

Computer Architecture

ARM 어셈블리 프로그래밍 실습

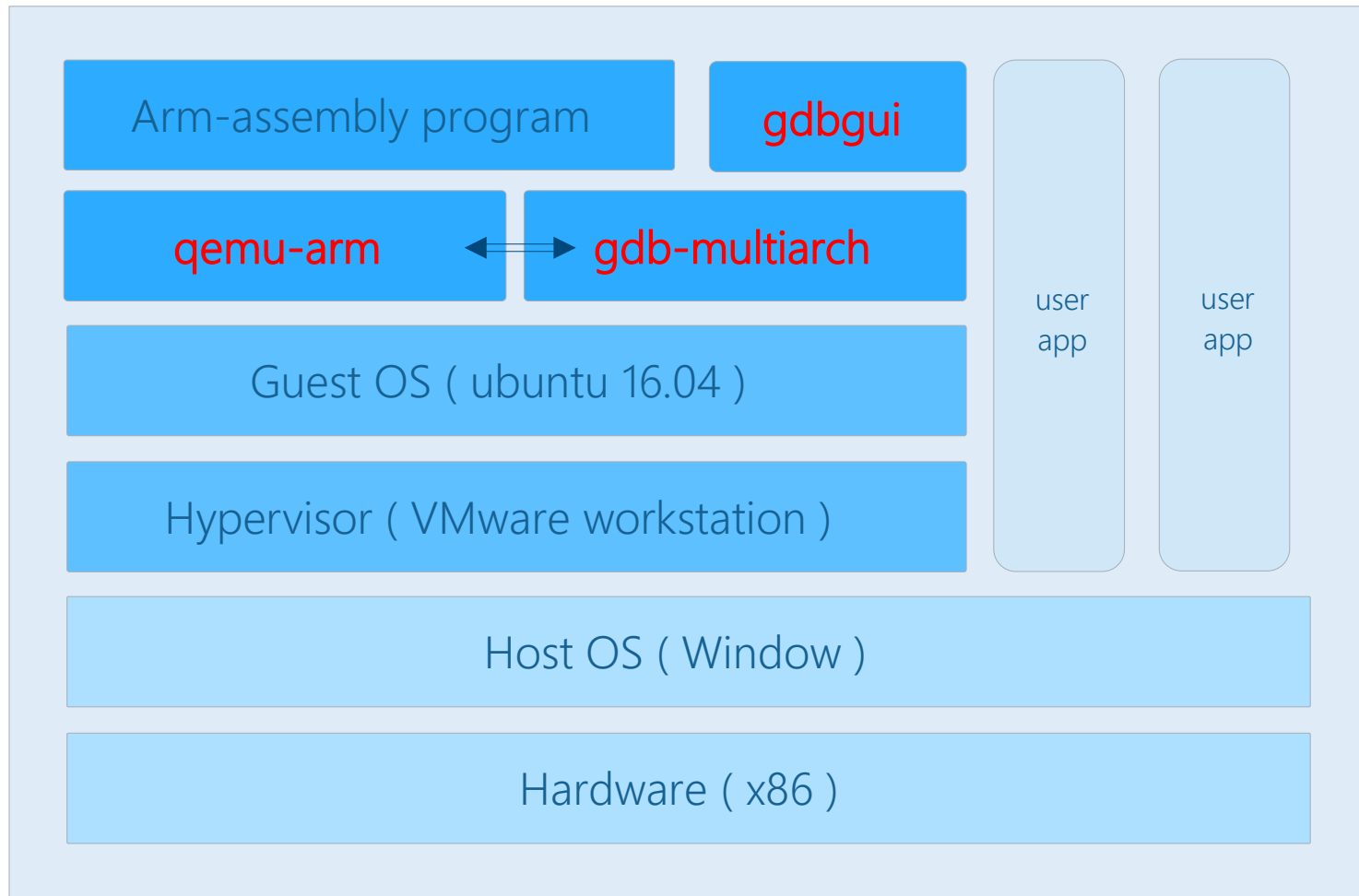
목차

- 실습목표
- ARM 어셈블리 프로그램 개발환경구축
- VMware
- GDB
- Qemu
- 설치과정 요약, 간단설명 및 설치 가이드
- 사용법
- 예제 0, 1, 2, 3
- 부록 - GDB 명령어
- 부록 - ARM Asembly Directive

실습목표

- Linux 환경에서 gdbgui 프로그램을 통해 GDB 사용법을 배운다
- ARM assembly에 대한 실습을 통해 이론으로 배웠던 ARM instruction들을 실제로 다루어 본다.
- Assembly로 코드를 작성하면서 메모리에 관점에서 생각해보는 시각을 키우고 메모리에 대한 이해도를 높인다.

ARM 어셈블리 프로그램 개발환경구축

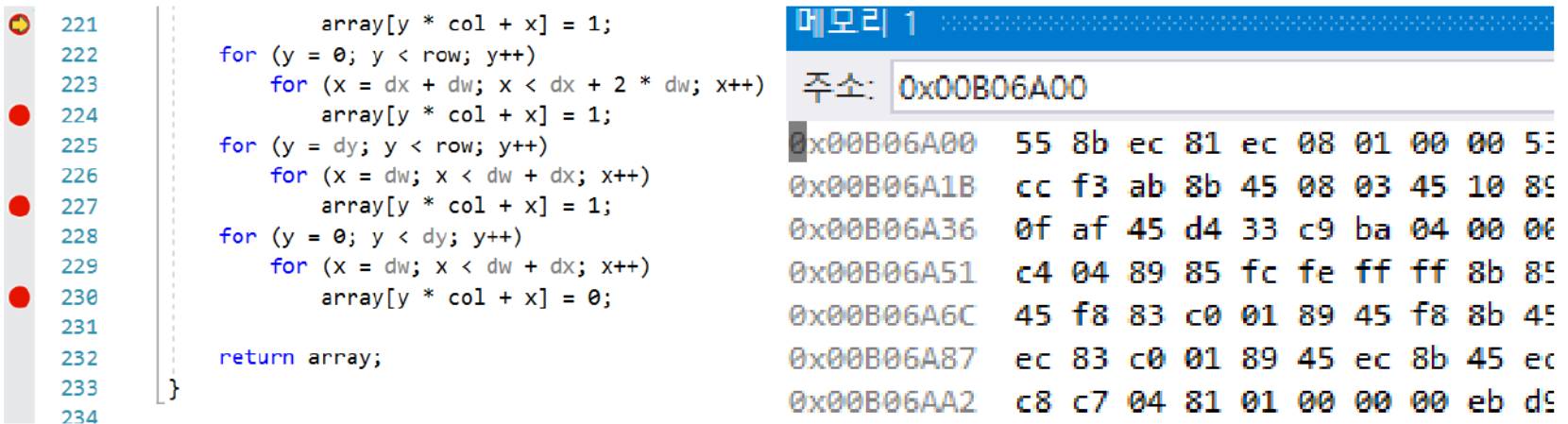


VMware

- 가상 머신에서 "가상"은 "의사", "유사" 혹은 "가짜" 라는 의미로, 가상 하드웨어, 즉 물리적 하드웨어로 구성된 실제 컴퓨터와 달리 소프트웨어적으로 모방한 가상 CPU, 가상 RAM, 가상 하드디스크 등으로 구성된 컴퓨터다.
- VMware는 가상 머신 유틸리티로 가상 머신을 생성할 수 있는 환경을 제공하는 소프트웨어로, 응용 프로그램과 동일한 계층에서 실행된다. 가상 머신 유틸리티를 이용하면 다수의 가상 머신을 생성할 수 있다.
- 예를 들면 윈도우 운영체제를 사용하고 있는 PC에 VMware를 설치하면 그 위에 다른 종류의 운영체제를 설치할 수 있다. 따라서 멀티부팅과는 달리 특정 시점에 서로 다른 운영체제를 가진 다수의 컴퓨터 시스템을 동시에 사용할 수 있다.

GDB

- GNU 소프트웨어 시스템을 위한 기본 디버거이다. GDB는 다양한 유닉스 기반의 시스템에서 동작하는 이식성 있는 디버거로, 에이다, C, C++, 포트란 등의 여러 프로그래밍 언어를 지원한다.
- 예를 들면 Visual Studio 에서 F5키를 누르면 보이는 화면이 디버깅 모드이고. breakpoint, step into, step over, step out 등의 기능으로 프로그램의 실행을 추적할 수 있다.



The screenshot displays a debugger interface with two main panels. The left panel shows a C program with several nested loops and a return statement. The right panel, titled '메모리 1' (Memory 1), shows a memory dump starting at address 0x00B06A00.

```
221     array[y * col + x] = 1;
222     for (y = 0; y < row; y++)
223         for (x = dx + dw; x < dx + 2 * dw; x++)
224             array[y * col + x] = 1;
225     for (y = dy; y < row; y++)
226         for (x = dw; x < dw + dx; x++)
227             array[y * col + x] = 1;
228     for (y = 0; y < dy; y++)
229         for (x = dw; x < dw + dx; x++)
230             array[y * col + x] = 0;
231
232     return array;
233 }
234
```

메모리 1

주소: 0x00B06A00

0x00B06A00	55 8b ec 81 ec 08 01 00 00 53
0x00B06A1B	cc f3 ab 8b 45 08 03 45 10 89
0x00B06A36	0f af 45 d4 33 c9 ba 04 00 00
0x00B06A51	c4 04 89 85 fc fe ff ff 8b 89
0x00B06A6C	45 f8 83 c0 01 89 45 f8 8b 45
0x00B06A87	ec 83 c0 01 89 45 ec 8b 45 ec
0x00B06AA2	c8 c7 04 81 01 00 00 00 eb d9

Qemu

- Qemu는 가상 머신 에뮬레이터(예: 실제 하드웨어는 intel CPU를 사용하고 있는데 마치 ARM CPU를 사용하는 것과 같은 환경을 제공), 가상화 솔루션이다. 가장 큰 장점은 host PC와 target board가 다른 환경에서 다양한 machine(ARM, SPARC, MIPS, PowerPC 등)을 테스트 할 수 있도록 도와준다.
- 현 수업에서 ARM architecture를 학습하기 때문에 ARM CPU 대상으로 테스트 환경을 구축할 것이다.

설치과정 요약

1. Vmware 설치(Vmware 16 workstation player) : 17버전도 가능
2. Ubuntu 설치(Ubuntu 16.04) : 반드시 이 버전 사용해야 함
3. Compiler 설치(gcc ARM 용 complier)
4. 가상 머신 에뮬레이터 설치(Qemu - 실제 하드웨어는 intel CPU를 사용하고 있는데 마치 ARM CPU를 사용하는 것과 같은 환경을 제공)
5. Debugger 설치(GNU debugger: GDB-multiarch, ARM 바이너리를 디버깅 할 수 있도록 해준다)
6. GUI에서의 Debugger 동작환경 설치(GDBGUI 0.13.0.0)

설치 과정 간단 설명 (1/2)

- 8 페이지 1번~2번에 해당 (1.Vmware / 2.Ubuntu 설치)
Vmware 및 ubuntu를 다운로드후 설치한다. (P11-P20 에 자세히 설명)
- 8 페이지 3번~5번에 해당 (3.컴파일러 / 4.에뮬레이터 / 5.gdb 설치)
아래의 설치 명령어들을 하나씩
수동으로 수행하여 설치하면서 진행한다. (P21-P23 에 자세히 설명)

```
$sudo apt-get install gcc-aarch64-linux-gnu  
$sudo apt-get install qemu  
$sudo apt-get install gdb-multiarch
```

설치 과정 간단 설명 (2/2)

▣ 8페이지 6번에 해당 (6. gdbgui 설치)

아래의 명령어들을 하나씩 수동으로 수행하여 설치 (P24~P25 에 설명)

```
$sudo apt install python-pip
$wget https://bootstrap.pypa.io/pip/3.5/get-pip.py -O /tmp/get-pip.py
$sudo python3 /tmp/get-pip.py
$pip install --user pipenv
$pip3 install --user pipenv
$echo "PATH=$HOME/.local/bin:$PATH" >> ~/.profile
$source ~/.profile
$whereis pip
$sudo pip install gdbgui==0.13.0.0
```

설치 가이드 - VMware

Ubuntu 버전은 반드시 16.04
이어야 함. 다른 버전에서는 본
환경이 설치가 잘 안됨!!

