一、实验基础信息

个人信息

201700130011 — 刘建东 — 17级菁英班

实验信息

日期：2019.11.18

题目：NACHOS 的 $Makefiles$（实验二）

实验任务

该实验在目录 $lab2$ 中完成。

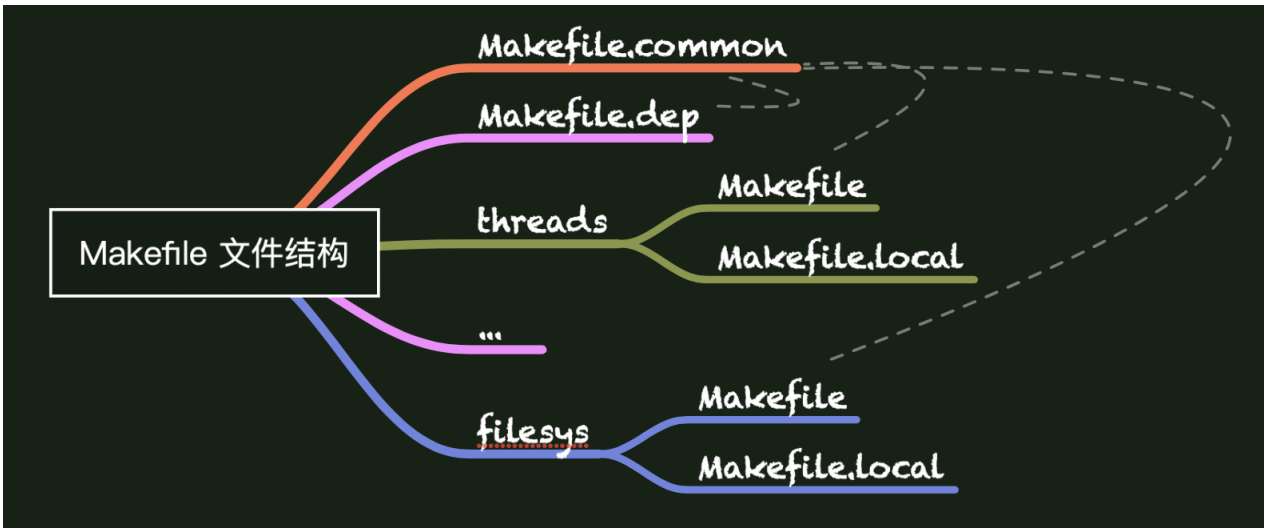
1. 熟悉 $Nachos$ 的 $makefiles$ 结构
2. 熟悉如何在几个 lab 文件目录中构造相应的 $Nachos$ 系统

二、实验基本方法

实验二主要是介绍了 $Nachos$ 系统中 $makefile$ 的一些细节，以及在其它目录中修改 $Nachos$ 代码并生成修改后的 $Nachos$ 系统的两种方法。

1. makefile 文件结构

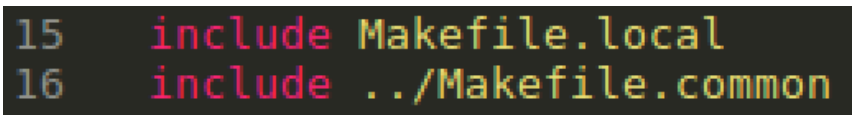
接下来我们需要了解一下 $Nachos$ 中 makefile 的结构。



从上图中我们可以看到，最外层文件夹中有 $.common$ 和 $.dep$ 两个文件，其中 $.common$ 文件包含了 $.dep$ 文件，可在 $.common$ 文件代码的第 44 行找到依据。

```
44  include ../Makefile.dep
```

而在内存文件夹中，则为 $Makefile$ 文件和 $.local$ 文件，其中 $Makefile$ 文件包含了 $.local$ 文件和外层文件夹中的 $.common$ 文件，可在其文件代码中找到依据。



接下来我们介绍一下上述文件结构中提到的 4 个文件。

2. Makefile 文件

子目录中 Makefile 文件的内容比较简单，主要就是如下两行，即包含了 Makefile.local 和 ../Makefile.common。

```
include Makefile.local
include ../Makefile.common
```

3. Makefile.local 文件

该文件的作用是对一些编译、链接以及运行时所使用的宏进行定义。

- CCFILES：指定在该目录下生成 $Nachos$ 时所涉及到的 C++ 源文件。
- INCPATH：指明所涉及的 C++ 源程序中的头文件（.h）所在的路径，以便利用 g++ 进行编译链接时通过这路径查找这些头文件。
- DEFINES：传递 g++ 的一些标号或者宏。

