实践项目 P1：交叉编译和跨平台应用仿真

杨浩然 10195501441

操作系统： ubuntu 20.04

2021-10-4

在 Ubuntu 上安装能够运行 aarch64 (64-bit ARM ISA)应用的 Qemu 虚拟机（qemu-aarch64）

安装之前可以先看看 https://wiki.archlinux.org/title/QEMU_(%E7%AE%80%E4%BD%93%E4%B8%AD%E6%96%87)

粗略地说，QEMU 有两种运行模式：

- **全系统模拟模式 (full-system emulation)**

  在该模式下, QEMU将会模拟一个完整的系统，包含一个或多个处理器以及各种外围设备。这种模式更加贴近真实的系统，且这种模式不要求被模拟的客户机系统是Linux，但它的速度较慢。

  QEMU中启用full-system模式的命令依照如下规则进行命名 `qemu-system-*目标机器架构*`, 例如 `qemu-system-x86_64` 用于模拟64位intel架构CPU, `qemu-system-i386` 模拟32位intel架构CPU, `qemu-system-arm` 模拟ARM架构(32 位), `qemu-system-aarch64` 模拟ARM架构(64位), 等等。

  如果模拟的CPU架构与宿主机的CPU架构相同, 那么即使在此模式下，QEMU仍有可能使用hypervisor(例如KVM or Xen)的技术对模拟机进行加速。

- **用户模式(Usermode emulation)**

  在此模式下, QEMU能够利用宿主机的系统资源来调用为其他架构编译的Linux可执行文件。当然, 里面依旧存在一些小问题, 比如说一些功能特性没有被实现, 采用动态链接的可执行文件无法直接在上面使用并且只支持Linux程序。

  QEMU中启用user模式的命令依照如下规则进行命名 `qemu-*目标机器架构*`, 例如 `qemu-x86_64` 用于模拟intel 64位的CPU。

最初我是安装的 full-system emulation，主要有几个不好的地方：

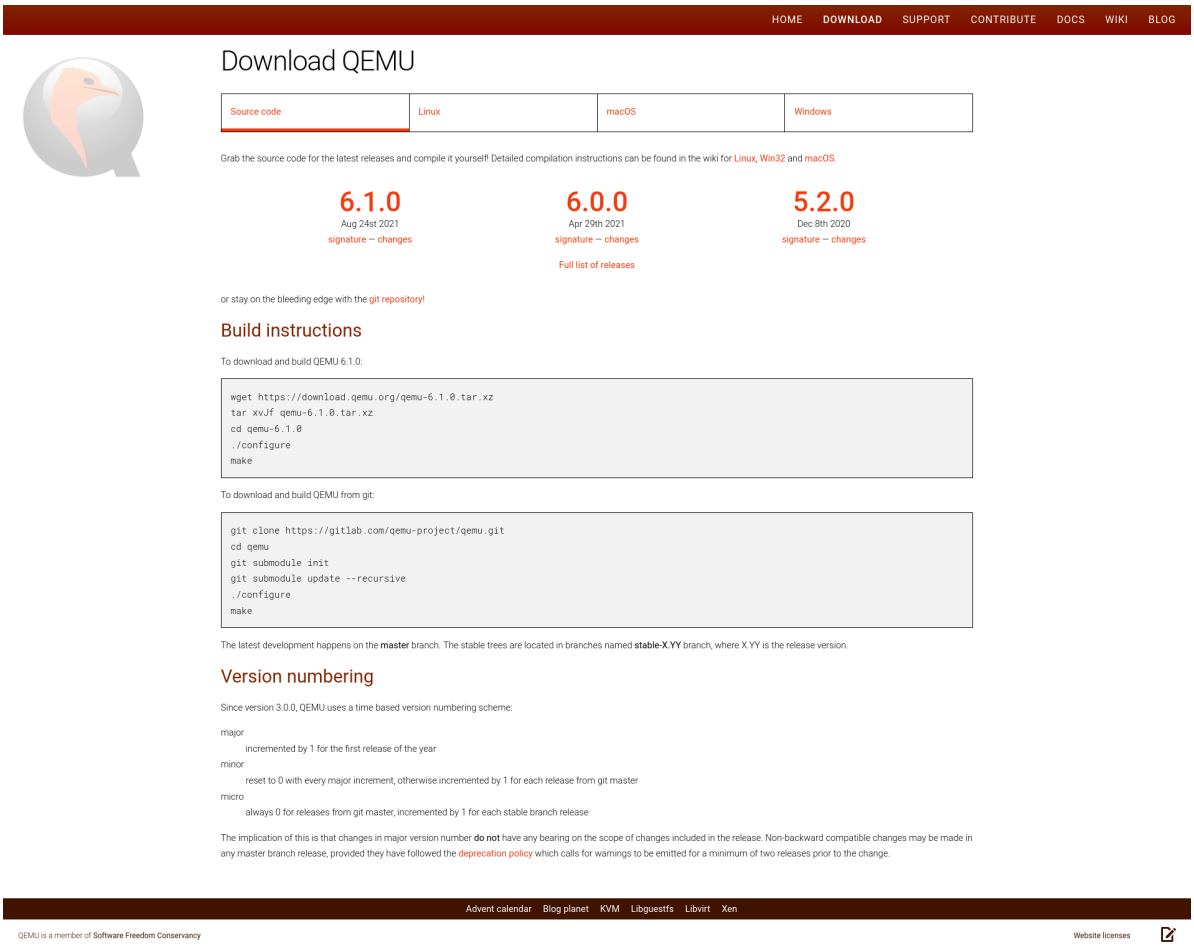
1. 需要不停安装相关的依赖，耗时比较严重。
2. 编译要注意控制选项，即 `make` 的时候要加上与 arrch64 相关的参数，避免编译其他无关的功能。
3. 编译好 QEMU 后，还需要下载 ARM 架构系统的系统镜像，在 QEMU 中进行安装。
   安装系统也是一件需要技术的事情，以一个技术博客给出的安装命令为例：

```
qemu-system-aarch64 -nographic -machine virt,gic-version=max -m 512M -cpu max
-smp 4 \
-netdev user,id=vnet,hostfwd=:127.0.0.1:0-:22 -device virtio-net-
pci,netdev=vnet \
-drive file=ubuntu-image.img,if=none,id=drive0,cache=writeback -device
virtio-blk,drive=drive0,bootindex=0 \
-drive file=flash0.img,format=raw,if=pflash -drive
file=flash1.img,format=raw,if=pflash
```

