NachOS - Scheduling

2024 / 11 / 26

Assignment

Assignment

- 在本次 Lab 中，必須完成兩項要求

1. 建立「-sche」Bash Option

- Bash Option 的用途：傳遞你的「目的」
- 例如建置環境時的指令 docker build **-t** nachos .

  「**-t**」的含義為「為 Docker image 加上標籤」。
- 例如作業一的指令 ../build.linux/nachos **-e** halt

  「**-e**」的含義為「執行」某個檔案。

所以在本 Lab 會用到指令範例如下

../build.linux/nachos **-sche** SJF

其含義為「我要使用 SJF 排程方法」

Assignment

- 在本次 Lab 中，必須完成兩項要求

2. 建立三種排程方法：**Priority、SJF、FCFS**，並且列印出正確結果。

- 結合前一要求，機測時必須能夠順利運行以下指令

../build.linux/nachos -sche Priority

../build.linux/nachos -sche SJF

../build.linux/nachos -sche FCFS

Assignment

- 測試方法：助教會在 threads/thread.cc 中提供測試用 Function：**SelfTest()**

  - SelfTest() 會建立數個 Thread，並且賦予 name、id、priority、burst time、start time 等等資訊。
  - Priority 排程法：priority **越小優先級越高**

    依start time先後次序進行排序，時候未到的thread不能執行；priority優先級高的先處理；

    priority相同依 ID 由小至大依序處理
  - SJF 排程法：根據 burst time 進行排序

    依start time先後次序進行排序，時候未到的thread不能執行；burst time小的先處理；

    priority相同依 ID 由小至大依序處理
  - FCFS 排程法：根據 start time 進行排序

    依start time先後次序進行排序，時候未到的thread不能執行；

    start time 相同依 ID 由小至大依序處理

Assignment

- threads/thread.cc - Thread::SelfTest()

  - 在這頁投影片中，「Thread 資訊」代表「priority、burst time、start time」。

  - Thread 的數量可能不止 3 個，也可能出現 priority、burst time、start time 相同等等的情境
    但 Thread 資訊**不會故意設計成 0、負數、小數**
    這種 Special Case，同學可以專注於「如何實現排程」這個目標。

  - SelfTest() 在機測過程可能會修改 Thread 資訊
    不過整體架構不變。

```
void Thread::SelfTest() {
    DEBUG(dbgThread, "Entering Thread::SelfTest") ;
    const int thread_num = 6 ;
    char *name[thread_num] = {"A", "B", "C", "D", "E", "F"} ;
    int priority[thread_num] = {7, 2, 4, 4, 6, 3} ;
    int burst[thread_num] = {3, 2, 4, 4, 5, 7} ;
    int start[thread_num] = {1, 1, 7, 7, 15, 15} ;

    Thread *t ;
    int i = 0 ;
    for ( i = 0 ; i < thread_num ; i ++ ) {
        t = new Thread( name[i], i ) ;
        t->setPriority(priority[i]) ;
        t->setBurstTime(burst[i]) ;
        t->setStartTime(start[i]) ;
        t->Fork((VoidFunctionPtr) SimulateTimeThread, (void *)NULL) ;
    } // for()
} // SelfTest()
```

Assignment

- 成功運行範例 (FCFS)
  - 螢幕輸出資訊已經準備在 threads/thread.cc 也就是這部分無需同學實作。
  - 範例僅供參考，指令有些微差異是當前路徑不同造成。

```
const int thread_num = 6 ;
char *name[thread_num] = {"A", "B", "C", "D", "E", "F"} ;
int priority[thread_num] = {7, 2, 4, 4, 6, 3} ;
int burst[thread_num] = {3, 2, 4, 4, 5, 7} ;
int start[thread_num] = {1, 1, 7, 7, 15, 15} ;
```

```
06:26:53 root@9e41f3e31240 test ±|main ✗|→ ../build.linux/nachos -sche FCFS
===== FCFS =====
*** thread A: remaining time 2
*** thread A: remaining time 1
*** thread A: remaining time 0
*** thread B: remaining time 1
*** thread B: remaining time 0
*** thread C: remaining time 3
*** thread C: remaining time 2
*** thread C: remaining time 1
*** thread C: remaining time 0
*** thread D: remaining time 3
*** thread D: remaining time 2
*** thread D: remaining time 1
*** thread D: remaining time 0
*** thread E: remaining time 4
*** thread E: remaining time 3
*** thread E: remaining time 2
*** thread E: remaining time 1
*** thread E: remaining time 0
*** thread F: remaining time 6
*** thread F: remaining time 5
*** thread F: remaining time 4
*** thread F: remaining time 3
*** thread F: remaining time 2
*** thread F: remaining time 1
*** thread F: remaining time 0
```

