2017

# Operating System Lab 0

Embedded System Lab.

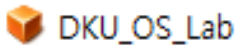
LAB 0 – 프로젝트 구성, 개발환경 구성, GDB

JUHYUNG SON, CHOI JONG MOO

## A. 프로젝트 구성

**본 수업에서는 우분투 가상 머신 이미지를 배포하여 실습을 진행하며 배포된 이미지는 아래와 같은 구성을 가지고 있다.**

1. **배포 실습 이미지.**




**다운로드 경로**

https://goo.gl/e3gNGl

2. **실습 이미지 구성**

- Image name : DKU_OS_Lab
- Operating System : Ubuntu 16.04 LTS
- Kernel Version : 4.4.0-31-generic
- Hardware Platform : x86_64
- ***login & root password : 1234***

3. **실습 이미지 수행 방법**

참조된 실습 환경 구축 PPT 를 참조한다.

https://goo.gl/tM0Dde

4. **실습 내용**

```
root@os-lecture:/home/os-lecture/DKU_OS_LAB# ls
lab1_sched  lab2_sync  lab3_fuse
```

i. **lab1_sched**

FCFS, RR, SPN,SRT, HRRN, MLFQ User Scheduler 구현 과제

https://goo.gl/FlDSGa

ii. **lab2_sync**

Multi threading 환경에서 공유 자원에 대한 race condition 해결 과제

https://goo.gl/PhpjZY

iii. **lab3_fuse**

ramdisk 상에서의 UNIX file system 구현 과제

https://goo.gl/WJBrbM

### 5. 과제 제출 방법

#### i. 구현 내용

압축하여 tooson9010@gmail.com 으로 제출

#### ii. 레포트

미디어 센터 506 호로 제출

## B. 개발 환경 구성

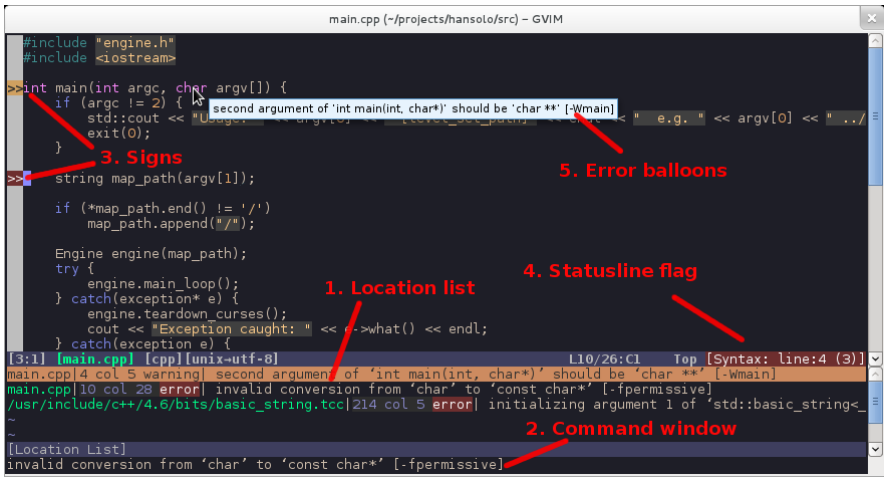
**본 수업에서는 배포된 우분투 실습 이미지의 vim editor 를 사용하여 개발한다. 베포된 vim editor 에는 개발에 유용하게 사용될 수 있는 몇가지 PlugIn 들이 설치되어 있다.**