参数实在是太多，要么严格按照别人博客的步骤安装（大多数博客相当过时，安装的虚拟机操作系统很过时），要么把过程完全搞懂，另辟蹊径。
我尝试过安装 ARM 架构的 CentOS，无法完成安装过程。我又尝试了安装 ARM 架构的 Ubuntu 服务器版，安装的时间极长，为了把编译好的 loop.aarch64.gcc 传入虚拟机也是大费周折，也没有成功。
最后还是放弃了从源码编译 QEMU 的 full-system emulation 的想法（我真的尝试了很多方法，搜索了不计其数的资料），转而投向通过 apt-get 安装 QEMU 的 usermode emulation。

(未采用)基于源码安装

QEMU 的官网给出了通过 source code 安装的方法(https://www.qemu.org/download/#source)



在安装 qemu 的过程中，一些依赖需要被安装：

- re2c
- Ninja

安装 re2c

re2c 是 Ninja 的依赖，所以首先安装 re2c。

1. 首先在 re2c 的 github 仓库中下载 (https://github.com/skvadrik/re2c/releases/tag/2.2)
2. 解压后通过 `sudo ./configure && make && make install` 进行安装 (https://re2c.org/build/build.html)
   注意一定要使用 `sudo`，否则在 `make && make install` 中会出现权限不足而无法执行的情况。

3. 最后可以用 `make check` 检查一下 re2c 是否完全安装，如下图。

```
(base) younghojan@younghojan-XPS-15-7590:~/下载/re2c-2.2$ make check
make  check-recursive
make[1]: 进入目录“/home/younghojan/下载/re2c-2.2”
Making check in .
make[2]: 进入目录“/home/younghojan/下载/re2c-2.2”
Reconfigure with --enable-docs to rebuild docs
make  re2c_test_range re2c_test_s_to_n32_unsafe re2c_test_ver_to_vernum re2c_test_argsubst
make[3]: 进入目录“/home/younghojan/下载/re2c-2.2”
  CXX      src/test/range/test.o
  CXXLD    re2c_test_range
  CXX      src/test/s_to_n32_unsafe/test.o
  CXXLD    re2c_test_s_to_n32_unsafe
  CXX      src/test/ver_to_vernum/test.o
  CXXLD    re2c_test_ver_to_vernum
  CXX      src/test/argsubst/test.o
  CXXLD    re2c_test_argsubst
make[3]: 离开目录“/home/younghojan/下载/re2c-2.2”
make  check-TESTS
make[3]: 进入目录“/home/younghojan/下载/re2c-2.2”
make[4]: 进入目录“/home/younghojan/下载/re2c-2.2”
PASS: run_tests.py
PASS: re2c_test_range
PASS: re2c_test_s_to_n32_unsafe
PASS: re2c_test_ver_to_vernum
PASS: re2c_test_argsubst
============================================================================
Testsuite summary for re2c 2.2
============================================================================
# TOTAL: 5
# PASS:  5
# SKIP:  0
# XFAIL: 0
# FAIL:  0
# XPASS: 0
# ERROR: 0
============================================================================
make[4]: 离开目录“/home/younghojan/下载/re2c-2.2”
make[3]: 离开目录“/home/younghojan/下载/re2c-2.2”
make[2]: 离开目录“/home/younghojan/下载/re2c-2.2”
make[1]: 离开目录“/home/younghojan/下载/re2c-2.2”
```

使用 `sudo apt-get install re2c` 也是可以的。

安装 Ninja

~~访问 Ninja 官网，参考通过 source code 安装的方法：~~

```
git clone git://github.com/ninja-build/ninja.git && cd ninja
git checkout release
cat README.mdhttps://www.qemu.org/download/#source
```