以下述的一个真实的 $.local$ 文件作为例子，其中 CCFILES 包含了所有 c++ 源文件，INCPATH 给定了查找 .h 文件时的搜索路径，DEFINES 则表示文件所定义的宏，可用于 .cc 文件中 #ifdef ... #endif 语句。

```
CCFILES = main.cc\
  list.cc\
  ...
  timer.cc

INCPATH += -I../threads -I../machine

DEFINES += -DTHREADS
```

除上述解释说明之外，这里还有一个细节需要注意，即 INCPATH 和 DEFINES 的赋值使用 += 号，表示将后续字符串直接添加到原字符串的尾部。

4. Makefile.dep 文件

该文件的主要功能是根据安装 $Nachos$ 时所使用的操作系统环境，定义一些相应的宏，供 g++ 使用。在其具体文件中，其主要用下述两个语句来实现相应操作系统的宏定义。

```
uname = $(shell uname)
ifeq($(uname),Linux)
```

由于我们使用的OS环境是Linux，因此我们主要研究一下Linux环境下定义的一些相应宏。

```
ifeq ($(uname),Linux)
HOST_LINUX=-linux
HOST = -DHOST_i386 -DHOST_LINUX
CPP=/lib/cpp
CPPFLAGS = $(INCDIR) -D HOST_i386 -D HOST_LINUX
arch = unknown-i386-linux
ifdef MAKEFILE_TEST
GCCDIR = /usr/local/mips/bin/decstation-ultrix-
LDFLAGS = -T script -N
ASFLAGS = -mips2
endif
endif
```

其中定义的 GCCDIR 表示 gcc mips 交叉编译器所在的路径及其前缀，这也是实验一中需要将交叉编译器装到 /usr/local/ 目录下的原因。

除了上述的一些约束之外，$.dep$ 文件还给出了 $.common$ 文件中需要使用的几个宏。

```
arch_dir = arch/$(arch)
obj_dir = $(arch_dir)/objects
bin_dir = $(arch_dir)/bin
depends_dir = $(arch_dir)/depends
```

其中 arch_dir 为 $Nachos$ 可执行程序对应的路径，obj_dir 表示目标文件对应路径，depends_dir 表示依赖文件对应路径。

5. Makefile.common 文件

这是 makefile 文件结构中最复杂的一个文件，其中定义了编译链接生成一个完整 $Nachos$ 可执行文件所需要的所有规则。

vpath

用 vpath 定义了一些编译时查找相关文件（.cc / .h / .s）的路径。

```
vpath %.cc  ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.h   ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.s   ../network:../filesys:../vm:../userprog:../threads:../machine
```

这里有一个细节，如果在当前目录下找不到相应的文件，编译器才会在 vpath 给出的路径中查找这些文件，因此编译器默认优先在当前文件夹中查找所需要的文件。

.s / .o / .cc 文件存放路径

下述代码告知了编译器 .s / .o / .cc 文件的存放路径。

```
s_ofiles = $(SFILES:%.s=$(obj_dir)/%.o)
c_ofiles = $(CFILES:%.c=$(obj_dir)/%.o)
cc_ofiles = $(CCFILES:%.cc=$(obj_dir)/%.o)
```

生成 nachos 可执行文件

下述代码为链接生成 $nachos$ 可执行文件，并在当前目录下建立一个指向该文件的符号链接文件 $nachos$ 的过程。

```
$(program): $(ofiles)
$(bin_dir)/% :                    # % 匹配任意的非空字符串
  @echo ">>> Linking" $@ "<<<"    # @ 指目标文件名（nachos）
  $(LD) $^ $(LDFLAGS) -o $@       # $^ 指所有依赖文件，$< 指第一个搜索到的依赖文件
  ln -sf $@ $(notdir $@)          # 在当前目录下建立文件
```

该文件的其余部分主要是 .o 文件和 .d 文件一些生成和存放相关的代码，此处不再赘述。

最后给出两个注意点。

1. g++ 参数 -MM 表示自动寻找源文件（.cc）直接或间接关联的头文件，并生成相应的依赖关系文件（.d）。

   这么做的目的是由 g++ 自动维护源文件与头文件之间的联系，不需要用户去指定，避免用户疏忽。

2. `include $(diles)` 将所有创建的 .d 文件包括进 makefile.common 中，目的是通知 g++，当 `$(diles)` 中任何一个 .d 文件所依赖的源文件以及头文件被修改后，.d 文件需要重新编译，相应的 .o 文件以及最终的可执行文件也都需要重新编译生成。