□ 설치도중 필요한 Ubuntu 이미지를 미리 다운로드함

□ Ubuntu 16.04-desktop-amd64.iso

다운로드 파일은 P16에서 사용예정

<http://old-releases.ubuntu.com/releases/16.04.0/>

MD5SUMS-metalink.gpg	2016-07-21 12:16	933
MD5SUMS.gpg	2019-02-28 16:26	916
SHA1SUMS	2019-02-28 16:25	3.8K
SHA1SUMS.gpg	2019-02-28 16:26	916
SHA256SUMS	2019-02-28 16:25	5.0K
SHA256SUMS.gpg	2019-02-28 16:26	916
source/	2018-08-02 10:53	-
ubuntu-16.04-desktop-amd64.iso	2016-04-20 22:30	1.4G
ubuntu-16.04-desktop-amd64.iso.torrent	2016-04-21 09:58	56K
ubuntu-16.04-desktop-amd64.iso.zsync	2016-04-21 09:58	2.8M
ubuntu-16.04-desktop-amd64.list	2016-04-20 22:30	4.4K
ubuntu-16.04-desktop-amd64.manifest	2016-04-20 22:25	63K
ubuntu-16.04-desktop-amd64.metalink	2016-07-21 12:16	45K

설치 가이드 - VMware

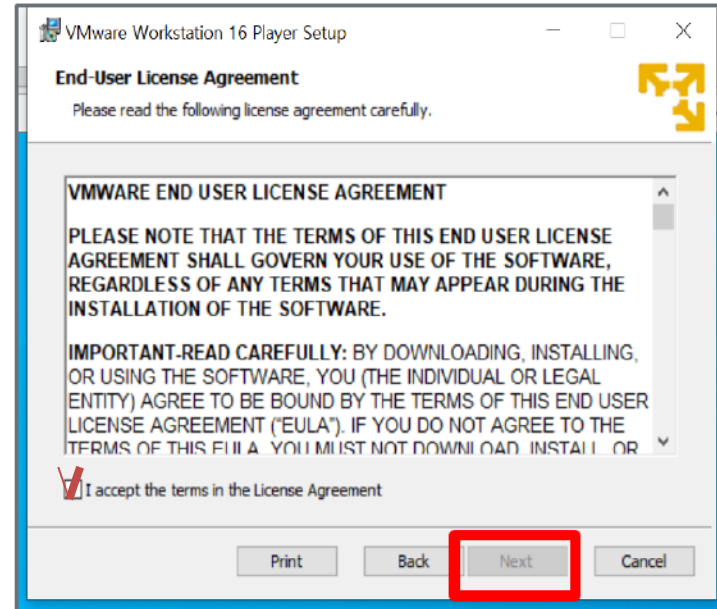
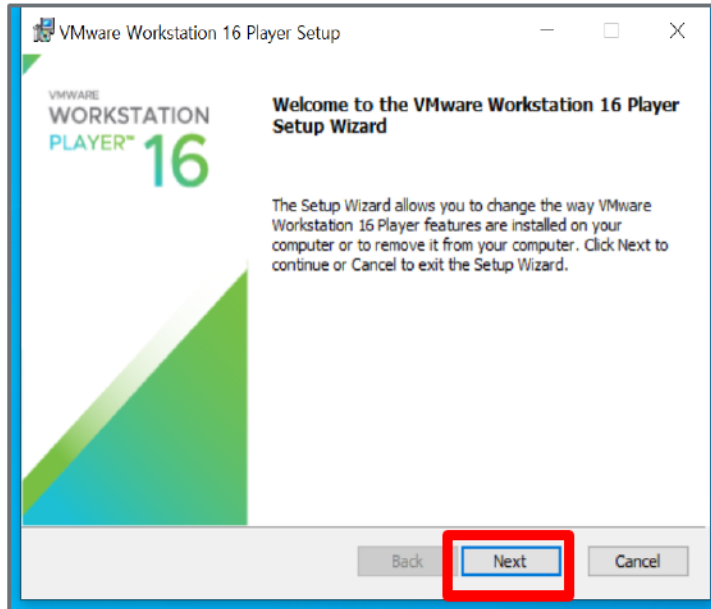
□ VMware 설치

<https://www.vmware.com/kr/products/workstation-player/workstation-player-evaluation.html>

-링크에서 vmware workstation player 17 다운로드

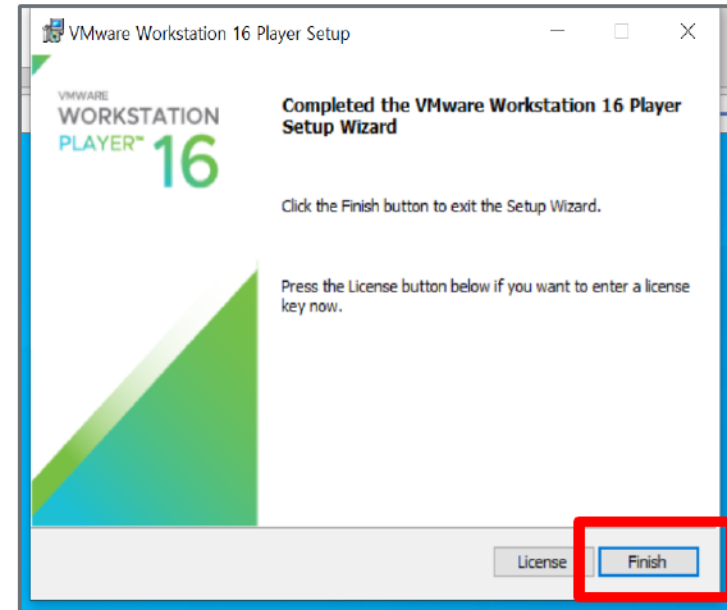
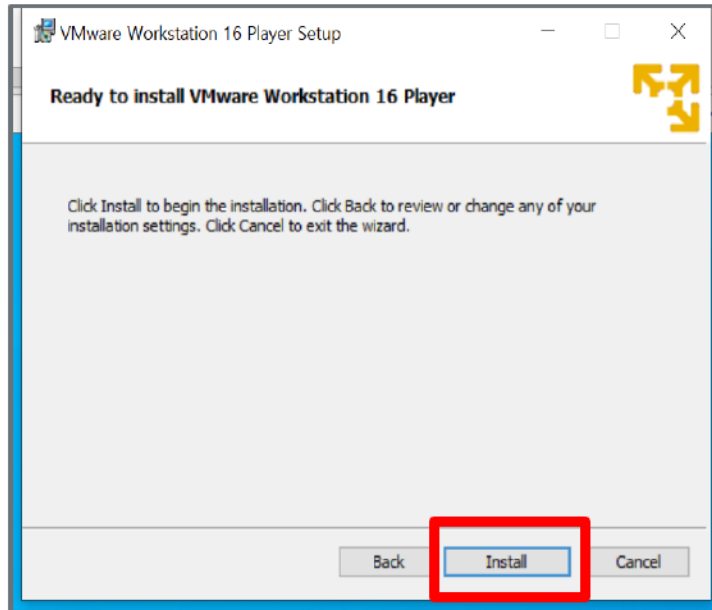
(다음 슬라이드의 Vmware 설치과정은 player 16 을 기반으로 설명하고 있으나 player17도 큰 차이가 없음.)

설치 가이드 - VMware



- 설치패키지 사용을 위해 다시 시작해야 한다고 나오면 다시시작 선택
- 위와 같은 설치 화면에서 accept 선택 후 next를 클릭

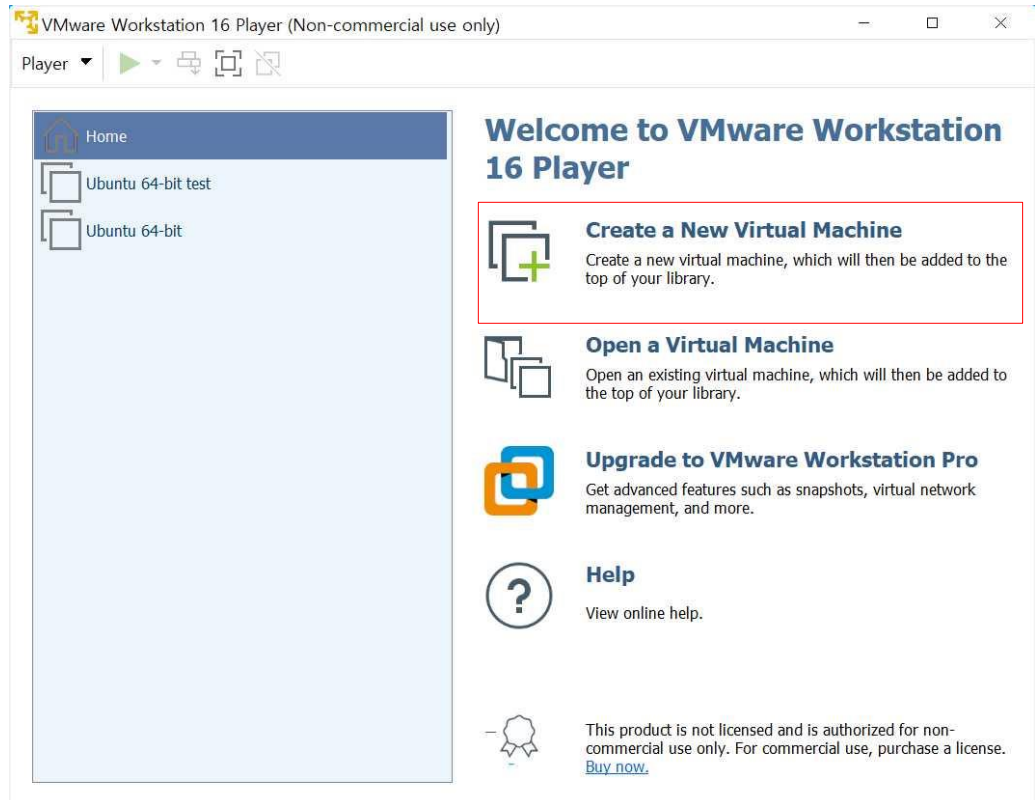
설치 가이드 - VMware



- 이 외에 다른 화면에서도 디폴트값 선택
- install을 클릭하고 finish를 클릭해서 설치를 마무리한다.

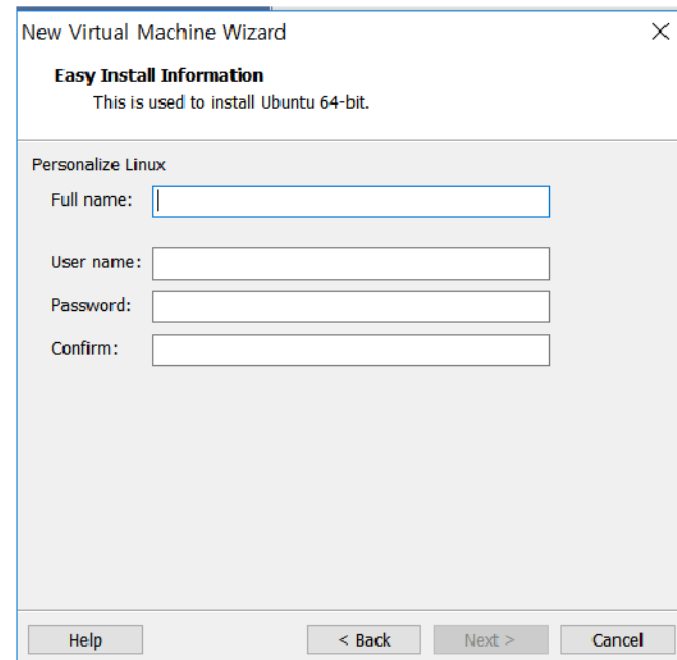
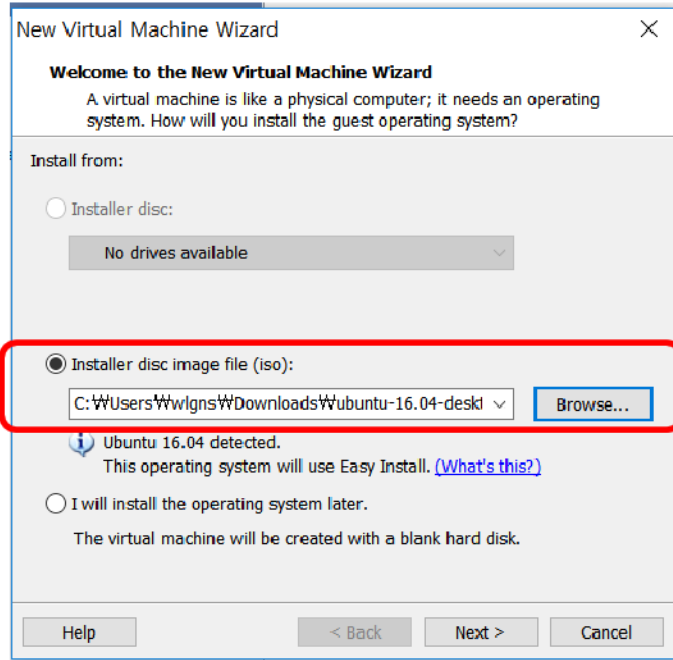
설치 가이드 - VMware

- 다음 절차에 따라 생성
 - Create 버튼 클릭



설치 가이드 - VMware

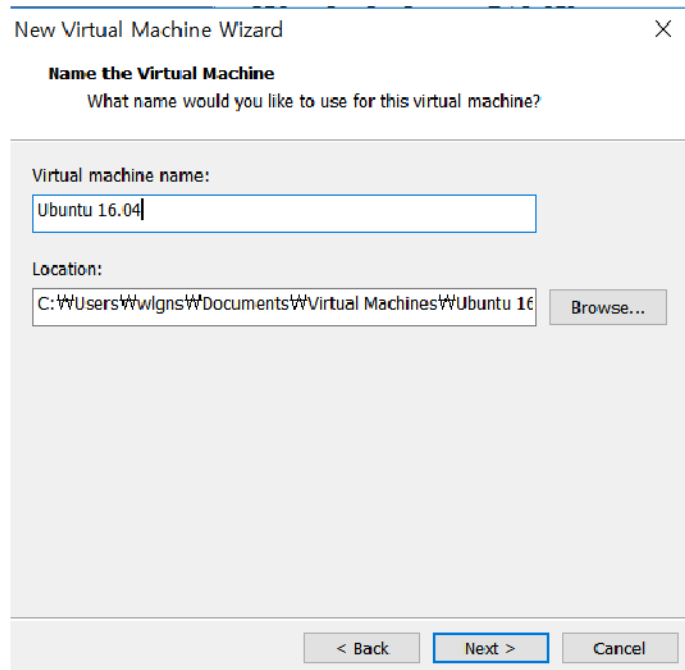
□ 다음 절차에 따라 생성



- P11에서 다운받은 Ubuntu image file의 경로를 선택 후 next
- 사용할 ID와 password 입력 후 next

설치 가이드 - VMware

□ 다음 절차에 따라 생성



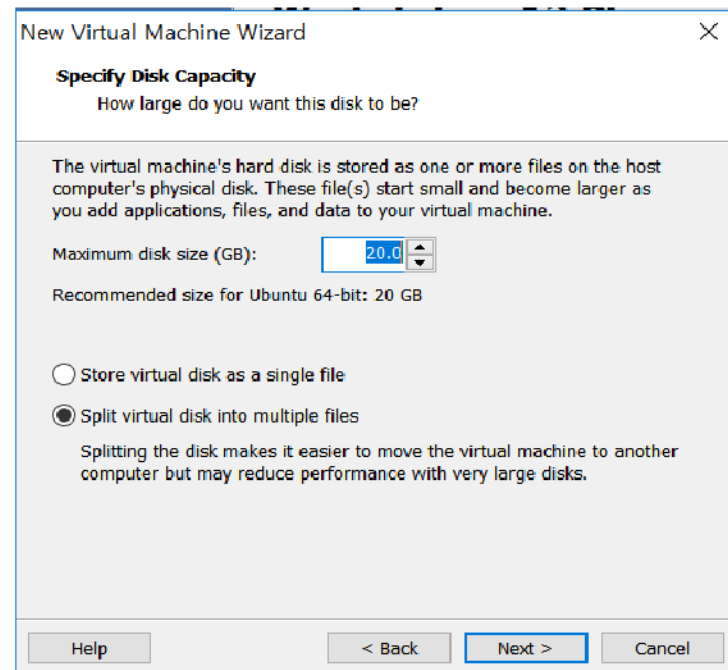
New Virtual Machine Wizard

Name the Virtual Machine
What name would you like to use for this virtual machine?