~~README.md 中提供了两种安装 Ninja 的方法：通过 Python 或 CMake。~~

~~使用 Python 安装 Ninja，需要 bootstrap 和 re2c(这也是为什么要先安装 re2c)。~~

~~执行 `./configure.py --bootstrap` 以安装。~~

使用上面的方法安装 Ninja，在安装 qemu 时仍然会报错 Cannot find Ninja。

Ubuntu 系统下通过 `sudo apt-get install ninja-build` 可以成功安装。

安装其他依赖

```
sudo apt-get install libglib2.0-dev
sudo apt-get install libpixman-1-dev
```

安装 qemu

在 qemu 的官网可以找到安装方法(https://www.qemu.org/download/#source):

```
wget https://download.qemu.org/qemu-6.1.0.tar.xz    # 若下载速度太慢，也可使用浏览器下载
tar xvJf qemu-6.1.0.tar.xz
cd qemu-6.1.0
./configure
make
```

`make` 引起了 9518 项编译操作，这个时候可以去喝一杯咖啡☕

```
  AS      multiboot.o
  BUILD   multiboot.img
  BUILD   multiboot.raw
  SIGN    multiboot.bin
  AS      linuxboot.o
  BUILD   linuxboot.img
  BUILD   linuxboot.raw
  SIGN    linuxboot.bin
  CC      linuxboot_dma.o
  BUILD   linuxboot_dma.img
  BUILD   linuxboot_dma.raw
  SIGN    linuxboot_dma.bin
  AS      kvmvapic.o
  BUILD   kvmvapic.img
  BUILD   kvmvapic.raw
  SIGN    kvmvapic.bin
  AS      pvh.o
  CC      pvh_main.o
  BUILD   pvh.img
  BUILD   pvh.raw
  SIGN    pvh.bin
make[1]: 离开目录“/home/younghojan/下载/qemu-6.1.0/build”
changing dir to build for make ""...
make[1]: 进入目录“/home/younghojan/下载/qemu-6.1.0/build”
[1/177] Generating qemu-version.h with a meson_exe.py custom command
[2/35] Generating QAPI test (include) with a custom command
make[1]: 离开目录“/home/younghojan/下载/qemu-6.1.0/build”
```

经过漫长的等待，终于编译好了.....

虚拟机使用 ARM 架构的 Ubuntu 20.04.3 LTS

主要参考这篇博客中的内容:

> https://futurewei-cloud.github.io/ARM-Datacenter/qemu/how-to-launch-aarch64-vm/

启动后如下图:

```
younghojan1129@ubuntu1129:~$ uname -a
Linux ubuntu1129 4.15.0-159-generic #167-Ubuntu SMP Mon Sep 20 23:06:52 UTC 2021 aarch64 aarch64 aarch64 GNU/Linux
younghojan1129@ubuntu1129:~$
```

使用 apt-get 安装

安装的方法极其简单，只需在 terminal 输入:

```
sudo apt-get install qemu
sudo apt-get install qemu-system
sudo apt-get install qemu-user
```

安装 aarch64 的 GCC 工具链(gcc-10-aarch64-linux-gnu)

在 terminal 中执行 `sudo apt-get install gcc-10-aarch64-linux-gnu` 即可安装。

```
younghojan@younghojan-XPS-15-7590:~/下载/qemu-6.1.0$ sudo apt-get install gcc-10-aarch64-linux-gnu
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  binutils-arm-linux-gnueabi cpp-10-arm-linux-gnueabi
  gcc-10-arm-linux-gnueabi-base libasan6-armel-cross libatomic1-armel-cross
  libc6-armel-cross libc6-dev-armel-cross libgcc-10-dev-armel-cross
  libgcc-s1-armel-cross libgomp1-armel-cross libstdc++6-armel-cross
  libubsan1-armel-cross linux-libc-dev-armel-cross
使用'sudo apt autoremove'来卸载它(它们)。
将会同时安装下列软件：
  binutils-aarch64-linux-gnu cpp-10-aarch64-linux-gnu
  gcc-10-aarch64-linux-gnu-base libasan6-arm64-cross libatomic1-arm64-cross
  libc6-arm64-cross libc6-dev-arm64-cross libgcc-10-dev-arm64-cross
  libgcc-s1-arm64-cross libgomp1-arm64-cross libitm1-arm64-cross
  liblsan0-arm64-cross libstdc++6-arm64-cross libtsan0-arm64-cross
  libubsan1-arm64-cross linux-libc-dev-arm64-cross
建议安装：
  binutils-doc gcc-10-locales gcc-10-doc
下列【新】软件包将被安装：
  binutils-aarch64-linux-gnu cpp-10-aarch64-linux-gnu gcc-10-aarch64-linux-gnu
  gcc-10-aarch64-linux-gnu-base libasan6-arm64-cross libatomic1-arm64-cross
  libc6-arm64-cross libc6-dev-arm64-cross libgcc-10-dev-arm64-cross
  libgcc-s1-arm64-cross libgomp1-arm64-cross libitm1-arm64-cross
  liblsan0-arm64-cross libstdc++6-arm64-cross libtsan0-arm64-cross
  libubsan1-arm64-cross linux-libc-dev-arm64-cross
升级了 0 个软件包，新安装了 17 个软件包，要卸载 0 个软件包，有 93 个软件包未被升级。
需要下载 38.1 MB 的归档。
解压缩后会消耗 143 MB 的额外空间。
您希望继续执行吗？ [Y/n] Y
```