除了上述的makefile文件结构之外，在该实验中我们还用到了touch与grep的命令，下面稍微介绍一下这两种方法。

touch 命令

touch fileName，若 fileName 已存在，则将 fileName 的时间标签更新为系统当前的时间（默认方式），文件数据不做修改。若 fileName 不存在，则新建文件名为 fileName 的空文件。

grep 命令

grep 全称为 ”全面搜索正则表达式并把行打印出来“，是一个文本搜索工具，能够使用正则表达式搜索文件，并将匹配的行打印出来。

三、源代码及注释

实验二主要是熟悉和了解 $Nachos$ 系统 makefile 的文件结构，因此与实验一一样并不需要自己来实现代码，所以此处主要列出一部分与该实验相关的源代码。（Makefile.common 文件）

```
ifndef MAKEFILE_COMMON
define MAKEFILE_COMMON
yes
endef

include ../Makefile.dep

vpath %.cc  ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.h   ../network:../filesys:../vm:../userprog:../threads:../machine
vpath %.s   ../network:../filesys:../vm:../userprog:../threads:../machine

CFLAGS = -g -Wall -Wshadow $(INCPATH) $(DEFINES) $(HOST) -DCHANGED

s_ofiles = $(SFILES:%.s=$(obj_dir)/%.o)
c_ofiles = $(CFILES:%.c=$(obj_dir)/%.o)
cc_ofiles = $(CCFILES:%.cc=$(obj_dir)/%.o)

ofiles = $(cc_ofiles) $(c_ofiles) $(s_ofiles)

program = $(bin_dir)/nachos

$(program): $(ofiles)

$(bin_dir)/% :
  @echo ">>> Linking" $@ "<<<"
  $(LD) $^ $(LDFLAGS) -o $@
  ln -sf $@ $(notdir $@)

$(obj_dir)/%.o: %.cc
  @echo ">>> Compiling" $< "<<<"
  $(CC) $(CFLAGS) -c -o $@ $<

$(obj_dir)/%.o: %.c
  @echo ">>> Compiling" $< "<<<"
  $(CC) $(CFLAGS) -c -o $@ $<

$(obj_dir)/%.o: %.s
  @echo ">>> Assembling" $< "<<<"
  $(CPP) $(CPPFLAGS) $< > $(obj_dir)/tmp.s
  $(AS) -o $@ $(obj_dir)/tmp.s
  rm $(obj_dir)/tmp.s

s_dfiles = $(SFILES:%.s=$(depends_dir)/%.d)
c_dfiles = $(CFILES:%.c=$(depends_dir)/%.d)
cc_dfiles = $(CCFILES:%.cc=$(depends_dir)/%.d)
```

```
dfiles = $(cc_dfiles) $(c_dfiles) $(s_dfiles)

$(depends_dir)/%.d: %.cc
  @echo ">>> Building dependency file for " $< "<<<"
  @$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \
  | sed '\''s@$*.o[ ]*:@$(depends_dir)/$(notdir $@) $(obj_dir)/&@g'\'' > $@'

$(depends_dir)/%.d: %.c
  @echo ">>> Building dependency file for" $< "<<<"
  @$(SHELL) -ec '$(CC) -MM $(CFLAGS) $< \
  | sed '\''s@$*.o[ ]*:@$(depends_dir)/$(notdir $@) $(obj_dir)/&@g'\'' > $@'

$(depends_dir)/%.d: %.s
  @echo ">>> Building dependency file for" $< "<<<"
  @$(SHELL) -ec '$(CPP) -MM $(CPPFLAGS) $< \
  | sed '\''s@$*.o[ ]*:@$(depends_dir)/$(notdir $@) $(obj_dir)/&@g'\'' > $@'

include $(dfiles)

clean:
  rm -f `find $(arch_dir) -type f -print | egrep -v '(CVS|cvsignore)'`
  rm -f nachos coff2noff coff2flat
  rm -f *.noff *.flat

tmpclean:
  rm -f tmp*

endif # MAKEFILE_COMMON
```

四、实验测试方法及结果

1. 在其它目录中修改代码并生成修改后的 Nachos 系统

由于课设需要对源码进行修改，但是我们又需要保存最初的代码，使得改错时仍可以恢复，因此我们需要将要修改的文件复制到实验目录下，然后在这些目录下利用 make 命令生成相应的 Nachos 系统。