Virtual machine name:
Ubuntu 16.04

Location:
C:\Users\wlgns\Documents\Virtual Machines\Ubuntu 16.04
Browse...

< Back Next > Cancel



New Virtual Machine Wizard

Specify Disk Capacity
How large do you want this disk to be?

The virtual machine's hard disk is stored as one or more files on the host computer's physical disk. These file(s) start small and become larger as you add applications, files, and data to your virtual machine.

Maximum disk size (GB): 20.0

Recommended size for Ubuntu 64-bit: 20 GB

☐ Store virtual disk as a single file

☒ Split virtual disk into multiple files

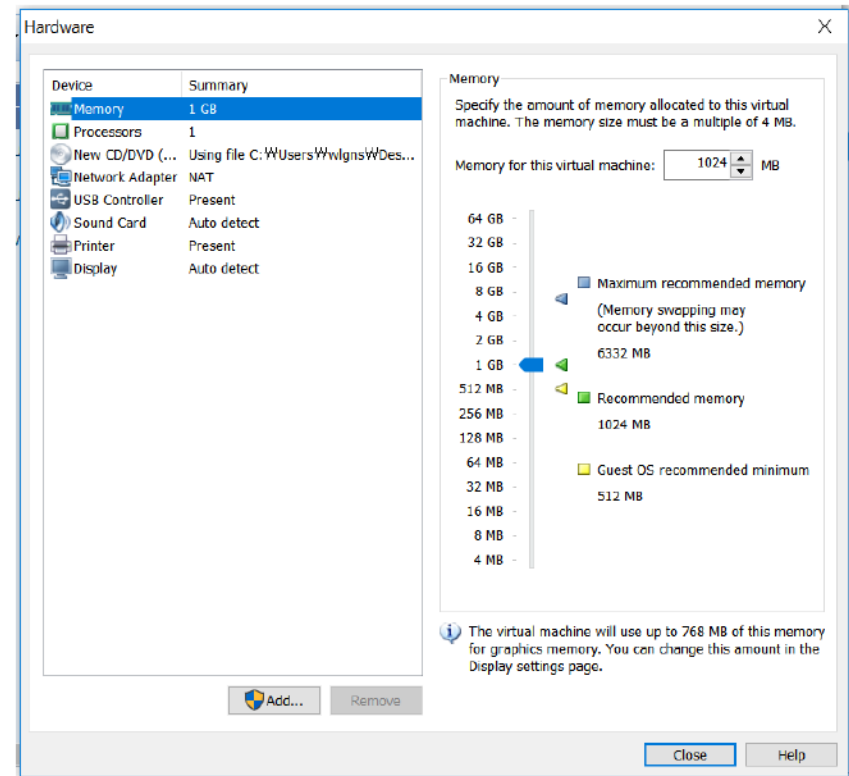
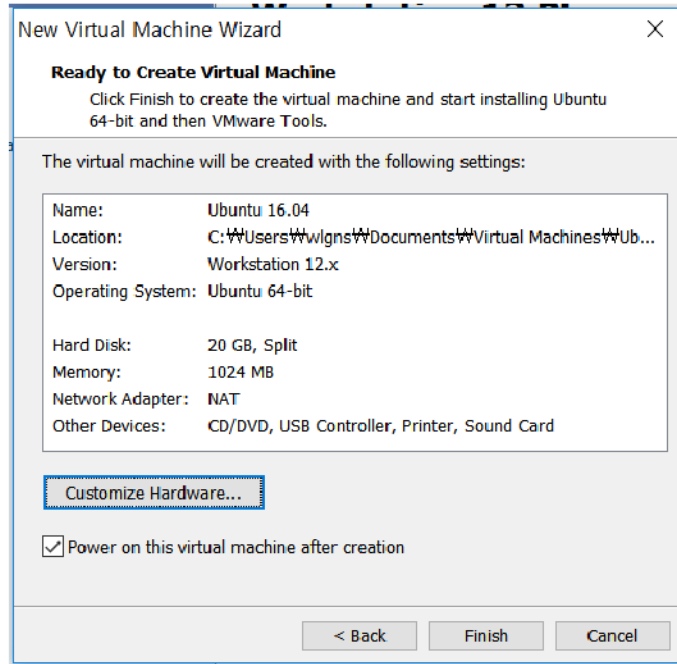
Splitting the disk makes it easier to move the virtual machine to another computer but may reduce performance with very large disks.

Help < Back Next > Cancel

□ 생성하는 Virtual machine 가상 storage size 설정 후 next

설치 가이드 - VMware

□ 다음 절차에 따라 생성



□ Finish를 누르면 ubuntu 설치 시작

설치 가이드 - VMware

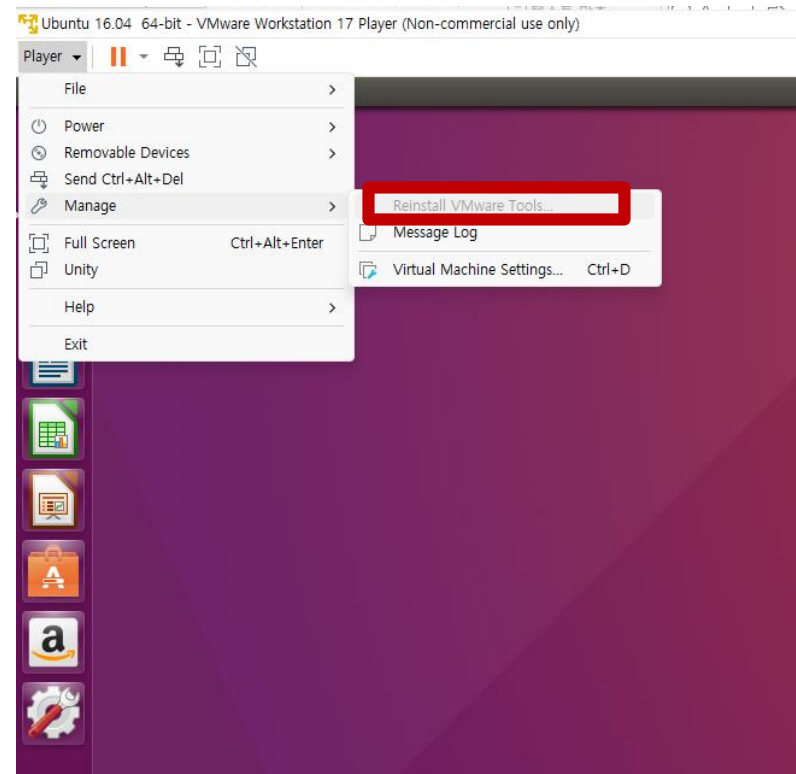
Ubuntu 버전은 반드시 16.04
이어야 함. 다른 버전에서는 본
환경이 설치가 잘 안됨!!

□ 리눅스로 부팅 한 후, ubuntu업그레이드 할 것인지
 물어볼 수도 있는데, 절대로 업그레이드 하지 말것

□ 부팅 후 아래 그림과 같이 Reinstall Vmware Tools 선택

Vmware Tools?

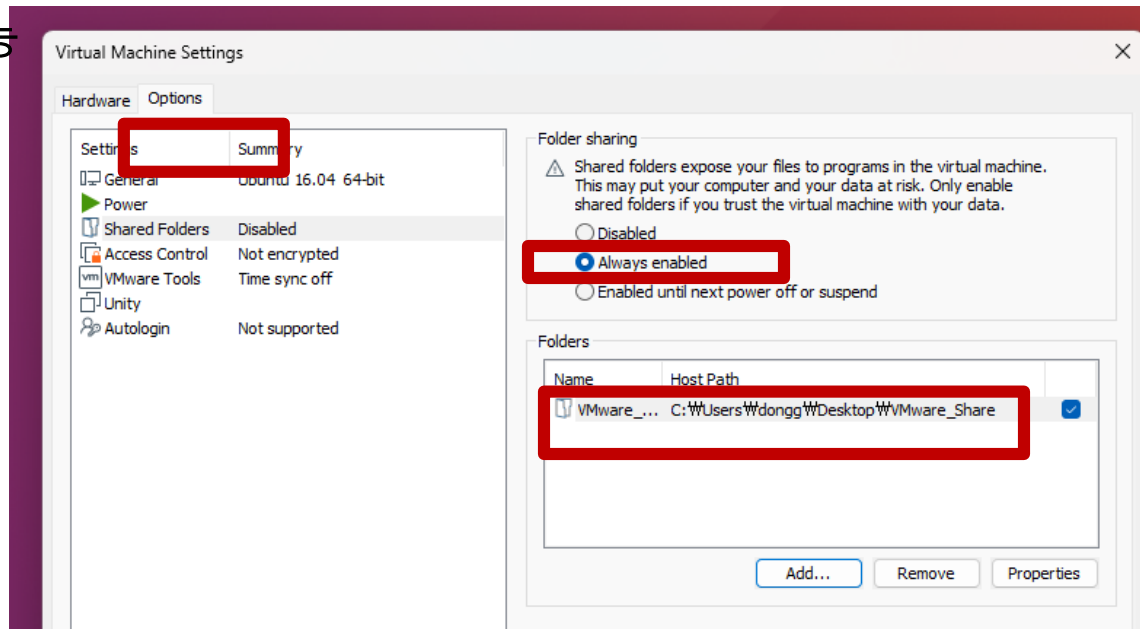
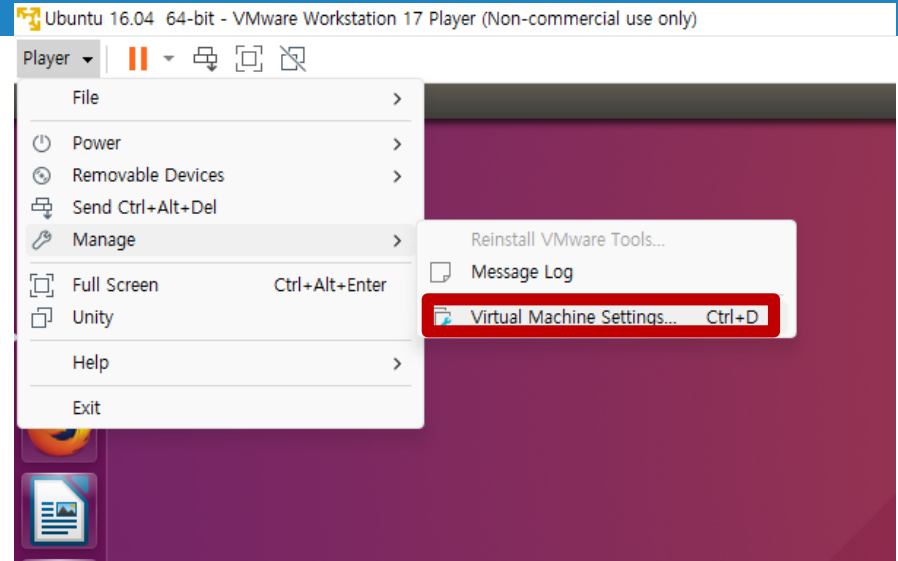
해당 게스트에 전용 드라이버
모음인 VMware Tools를 설치하면,
모든 장치드라이버를 자동으로
설치할 수 있으며 호스트에서
"커스텀 하드웨어"에 직접 접근해
속도가 비약적으로 상승한다.
부가기능인 파일을
드래그한다든지 클립보드 공유나
시간 동기화, Unity 모드와 디버깅
등을 하기 위해서도 반드시
설치해야 한다.



설치 가이드 - VMware

□ 윈도우-리눅스 간 공유 디렉토리 설정

- 윈도우에서 공유할 디렉토리 하나 생성
- Vmware에서 오른쪽 그림들과 같이 파일 공유 설정
- 이후 리눅스머신의 /mnt/hgfs/ 디렉토리 아래 설정한 공유 디렉토리가 보임
- 윈도우-리눅스간 파일 공유 가능



설치 가이드 - Compiler

□ gcc 베이스 오픈 소스 **64비트ARM용 컴파일러인 gcc-aarch64-linux-gnu-gcc**를 사용한다.

