为 SimpleKernel 添加 UEFI 启动支持

基本信息

- 个人信息

  姓名：葛岩

  学校：杭州电子科技大学

  专业：计算机技术

- 项目PR

  x86_64 的 UEFI 支持与对重写构建系统

  https://github.com/Simple-XX/SimpleKernel/pull/157

  x86_64 下的 UEFI 输出

  https://github.com/Simple-XX/SimpleKernel/pull/158

  x86_64 下内存信息的获取

  https://github.com/Simple-XX/SimpleKernel/pull/159

- 更多内容请看简历

项目详细方案

项目目标是将现有的 grub2-multiboot 方案替换为 uefi，为了实现此目标，将整个工作分为如下几步：

1. 分析现有代码，找出需要改动的部分
2. 研究 uefi 启动流程，规划要添加的代码框架
3. 将代码添加到内核，此阶段保留旧的引导方式
4. 测试-修复迭代
5. 删除旧的引导

以上所有内容均已完成，体现在上文提到的三个 PR 中。

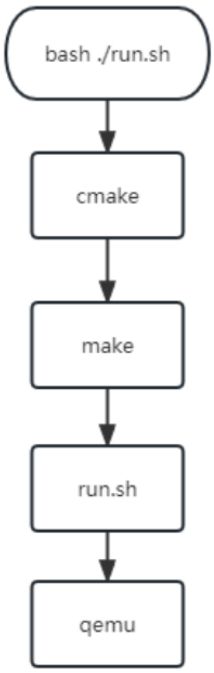
除以上五点，我还计划在 uefi 阶段做更多的工作，如对内存的初始化，这样在进入内核时就能够使用 c++ 容器，全局对象的构造也可以更加灵活。

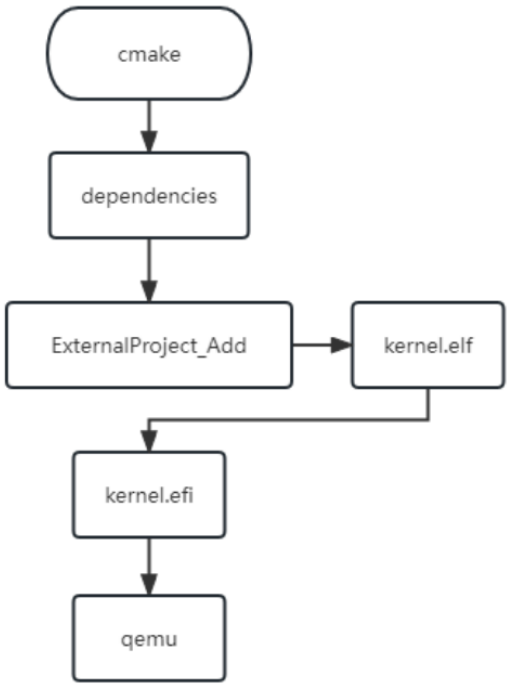
下面我将描述我的思路。

构建系统重构

可以看到，旧的启动流程需要运行 run.sh，设置并传递环境变量给 cmake 进行生成，编译好内核后控制流再从 make 返回 run.sh，进入 qemu。

该过程逻辑复杂不够简洁，因此我做出了右侧的改进。





改进后的构建系统，只需要运行 cmake 即可完成对环境变量的设置，内核的生成，qemu 的模拟等操作，所有的指令以 `make xxx` 的形式提供，无需有 run.sh 与 env.sh，大大简化了流程。

此外，我还对 i386/x86_64 做出了取舍，将 i386 相关代码整合到 x86_64 中，这一举措也减少了很多为了区分 32/64 的冗余代码。

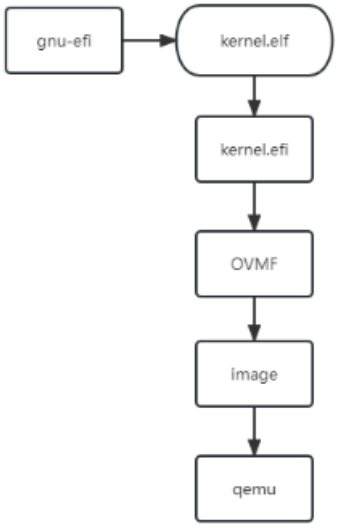
UEFI 引导

得益于构建系统的改进，引入 UEFI 引导也变得容易。

首先将 gnu-efi 进行编译，将其二进制文件编入内核中，得到 kernek.elf，再使用 objcopy 命令将内核转换为 efi 格式，这样就获得了目标内核。