Assignment

- 成功運行範例 (SJF)
  - 螢幕輸出資訊已經準備在 threads/thread.cc 也就是這部分無需同學實作。
  - 範例僅供參考，指令有些微差異是當前路徑不同造成。

```
const int thread_num = 6 ;
char *name[thread_num] = {"A", "B", "C", "D", "E", "F"} ;
int priority[thread_num] = {7, 2, 4, 4, 6, 3} ;
int burst[thread_num] = {3, 2, 4, 4, 5, 7} ;
int start[thread_num] = {1, 1, 7, 7, 15, 15} ;
```

```
06:28:09 root@9e41f3e31240 test ±|main ✗|→ ../build.linux/nachos -sche SJF
===== SJF =====
*** thread B: remaining time 1
*** thread B: remaining time 0
*** thread A: remaining time 2
*** thread A: remaining time 1
*** thread A: remaining time 0
*** thread C: remaining time 3
*** thread C: remaining time 2
*** thread C: remaining time 1
*** thread C: remaining time 0
*** thread D: remaining time 3
*** thread D: remaining time 2
*** thread D: remaining time 1
*** thread D: remaining time 0
*** thread E: remaining time 4
*** thread E: remaining time 3
*** thread E: remaining time 2
*** thread E: remaining time 1
*** thread E: remaining time 0
*** thread F: remaining time 6
*** thread F: remaining time 5
*** thread F: remaining time 4
*** thread F: remaining time 3
*** thread F: remaining time 2
*** thread F: remaining time 1
*** thread F: remaining time 0
```

Assignment

- 成功運行範例 (Priority)
  - 螢幕輸出資訊已經準備在 threads/thread.cc 也就是這部分無需同學實作。
  - 範例僅供參考，指令有些微差異是當前路徑不同造成。

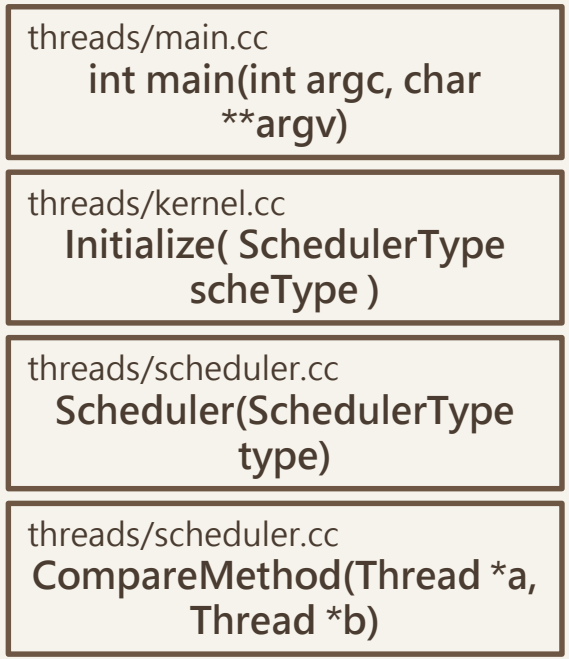
```
const int thread_num = 6 ;
char *name[thread_num] = {"A", "B", "C", "D", "E", "F"} ;
int priority[thread_num] = {7, 2, 4, 4, 6, 3} ;
int burst[thread_num] = {3, 2, 4, 4, 5, 7} ;
int start[thread_num] = {1, 1, 7, 7, 15, 15} ;
```

```
06:26:50 root@9e41f3e31240 test ±|main ✗|→ ../build.linux/nachos -sche Priority
===== Priority =====
*** thread B: remaining time 1
*** thread B: remaining time 0
*** thread A: remaining time 2
*** thread A: remaining time 1
*** thread A: remaining time 0
*** thread C: remaining time 3
*** thread C: remaining time 2
*** thread C: remaining time 1
*** thread C: remaining time 0
*** thread D: remaining time 3
*** thread D: remaining time 2
*** thread D: remaining time 1
*** thread D: remaining time 0
*** thread F: remaining time 6
*** thread F: remaining time 5
*** thread F: remaining time 4
*** thread F: remaining time 3
*** thread F: remaining time 2
*** thread F: remaining time 1
*** thread F: remaining time 0
*** thread E: remaining time 4
*** thread E: remaining time 3
*** thread E: remaining time 2
*** thread E: remaining time 1
*** thread E: remaining time 0
```