□ 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gcc-aarch64-linux-gnu
```

□ 컴파일러 설치 확인

```
qwer@ubuntu:~$ ls /usr/bin/aarch64-linux-gnu*  
/usr/bin/aarch64-linux-gnu-addr2line /usr/bin/aarch64-linux-gnu-gcov  
/usr/bin/aarch64-linux-gnu-ar /usr/bin/aarch64-linux-gnu-gcov-7  
/usr/bin/aarch64-linux-gnu-as /usr/bin/aarch64-linux-gnu-gcov-dump  
/usr/bin/aarch64-linux-gnu-c++filt /usr/bin/aarch64-linux-gnu-gcov-dump-7  
/usr/bin/aarch64-linux-gnu-cpp /usr/bin/aarch64-linux-gnu-gcov-tool  
/usr/bin/aarch64-linux-gnu-cpp-7 /usr/bin/aarch64-linux-gnu-gcov-tool-7  
/usr/bin/aarch64-linux-gnu-dwp /usr/bin/aarch64-linux-gnu-gprof  
/usr/bin/aarch64-linux-gnu-elfedit /usr/bin/aarch64-linux-gnu-ld  
/usr/bin/aarch64-linux-gnu-g++ /usr/bin/aarch64-linux-gnu-ld.bfd  
/usr/bin/aarch64-linux-gnu-g++-7 /usr/bin/aarch64-linux-gnu-ld.gold  
/usr/bin/aarch64-linux-gnu-gcc /usr/bin/aarch64-linux-gnu-nm  
/usr/bin/aarch64-linux-gnu-gcc-7 /usr/bin/aarch64-linux-gnu-objcopy  
/usr/bin/aarch64-linux-gnu-gcc-ar /usr/bin/aarch64-linux-gnu-objdump  
/usr/bin/aarch64-linux-gnu-gcc-ar-7 /usr/bin/aarch64-linux-gnu-ranlib  
/usr/bin/aarch64-linux-gnu-gcc-nm /usr/bin/aarch64-linux-gnu-readelf  
/usr/bin/aarch64-linux-gnu-gcc-nm-7 /usr/bin/aarch64-linux-gnu-size  
/usr/bin/aarch64-linux-gnu-gcc-ranlib /usr/bin/aarch64-linux-gnu-strings  
/usr/bin/aarch64-linux-gnu-gcc-ranlib-7 /usr/bin/aarch64-linux-gnu-strip
```

설치 가이드 - Qemu

- Host PC(Intel CPU)에서 ARM 프로그램을 테스트 할 수 있도록 해주는 가상 에뮬레이터인 Qemu를 설치한다.
- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install qemu
```

- Qemu 설치 확인

```
adio2000@ubuntu:~$ ls /usr/bin/qemu*  
/usr/bin/qemu-aarch64      /usr/bin/qemu-ppc64  
/usr/bin/qemu-alpha       /usr/bin/qemu-ppc64abi32  
/usr/bin/qemu-arm         /usr/bin/qemu-ppc64le  
/usr/bin/qemu-armeb       /usr/bin/qemu-s390x  
/usr/bin/qemu-cris        /usr/bin/qemu-sh4  
/usr/bin/qemu-i386        /usr/bin/qemu-sh4eb  
/usr/bin/qemu-img         /usr/bin/qemu-sparc  
/usr/bin/qemu-io          /usr/bin/qemu-sparc32plus
```

설치 가이드 – Multiarch Debugger

- gdb-multiarch는 여러 가지 architecture를 지원하는 GNU debugger이다(하나의 GDB이다). ARM 바이너리를 디버깅 할 수 있도록 해준다.

- 아래와 같이 커맨드를 이용하여 설치

```
$ sudo apt-get install gdb-multiarch
```

- gdb-multiarch 설치 확인

```
qwer@ubuntu:~$ ls /usr/bin/gdb-multiarch  
/usr/bin/gdb-multiarch
```

설치 가이드 - gdbgui

```
$sudo apt install python-pip
$wget https://bootstrap.pypa.io/pip/3.5/get-pip.py -O /tmp/get-pip.py
$sudo python3 /tmp/get-pip.py
$pip install --user pipenv
$pip3 install --user pipenv
$echo "PATH=$HOME/.local/bin:$PATH" >> ~/.profile
$source ~/.profile
$whereis pip
$sudo pip install gdbgui==0.13.0.0
```

(sudo pip install gdbgui==0.13.0.0 설치 중 다음과 같은 에러가 발생 할 수 있으나, 설치는 잘 되고, 동작에 문제 없음)

```
Collecting brotli
  Downloading Brotli-1.1.0.tar.gz (7.4 MB)
    | 7.4 MB 1.8 MB/s
Installing build dependencies ... done
Getting requirements to build wheel ... error
ERROR: Command errored out with exit status 1:
  command: /usr/bin/python3 /usr/local/lib/python3.5/dist-packages/pip/_vendor/pep517/_in_process.py get_requires_for_build_wheel /tmp/tmpzzq8eh7
  cwd: /tmp/pip-install-41s6f25d/brotli_1.1.0
Complete output (17 lines):
Traceback (most recent call last):
  File "/usr/local/lib/python3.5/dist-packages/pip/_vendor/pep517/_in_process.py", line 200, in _main
    main()
  File "/usr/local/lib/python3.5/dist-packages/pip/_vendor/pep517/_in_process.py", line 203, in main
    json_out['return_val'] = hook(**hook_input['kwargs'])
  File "/usr/local/lib/python3.5/dist-packages/pip/_vendor/pep517/_in_process.py", line 114, in get_requires_for_build_wheel
    return hook(config_settings)
  File "/usr/local/lib/python3.5/dist-packages/setuputils/build_meta.py", line 150, in get_requires_for_build_wheel
    config_settings, requirements=['wheel'])
  File "/usr/local/lib/python3.5/dist-packages/setuputils/build_meta.py", line 130, in get_build_requires
    self.run_setup()
  File "/usr/local/lib/python3.5/dist-packages/setuputils/build_meta.py", line 145, in run_setup
    exec(compile(code, __file__, 'exec'), locals())
  File "setup.py", line 30
    s = re.match(r'^(?!(?:[a-zA-Z0-9_]+\.?)+$)', line)
    ^
SyntaxError: invalid syntax
WARNING: Discarding https://files.pythonhosted.org/packages/2f/c2/f9e977608bdf958650638c3f1e28f85a1b075f075ebbe77db8555463787b/Brotli-1.1.0.tar.gz#sha256=81d
e88ac11bcb85841e440c13611c00b67d3bf82698314928d8b676362546724 (from https://pypi.org/simple/brotli/). Command errored out with exit status 1: /usr/bin/python
3 /usr/local/lib/python3.5/dist-packages/pip/_vendor/pep517/_in_process.py get_requires_for_build_wheel /tmp/tmpzzq8eh7 Check the logs for full command outp
ut.
Downloading Brotli-1.1.0-cp35-cp37m-macosx10.9-x86_64.whl (257 kb)
```


설치 가이드 - gdbgui

□ gdbgui 설치 확인

```
$ gdbgui -g gdb-multiarch
```

```
yj@ubuntu:~/lab0$ gdbgui -g gdb-multiarch
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
```



사용법 – Compiler

□ 컴파일 형식

```
$ aarch64-linux-gnu-gcc [options] files ...
```

- aarch64-linux-gnu-gcc --help 를 통해 자세한 내용 참고

실행방법은 따라하지
말고, 일단
참고하세요. 뒤에
example 예제에서
따라하면 됩니다.

□ 실행 방법(예)

```
$ aarch64-linux-gnu-gcc -o lab0 assembler.S main.c -static
```

- main.c와 assembler.S file을 compile, assemble 한 뒤 lab0 이름의 실행 가파일이 생성한다.
- -o 옵션 다음에는 컴파일된 실행 파일의 이름을 지정
- -static 은 정적 링킹 옵션

사용법 – Qemu

□ Qemu 실행 command format

```
qemu-aarch64 [options] program [ arguments ... ]
```

- ▣ qemu-aarch64 --help 를 통해 자세한 내용 참고

실행방법은 따라하지
말고, 일단
참고하세요. 뒤에
example 예제에서
따라하면 됩니다.

□ 실행 방법(예)

```
$ qemu-aarch64 -g 8080 ./lab0 (포트번호는 임의 지정)
```

- ▣ -g 옵션으로 gdb 연결을 위한 포트 번호를 설정할 수 있다.
(포트번호는 임의 지정)
- ▣ 위 명령어를 수행하면 연결대기 상태로 들어간다. 다른 터미널창에서
다음 페이지의 gdbgui 명령어를 수행한다.

사용법 – gdbgui

□ gdbgui 실행 command format

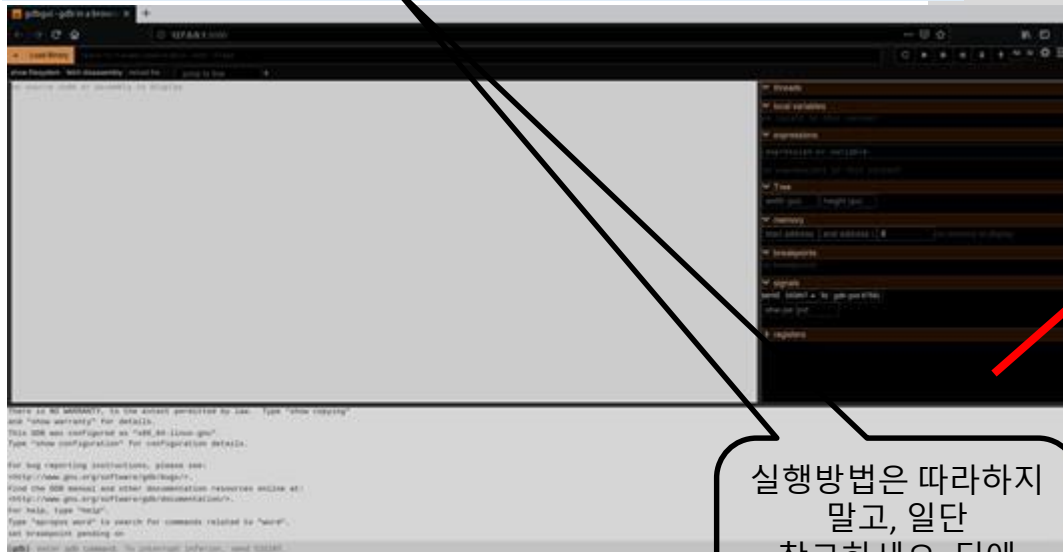
```
$ gdbgui [options] [args]
```

- ▣ gdbgui --help 를 통해 자세한 내용 참고

□ 실행 방법(예)

```
$ gdbgui -g gdb-multiarch
```

실행후 gdbgui 0.15.1.0 으로
upgrade 하라는 추천 메시지가 나오는데 **upgrade** 하지
마세요! 그버전은 아직 불안
정합니다.



실행방법은 따라하지
말고, 일단
참고하세요. 뒤에
example 예제에서
따라하면 됩니다.

뒤에 나올 예제0 ~ 3 통
해, **gdbgui** 화면에서
레지스터값 확인 및
수정, 메모리 값 확인
및 수정 방법을 설명

예제0. 실습 목표

- copyarray 함수를 실행시켜 잘 실행되는지 확인
 - copyarray는 배열 값들을 복사하는 어셈블리어 함수이다.
 - copyarray 함수를 사용하는 프로그램을 앞서 설치한 크로스 컴파일러를 활용해 컴파일한다.
 - Qemu를 사용하여 프로그램을 수행하고 결과값을 확인해본다.

```
Array Original :2 12 -1 10 7  
Array Copy :0 0 0 0 0  
Array Original :2 12 -1 10 7  
Array Copy :2 12 -1 10 7
```

초기조건

copyarray함수
수행 후 결과

예제0. 소스 코드

주의사항1: ADDI, SUBI 등 Immediate명령어
는 지원 안함 → ADD, SUB 명령어에 통합

주의사항2: LR(X30레지스터 이름) 대신
X30이라고 써야함.

주의사항3: 64비트 컴퓨터도 int는 32비트임 (long:64비트)

□ main.c

```
adio2000@ubuntu: ~/assembly64/lab0
#include <stdio.h>

extern void copyarray(long arr1[], long arr2[], long n);

void printarr(long arr[], long n){
    for(long i = 0; i < n ; i++){
        printf("%ld ",arr[i]);
        printf("\n");
    }
}

int main(){
    long arrOri[5] = {2,12,-1,10,7};
    long arrCpy[5] = {0, };
    long num = 5;
    printf("Array Original :");
    printarr(arrOri,num);

    printf("Array Copy :");
    printarr(arrCpy,num);

    copyarray(arrOri, arrCpy, num);

    printf("Array Original :");
    printarr(arrOri,num);

    printf("Array Copy :");
    printarr(arrCpy,num);

    return 0;
}
```

□ assem.S

```
adio2000@ubuntu: ~/assembly64/lab0
.text
.global copyarray
.type copyarray,STT_FUNC

copyarray:
    sub    sp, sp, #32
    stur   x19, [sp,#0]
    stur   x20, [sp,#8]
    stur   x21, [sp,#16]
    stur   x22, [sp,#24]
    add    x19, xzr, xzr

cwhile:
    cmp    x2, x19
    b.le   cexit
    lsl    x20, x19, #3
    add    x21, x0, x20
    ldur   x22, [x21,#0]
    add    x21, x1, x20
    stur   x22, [x21,#0]
    add    x19, x19, #1
    b      cwhile

cexit:
    ldur   x22, [sp,#24]
    ldur   x21, [sp,#16]
    ldur   x20, [sp,#8]
    ldur   x19, [sp,#0]
    add    sp, sp, #32
    br     x30

.end
```

예제0. 실습 과정

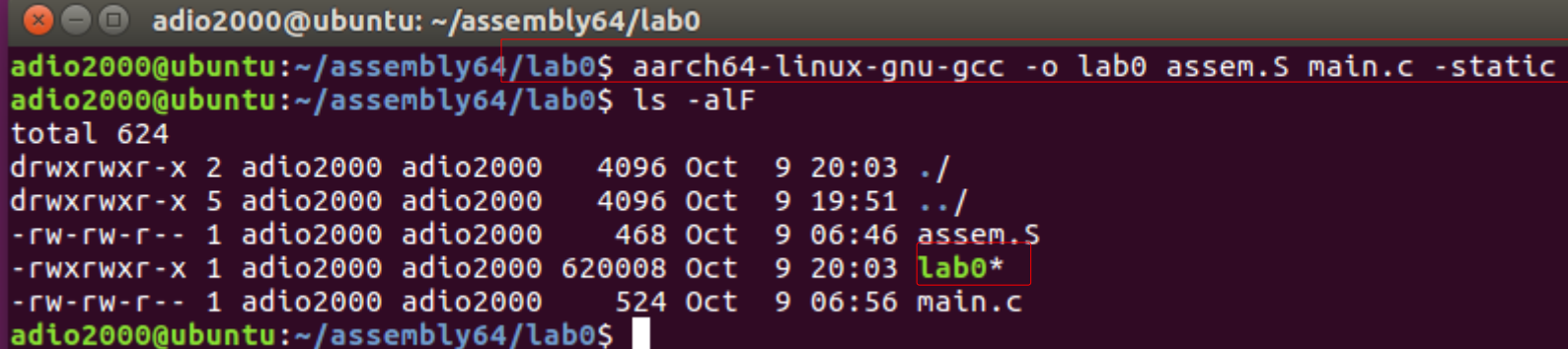
- 소스코드 다운로드 및 압축 해제
 - ▣ Ubuntu에서 적당한 위치에 lab0 디렉토리 생성
 - ▣ U-Saint 강의 홈페이지에서.zip (소스코드 압축 파일)를 위에서 만든 lab0 디렉토리에 다운로드
(윈도우에서 받아서 공유폴더를 이용해 Ubuntu로 가져와도 됨)
 - ▣ lab0 디렉토리로 이동 (이후 모든 작업들은 이 디렉토리에서 수행)
 - ▣ unzip 명령어를 이용하여 lab0.zip 압축해제

예제0. 실습 과정

- 소스코드를 압축 해제 했다면, 컴파일을 해보자.

```
$ aarch64-linux-gnu-gcc -o lab0 assem.S main.c -static
```

- 컴파일 되었다면, 아래와 같이 lab0 이름의 실행 파일이 생성된다.