***vim editor 내부에서 분리된 창간의 이동은 ctrl + w + 방향키 를 통해 이동할 수 있다.***

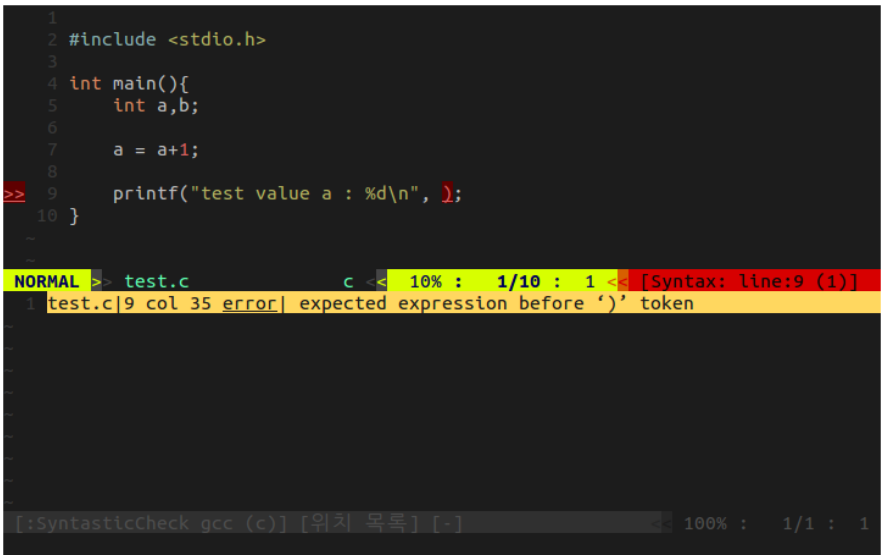
### 1. 설치된 vimrc 플러그인 및 툴

#### i. Syntastic

문법 체크 기능 플러그인으로 소스 작성 후 컴파일 과정에서 error 를 확인하는 것이 아닌 vim 에서 error 및 warning 을 확인할 수 있도록 해주는 플러그인으로 아래와 같이 다양한 기능을 제공한다




예시로 아래와 같은 9 번째 줄 printf 함수에 출력 대상을 지정하지 않은 예제 코드가 있다고 할 때, vim 창의 하단에 어느 소스코드의 몇번 째 line 에




무슨 에러가 발생 하였는지의 정보를 나타내며 상단의 소스코드 화면에서 문제가 발생한 line 을 표시해 준다. 이를 통해 vim 을통해 개발하며 error 를 확인할 수 있다. 소스코드 작성 후, 저장함으로써 에러를 업데이트하여 확인할 수 있다.

- *ctrl + w + 방향키* 를 통해 아래 창으로 이동하여 해당 에러가 난 위치로 갈 수 있다.
- 참고

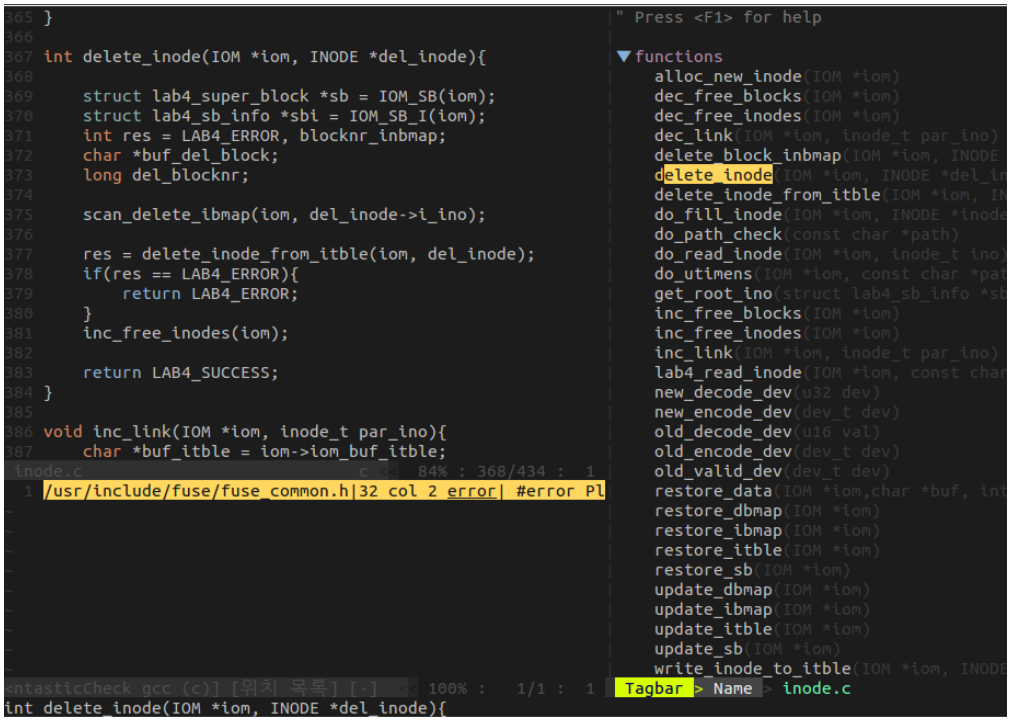
  https://github.com/vim-syntastic/syntastic

### ii. Tagbar

본 과제를 수행하다 보면 test.c 라는 간단한 소스코드가 아닌 다양한 함수 및 매크로, 변수 등을 사용할 수 있다. 이때 현재까지 작성한 소스코드의 손쉬운 관리를 위해 함수, 변수, 매크로 등을 확인하고 해당 위치로 이동하는 기능을 제공하는 플러그인이다.