下一步是再 qemu 上运行起来，这里将 OVMF 与 kernel.efi 一起写入到一个 fatfs 中，将这个 fatfs 传递给 qemu 即可。

```
# 制作内核
# 将内核调整为 efi
add_custom_target(kernel_efi DEPENDS kernel)
add_custom_command(TARGET kernel_efi
        WORKING_DIRECTORY ${CMAKE_RUNTIME_OUTPUT_DIRECTORY}
        COMMENT "Gen kernel.efi"
        COMMAND ${CMAKE_OBJCOPY} --target=efi-app-x86-64 ${KernelName}
kernel.efi
        -g -R .comment -R .gnu_debuglink -R .note.gnu.build-id
        -R .gnu.hash -R .plt -R .rela.plt -R .dynstr -R .dynsym -R .rela.dyn
        -S -R .eh_frame -R .gcc_except_table
        )
```



启动信息的处理

UEFI 启动后，会将相关参数传递过来，其入口原型为：

```
extern "C" EFI_STATUS
efi_main(EFI_HANDLE image, EFI_SYSTEM_TABLE* systemTable);
```

类似的，在初始化完成后，将参数交给内核：

```
void kernel_main(void* _systemtable);
```

```
/**
 * @brief 内核主要逻辑
 * @note 这个函数不会返回
 */
void kernel_main(void* _systemtable) {
#if defined(__x86_64__)
    EFI_SYSTEM_TABLE* systemTable = (EFI_SYSTEM_TABLE*)_systemtable;
    EFI_STATUS             Status;
    EFI_MEMORY_DESCRIPTOR* EfiMemoryMap;
    ptrdiff_t              EfiMemoryMapSize;
    ptrdiff_t              EfiMapKey;
    ptrdiff_t              EfiDescriptorSize;
    UINT32                 EfiDescriptorVersion;

    //
    // Get the EFI memory map.
    //
    EfiMemoryMapSize = 0;
    EfiMemoryMap     = NULL;
    Status = uefi_call_wrapper(systemTable->BootServices->GetMemoryMap, 5,
                               &EfiMemoryMapSize, EfiMemoryMap, &EfiMapKey,
                               &EfiDescriptorSize, &EfiDescriptorVersion);
    ASSERT(Status == EFI_BUFFER_TOO_SMALL);

    //
    // Use size returned for the AllocatePool.
    //
    EfiMemoryMap
```

```
      = (EFI_MEMORY_DESCRIPTOR*)AllocatePool(EfiMemoryMapSize
                                             + 2 * EfiDescriptorSize);
    ASSERT(EfiMemoryMap != NULL);
    Status = uefi_call_wrapper(systemTable->BootServices->GetMemoryMap, 5,
                               &EfiMemoryMapSize, EfiMemoryMap, &EfiMapKey,
                               &EfiDescriptorSize, &EfiDescriptorVersion);
    if (EFI_ERROR(Status)) {
        FreePool(EfiMemoryMap);
    }

    //
    // Get descriptors
    //
    EFI_MEMORY_DESCRIPTOR* EfiEntry = EfiMemoryMap;
    do {
        // ... do something with EfiEntry ...
        EfiEntry = NextMemoryDescriptor(EfiEntry, EfiDescriptorSize);

    } while ((UINT8*)EfiEntry < (UINT8*)EfiMemoryMap + EfiMemoryMapSize);
#endif
    return;
}
```