Hint - Trace Code

Trace Code

- 執行../build.linux/nachos -sche RR 時，發生了什麼事？



Trace Code

- threads/scheduler.h - **enum**
  - **實作 0**：事先列舉需要的排程方法。

```
/* Lab2 - Scheduling - Start */

enum SchedulerType {
      ➡ RR,
} ;

/* Lab2 - Scheduling - End */
```

Trace Code

- threads/threads.h - **class Thread**
  - **實作 1**：class Thread 有 **public** 與 **private** 兩個權限區塊。

    因為原生的 nachOS Thread 並沒有 priority、burst time、start time 這些資訊

    請同學在 private 區塊建立上述變數，型別為 int。

  - **實作 2**：承上，由於變數權限為 private，因此需要相對應的函式才能更改變數與讀取變數內容

    請同學補齊大括號內之內容。

    這些函式在測試用 function 會用到，所以函式名勿更改。

    換句話說，如果函式內容寫錯，就會無法賦予 thread 相關資訊。

```
/* Lab2 - Scheduling - Start */

int getBurstTime() {}
int getPriority() {}
int getStartTime() {}
void setBurstTime(int x) {}
void setStartTime(int x) {}
void setPriority(int x) {}

/* Lab2 - Scheduling - Start */
```

Trace Code

- threads/threads.h - **class Thread**

```
/* Lab2 - Scheduling - Start */

int getBurstTime() {}
int getPriority() {}
int getStartTime() {}
void setBurstTime(int x) {}
void setStartTime(int x) {}
void setPriority(int x) {}

/* Lab2 - Scheduling - Start */
```

```
void Thread::SelfTest() {
    const int thread_num = 3;
    char *name[thread_num] = {"A", "B", "C"};
    int priority[thread_num] = {7, 4, 6};
    int burst[thread_num] = {5, 19, 3};
    int start[thread_num] = {2, 1, 3};

    Thread *t;
    int i = 0;
    for (i = 0; i < thread_num; i ++) {
        t = new Thread(name[i], i);
        t->setPriority(priority[i]);
        t->setBurstTime(burst[i]);
        t->setStartTime(start[i]);
        t->Fork((VoidFunctionPtr) SimpleThread, (void *)NULL);
    }
}
```

Trace Code

- threads/main.cc - **int main(int argc, char **argv)**
  - (暫時) 換個例子，如果執行的指令是 ../build.linux/nachos **-d** + -sche SJF 也就是執行 Debug mode。
  - 在這個 for() 迴圈會處理使用者輸入的指令。

int main(int argc, char **argv)

Initialize(SchedulerType scheType)

Scheduler(SchedulerType type)

CompareMethod(Thread *a, Thread *b)

```
// some command line arguments are handled here.
// those that set kernel parameters are handled in
// the Kernel constructor
for (i = 1; i < argc; i++) {
    if (strcmp(argv[i], "-d") == 0) {
        ASSERT(i + 1 < argc);  // next argument is debug string
        debugArg = argv[i + 1];
        i++;
    } else if (strcmp(argv[i], "-z") == 0) {
        cout << copyright << "\n";
    } else if (strcmp(argv[i], "-x") == 0) {
        ASSERT(i + 1 < argc);
        userProgName = argv[i + 1];
        i++;
    } else if (strcmp(argv[i], "-K") == 0) {
        threadTestFlag = TRUE;
    } else if (strcmp(argv[i], "-C") == 0) {
        consoleTestFlag = TRUE;
    } else if (strcmp(argv[i], "-N") == 0) {
        networkTestFlag = TRUE;
    }
```

Trace Code

- threads/main.cc - **int main(int argc, char **argv)**
  - **實作 3**：在 for() 已經預留好框架與註解提示