用 `aarch64-linux-gnu-gcc-10 -v` 验证安装：

```
younghojan@younghojan-XPS-15-7590:~/下载/qemu-6.1.0$ aarch64-linux-gnu-gcc-10 -v
Using built-in specs.
COLLECT_GCC=aarch64-linux-gnu-gcc-10
COLLECT_LTO_WRAPPER=/usr/lib/gcc-cross/aarch64-linux-gnu/10/lto-wrapper
Target: aarch64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 10.3.0-1ubuntu1~20.04' --with-bugurl=file:///usr/share/doc/gcc-10/README.Bugs --enable-languages=c,ada,c++,go,d,fortran,objc,obj-c++,m2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-10 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-libquadmath --disable-libquadmath-support --enable-plugin --enable-default-pie --with-system-zlib --enable-libphobos-checking=release --without-target-system-zlib --enable-multiarch --enable-fix-cortex-a53-843419 --disable-werror --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=aarch64-linux-gnu --program-prefix=aarch64-linux-gnu- --includedir=/usr/aarch64-linux-gnu/include --with-build-config=bootstrap-lto-lean --enable-link-mutex
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 10.3.0 (Ubuntu 10.3.0-1ubuntu1~20.04)
```

用 aarch64 的 GCC 工具链交叉编译 loop.c (-O2)，生成可执行文件 loop.aarch64.gcc，并用 qemu-aarch64 运行 loop.aarch64.gcc

编译 loop.c

编译的命令为

```
aarch64-linux-gnu-gcc-10 -O2 -o loop.aarch64.gcc loop.c
```

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ aarch64-linux-gnu-gcc-10 -O2 -o loop.aarch64.gcc loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ls
fasttime.h  loop.aarch64.gcc  loop.c
```

可以看到，成功生成了文件 loop.aarch64.gcc

用 qemu-aarch64 运行 loop.aarch64.gcc

在终端输入：

```
qemu-aarch64 loop.aarch64.gcc 50
```

首先有两个事情要注意：

1. 运行 loop.aarch64.gcc 时需要带一个大于 0 且小于等于 100 的参数
2. 不要输错字。我一直把 aarch64 写成 arrch64 还不自知，浪费了很多时间。

当然，这样运行是出不来结果的：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc
/lib/ld-linux-aarch64.so.1: No such file or directory
```

提示没有 /lib/ld-linux-aarch64.so.1 这个文件，这说明我们缺少这个动态链接库。这就是 usermode emulation 一个不好的地方了。解决方法有两个：

1. 在编译 loop.c 时使用静态编译，即加上参数 `-staic`。

   ```
   younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ aarch64-linux-gnu-gcc-10 -static -O2 -o loop.aarch64.gcc loop.c
   younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 50
   Elapsed execution time: 15.157701 sec; N: 1024, I: 5000000, _OP_: +, _TYPE_: uint32_t
   ```

   这样显然是可以成功运行程序的。

2. 既然缺少动态链接库，那么在对应的地方补上就可以了。下载 ld-linux-aarch64.so.1 并将其保存到 /lib；下载 libc.so.6 并将其保存到 /lib64。

   ```
   younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ aarch64-linux-gnu-gcc-10 -O2 -o loop.aarch64.gcc loop.c
   younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 50
   Elapsed execution time: 16.798154 sec; N: 1024, I: 5000000, _OP_: +, _TYPE_: uint32_t
   ```

   可以成功编译并且运行。

用 clang 交叉编译 loop.c(-O2)，生成可执行文件 loop.aarch64.clang，并用 qemu-aarch64 运行 loop.aarch64.clang

编译 loop.c

参考：https://llvm.org/docs/HowToCrossCompileLLVM.html

> In addition to the ones above, you'll also need:
>
> - `'-target arm-linux-gnueabihf'` or whatever is the triple of your cross GCC.
> - `'--sysroot=/usr/arm-linux-gnueabihf'`, `'--sysroot=/opt/gcc/arm-linux-gnueabihf'` or whatever is the location of your GCC's sysroot (where /lib, /bin etc are).
> - Appropriate use of `-I` and `-L`, depending on how the cross GCC is installed, and where are the libraries and headers.

编译命令为：

```
clang -static -target aarch64-linux-gnu -O2 -o loop.aarch64.clang loop.c --
sysroot=/usr/aarch64-linux-gnu
```