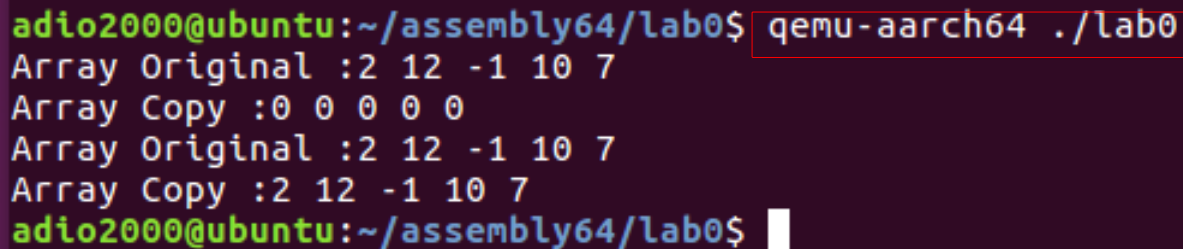


```
adio2000@ubuntu: ~/assembly64/lab0
adio2000@ubuntu:~/assembly64/lab0$ aarch64-linux-gnu-gcc -o lab0 assem.S main.c -static
adio2000@ubuntu:~/assembly64/lab0$ ls -alF
total 624
drwxrwxr-x 2 adio2000 adio2000 4096 Oct 9 20:03 ./
drwxrwxr-x 5 adio2000 adio2000 4096 Oct 9 19:51 ../
-rw-rw-r-- 1 adio2000 adio2000 468 Oct 9 06:46 assem.S
-rwxrwxr-x 1 adio2000 adio2000 620008 Oct 9 20:03 lab0*
-rw-rw-r-- 1 adio2000 adio2000 524 Oct 9 06:56 main.c
adio2000@ubuntu:~/assembly64/lab0$
```


예제0. 실습 과정

- 실행파일을 qemu로 동작시킨다.

```
$ qemu-aarch64 ./lab0
```



```
audio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 7
audio2000@ubuntu:~/assembly64/lab0$
```

- Original 배열이 Copy 배열로 복사된 것을 확인할 수 있다.

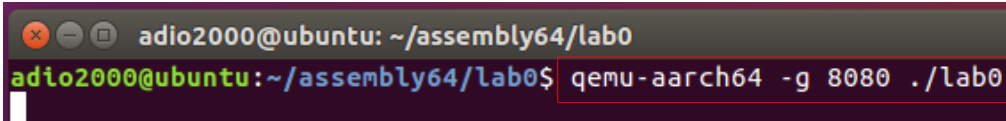
예제1. 실습 목표

- GDB를 사용하여 다음 페이지 copyarray 함수 분석
 - ▣ 예제 0의 lab0 이라는 실행파일이 생성된 후,
Qemu와 GDB 환경 위에서 이 프로그램을 수행하고, breakpoint 활용,
레지스터 출력, 메모리 출력 등의 GDB 명령어를 사용해본다.
 - ▣ copyarray 함수를 수행할 때 레지스터와 메모리 변화를 분석한다.

예제1. 실습 과정

- 실행파일을 디버깅 모드로 qemu 위에서 동작시킨다.
 - ▣ qemu를 gdb-multiarch와 연결시키기 위해 아래와 같이 실행
 - ▣ 실행하면 아래와 같이 gdb-multiarch 연결 대기 상태가 된다.

```
$ qemu-aarch64 -g 8080 ./lab0
```



```
audio2000@ubuntu: ~/assembly64/lab0
audio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
```

예제1. 실습 과정

- Qemu는 대기 상태로 두고, 새로운 터미널을 연다.
 - 새로운 터미널에서 lab0 디렉토리로 이동 후,
gdbgui를 사용하여 gdb-multiarch를 실행시킨다.
 - 터미널 단축키는 ctrl+alt+T 혹은 기존 터미널에서 ctrl+shift+T

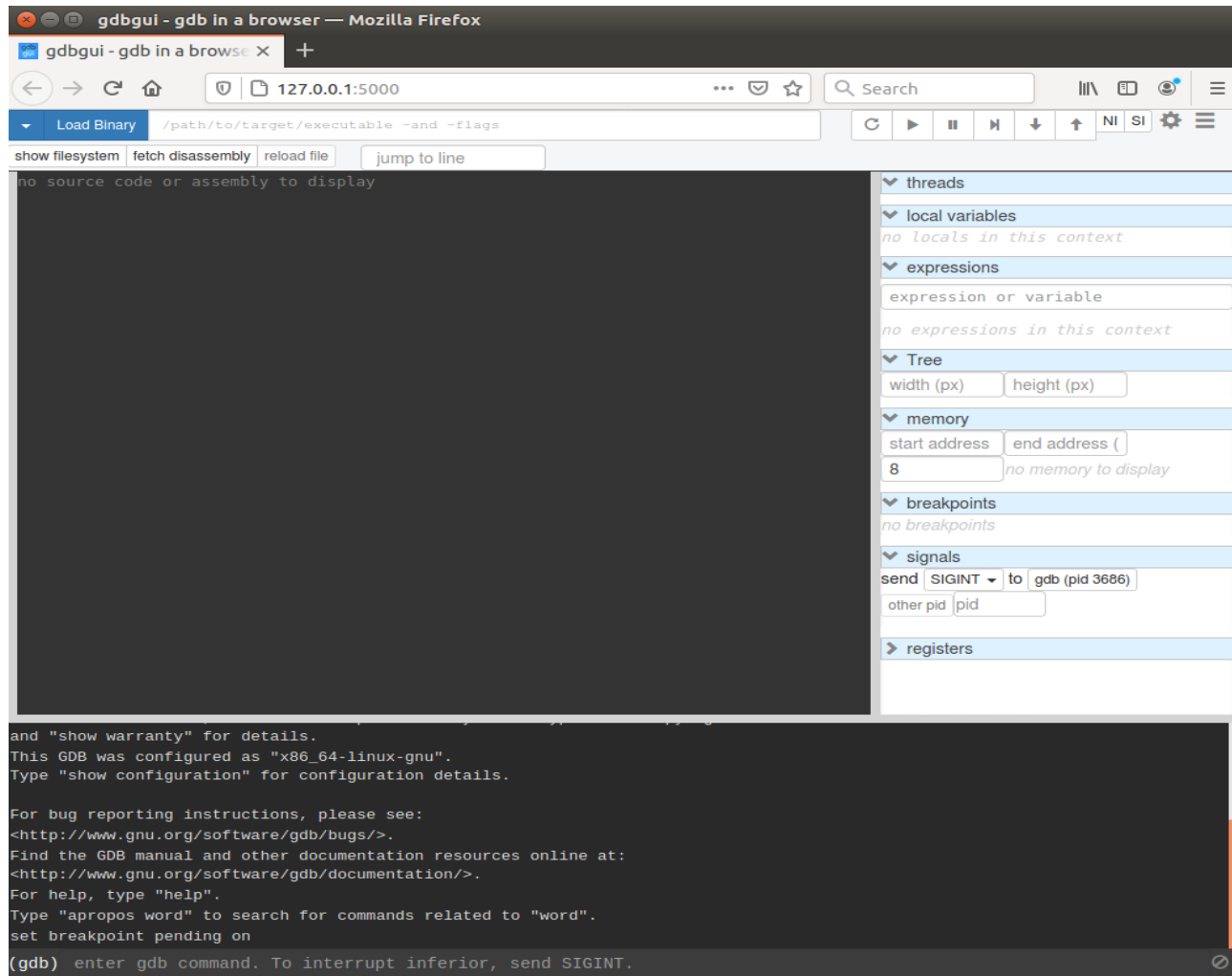
```
$ gdbgui -g gdb-multiarch
```

```
radio2000@ubuntu:~$ gdbgui -g gdb-multiarch
failed to initialize socketio app with async mode "gevent". Continuing with async
mode "threading".
Opening gdbgui with default browser at http://127.0.0.1:5000
exit gdbgui by pressing CTRL+C
WebSocket transport not available. Install eventlet or gevent and gevent-websock
et for improved performance.
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [09/Oct/2023 22:09:33] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Oct/2023 22:09:36] "GET /static/css/gdbgui.css?_undefined HTTP
/1.1" 200 -
127.0.0.1 - - [09/Oct/2023 22:09:36] "GET /static/vendor/css/gdbgui_awesomeplete
.css HTTP/1.1" 200 -
```

예제1. 실습 과정

gdbgui 0.15.1.0 으로
upgrade 하라는 추천 메시지가 나오는데 **upgrade** 하지
마세요! 그버전은 아직 불안함

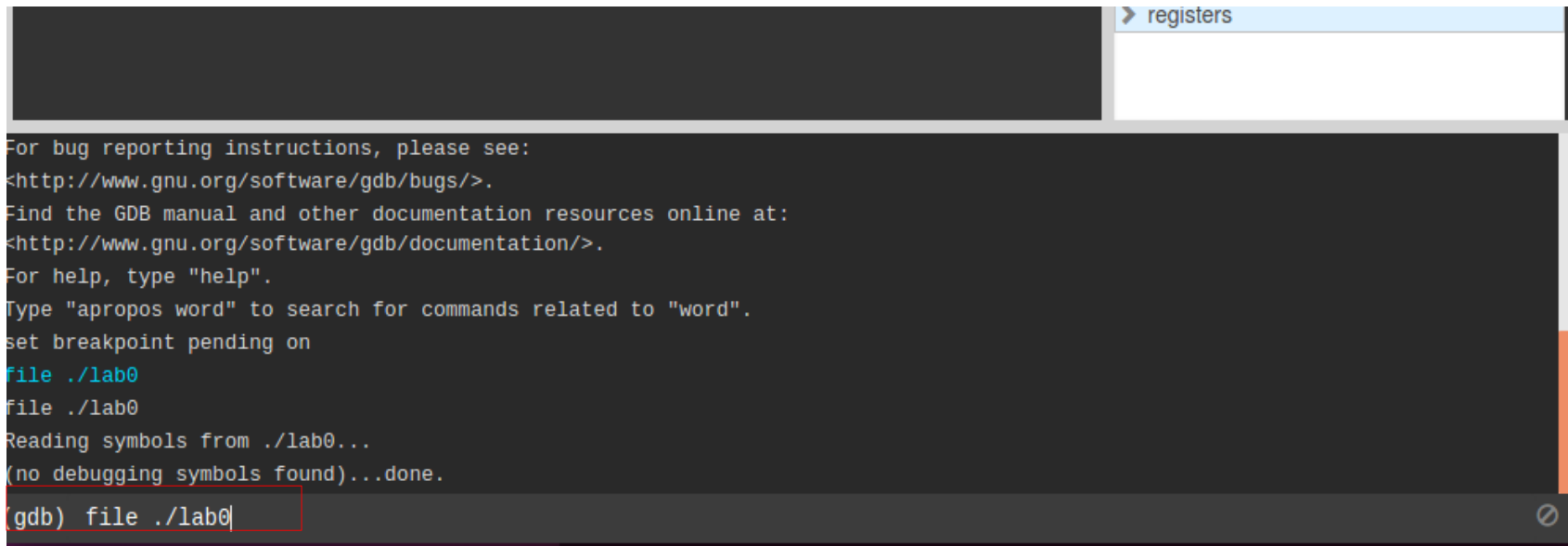
□ gdbgui 실행 화면



예제1. 실습 과정

- gdbgui 하단의 터미널에서 동작할 file을 알려준다.

```
(gdb) file ./lab0
```



The screenshot shows the gdbgui interface. At the top, there is a dark grey header bar. Below it, on the left, is a large terminal window with a dark background. On the right, there is a sidebar with a light blue header labeled 'registers'. The terminal window displays the following text:

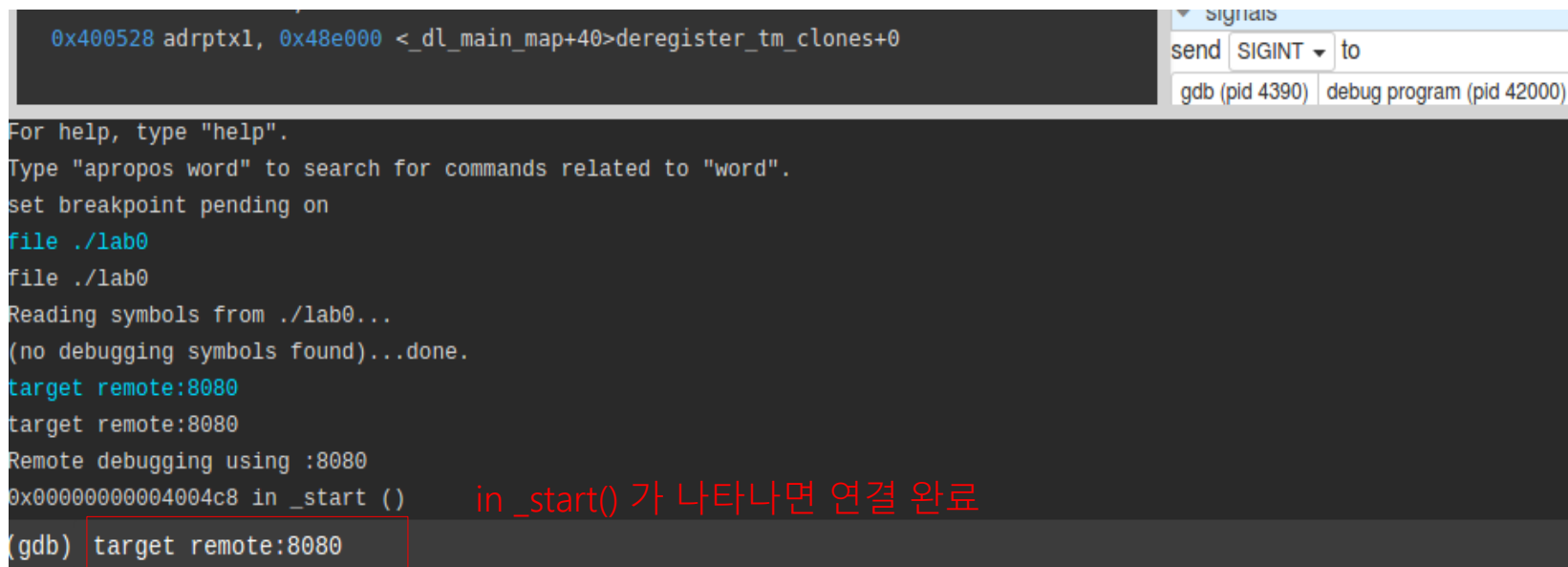
```
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
set breakpoint pending on
file ./lab0
file ./lab0
Reading symbols from ./lab0...
(no debugging symbols found)...done.
(gdb) file ./lab0
```

The last line of the terminal output, '(gdb) file ./lab0', is highlighted with a red rectangular box.