其主要步骤如下：

1. 将目录 code/threads 中的 code/threads/arch 及其子目录全部复制到目录 lab2 中，并将 arch/unknown-i386-linux/bin/、arch/unknown-i386-linux/depends/ 以及 arch/unknown-i386-linux/objects/ 清空。
2. 将 code/threads 中 Makefile 和 Makefile.local 复制到目录 lab2 中。
3. 修改 Makefile.local，使得可以正确编译修改后的 Nachos 系统。

由实验第二部分可知，Makefile.local 中主要包含了 CCFILES 和 INCPATH，其中make会根据 vpath 去寻找C++源程序，因此CCFILES不需要修改，即我们主要需要修改的是INCPATH文件。

2. 方法 1 修改 INCPATH 文件

在INCPATH中将目录 -I../lab2 添加到../threads之前，如下所示：

```
INCPATH += -I../lab2 -I../threads -I../machine
```

接下来在其它目录中生成nachos系统，查看修改之后的效果。

make前文件夹中的内容如下。

```
gene@ubuntu:~/Desktop/OS/code/lab2$ ls
arch  Makefile  Makefile.local  scheduler.cc  scheduler.h
```

make之后文件夹中的内容如下。

```
gene@ubuntu:~/Desktop/OS/code/lab2$ ls
arch  Makefile  Makefile.local  nachos  scheduler.cc  scheduler.h
```

touch scheduler.h 之后再次make，会重新编译，说明编译内容发生了变化，但仅只有scheduler.cc被重新编译了。

```
>>> Compiling scheduler.cc <<<
g++ -g -Wall -Wshadow -I../lab2 -I../threads -I../machine -DTHREADS -DHOST_i386
-DHOST_LINUX -DCHANGED -c -o arch/unknown-i386-linux/objects/scheduler.o schedul
er.cc
In file included from ../threads/list.h:17:0,
                 from scheduler.h:13,
                 from scheduler.cc:22:
```

touch ../threads/scheduler.h 之后其余文件均被重新编译。

```
>>> Compiling ../machine/interrupt.cc <<<
g++ -g -Wall -Wshadow -I../lab2 -I../threads -I../machine -DTHREADS -DHOST_i386
-DHOST_LINUX -DCHANGED -c -o arch/unknown-i386-linux/objects/interrupt.o ../mach
ine/interrupt.cc
In file included from ../threads/list.h:17:0,
                 from ../machine/interrupt.h:39,
                 from ../machine/interrupt.cc:24:
```

而这是因为在code目录makefile.common中，g++的参数-MM表示搜索与.cc文件相同目录下的.h文件，因此修改../threads/scheduler.h之后，再次编译main.cc文件时就会优先在当前目录中寻找scheduler.h，而由于scheduler.h修改过，main.cc文件也需要重新编译。

但我们希望当lab2/scheduler.h文件被修改后，所有间接或直接使用该文件的程序都重新编译，因此我们需要将../threads中直接或间接使用scheduler.h的文件都复制到lab2目录中。

grep 命令

我们使用grep命令来寻找包含scheduler.h字符串的文件。

```
gene@ubuntu:~/Desktop/OS/code/threads$ grep scheduler.h *
grep: arch: 是一个目录
匹配到二进制文件 nachos
scheduler.cc:#include "scheduler.h"
scheduler.h:// scheduler.h
system.h:#include "scheduler.h"
```

由于system.h中也包含scheduler.h，因此继续查看system.h的文件。

```
gene@ubuntu:~/Desktop/OS/code/threads$ grep system.h *
grep: arch: 是一个目录
main.cc:#include "system.h"
匹配到二进制文件 nachos
scheduler.cc:#include "system.h"
synch.cc:#include "system.h"
synchtest.cc:#include "system.h"
system.cc:#include "system.h"
system.h:// system.h
thread.cc:#include "system.h"
threadtest.cc:#include "system.h"
```

将上述所有涉及到的文件取一个并集，然后加入到lab2文件夹中，再次touch scheduler.h，然后make。

```
>>> Compiling main.cc <<<
g++ -g -Wall -Wshadow -I../lab2 -I../threads -I../machine -DTHREADS -DHOST_i386
-DHOST_LINUX -DCHANGED -c -o arch/unknown-i386-linux/objects/main.o main.cc
In file included from main.cc:55:0:
../threads/utility.h:106:8: warning: extra tokens at end of #endif directive [-W
endif-labels]
```

再次touch ../threads/scheduler.h，然后make，发现nachos不再重新编译。

```
gene@ubuntu:~/Desktop/OS/code/threads$ make
make: 'arch/unknown-i386-linux/bin/nachos' is up to date.
```