用 qemu-aarch64 运行 loop.aarch64.clang

上述采用静态编译，运行结果如下：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ clang -target aarch64-linux-gnu -O2 -o loop.aarch64.clang loop.c --sysroot=/usr/aarch64-linux-
gnu
/usr/bin/aarch64-linux-gnu-ld: 找不到 /usr/aarch64-linux-gnu/lib/libc.so.6 于 /usr/aarch64-linux-gnu 内部
/usr/bin/aarch64-linux-gnu-ld: 找不到 /usr/aarch64-linux-gnu/lib/libc_nonshared.a 于 /usr/aarch64-linux-gnu 内部
/usr/bin/aarch64-linux-gnu-ld: 找不到 /usr/aarch64-linux-gnu/lib/ld-linux-aarch64.so.1 于 /usr/aarch64-linux-gnu 内部
clang: error: linker command failed with exit code 1 (use -v to see invocation)
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ clang -static -target aarch64-linux-gnu -O2 -o loop.aarch64.clang loop.c --sysroot=/usr/aarch6
4-linux-gnu
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 50
Elapsed execution time: 39.015724 sec; N: 1024, I: 5000000, _OP_: +, _TYPE_: uint32_t
```

为什么我可以在 x86_64 上运行 ARM 程序?

这是随手一试的发现，我可以在 Ubuntu（而非 qemu-aarch64）中运行交叉编译的 loop.aarch64.gcc!

过程如下：

1. 我用 aarch64-linux-gnu-gcc-10 和 gcc 分别编译出 loop.aarch64.gcc 和 loop.x86_64.gcc，并且使用 `file` 命令分别查看两个文件。

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ aarch64-linux-gnu-gcc-10 -O2 -o loop.aarch64.gcc loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ gcc -O2 -o loop.x86_64.gcc loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ file loop.aarch64.gcc
loop.aarch64.gcc: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, BuildID[sha1]=2c5581a3f18d1dbaf7ce20722352fec4444607ca, for GNU/Linux 3.7.0, not stripped
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ file loop.x86_64.gcc
loop.x86_64.gcc: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=7193bc7881f8ab4cabf039c9c4caba776043c676, for GNU/Linux 3.2.0, not stripped
```

   编译的结果应该是正确的，loop.aarch64.gcc 显示是 ARM aarch64，loop.x86_64.gcc 显示是 x86-64。

2. 使用 qemu-aarch64 运行这两个文件。

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 10
Elapsed execution time: 3.166569 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.x86_64.gcc 10
loop.x86_64.gcc: Invalid ELF image for this architecture
```

   loop.aarch64.gcc 是可以正确运行并且输出运行结果的，而 loop.x86_64.gcc 输出 Invalid ELF image for this architecture，无法在 ARM aarch64 架构的 QEMU 虚拟机上执行。

3. 在 Ubuntu 中使用 `./` 运行这两个文件。

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.aarch64.gcc 10
Elapsed execution time: 3.076947 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x86_64.gcc 10
Elapsed execution time: 0.480844 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
```

   loop.aarch64.gcc 和 loop.x86_64.gcc 均可成功运行，区别是 loop.aarch64.gcc 的运行时间比 loop.x86_64.gcc 的运行时间多出一个数量级。

用 qemu-aarch64 分别运行前面编译出来的 loop.aarch64.gcc 和 loop.aarch64.clang（分别用参数 5、10、20、40、80 进行测试），记下每次测试的执行时间并以图形方式呈现

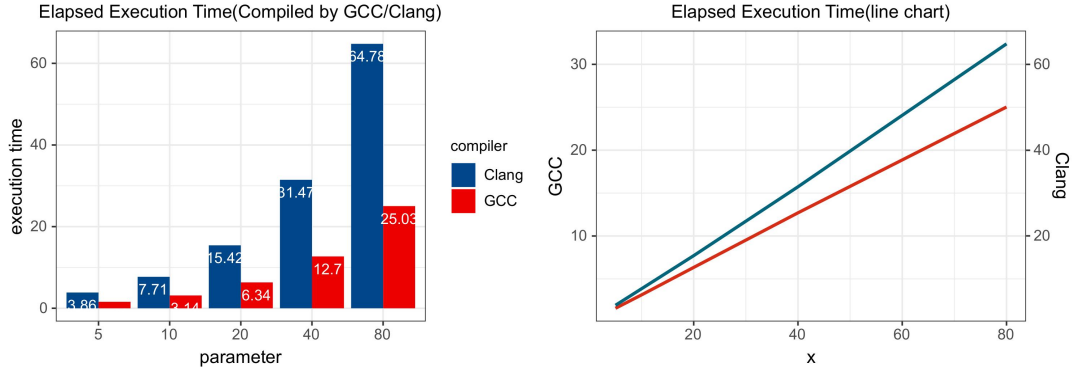
查看文件大小：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ls -l loop.aarch64.clang loop.aarch64.gcc
-rwxrwxrwx 1 younghojan younghojan 601016 10月 19 21:18 loop.aarch64.clang
-rwxrwxrwx 1 younghojan younghojan  13928 10月 19 21:26 loop.aarch64.gcc
```

运行结果如下：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 5
Elapsed execution time: 1.575265 sec; N: 1024, I: 500000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 10
Elapsed execution time: 3.137358 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 20
Elapsed execution time: 6.337606 sec; N: 1024, I: 2000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 40
Elapsed execution time: 12.703162 sec; N: 1024, I: 4000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc 80
Elapsed execution time: 25.026666 sec; N: 1024, I: 8000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 5
Elapsed execution time: 3.859032 sec; N: 1024, I: 500000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 10
Elapsed execution time: 7.713013 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 20
Elapsed execution time: 15.424696 sec; N: 1024, I: 2000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 40
Elapsed execution time: 31.465448 sec; N: 1024, I: 4000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.clang 80
Elapsed execution time: 64.778084 sec; N: 1024, I: 8000000, __OP__: +, __TYPE__: uint32_t
```