예제1. 실습 과정

- Qemu에서 대기하고 있는 GDB 연결 포트 번호를 입력한다.

(gdb) **target remote:8080** (포트번호는 qemu 실행시 썼던 것)



The screenshot shows a GDB terminal window with the following content:

```
0x400528 adrptx1, 0x48e000 <_dl_main_map+40>deregister_tm_clones+0
```

On the right side, there is a 'signals' panel with a dropdown menu set to 'SIGINT' and buttons for 'send' and 'to'. Below this, it shows 'gdb (pid 4390)' and 'debug program (pid 42000)'.

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
set breakpoint pending on
file ./lab0
file ./lab0
Reading symbols from ./lab0...
(no debugging symbols found)...done.
target remote:8080
target remote:8080
Remote debugging using :8080
0x00000000004004c8 in _start ()
(gdb) target remote:8080
```

Below the terminal output, the text "in _start() 가 나타나면 연결 완료" is written in red.

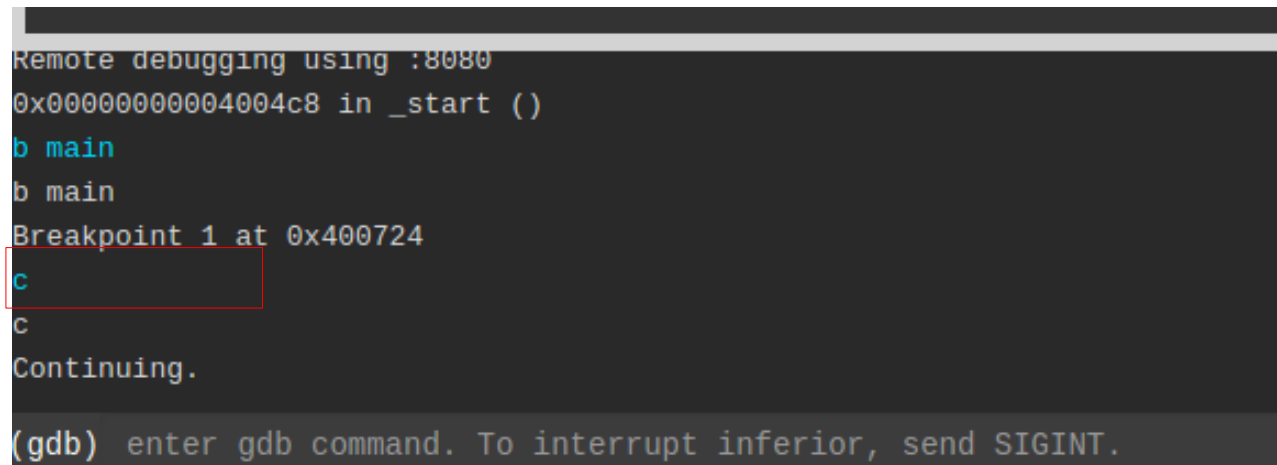
예제1. 실습 과정

- main 함수에 breakpoint 설정
 - ▣ (gdb) break main 또는 b main

```
Reading symbols from ./lab0...
(no debugging symbols found)...done.
-data-disassemble: Invalid filename.
target remote:8080
target remote:8080
Remote debugging using :8080
0x00000000004004c8 in _start ()
b main
b main
Breakpoint 1 at 0x400724
(gdb) |enter gdb command. To interrupt inferior, send SIGINT.
```


예제1. 실습 과정

- main 함수의 시작까지 실행
 - ▣ (gdb) continue 또는 c 입력



```
Remote debugging using :8080
0x00000000004004c8 in _start ()
b main
b main
Breakpoint 1 at 0x400724
c
c
Continuing.

(gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

- ▣ main함수의 시작 주소까지 실행된 것임

예제1. 실습 과정

- main 함수의 시작 까지 실행
 - ▣ 왼쪽 창에 지금 수행하고 있는 소스코드 라인이 표시됨
 - ▣ 오른쪽 창에 함수의 시작 주소, 변수값, 메모리 등 정보가 나옴

The screenshot shows a debugger interface with a dark theme. The main window displays assembly code for a function named 'main'. The first line, '0x400724 ldrtx1, [x0] main+16', is highlighted with a red box. The code continues with various instructions like 'strtx1', 'movtx1', 'adrptx0', 'addtx1', 'addtx0', 'ldptx2', 'stptx2', 'ldptx2', 'stptx2', 'ldrtx1', 'strtx1', 'stptx2', 'stptx2', 'stptx2', 'movtx0', 'strtx0', 'adrptx0', 'addtx0', 'blt0x406950', 'addtx0', 'ldrtx1', 'blt0x4006ac', 'adrptx0', and 'addtx0'.

On the right side, there are several panels:

- threads**: A table showing the current thread. The first row is selected and shows 'main' at address '0x400724'.
- local variables**: A section titled 'no locals in this context'.
- expressions**: A section titled 'no expressions in this context'.
- Tree**: A section with input fields for 'width (px)' and 'height (px)'.
- memory**: A section with input fields for 'start address' and 'end address (8)'.
- breakpoints**: A section with a checked checkbox and the text '(unknown function) thread groups: 11'.
- signals**: A section with a dropdown menu showing 'SIGINT' and a 'to' field.

예제1. 실습 과정

- main 함수까지 실행

- ▣ Main의 시작 지점까지 수행 되었으므로,
qemu 창에 아직은 아무것도 프린트 되지 않음

```
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
```

예제1. 실습 과정

- copyarray 함수에 breakpoint 설정하고 계속 실행
 - ▣ (gdb) b copyarray
 - ▣ (gdb) c

The screenshot displays the GDB (GNU Debugger) interface. The top panel shows assembly code for the `copyarray` function, with instructions like `addtx19, xzr, xzr` and `cmptx2, x19`. The right panel shows the `threads` list, indicating the `copyarray` thread is selected. Below the threads, the `local variables` and `expressions` panels are empty. The `memory` panel shows the start address and end address (8). The `breakpoints` panel shows a breakpoint set at `0x40066c` in the `copyarray` function.

```
> 0x40066c addtx19, xzr, xzr
0x400670 cmptx2, x19
0x400674 b.lt0x400694 <cexit>
0x400678 ls1tx20, x19, #3
0x40067c addtx21, x0, x20
0x400680 ldurtx22, [x21]
0x400684 addtx21, x1, x20
0x400688 sturtx22, [x21]
0x40068c addtx19, x19, #0x1
0x400690 bt0x400670 <cwhile>
0x400694 ldurtx22, [sp, #24]
0x400698 ldurtx21, [sp, #16]
0x40069c ldurtx20, [sp, #8]
0x4006a0 ldurtx19, [sp]
0x4006a4 addtsp, sp, #0x20
0x4006a8 brtx30
0x4006ac stptx29, x30, [sp, #-48]!
0x4006b0 movtx29, sp
0x4006b4 strtx0, [x29, #24]
0x4006b8 strtx1, [x29, #16]
0x4006bc strtxzr, [x29, #40]
0x4006c0 bt0x4006f0 <printarr+68>
0x4006c4 ldrtx0, [x29, #40]
0x4006c8 ls1tx0, x0, #3
0x4006cc ldrtx1, [x29, #24]
```

copyarray+20
cwhile+0
cwhile+4
cwhile+8
cwhile+12
cwhile+16
cwhile+20
cwhile+24
cwhile+28
cwhile+32
cexit+0
cexit+4
cexit+8
cexit+12
cexit+16
cexit+20
printarr+0
printarr+4
printarr+8
printarr+12
printarr+16
printarr+20
printarr+24
printarr+28
printarr+32

threads
selected Remote target, id 1, stopped

func	file	addr	args
copyarray		0x40066c	
main		0x4007a8	

local variables
no locals in this context

expressions
expression or variable
no expressions in this context

Tree
width (px) height (px)

memory
start address: end address (8)
memory to display

breakpoints
☒ (unknown function) thread groups: if
main
(file not cached)
☒ (unknown function) thread groups: if

```
copyarray
copyarray
Breakpoint 2 at 0x40066c
Continuing.

Breakpoint 2, 0x000000000040066c in copyarray ()
gdb) enter gdb command. To interrupt inferior, send SIGINT.
```

예제1. 실습 과정

- copyarray 함수의 시작까지 실행
 - ▣ Copyarray 시작 지점까지 수행 되었으므로,
qemu 창에 아래와 같이 프린트
(main에서copyarray 부르기 전에 프린트한 것임)

```
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
```

예제1. 실습 과정

- GDB를 활용하여 copyarray 함수 분석
 - ▣ 우측 창 맨 밑에 registers 섹션 클릭 후 레지스터들 확인

show filesystemfetch disassemblyreload fileinteljump to line

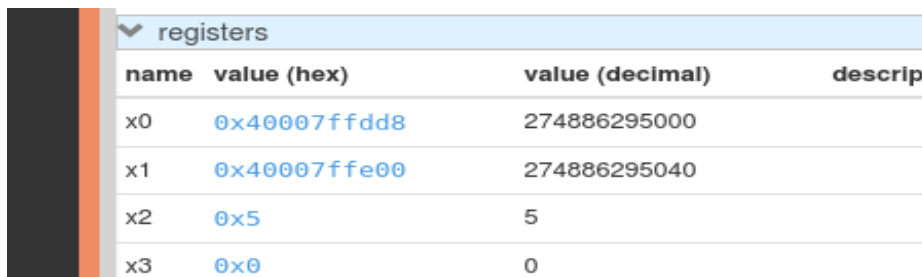
```
> 0x40066c addtx19, xzr, xzr          copyarray+20
0x400670 cmptx2, x19                cwhile+0
0x400674 b.let0x400694 <cexit>       cwhile+4
0x400678 lsltx20, x19, #3           cwhile+8
0x40067c addtx21, x0, x20            cwhile+12
0x400680 ldurtx22, [x21]             cwhile+16
0x400684 addtx21, x1, x20            cwhile+20
0x400688 sturtx22, [x21]             cwhile+24
0x40068c addtx19, x19, #0x1          cwhile+28
0x400690 bt0x400670 <cwhile>        cwhile+32
0x400694 ldurtx22, [sp,#24]          cexit+0
0x400698 ldurtx21, [sp,#16]          cexit+4
0x40069c ldurtx20, [sp,#8]           cexit+8
0x4006a0 ldurtx19, [sp]             cexit+12
0x4006a4 addtsp, sp, #0x20           cexit+16
0x4006a8 brtx30                     cexit+20
0x4006ac stptx29, x30, [sp,#-48]!    printarr+0
0x4006b0 movtx29, sp                printarr+4
0x4006b4 strtx0, [x29,#24]           printarr+8
0x4006b8 strtx1, [x29,#16]           printarr+12
0x4006bc strtxzr, [x29,#40]          printarr+16
```

x21	0x400f08	4198152
x22	0x0	0
x23	0x0	0
x24	0x4001d8	4194776
x25	0x0	0
x26	0x0	0
x27	0x0	0
x28	0x0	0
x29	0x40007ffdc0	274886294976
x30	0x4007a8	4196264
sp	0x40007ffda0	274886294944
pc	0x40066c	4195948
cpsr	0x60000000	1610612736
v0		

예제1. 실습 과정

□ GDB를 활용하여 copyarray 함수 분석

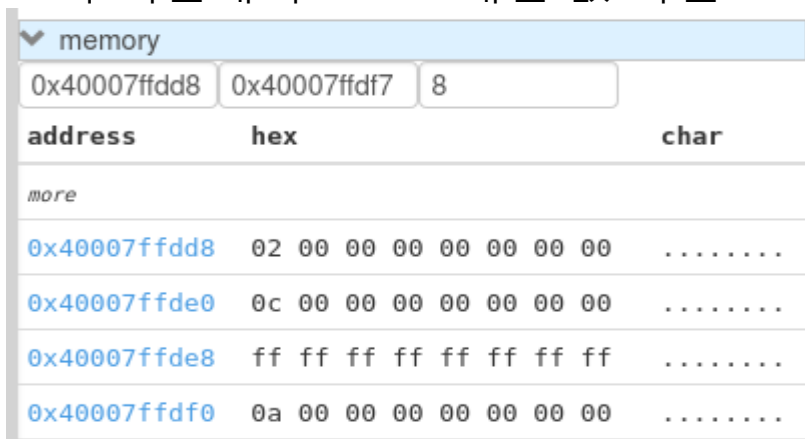
- 아래 registers 섹션에서 \$X0 레지스터에는 copyarray 함수의 첫 번째 인자인 arrOri 배열의 시작 주소값이 저장됨



The screenshot shows the 'registers' window in GDB. It contains a table with the following data:

name	value (hex)	value (decimal)	description
x0	0x40007ffdd8	274886295000	
x1	0x40007ffe00	274886295040	
x2	0x5	5	
x3	0x0	0	

- 위에서 X0의 내용값 (파란색 주소값) 클릭하고 메모리 섹션에서 arrOri 배열 값 확인



The screenshot shows the 'memory' window in GDB. It displays the memory contents starting from address 0x40007ffdd8. The table shows the address, hex value, and character representation.