- F8 을 눌러 해당 기능을 on/off 할 수 있다.
- *ctrl + w + 방향키* 를 통해 우측 창으로 이동하여 해당 함수, 매크로 변수 등으로 이동할 수 있다.
- 참고

  https://github.com/majutsushi/tagbar




### iii. AutoComplPop

자동완성기능을 해주는 플러그인이다.

- 참고

  https://github.com/vim-scripts/AutoComplPop

### iv. NERDTree

탐색기 기능을 하는 플러그인으로써 vim ediitor 에서 작업 도중 다른 파일로 옮겨가야 할 때 유용하게 사용할 수 있다. 아래와 같이 inode.c 라는 파일을 작성 중 file.c 등 다른 파일로 이동해야 할 때, NERDTree 플러그인을 통해 이동 및 조회가 가능하다.

- F7 을 눌러 해당 기능을 on/off 할 수 있다.
- *ctrl + w + 방향키* 를 통해 좌측 창으로 이동하여 다른 위치의 파일들을 열 수 있다.
- 참고

  https://github.com/scrooloose/nerdtree

### v. Ctags

Ctags 는 프로그래밍 소스코드의 태그(전역변수 선언, 함수 정의, 매크로 정의)들의 데이터 베이스(tags 파일)을 생성하는 명령어이다. Ctags 를 사용하고자하는 directory 로 이동하여 “# ctags –R”명령어를 이용하여 tags 파일을 생성한다.

- 사용하려는 directory 에서 tag 파일을 생성해 주어야 한다.

  # ctags -R

- vim editor 의 ex 모드에서 ":tj 함수명 or 구조체명"을 통해서 원하는 source code 를 찾을 수 있다.

  : tj 함수명

  : tj 구조체 명.

- 혹은 vi 를 열어서 source code 를 분석하는 중에 함수 원형이나 구조체이름에 커서를 위치시키고 "ctrl + ]"를 누르면 자동으로 태그를 찾아간다.이전으로 다시 되돌아 오고 싶은경우 "ctrl + t"를 사용해서 돌아올 수 있다.

- 참고

  https://en.wikipedia.org/wiki/Ctags

## C. GDB 를 통한 디버깅

### 실습을 구현하는데 있어 gdb 를 사용하여 디버깅을 수행할 수 있다.

#### 1. GDB 란?

GNU 의 디버거 프로그램으로, 프로그램 실행 동안 내부에서 진행되고 있는 상황들을 Line by Line 으로 훑어 볼 수 있는 툴이다. 컴파일 시, g 옵션을 통해 디버깅 정보를 삽입하여 디버깅을 할 수 있으며, 최적화 옵션을 주면 어셈블리 코드 자체에 많은 변경이 가능해져 C 소스의 한 행과 대응되는 어셈블리 묶음이 흐트러지기 때문에 최적화 옵션 (-O) 를 주지 않는 것이 좋다. 또한 Terminal User Interface 를 통해 아래와 같이 더 편리한 디버깅이 가능하다.


## 2. 기본 사용법

### i. 실행 방법

- *gdb [프로그램 명]*
- *gdb [프로그램 명] [core 파일 명]*
- *gdb [프로그램 명] [현재 수행중인 프로세스의 PID]*

### ii. 종료 방법

- *(gdb ) q*
- *(gdb) ctrl + d*

### iii. 소스 보기

l(list) 명령을 통해 소스 보기 가능. 그 후 l 또 누르거나 Enter 를 통해 소스의 나머지 부분 볼 수 있다. gdb 에서 아무 입력 없이 Enter 는 전 명령의 반복을 의미한다.

- *(gdb) l*
- *(gdb) l [행 번호]*
- *(gdb) l [함수 명]*
- *(gdb) l [파일 명] : [함수 명]*

### iv. 브레이크 포인트

r(run) 명령을 통해, 프로그램을 일단 gdb 에서 돌리고, 프로그램이 이상 종료되었을 시, 어떤 함수, 어떤 라인에서 종료된 건지 파악 하기 위해 사용한다. 함수 호출에 사용된 인자도 아래와 같이 파악이 가능하다.