这样，我们就获得了内存信息。

基于 UEFI 的调试输出

可直接借用 UEFI 提供的接口实现，但要注意 utf 字符编码的问题，相关设置如下

```
# 通用编译选项
set(CMAKE_C_FLAGS
        "${CMAKE_C_FLAGS} \
-Wall -Wextra \
-no-pie -nostdlib \
-fPIC -ffreestanding -fexceptions -fshort-wchar \
-DGNU_EFI_USE_MS_ABI")

# 架构相关编译选项
if (ARCH STREQUAL "riscv64")
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -march=rv64imafdc")
elseif (ARCH STREQUAL "x86_64")
    set(CMAKE_C_FLAGS
            "${CMAKE_C_FLAGS} \
-march=corei7 -mtune=corei7 -m64 -mno-red-zone \
-z max-page-size=0x1000 \
-Wl,-shared -Wl,-Bsymbolic")
elseif (ARCH STREQUAL "aarch64")
    set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -march=armv8-a -mtune=cortex-a72")
endif ()
```

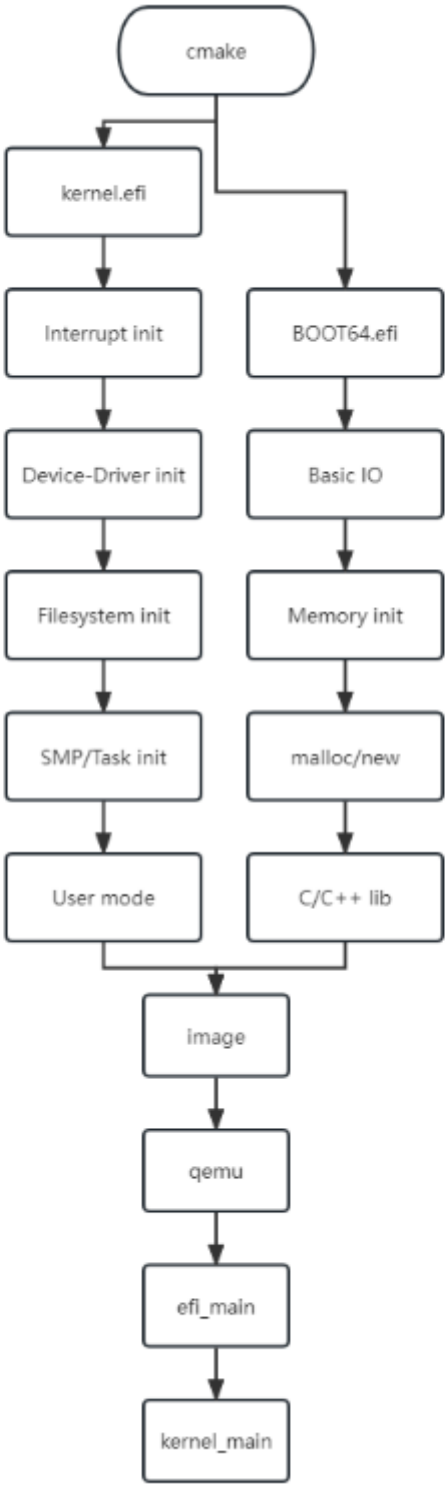
在编译时，需要添加 `-fshort-wchar` 以支持 utf 编码，`-Wl,-shared -Wl,-Bsymbolic` 则用于定位链接脚本中的 `ImageBase` 变量。

```
...
/* 指定输出架构 */
OUTPUT_ARCH(i386:x86-64)
/* 设置入口点 */
ENTRY(_start)
/* 设置各个 section */
SECTIONS {
    ImageBase = 0;
    ...
}
```

```
/**
 * @brief 内核主要逻辑
 * @note 这个函数不会返回
 */
void kernel_main(void* _systemtable) {
#if defined(__x86_64__)
    EFI_SYSTEM_TABLE* systemTable = (EFI_SYSTEM_TABLE*)_systemtable;
    EFI_STATUS        status
      = uefi_call_wrapper(systemTable->ConOut->OutputString, 2,
                          systemTable->ConOut, L"Hello UEFI111!\n");
#endif
    return;
}
```