可视化如下：



使用 GCC 工具链编译的程序，运行时间几乎小于 Clang 编译的程序的 1/2。

注意到， GCC 工具链编译的程序是动态编译的，而 Clang 编译的程序是静态编译的。按道理来说，动态链接库内的函数只有在实际使用到时才加载到内存，所以动态编译的程序运行应该比静态编译的程序慢，文件大小比静态编译的程序的小。

但是在本次执行之前，loop.aarch64.gcc 已经被执行过，也许相关的动态链接库已经被加载到内存中，之后再执行则加快了速度。

使用 GCC 工具链静态编译 loop.c（为了控制变量），再次进行测试：

```
aarch64-linux-gnu-gcc-10 -static -O2 -o loop.aarch64.gcc loop.c
```

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ aarch64-linux-gnu-gcc-10 -static -O2 -o loop.aarch64.gcc_static loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ file loop.aarch64.gcc_static
loop.aarch64.gcc_static: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=e7cc2602a29c480d781862e60c10207412d56ce7, for GNU/Linux 3.7.0, not stripped
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ls -l loop.aarch64.gcc_static
-rwxrwxrwx 1 younghojan younghojan 596744 10月 20 11:00 loop.aarch64.gcc_static
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ls -l loop.aarch64.clang
-rwxrwxrwx 1 younghojan younghojan 601016 10月 19 21:18 loop.aarch64.clang
```

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc_static 5
Elapsed execution time: 1.517827 sec; N: 1024, I: 500000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc_static 10
Elapsed execution time: 3.108130 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc_static 20
Elapsed execution time: 6.059263 sec; N: 1024, I: 2000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc_static 40
Elapsed execution time: 12.109701 sec; N: 1024, I: 4000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ qemu-aarch64 loop.aarch64.gcc_static 80
Elapsed execution time: 24.215998 sec; N: 1024, I: 8000000, __OP__: +, __TYPE__: uint32_t
```

运行时间略有提升。

用 host 机器上的 gcc 和 clang 分别编译(-O2)出 loop.x64.gcc 和 loop.x64.clang，并对这两个执行文件分别用参数 5、10、20、40、80 进行测试，记下每次测试的执行时间并以图形方式呈现，进而与前一步 qemu 仿真测试的结果进行比较

编译命令如下：

```
gcc -O2 -o loop.x64.gcc loop.c
clang -O2 -o loop.x64.clang loop.c
```

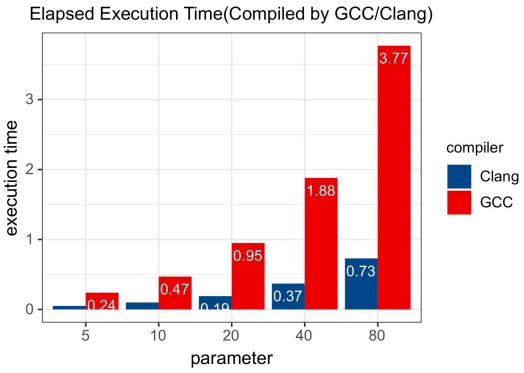
```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ gcc -O2 -o loop.x64.gcc loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ clang -O2 -o loop.x64.clang loop.c
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ file loop.x64.gcc
loop.x64.gcc: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=7193bc7881f8ab4cabf039c9c4caba776043c676, for GNU/Linux 3.2.0, not
 stripped
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ file loop.x64.clang
loop.x64.clang: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e991dcd848cb9572a2e323a152d719aaa29af609, for GNU/Linux 3.2.0, not
stripped
```