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000040053c in func (sw=1, str=0x0) at bugprg.c:12
12          printf("first char = %c\n", *str);
```

위의 예시에서 func 는 sw 와 str 의 두개의 인자를 가지며, 각각 1, 0x0 의 값이 전달되었다는 것을 확인 할 수 있으며, 어떻게 func 함수가 호출 되었는지 또한 확인이 가능하다.

- *(bdb) bt*

```
(gdb) bt
#0  0x000000000040053c in func (sw=1, str=0x0) at bugprg.c:12
#1  0x00000000004005ca in main () at bugprg.c:28
```

bt 는 프로그램의 스택 프레임을 순차적으로 따라가며 해석해 현재 수행 코드까지 어떤 함수들의 호출이 있었는지 파악 할 수 있도록 해준다.

- *(gbb) b func*

func 함수에 브레이크 포인트 설정

- *(gbb) b 10*

10 행에 브레이크 포인트 설정

- *(gbb) b a.c:func*

a.c 파일의 func 함수에 브레이크 포인트 설정

- *(gbb) b a.c:10*

a.c 파일의 10 행에 브레이크 포인트 설정

- *(gbb) b +2*

위의 활용 예시처럼 상황에 따라 조건을 주어 breakpoint 를 줄 수 있으며, rb 명령으로 정규 표현식을 활용한 breakpoint 설정도 가능하다.

### v. 임시 브레이크 포인트

b 명령은 gdb 가 끝날 때까지 유효한 breakpoint 를 생성하는 명령어 지만, tb 명령어를 통해 임시적으로 한번만 유효한 breakpoint 를 설정도 가능하다. 이를 통해 임시 중단점을 사용하는 것으로 한번만 설정되며, 그 이후에는 삭제된다

- *(gbb) tb*

### vi. 브레이크 포인트 조회

info 명령을 통해 breakpoint 조회를 할 수 있다.

- *(gdb)  info b*
- *(gdb) info breakpoint*

### vii. 브레이크 포인트 삭제

cl, disable, enable, d, 등의 명령어를 사용하여 설정된 breakpoint 를 삭제, 설정할 수 있다.

- *(gdb) cl func*

func 함수의 시작 부분에 breakpoint 지움

- *(gdb) cl 10*

10 행의 breakpoint 지움

- *(gdb) delete 1*

고유번호 1 번의 breakpoint 를 지움

- *(gdb) cl a.c:func*

a.c 파일의 func 함수의 breakpoint 지움

- *(gdb) cl a.c:10*

a.c 파일의 10 행의 breakpoint 지움

- *(gdb) d*

모든 breakpoint 지움

- *(gdb) disable br*

모든 breakpoint 비활성화

- *(gdb) disable br 1*

1 번 breakpoint 비활성화

- *(gdb) enable br*

모든 breakpoint 포인트 활성화

- *(gdb) enable br 1*

1 번 breakpoint 활성화

### viii. 프로그램 실행

run 명령어를 통해 프로그램 실행이 가능하다. 이미 실행 중일 때, r 명령을 통해 재 실행할 수 있으며, 인자를 주는 것도 가능하다.

- *(gdb ) r*
- *(gdb ) run*
- *(gdb ) r -M create –T 6 –C 1000000*

### ix. breakpoint 도중 실행.

브레이크 포인트를 설정하고, 프로그램을 수행하며 디버깅 시, 프로시저 내부까지 들어가거나, 프로시저 단위로 디버깅을 수행 할 수 있다. step 명령을 통해 프로시저 내부 호출하며 실행(step)을 하고, next 명령을 통해 프로시저를 넘어가며 실행하며, 반복문을 빠져나오기 위해 until 명령을 사용할 수 있다. 또한 함수가 끝난 시점까지 실행되도록 한 후 함수 끝 시점으로 이동하려면 finish, 함수의 나머지 부분을 실행하지 않고 함수를 종료 하려면 return 명령 사용한다.

- *(gdb ) s*
- *(gdb ) s 6*
- *(gdb ) n*
- *(gdb ) n 6*
- *(gdb ) c*
- *(gdb ) u*
- *(gdb ) finish*
- *(gdb ) return*
- *(gdb ) return [리턴 값]*

### x. watch point 설정

프로그램의 변수 등의 변화를 monitoring 하며 디버깅할 시 설정. 변수 값이 어떻게 바뀌는지, 어떤 코드가 바뀌는지 확인 할 때 편리하게 사용할 수 있다. 그냥 watch 는 변수에 값이 써질 때, rwatch 는 변수의 값이 읽혀질 때, awatch 는 읽거나 쓰거나 모든 경우에 사용한다. 지역 변수에 watch point 를 설정하려면 프로그램 수행 context 가 해당 지역 변수가 정의된 함수 내에 있을 때, watch point 를 설정 해야 한다. (지역 변수는 해당 함수 내에만 존재하기 때문에 해당 함수를 실행중이지 않다면, 지역변수의 위치를 찾을 수 없다.)

- *(gdb ) watch [변수 명]*
- *(gdb ) rwatch [변수 명]*
- *(gdb ) awatch [변수 명*

### xi. 변수, 함수 출력

info 명령어를 통해 지역변수, 전역 변수의 현재 상태 조회가 가능하다.