更多工作

1. UEFI 阶段能做的事不只是将控制权转移给内核，至少应该支持虚拟内存与堆，提供兼容 c/c++ 标准的内存分配函数。
2. 基于上面的成果，我们可以将 bootloader 与内核分离，编译出 BOOT64.efi 与 kernel.efi 两个文件，实现功能解耦。




```
# 引导程序
ExternalProject_Add(
        boot
        BUILD_ALWAYS 1
        PREFIX ${CMAKE_BINARY_DIR}
        SOURCE_DIR ${CMAKE_SOURCE_DIR}/src/boot
        BINARY_DIR ${CMAKE_BINARY_DIR}/src/boot
        CMAKE_ARGS
        -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
        -DCMAKE_VERBOSE_MAKEFILE=${CMAKE_VERBOSE_MAKEFILE}
        -
DCMAKE_TOOLCHAIN_FILE=${CMAKE_SOURCE_DIR}/cmake/${ARCH}-${CMAKE_HOST_SYSTEM_
PROCESSOR}.cmake
```

```
        -DMACHINE=${MACHINE}
        -DARCH=${ARCH}
        -DCMAKE_C_COMPILER_WORKS=${CMAKE_C_COMPILER_WORKS}
        -DCMAKE_CXX_COMPILER_WORKS=${CMAKE_CXX_COMPILER_WORKS}
        -
DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=${CMAKE_FIND_ROOT_PATH_MODE_PROGRAM}
        -
DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=${CMAKE_FIND_ROOT_PATH_MODE_LIBRARY}
        -
DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=${CMAKE_FIND_ROOT_PATH_MODE_INCLUDE}
        -
DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=${CMAKE_FIND_ROOT_PATH_MODE_PACKAGE}
        -DCMAKE_C_STANDARD=${CMAKE_C_STANDARD}
        -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
        -DCMAKE_C_FLAGS=${CMAKE_C_FLAGS}
        -DCMAKE_CXX_FLAGS=${CMAKE_CXX_FLAGS}
        -DCMAKE_ASM_FLAGS=${CMAKE_ASM_FLAGS}
        -DCMAKE_ARCHIVE_OUTPUT_DIRECTORY=${CMAKE_ARCHIVE_OUTPUT_DIRECTORY}
        -DCMAKE_LIBRARY_OUTPUT_DIRECTORY=${CMAKE_LIBRARY_OUTPUT_DIRECTORY}
        -DCMAKE_RUNTIME_OUTPUT_DIRECTORY=${CMAKE_RUNTIME_OUTPUT_DIRECTORY}
        -DCMAKE_MODULE_PATH=${CMAKE_MODULE_PATH}
        -DKernelName=${KernelName}.boot
        BUILD_COMMAND ${GENERATOR_COMMAND}
        INSTALL_COMMAND ""
)

# 内核本体
ExternalProject_Add(
        kernel
        BUILD_ALWAYS 1
        PREFIX ${CMAKE_BINARY_DIR}
        SOURCE_DIR ${CMAKE_SOURCE_DIR}/src
        BINARY_DIR ${CMAKE_BINARY_DIR}/src
        CMAKE_ARGS
        -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
        -DCMAKE_VERBOSE_MAKEFILE=${CMAKE_VERBOSE_MAKEFILE}
        -
DCMAKE_TOOLCHAIN_FILE=${CMAKE_SOURCE_DIR}/cmake/${ARCH}-${CMAKE_HOST_SYSTEM_
PROCESSOR}.cmake
        -DMACHINE=${MACHINE}
        -DARCH=${ARCH}
        -DCMAKE_C_COMPILER_WORKS=${CMAKE_C_COMPILER_WORKS}
        -DCMAKE_CXX_COMPILER_WORKS=${CMAKE_CXX_COMPILER_WORKS}
        -
DCMAKE_FIND_ROOT_PATH_MODE_PROGRAM=${CMAKE_FIND_ROOT_PATH_MODE_PROGRAM}
        -
DCMAKE_FIND_ROOT_PATH_MODE_LIBRARY=${CMAKE_FIND_ROOT_PATH_MODE_LIBRARY}
        -
DCMAKE_FIND_ROOT_PATH_MODE_INCLUDE=${CMAKE_FIND_ROOT_PATH_MODE_INCLUDE}
        -
DCMAKE_FIND_ROOT_PATH_MODE_PACKAGE=${CMAKE_FIND_ROOT_PATH_MODE_PACKAGE}
        -DCMAKE_C_STANDARD=${CMAKE_C_STANDARD}
        -DCMAKE_CXX_STANDARD=${CMAKE_CXX_STANDARD}
        -DCMAKE_C_FLAGS=${CMAKE_C_FLAGS}
        -DCMAKE_CXX_FLAGS=${CMAKE_CXX_FLAGS}
        -DCMAKE_ASM_FLAGS=${CMAKE_ASM_FLAGS}
        -DCMAKE_ARCHIVE_OUTPUT_DIRECTORY=${CMAKE_ARCHIVE_OUTPUT_DIRECTORY}
        -DCMAKE_LIBRARY_OUTPUT_DIRECTORY=${CMAKE_LIBRARY_OUTPUT_DIRECTORY}
```

```
        -DCMAKE_RUNTIME_OUTPUT_DIRECTORY=${CMAKE_RUNTIME_OUTPUT_DIRECTORY}
        -DCMAKE_MODULE_PATH=${CMAKE_MODULE_PATH}
        -DKernelName=${KernelName}
        BUILD_COMMAND ${GENERATOR_COMMAND}
        INSTALL_COMMAND ""
)
```