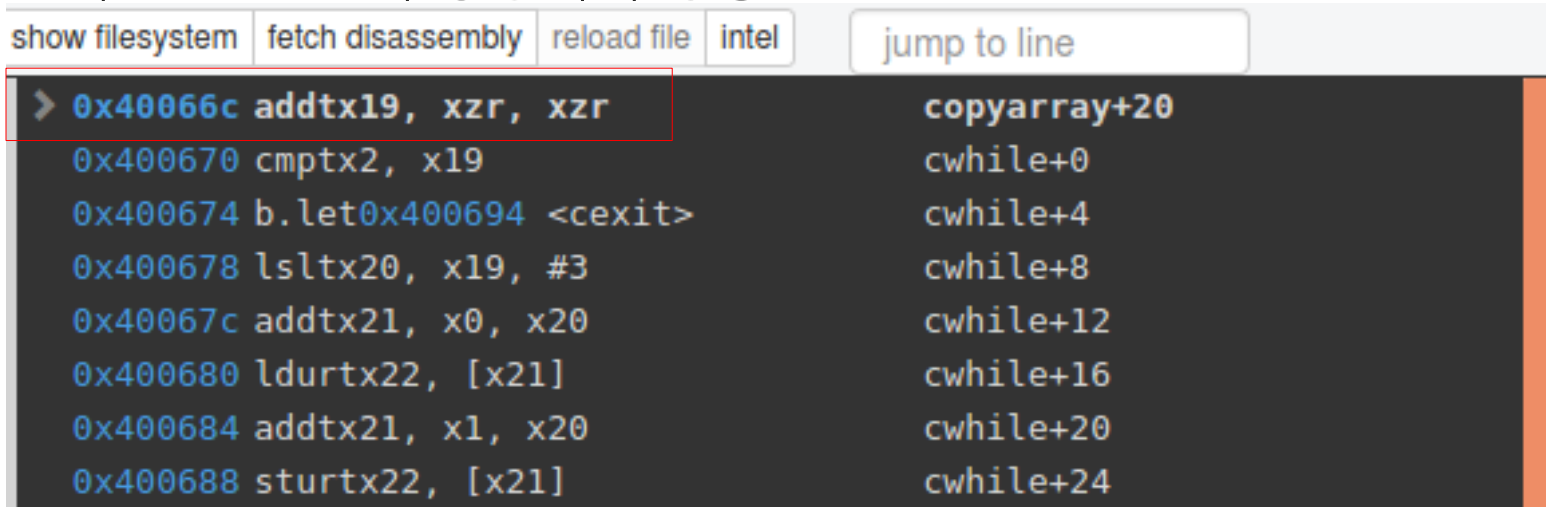
address	hex	char
0x40007ffdd8	02 00 00 00 00 00 00 00
0x40007ffde0	0c 00 00 00 00 00 00 00
0x40007ffde8	ff ff ff ff ff ff ff ff
0x40007ffdf0	0a 00 00 00 00 00 00 00

Little endian으로 저장된 64비트 data임.
2, 12, -1, 10 등이 한 줄씩 차례로 저장된 것을 확인

예제1. 실습 과정

□ GDB를 활용하여 copyarray 함수 분석

▣ 현재 40066c 번지 명령어 수행중



```
show filesystem  fetch disassembly  reload file  intel  jump to line
> 0x40066c addtx19, xzr, xzr                copyarray+20
0x400670 cmptx2, x19                        cwhile+0
0x400674 b.let0x400694 <cexit>              cwhile+4
0x400678 ls1tx20, x19, #3                  cwhile+8
0x40067c addtx21, x0, x20                   cwhile+12
0x400680 ldurtx22, [x21]                   cwhile+16
0x400684 addtx21, x1, x20                   cwhile+20
0x400688 sturtx22, [x21]                   cwhile+24
```

▣ registers 섹션에서 pc 값 확인: 0x40066c 임을 확인

▣ 다음 라인을 수행하려면

(gdb) si 를 입력

예제1. 실습 과정

- si 입력 후 0x400670 명령어 수행됨

pc 레지스터 값도 이 주소로 바뀌었음을 확인

0x40066c	addtx19, xzr, xzr	copyarray+20	x24	0x4001d8	4194776
> 0x400670	cmptx2, x19	cwhile+0	x25	0x0	0
0x400674	b.let0x400694 <cexit>	cwhile+4	x26	0x0	0
0x400678	lsltx20, x19, #3	cwhile+8	x27	0x0	0
0x40067c	addtx21, x0, x20	cwhile+12	x28	0x0	0
0x400680	ldurtx22, [x21]	cwhile+16	x29	0x40007ffdc0	274886294976
0x400684	addtx21, x1, x20	cwhile+20	x30	0x4007a8	4196264
0x400688	sturtx22, [x21]	cwhile+24	sp	0x40007ffda0	274886294944
0x40068c	addtx19, x19, #0x1	cwhile+28	pc	0x400670	4195952
0x400690	bt0x400670 <cwhile>	cwhile+32	cpsr	0x60000000	1610612736
0x400694	ldurtx22, [sp, #24]	cexit+0			
0x400698	ldurtx21, [sp, #16]	cexit+4			
0x40069c	ldurtx20, [sp, #8]	cexit+8			
0x4006a0	ldurtx19, [sp]	cexit+12			
0x4006a4	addtsp, sp, #0x20	cexit+16			

예제1. 실습 과정

□ GDB를 활용하여 copyarray 함수 분석

■ si 명령어를 계속 타이핑하면서 레지스터 및 메모리 값들의 변화를 살펴볼 것

참고) 기본적인 gdb 명령어

명령어	기능
c (Continue) 	프로그램 수행
k (Kill)	프로그램 수행 종료
s (Step) 	현재 행 수행 후 정지, 함수 호출시 함수 안으로 들어감 (s #: #번 연속 수행)
n (Next) 	현재 행 수행 후 정지, 함수 호출시 함수 수행 다음 행으로 이동 (n #: #번 연속 수행)
si (Step Instruction)	어셈블리 명령어 단위의 수행 (진행은 Step과 같음)
ni (Next Instruction)	어셈블리 명령어 단위의 수행 (진행은 Next와 같음)
b* [function]	특정 함수에 breakpoint 설정
b* [address]	특정 주소에 breakpoint 설정

예제1. 실습 과정

□ GDB를 활용하여 copyarray 함수 분석

- 원하는 분석이 끝났다면 c 명령어를 사용하여 프로그램 재개
(gdb) continue 또는 c 입력

이후에 **Breakpoint**가 없기때문에 프로그램 끝까지 실행 후 정상 종료함
빨간색 메시지는 memory section에 입력된 값이 유지되고 있기 때문에 출력 됨

- qemu 창에는 프로그램의 최종 수행 결과가 출력됨

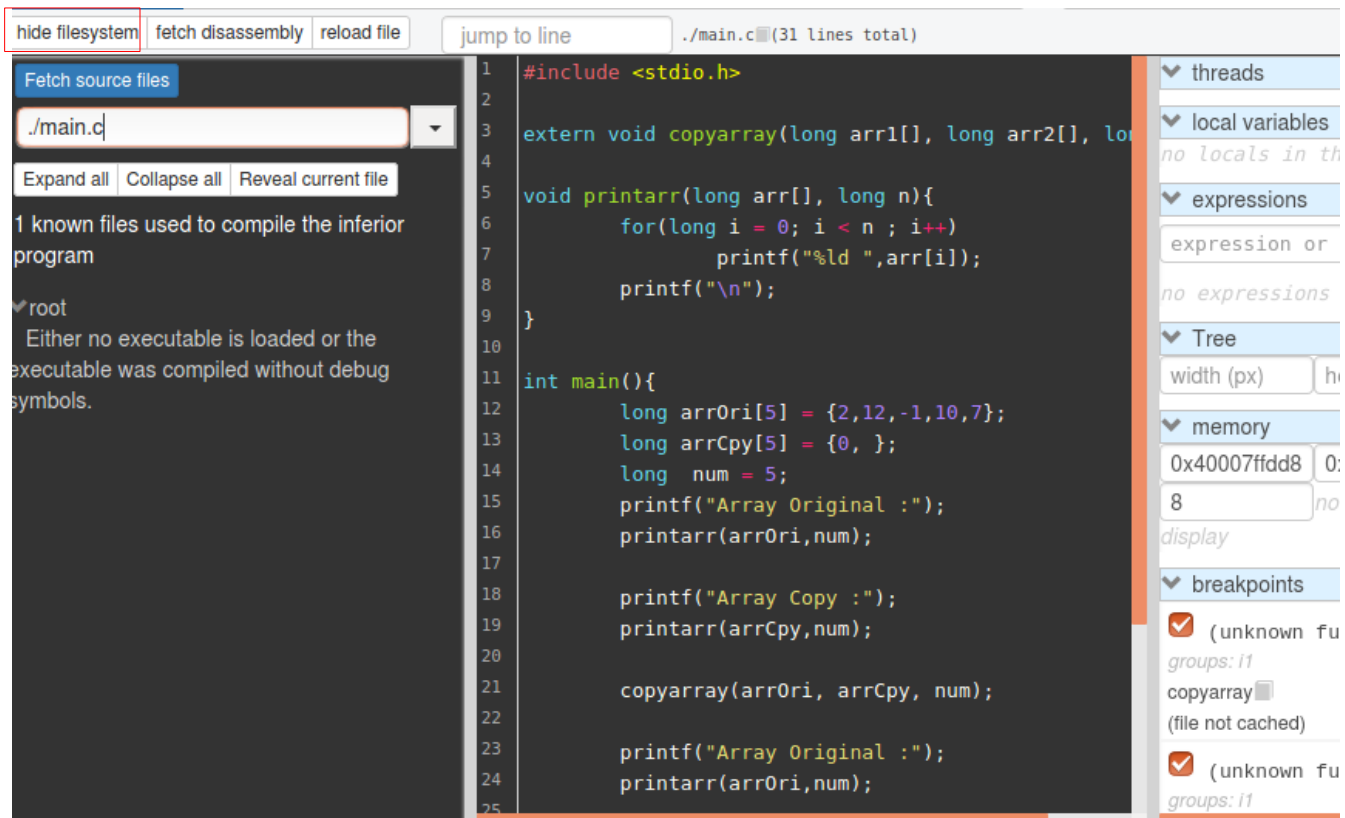
```
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 7
adio2000@ubuntu:~/assembly64/lab0$
```

- Qemu에서 lab0를 다시 디버그하고 싶다면 아래 명령어 재수행
\$ qemu-aarch64 -g 8080 ./lab0

예제1. 실습 과정

□ 혹시 소스코드가 보고 싶을 경우

다른 터미널을 열어서 vi 로 보거나, 아래와 같이 gdbgui의 위쪽 왼편의 show filesystem 탭을 열어 경로를 입력하면 됨. 이후, breakpoint 설정하고 c 로 진행시키면 다시 어셈블리 코드로 복귀함



예제2. 실습 목표

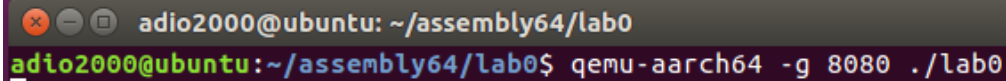
□ gdb에서는 실행 중간에 레지스터 값을 임의로 바꿀 수 있음

□ 레지스터 값을 임의로 변경 후 수행 결과를 확인해보자

- copyarray 함수를 호출할 때 \$X2 레지스터에는 세 번째 인자인 num 값이 저장된다. (num == 5)
- copyarray 함수 시작 부분에서 \$X2 레지스터 값을 1 감소시키고 계속해서 수행한다.
- 출력 결과가 어떻게 변하는지 확인한다.

예제2. 실습 과정

- 프로그램 다시 수행하기
 - ▣ 실행파일을 qemu로 재동작시킨다



```
adio2000@ubuntu: ~/assembly64/lab0
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
```

예제2. 실습 과정

- GDB를 Qemu와 연결하기
 - ▣ gdbgui를 실행하고 file 불러오기 및 qemu와의 연결을 시도

```
(gdb) file ./lab0
```

```
(gdb) target remote:8080
```

예제2. 실습 과정

- breakpoint 설정하기
 - copyarray 함수에 breakpoint를 설정하고 계속 실행한다
 - (gdb) b copyarray
 - (gdb) c

예제2. 실습 과정

□ 함수 인자로 넘어온 레지스터 값 확인하기

▣ 현재 레지스터 값 확인한다.

x0: arrOri 배열의 시작 주소
x1: arrCpy 배열의 시작 주소
x2: num (5)

▼ registers		
name	value (hex)	value (decimal)
x0	0x40007ffdd8	274886295000
x1	0x40007ffe00	274886295040
x2	0x5	5

x0, x1, x2 레지스터에는 **copyarray** 함수를 호출할 때 입력한 인자들이 저장되어있음.

예제2. 실습 과정

- 레지스터 값 변경 후 프로그램 출력 결과 확인하기
 - ▣ num 값이 저장되어 있는 \$x2 레지스터의 값을 1감소 시키고 결과를 확인한다.
 - ▣ (gdb) set \$x2 = 4

✓ registers			
name	value (hex)	value (decimal)	descr
x0	0x40007ffdd8	274886295000	
x1	0x40007ffe00	274886295040	
x2	0x4	4	

\$x2 레지스터 값이 5 => 4 로 변경됨

예제2. 실습 과정

- 레지스터 값 변경 후 프로그램 출력 결과 확인하기
 - ▣ 프로그램을 끝까지 수행하고 출력값을 확인한다.

(gdb) c

```
Breakpoint 1, 0x000000000040066c in copyarray ()
set $x2 = 4
set $x2 = 4
c
c
Continuing.
[Inferior 1 (Remote target) exited normally]
(gdb) |enter gdb command. To interrupt inferior, send SIGINT.
```

```
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :2 12 -1 10 7
Array Copy :2 12 -1 10 0
adio2000@ubuntu:~/assembly64/lab0$
```

기존 **num**에 해당하는 5개 값을 복사한 것이 아닌, 1이 감소된 4개 값을 복사했다.