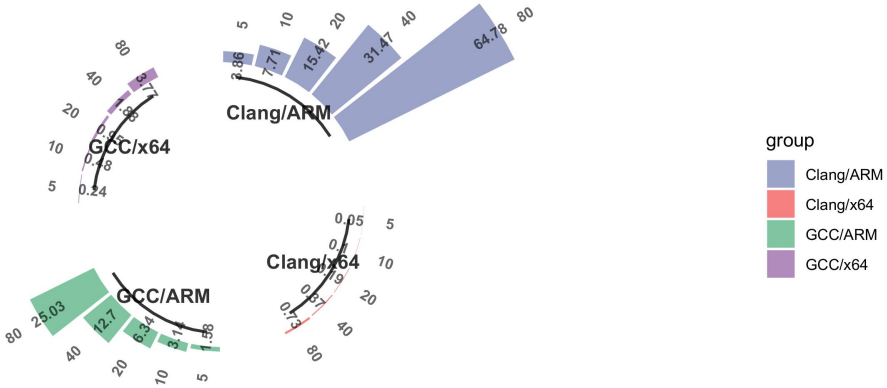
运行结果：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.gcc 5
Elapsed execution time: 0.241915 sec; N: 1024, I: 500000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.gcc 10
Elapsed execution time: 0.476765 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.gcc 20
Elapsed execution time: 0.952233 sec; N: 1024, I: 2000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.gcc 40
Elapsed execution time: 1.882462 sec; N: 1024, I: 4000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.gcc 80
Elapsed execution time: 3.769242 sec; N: 1024, I: 8000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.clang 5
Elapsed execution time: 0.050449 sec; N: 1024, I: 500000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.clang 10
Elapsed execution time: 0.096310 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.clang 20
Elapsed execution time: 0.189003 sec; N: 1024, I: 2000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.clang 40
Elapsed execution time: 0.366320 sec; N: 1024, I: 4000000, __OP__: +, __TYPE__: uint32_t
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ ./loop.x64.clang 80
Elapsed execution time: 0.728133 sec; N: 1024, I: 8000000, __OP__: +, __TYPE__: uint32_t
```

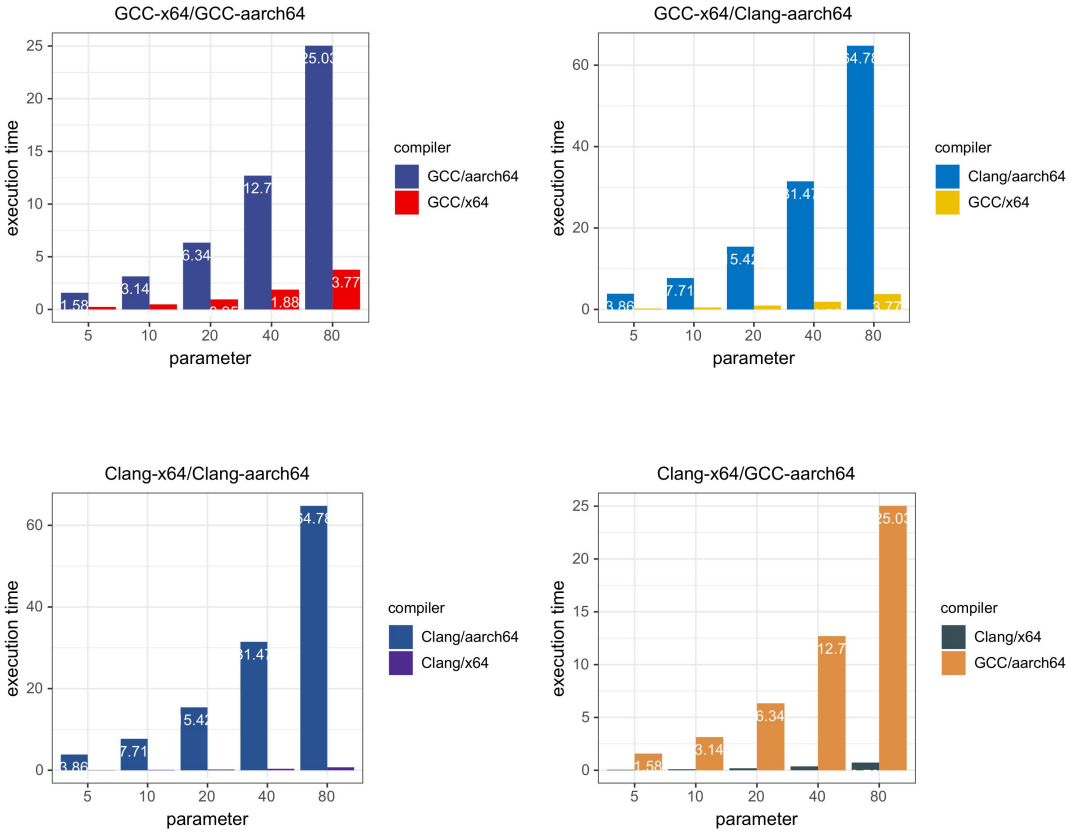
跟 QEMU 仿真测试的结果不是一个数量级的。

可视化如下：


Elapsed Execution Time(Compiled by GCC/Clang)
execution time
parameter
compiler
Clang
GCC
0.24
0.47
0.19
0.95
0.37
1.88
0.73
3.77


GCC/x64
Clang/ARM
GCC/ARM
Clang/x64
group
Clang/ARM
Clang/x64
GCC/ARM
GCC/x64

交叉对比，共有 4 种组合：{'GCC/x64, GCC/ARM', 'GCC/x64, Clang/ARM', 'Clang/x64, Clang/ARM', 'Clang/x64, GCC/ARM'}



安装支持多 ISA 的 gdb 调试器（gdb-multiarch）

在 terminal 中用 `apt-get install` 命令安装即可：

```
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ sudo apt-get install gdb-multiarch
[sudo] younghojan 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列【新】软件包将被安装：
  gdb-multiarch
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 38 个软件包未被升级。
需要下载 3,794 kB 的归档。
解压缩后会消耗 15.2 MB 的额外空间。
获取:1 http://cn.archive.ubuntu.com/ubuntu focal-updates/universe amd64 gdb-multiarch amd64 9.2-0ubuntu1~20.04 [3,794 kB]
已下载 3,794 kB，耗时 3秒 (1,323 kB/s)
正在选中未选择的软件包 gdb-multiarch。
(正在读取数据库 ... 系统当前共安装有 229050 个文件和目录。)
准备解压 .../gdb-multiarch_9.2-0ubuntu1~20.04_amd64.deb  ...
正在解压 gdb-multiarch (9.2-0ubuntu1~20.04) ...
正在设置 gdb-multiarch (9.2-0ubuntu1~20.04) ...
younghojan@younghojan-XPS-15-7590:/media/younghojan/Study/undergraduate/junior/SEM1/软件系统优化/project/p1/P1_loop$ gdb-multiarch
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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)
```

用 gdb-multiarch 结合 qemu-aarch64 对 loop.aarch64.gcc.debug 进行源码级调试

参考：https://www.starduster.me/2020/09/29/build-multiarch-environment-with-qemu/

1. 使用 GCC 工具链的 `-g` 编译选项对 loop.c 进行编译