```
(gdb) info locals
lval = 2331
i = 4195376
lstr = 0x400693 "I linke you."
pt = 0x601060 <gtime>
```

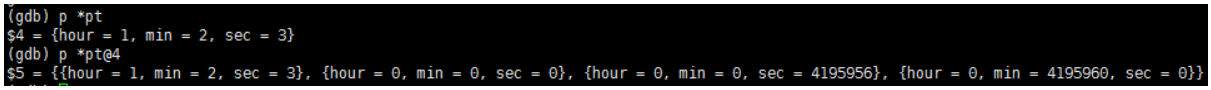
- *(gdb ) info locals*
- *(gdb ) info variables*

변수 명 조회를 통해 변수의 값 확인 및 함수 명 조회를 통해 해당 함수의 주소도 출력이 가능하다. 포인터형 변수의 경우 그 내용을 확인하려면 포인터값 출력처럼 p 명령어에 사용해 주면 값을 확인 할 수 있다.

```
(gdb) p pt
$2 = (struct time *) 0x601060 <gtime>
(gdb) p *pt
$3 = {hour = 1, min = 2, sec = 3}
```

확인하려는 값이 배열일 경우, 배열 전체를 출력하려면 gdb 에게 p [변수 명] 뿐만이 아니라, 배열임을 알려주어야 하며, 변수의 이름이 겹칠 경우 특정 함수

또는 파일로 지정하여 조회할 수 있고, 출력 형식을 지정하여 출력을 할 수도 있다.




void 형 포인터의 경우 해당 내용을 p 를 통해 출력하여도 void* 로 디버깅 정보에 저장되어 있기 때문에 그 값을 알 수 없다. 따라서 적절히 형 변환 하여 출력해 주어야 한다. 또한 디버깅 도중 p 명령어를 통해 변수 값을 설정 해 줄 수 있다.

- *(gdb ) p [변수 명]*
- *(gdb ) p [함수 명]*
- *(gdb ) p [변수 명]@[배열의 크기]*
- *(gdb ) p [함수 명] :: [변수 명]*
- *(gdb ) p '[파일 명]' :: [변수 명]*
- *(gdb ) p/[출력형식] [변수 명]*

  t(2 진수 출력), o(8 진수 출력), d(부호 있는 10 진수), u(부호 없는 10 진수), x(16 진수), c(최초 1 byte 값을 문자 형으로 출력), f(부동소수점 형식 출력), a(가장 가까운 심볼과의 offset 출력)

- *예시 : (gdb ) p (char *) [변수 명]*

display 명령을 통해 화면에 변수를 자동으로 출력이 가능하다, 특정 함수의 지역변수 출력 시, context 가 해당 함수에서 나가게 되면 더이상 출력 안됨. undisplay 를 통해 해제를 해주어야 하며 display 시에도 출력 형식을 지정할 수 있다.

- *(gdb) display [변수 명]*
- *(gdb) undisplay [display 번호]*
- *(gdb) display/[출력형식] [변수 명]*
- *(gdb) enable display [display 번호]*
- *(gdb) disable display [display 번호]*

### xii. 프로그램 종료

k 명령을 통해 작업중이던 디버깅을 종료할 수 있다.

- *(gdb) k*

## 3. 참고 자료

**이외에도 스택 프레임의 검사, 메모리 및 레지스터 상태 검사 등 다양하게 gdb 를 활용할 수 있다.**

**http://beej.us/guide/bggdb/**