因此方法一修改INCPATH的方法，总结一下就是将所有与要修改文件有关的文件全部放到实验文件夹中。这种方式比较繁琐，因此我们介绍方法二，更简单地完成要求。

3. 方法 2 修改 INCPATH 文件

我们在INCPATH中加入-I-，表示编译时强制预处理程序从紧随-I-其后所给出的目录中查找源程序所使用的头文件，而不是默认优先查找.cc源程序所在的目录中的头文件。

```
INCPATH += -I- -I../lab2 -I../threads -I../machine
```

修改INCPATH之后再次touch ../threads/scheduler.h，查看效果。

```
gene@ubuntu:~/Desktop/OS/code/lab2$ touch ../threads/scheduler.h
gene@ubuntu:~/Desktop/OS/code/lab2$ make
make: 'arch/unknown-i386-linux/bin/nachos' is up to date.
```

再touch lab2/scheduler.h，查看效果。

```
gene@ubuntu:~/Desktop/OS/code/lab2$ touch scheduler.h
gene@ubuntu:~/Desktop/OS/code/lab2$ make
>>> Building dependency file for  ../machine/timer.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../machine/sysdep.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../machine/interrupt.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../threads/synchtest.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../threads/threadtest.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../threads/thread.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../threads/system.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
>>> Building dependency file for  ../threads/synch.cc <<<
cc1plus: note: obsolete option -I- used, please use -iquote instead
```

可以发现，其它与scheduler.h有关的文件均会重新编译，即使它们不在lab2文件夹中。

至此实验结束，可以发现方法2可以更简单有效的实现我们的需求，也是今后实验所推荐的方法。

五、实验体会

1. 此次实验一开始是对makefile的一个介绍，之后就主要是与INCPATH有关的内容。其中一共有两个重点，一个是g++的-MM参数，一个是INCPATH中-I-的参数。
2. 首先是g++的-MM参数，表示编译寻找.cc文件相关联的头文件时，默认优先从.cc文件所在文件夹中寻找，也正是因为这个原因，在方法1中INCPATH未添加-I-时，touch lab2/scheduler.h，然后再make。所有threads文件夹中与scheduler.h相关的文件均未重新编译，其原因在于这些文件在编译寻找头文件时，优先在当前文件夹中寻找，即优先寻找到了threads/scheduler.h文件，而该文件并未修改过，所以threads中的文件都未重新编译。

   而当用grep命令将所有包含scheduler.h的文件放入lab2文件夹后，touch lab2/scheduler.h再make，所有与之相关的.cc文件都重新编译了。因此它们编译时优先从当前文件夹中寻找头文件，而由于当前文件夹中的头文件修改过了，所以它们都重新编译了。
3. 也正是由于g++中-MM的参数，即使不加实验1中的-I../lab2语句也能达到一样的实验效果，因为加与不加，.cc文件寻找头文件时都是默认优先在当前文件夹中寻找，因此-I../lab2语句在实验1加与不加并没有什么区别。
4. 其次就是INCPATH中-I-的参数，该参数表示编译时不再按照默认的查找方式查找头文件，而是按照INCPATH中给出的路径来查找头文件。也正是因为这个原因，touch threads/scheduler.h再make，其它与scheduler.h相关的文件都不会再重新编译，因此默认查找路径中的lab2/scheduler.h并没有修改，因此对编译没有影响。而当touch lab2/scheduler.h再make时，所有与之相关的.cc文件都将重新编译，完成了我们所期待的功能。
5. 总结一下本次实验中的四种编译选项。
   - INCPATH += -I../threads -I../machine
   - INCPATH += -I../lab2 -I../threads -I../machine

- INCPATH += -I../lab2 -I../threads -I../machine 并用 grep 语句将所有与 scheduler.h 的文件放入 lab2 文件夹中
- INCPATH += -I- -I../lab2 -I../threads -I../machine

其中第一、二种均为 touch ../threads/scheduler.h 再 make 后，所有与 scheduler.h 相关的 .cc 文件都会重新编译。而第三、四种，只需要 touch ../lab2/scheduler.h 再 make，所有与之相关的 .cc 文件都会重新编译，而第四种方式最简单，因此之后的实验中我们主要采用第四种方式进行 make。

6. 实验二与实验一一样，仍然未涉及到自己实现代码的部分，仍然属于为之后的实验打基础的部分，让读者了解 $Nachos$ 中 makefile 的方式，便于之后实验的开展。