```
aarch64-linux-gnu-gcc-10 -g -static -o loop.aarch64.gcc.debug loop.c
```

2. 使用 qemu-aarch64 在后台运行 loop.aarch64.gcc.debug 并且等待 gdb 连接 2333 端口（注意：参数添加应在此步）

```
qemu-aarch64 -g 2333 -L /usr/aarch64-linux-gnu ./loop.aarch64.gcc.debug 10 &
```

3. 启动 gdb-multiarch

```
gdb-multiarch ./loop.aarch64.gcc.debug
```

4. 在 gdb 中将架构切换为 aarch64

```
set arch aarch64
```

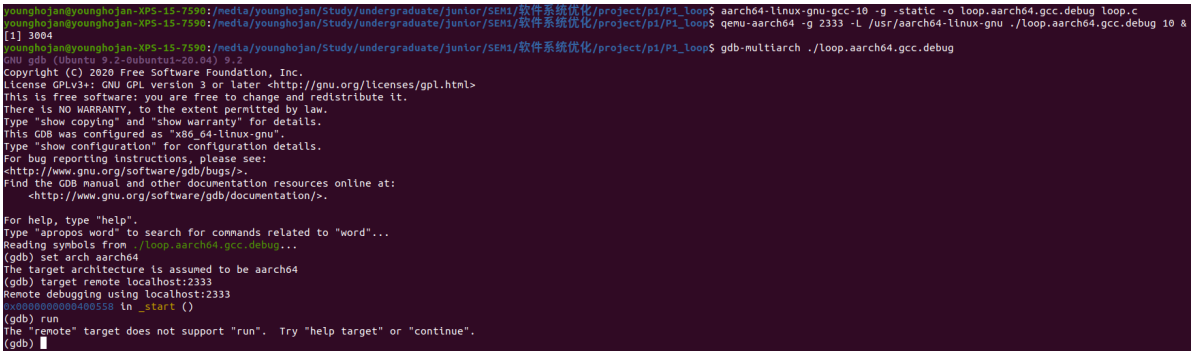
5. 连接到 localhost: 2333

```
target remote localhost:2333
```

6. 使用 `run` 执行

```
run
```

运行过程如图：



然而，remote target 的 gdb 无法使用 `run`，可以用 `continue` 替代。

```
(gdb) c
Continuing.
Elapsed execution time: 5.836881 sec; N: 1024, I: 1000000, __OP__: +, __TYPE__: uint32_t
[Inferior 1 (process 1) exited normally]
```

其他 gdb 相关的操作均可实现，只是无法使用 `run` 而已。

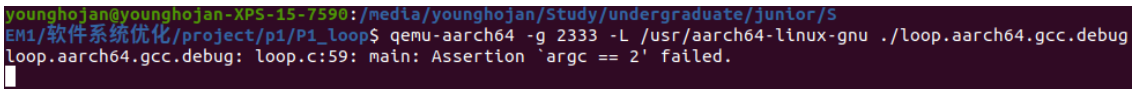
注意：这个部分的实践中的注意事项和遇见的困难主要是以下几个

1. No symbol table is loaded.

```
(gdb) b main
No symbol table is loaded.  Use the "file" command.
Make breakpoint pending on future shared library load? (y or [n])
```

   错误信息为没有加载符号表。最终在一篇博客 https://www.1024sky.cn/blog/article/19927 中无意间看见了别人的调试过程，照着试了一下，发现问题被解决了。解决方法，即在启动 gdb-multiarch 时加上参数 `./loop.aarch64.gcc.debug`。

2. loop.aarch64.gcc.debug: loop.c:59: main: Assertion `argc == 2' failed.



由于对 gdb 使用的不熟悉，这个问题困扰了我很久。

最初的解决方案是在 gdb 中使用 `set args <arg>` 来为调试过程指定参数，但是没有成功。正确的解决方案是在 qemu-aarch64 运行 loop.aarch64.gcc.debug 时加上参数（如 10），即 `qemu-aarch64 -g 2333 -L /usr/aarch64-linux-gnu ./loop.aarch64.gcc.debug 10 &`。

3. 在使用 GCC 工具链编译 loop.aarch64.gcc.debug 时要使用静态编译（动态编译似乎在调试的时候会有有点问题，忘记记录了）。

总结

至此，P1 的内容已经全部圆满完成了。回顾整个过程，遇到的困难已按时序在上文总结出来了。对于 P1 的大部分内容都是首次接触，没有经验可言，全是一个人慢慢摸索出来的，对于我这种小镇作题家还是有一定挑战，不过完成时的成就感给我带来的满足无可比拟。