch [변수 명]	변수 값이 바뀔 때 마다 값을 확인한다.
info locals	전체 지역 변수를 출력한다.
p [변수 명]	변수의 값을 출력한다.
p *주소	주소가 가리키는 위치의 값을 출력한다.
p $레지스터	해당 레지스터의 값을 출력한다.
info register	전체 레지스터 값 출력한다.
p *pt@4	4의 크기를 갖는 구조체 배열을 출력한다.

따라 하기 7. 변수 및 레지스터 확인

- **변수 값 출력 형식 설정**
  - **p/[출력 형식] [변수 명]**
  - **p/x arg**

명령	설 명
t	2진수로 출력한다.
o	8진수로 출력한다.
d	부호가 있는 10진수로 출력(int) 한다.
u	부호가 없는 10진수로 출력(unsigned int) 한다.
x	16진수로 출력한다.
c	최초 1바이트 값을 문자 형으로 출력한다.
f	부동 소수점 값 형식으로 출력한다.
a	가장 가까운 심볼의 오프셋을 출력한다.

따라 하기 7. 변수 및 레지스터 확인

❖ 변수 값의 변화 확인

```
(gdb) display sum
2: sum = 13257541771950589499
(gdb) n
24              if ((argc==2)&&isdigit(*(argv[1])))
2: sum = 0
(gdb)
26              if(max_addend>MAX||max_addend==0){
2: sum = 0
(gdb)
31              sum = sum_till_MAX(0);
2: sum = 0
(gdb)

Breakpoint 6, sum_till_MAX (n=0) at gdb_test.c:14
14              n++;
1: sum = 5777311251110205388
(gdb)
15              sum=n;
```

sum 변수를 display 설정

매번 sum 변수의 값을 확인

명령	설 명
display 변수	변수 값을 매번 화면에 출력한다.
display/출력형식 변수	변수 값을 설정한 출력 형식으로 출력한다.
undisplay 디스플레이 번호	디스플레이 설정을 없앤다.
disable display 디스플레이 번호	디스플레이를 일시 중단한다.
enable display 디스플레이 번호	디스플레이를 다시 활성화 시킨다.

따라 하기 8. 메모리 상태 검사

❖ 메모리 상태 검사

- -g 옵션을 사용하지 않고 컴파일 한 실행 파일의 디버깅에 사용한다.
- x/[출력 횟수] [출력 형식] [출력 단위] [출력 위치]

```
(gdb) x/10xb main
0x804853a <main>:       0x55    0x89    0xe5    0x83    0xe4    0xf0    0x83    0xec
0x8048542 <main+8>:     0x20    0xc7
(gdb) x/8xw main
0x804853a <main>:       0x83e58955      0xec83f0e4      0x2444c720      0x00000018
0x804854a <main+16>:    0x2444c700      0x0000001c      0x087d8300      0xe84d7502
```

main 의 주소 부터 16진수로 1바이트 씩 10개를 출력

main 의 주소 부터 16진수로 4바이트 씩 8개를 출력

명령	설 명
t	2진수로 출력한다.
o	8진수로 출력한다.
d	부호가 있는 10진수로 출력(int) 한다.
u	부호가 없는 10진수로 출력(unsigned int) 한다.
x	16진수로 출력한다.
c	최초 1바이트 값을 문자 형으로 출력한다.
f	부동 소수점 값 형식으로 출력한다.
a	가장 가까운 심볼의 오프셋을 출력한다.
s	문자 열로 출력한다.
i	어셈블리형식으로 출력한다.

명령	설 명
b	1바이트 단위 (byte)
h	2바이트 단위 (half word)
w	4바이트 단위 (word)
g	8바이트 단위 (giant word)

따라 하기 9. 어셈블리 코드 보기

- gdb에서 어셈블리 코드를 볼 수 있다.
  - -g 옵션으로 컴파일 되지 않은 실행 파일 디버깅에 사용한다.
  - disas [함수 명] : 함수의 어셈블리 코드를 출력한다.
  - disas [시작 주소] [끝 주소] : 주소 범위의 어셈블리 코드를 출력한다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804853a <+0>:     push   %ebp
   0x0804853b <+1>:     mov    %esp,%ebp
   0x0804853d <+3>:     and    $0xfffffff0,%esp
   0x08048540 <+6>:     sub    $0x20,%esp
   0x08048543 <+9>:     movl   $0x0,0x18(%esp)
   0x0804854b <+17>:    movl   $0x0,0x1c(%esp)
   0x08048553 <+25>:    cmpl   $0x2,0x8(%ebp)
   0x08048557 <+29>:    jne    0x80485a6 <main+108>
   0x08048559 <+31>:    call   0x80483e0 <__ctype_b_loc@plt>
   0x0804855e <+36>:    mov    (%eax),%edx
   0x08048560 <+38>:    mov    0xc(%ebp),%eax
   0x08048563 <+41>:    add    $0x4,%eax
   0x08048566 <+44>:    mov    (%eax),%eax
   0x08048568 <+46>:    movzbl (%eax),%eax
   0x0804856b <+49>:    movsbl %al,%eax
```

제출 사항

- 모든 따라 하기를 진행한 내용을 모두 보고서로 작성하여 이메일과 서면으로 제출
  - sihyeong@cnu.ac.kr
  - 제목 양식 : [sys02]HW05_학번_이름
  - 파일 제목 : [sys02]HW05_학번_이름
  - 반드시 메일 제목과 파일 양식을 지켜야 함. (위반 시 감점)
  - 보고서는 제공된 양식 사용

- 자신이 실습한 내용을 증명할 것 (자신의 학번이 항상 보이도록)

- 제출 일자
  - 이메일 : 2014년 10월 22일 23시 59분 59초
  - 서면 : 2014년 10월 23일 수업시간