这样我们在一个 cmake 中生成了两个二进制文件，只需要这两个文件放入 image/ 传递给 qemu 即可。

3. c/c++ 支持，目前的 c/c++ 支持是可以优化的，我计划在主要工作完成后尝试移植标准库。

   目前的 C/C++ 库是完全独立实现的，同时受到裸机制约，但实际上可以借助 -ffreestanding 参数，使用由 GCC 提供的可以直接在裸机环境使用的标准库，在下表列出。

	Headers required for a *freestanding* implementation
Types	`cstddef`
Implementation properties	`limits` `cfloat` `climits` (since C++11) `version` (since C++20)
Integer types	`cstdint` (since C++11)
Start and termination	`cstdlib` (partial)[1]
Dynamic memory management	`new`
Type identification	`typeinfo`
Source location	`source_location` (since C++20)
Exception handling	`exception`
Initializer lists	`initializer_list` (since C++11)
Comparisons	`compare` (since C++20)
Coroutines support	`coroutine` (since C++20)
Other runtime support	`cstdarg`
Fundamental library concepts	`concepts` (since C++20)
Type traits	`type_traits` (since C++11)
Bit manipulation	`bit` (since C++20)
Atomics	`atomic` (since C++11)[2]
Utility components	`utility` (since C++23)
Tuples	`tuple` (since C++23)
Memory	`memory` (since C++23) (partial)
Function objects	`functional` (since C++23) (partial)
Compile-time rational arithmetic	`ratio` (since C++23)
Iterators library	`iterator` (since C++23) (partial)
Ranges library	`ranges` (since C++23) (partial)
Deprecated headers	`ciso646` (until C++20) `cstdalign` (since C++11)(until C++20) `cstdbool` (since C++11)(until C++20)

目前 c++ 标准准备提供更多的 ffreestanding 头文件，甚至会有容器的支持，这一方式是符合发展进程的。

```
Notes
It's important to note that some compiler vendors may not fully support
freestanding implementation. For example, GCC libstdc++ has had
implementation and build issues before version 13, while LLVM libcxx and
MSVC STL do not support freestanding.

In C++23, many features will be made freestanding with partial headers.
However, it's still up for discussion in WG21 whether headers like <array>
will be made freestanding in the future standard (C++26, maybe). Regardless,
containers like vector, list, deque, and map will never be freestanding due
to their dependencies on exceptions and heap.

GCC 13 provides more headers, such as optional, span, array, and bitset, for
freestanding. However, it's important to note that these headers may not be
portable or provide the same experience as a hosted implementation. It's
better to avoid using them in a freestanding environment, even if the
toolchain provides them.
```

项目开发时间计划

日期	工作内容
06/26-06/30	与导师沟通
07/01-07/15	完成对构建系统的优化
07/15-07/31	完成 x86_64 平台的开发工作
08/01-08/15	完成 aarch64 平台的开发工作
08/15-08/31	boot 与 kernel 分离
09/01-09/15	在 boot 中做更多初始化工作
09/15-09/25	C/C++ 库支持
09/25-09/30	撰写结项报告