예제3. 실습 목표

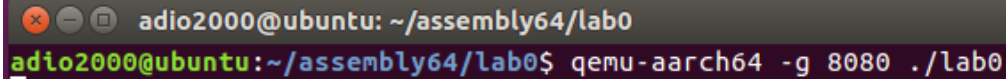
□ gdb에서는 실행 중간에 메모리 값을 임의로 바꿀 수 있음

□ 메모리 값을 임의로 변경 후 수행 결과를 확인해보자

- copyarray 함수를 호출할 때 \$x0 레지스터에는 첫 번째 인자인 arrOri 배열의 시작 주소값이 저장된다.
- 프로그램 수행 도중에 \$x0 레지스터가 가리키는 메모리 영역에 접근해 배열의 첫 번째 값을 변경하고 계속해서 수행한다.
- 출력 결과가 어떻게 변하는지 확인한다.

예제3. 실습 과정

- 프로그램 다시 수행하기
 - ▣ 실행파일을 qemu로 재동작시킨다



```
adio2000@ubuntu: ~/assembly64/lab0
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
```

A terminal window with a dark background. The title bar shows the user 'adio2000' on 'ubuntu' at the directory '~/assembly64/lab0'. The prompt is 'adio2000@ubuntu:~/assembly64/lab0\$'. The command entered is 'qemu-aarch64 -g 8080 ./lab0'. A cursor is visible on the line following the command.

예제3. 실습 과정

- GDB를 Qemu와 연결하기
 - ▣ gdbgui를 실행하고 file 불러오기 및 qemu와의 연결을 시도

```
(gdb) file ./lab0
```

```
(gdb) target remote:8080
```

예제3. 실습 과정

- breakpoint 설정하기
 - ▣ copyarray 함수에 breakpoint를 설정하고 계속 실행한다
 - ▣ (gdb) b copyarray
 - ▣ (gdb) c

예제3. 실습 과정

□ 메모리 값 변경 후 프로그램 출력 결과 확인하기

- ▣ \$x0 레지스터에 저장된 주소는 arrOri 배열의 시작주소를 나타낸다.
- ▣ \$x0 레지스터 값(빨간 네모)을 클릭해 arrOri 배열의 값들을 확인한다.

▼ registers		
name	value (hex)	value (decimal)
x0	0x40007ffdd8	274886295000
x1	0x40007ffe00	274886295040
x2	0x5	5

0x40007ffdd8	02 00 00 00 00 00 00 00
0x40007ffde0	0c 00 00 00 00 00 00 00
0x40007ffde8	ff ff ff ff ff ff ff ff
0x40007ffdf0	0a 00 00 00 00 00 00 00
0x40007ffdf8	07 00 00 00 00 00 00 00

2

12

-1

10

7

예제3. 실습 과정

- 메모리 값 변경 후 프로그램 출력 결과 확인하기
 - ▣ arrOri 배열의 메모리 영역에 접근해 첫 번째 값을 변경한다.
 - ▣ (gdb) set {long} 0x40007ffdd8 = 1
(타입 입력시 중괄호 주의!!)

```
0x4006cc ldr tx1, [x29, #24]      print arr+32
b copyarray
Note: breakpoint 1 also set at pc 0x40066c.
Breakpoint 2 at 0x40066c
/c
/c
Continuing.

Breakpoint 1, 0x000000000040066c in copyarray ()
set {long} 0x40007ffdd8 = 1
set {long} 0x40007ffdd8 = 1
(gdb) set {long} 0x40007ffdd8 = 1
```

address	hex	
more		arrOri[0] 값이 바뀐 것을 확인 (2->1로)
0x40007ffdd8	01 00 00 00 00 00 00 00	1
0x40007ffde0	0c 00 00 00 00 00 00 00	12
0x40007ffde8	ff ff ff ff ff ff ff ff	-1
0x40007ffdf0	0a 00 00 00 00 00 00 00	10
0x40007ffdf8	07 00 00 00 00 00 00 00	7
0x40007ffe00	00 00 00 00 00 00 00 00	
0x40007ffe08	00 00 00 00 00 00 00 00	

예제3. 실습 과정

- 메모리 값 변경 후 프로그램 출력 결과 확인하기
 - ▣ 프로그램을 끝까지 수행하고 출력값을 확인한다.

(gdb) c

```
adio2000@ubuntu:~/assembly64/lab0$ qemu-aarch64 -g 8080 ./lab0
Array Original :2 12 -1 10 7
Array Copy :0 0 0 0 0
Array Original :1 12 -1 10 7
Array Copy :1 12 -1 10 7
adio2000@ubuntu:~/assembly64/lab0$
```

arrOri 배열의 첫 번째 요소 값이 2임에도 arry Copy 에는 1로 변경되어 복사된 것을 볼 수 있다.
(첫줄에 **arrOri**의 첫번째 항목이 2로 찍힌 것은 1로 바뀌기전에 프린트 되었기 때문)

부록 - GDB 명령어

- gdbgui 환경이 아닌 터미널에서 gdb를 사용할 경우 다양한 명령어들의 숙지가 필요하다.
- 어셈블리 프로그램 뿐만 아니라 다른 고급 프로그램 디버깅시 매우 유용하다.

```
desktop-yoonjoon@ubuntu:~/ca_lab/ex1$ gdb-multiarch
GNU gdb (Ubuntu 7.11.1-0ubuntu1-16.5) 7.11.1
Copyright (C) 2016 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 "x86_64-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".
(gdb) █
```


```
(gdb) disass main
Dump of assembler code for function main:
0x00010660 <+0>:  push    {r11, lr}
0x00010664 <+4>:  add     r11, sp, #4
0x00010668 <+8>:  sub     sp, sp, #48      ; 0x30
0x0001066c <+12>: ldr     r3, [pc, #224]    ; 0x10754 <main+244>
0x00010670 <+16>: ldr     r3, [r3]
0x00010674 <+20>: str     r3, [r11, #-8]
0x00010678 <+24>: ldr     r3, [pc, #216]    ; 0x10758 <main+248>
0x0001067c <+28>: sub     r12, r11, #48     ; 0x30
0x00010680 <+32>: mov     lr, r3
0x00010684 <+36>: ldm     lr!, {r0, r1, r2, r3}
0x00010688 <+40>: stmla   r12!, {r0, r1, r2, r3}
0x0001068c <+44>: ldr     r3, [lr]
```

```
(gdb) info reg
r0             0xf6ffefb4      -150999116
r1             0xf6ffefc8      -150999096
r2             0x5             5
r3             0xf6ffefb4      -150999116
r4             0xf6ffef8       -150999048
r5             0x10eb8         69304
r6             0x0             0
r7             0x0             0
r8             0x0             0
r9             0x0             0
r10            0x96f6c         618348
r11            0xf6ffefe4      -150999068
r12            0x0             0
cn             0xf6ffefba      0xf6ffefba
```

```
(gdb) x/14l 0x00105b4
=> 0x105b4 <copy_array>:  sub     sp, sp, #8
0x105b8 <copy_array+4>:  str     r3, [sp]
0x105bc <copy_array+8>:  str     r4, [sp, #4]
0x105c0 <copy_array+12>: mov     r4, #0
0x105c4 <cwhile>:        cmp     r2, r4
0x105c8 <cwhile+4>:      ble     0x105dc <cexit>
0x105cc <cwhile+8>:      ldr     r3, [r0, r4, lsl #2]
0x105d0 <cwhile+12>:     str     r3, [r1, r4, lsl #2]
0x105d4 <cwhile+16>:     add     r4, r4, #1
0x105d8 <cwhile+20>:     b       0x105c4 <cwhile>
0x105dc <cexit>:         ldr     r4, [sp, #4]
0x105e0 <cexit+4>:       ldr     r3, [sp]
0x105e4 <cexit+8>:      add     sp, sp, #8
```

부록 - GDB 명령어

□ 기본적인 GDB 명령어

명령어	기능
c (Continue) 	프로그램 수행
k (Kill)	프로그램 수행 종료
s (Step) 	현재 행 수행 후 정지, 함수 호출시 함수 안으로 들어감 (s #: #번 연속 수행)
n (Next) 	현재 행 수행 후 정지, 함수 호출시 함수 수행 다음 행으로 이동 (n #: #번 연속 수행)
si (Step Instruction)	어셈블리 명령어 단위의 수행 (진행은 Step과 같음)
ni (Next Instruction)	어셈블리 명령어 단위의 수행 (진행은 Next와 같음)
disass 함수이름	특정 함수를 disassemble
list	소스코드 출력

부록 - GDB 명령어

□ 기본적인 GDB 명령어

명령어	기능
b* [function]	특정 함수에 breakpoint 설정
b* [address]	특정 주소에 breakpoint 설정
x/[n]x [\$register]	특정 레지스터로부터 n워드 만큼 출력
x/[n]x 주소	특정 주소로부터 n 워드 만큼 출력
info reg	현재 레지스터 정보를 출력
set {type} [address] = [value]	특정 메모리 값 변경
set [\$register] = [value]	특정 레지스터 값 변경
cl	브레이크 포인트 지우기
d	모든 브레이크 포인트 지우기

부록 - GDB 명령어

- 명령어 참고 블로그

<https://mintnlatte.tistory.com/581>

- GDB의 사용법에 대해 더 알아보고 싶다면, 아래의 링크 (GDB wiki 참고)

<http://www.gnu.org/software/gdb/documentation/>

- GDB를 사용한 디버깅 예시를 알고 싶다면 아래의 링크 (GNU KOREA 참조)

<http://korea.gnu.org/manual/release/gdb/gdb.html>

부록 - ARM Assembly Directive

- 의사 명령어 혹은 지시어(Directive Language)란 어셈블러에게 지시를 내리는 문장이다. 예를 들어 copyarray 코드에서 .global, .type, .text, .end 등을 말한다. 지시어는 기계어로 변환되지 않고 단지 개발자가 어셈블리어 프로그래밍하는 것을 도와준다.
- 지시어는 컴파일러마다 문법이 다르며 다음에 나오는 예제에서는 gcc를 기준으로 설명한다.

부록 - ARM Assembly Directive

예제

```
1: .include "config.inc" @#include
2: .text @RO-Data
3: .code 32 @16=Thumb, 32=Arm
4: .extern _main @extern
5: .global _start @global label
6: .equ REG_SYSTEM, 0x20000000 @#define
7:
8: _start: @_start 레이블
9:  MOV r0,REG_SYSTEM @define 상수 이용.
10:  BL _func1 @Branch Command
11:  ....
12: _func1:
13:  ....
14:  MOV pc,lr @return
15: MSG: .ascii "ARM Assembly Guide",0 @ascii 상수
16: VAR1: .byte 0x42,'A' @byte형 상수
17:  .end @end directive
```


부록 - ARM Assembly Directive

- **including (예제 소스 line no. 1)**
.include는 C언어에서 #include 지시자와 같은 역할입니다. 다른 파일을 load합니다. 주의할 점은 include하는 파일안에 파일의 끝을 뜻하는 .end 지시자가 사용되면 그 이후 모든 내용은 무시되기 때문에, include하는 파일에는 .end를 사용하지 않아야 합니다.
- **프로그램 영역 선언 (예제 소스 line no. 2)**
GCC에서 영역은 .text 영역과 .data영역으로 구분됩니다. .text영역은 program code나 상수 등 프로그램의 RO-Data가 들어가는 영역이라고 생각하시면 됩니다. .data영역은 변수 데이터, 즉 프로그램의 RW-Data에 들어가는 영역입니다.
- **ARM/THUMB 모드 (예제 소스 line no. 3)**
.code는 데이터가 16bit인지 32bit인지 설정하는 지시어입니다. 16이면 Thumb모드를 뜻하고, 32이면 Arm 모드를 뜻합니다.

부록 - ARM Assembly Directive

- **extern/global 선언 (예제 소스 line no. 4~5)**
.extern은 C언어와 마찬가지로 외부에 선언된 객체(label)을 해당 파일에서 사용할 때 사용합니다. .global은 해당 파일에서 선언된 label을 다른 외부 파일에서 사용할 수 있도록 해줍니다.
- **Pre-Procssing 상수 (예제 소스 line no. 6~7)**
.equ는 C언어에서 전처리 상수(#define)같은 개념입니다. 임베디드 시스템에서 어셈블리로 코딩할 때 자주 사용되니 꼭 알아 두세요.
- **레이블 선언 (예제 소스 line no. 8,12)**
어셈블리 코딩은 label 또는 명령으로 구분할 수 있습니다. label은 특정 메모리 주소를 가리키는 지표가 되며, 해당 라인에 레이블명 앞에 빈칸이 없이 [레이블 이름:] 으로 선언할 수 있습니다. 명령은 앞에 빈칸이 있어야 합니다. 즉, 해당 라인 앞에 빈칸이 없으면 레이블이고, 빈칸이 있으면 명령으로 해석하시면 되겠습니다. 레이블은 C언어에서 변수, 상수, 함수의 역할을 한다고 보시면 됩니다. C언어를 컴파일하면 각각의 변수, 상수, 함수가 각각의 주소를 갖는 label로 변환됩니다.

부록 - ARM Assembly Directive

□ 데이터 쓰기 (예제소스 line no. 15~16)

1byte를 Write할 때는 .byte, 2byte는 .hword, 4byte는 .word를 사용합니다. (32bit 시스템으로 가정) 문자열일 경우 .ascii를 사용하고, 끝에 Null을 뜻하는 0를 넣어줘야 합니다. .ascii보다 더 편리한 .asciz 지시자는 null을 자동으로 넣어줍니다.

또한, 콤마(,)를 이용해서 연속으로 데이터를 선언할 수 있습니다.

□ End directive (예제소스 line no. 17)

.end는 해당 파일의 끝을 뜻합니다. 이 지시자 뒤에 나오는 모든 내용은 무시 됩니다.

□ 참조

<https://m.blog.naver.com/PostView.nhn?blogId=gangst11&logNo=145839687&proxyReferer=https%3A%2F%2Fwww.google.com%2F>