    請同學根據系統預設的 Bash Option 以此類推該如何建立作業所需之 Bash Option。

```
/* Lab2 - Scheduling - Start */

// Hint : You should write something in "if()" to implement a new bash option.
//        At the same time, don't remove the "threadTestFlag" and "ASSERT()"
else if () {
    threadTestFlag = TRUE;
    ASSERT(i + 1 < argc) ;

    // Hint : This example shows you how to handle the parameter after a bash option
    //        "cout" is a debug message, "scheType" is a variable to record the scheduling method.
    if (strcmp(argv[i + 1], "RR") == 0) {
        cout << "===== RR =====" << endl ;
        scheType = RR ;
    } // if()

    i++ ;
} // else if()

/* Lab2 - Scheduling - End */
```

Trace Code

- threads/main.cc - **int main(int argc, char **argv)**

```
    DEBUG(dbgThread, "Entering main");

    kernel = new Kernel(argc, argv);

    // kernel->Initialize();

    /* Lab2 - Scheduling - Start */

    kernel->Initialize(scheType) ;

    /* Lab2 - Scheduling - End */

    CallOnUserAbort(Cleanup);  // if user hits ctl-C

    // at this point, the kernel is ready to do something
    // run some tests, if requested
    if (threadTestFlag) {
        kernel->ThreadSelfTest();  // test threads and synchronization
        return 1 ;
    }
```

Trace Code

- threads/kernel.cc - **Kernel::Initialize(SchedulerType scheType)**
  - 建立 nachOS Kernel 並初始化，同時會將排程類型傳入。

```
//----------------------------------------------------------------------
// Kernel::Initialize( SchedulerType scheType )
//----------------------------------------------------------------------

void Kernel::Initialize( SchedulerType scheType ) {

    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(RUNNING);

    stats = new Statistics();
    interrupt = new Interrupt;
    scheduler = new Scheduler( scheType );
    alarm = new Alarm(randomSlice);
    machine = new Machine(debugUserProg);
    synchConsoleIn = new SynchConsoleInput(consoleIn);
    synchConsoleOut = new SynchConsoleOutput(consoleOut);
    synchDisk = new SynchDisk();
    interrupt->Enable();
}
```

Trace Code

- threads/scheduler.cc - **Scheduler::Scheduler(SchedulerType type)**
  - **實作 4**：Scheduler::Scheduler(SchedulerType type) 是自定義的 Function
    此 Function 的目的是根據傳入的排程方法進行分類，已經預留好框架。
    同學只需要將剩下的 case 補齊即可。

```
//----------------------------------------------------------------------
// Scheduler::Scheduler(SchedulerType type)
//----------------------------------------------------------------------

Scheduler::Scheduler(SchedulerType type) {
        schedulerType = type;
        switch(type)
        {
            case RR:
                readyList = new List<Thread *> ;
                break;

            case /* scheduler type */ :
                readyList = new SortedList<Thread *>( /*Your Compare Method*/ );
                break;
        }
        toBeDestroyed = NULL;
} // Scheduler()
```

Trace Code

- threads/scheduler.cc - **Scheduler::Scheduler(SchedulerType type)**

  - 以 RR 為例子，此 function 被呼叫時，RR 這個 scheduler type 會被傳遞到此 function。

    透過 switch case 判斷該如何，因為 nachOS 預設就是 RR，因此不需要 Compare method。

    Q : 如果有其他排程方法？

```
//----------------------------------------------------------------------
// Scheduler::Scheduler(SchedulerType type)
//----------------------------------------------------------------------

Scheduler::Scheduler(SchedulerType type) {
        schedulerType = type;
        switch(type)
        {
            case RR:
                readyList = new List<Thread *> ;
                break;

            case /* scheduler type */ :
                readyList = new SortedList<Thread *>( /*Your Compare Method*/ );
                break;
        }
        toBeDestroyed = NULL;
} // Scheduler()
```

Trace Code

- threads/scheduler.cc - **Scheduler::Scheduler(SchedulerType type)**

  - 以 RR 為例子，此 function 被呼叫時，RR 這個 scheduler type 會被傳遞到此 function。

    透過 switch case 判斷該如何，因為 nachOS 預設就是 RR，因此不需要 Compare method。

    Q : 如果有其他排程方法？

```
//----------------------------------------------------------------------
// Scheduler::Scheduler(SchedulerType type)
//----------------------------------------------------------------------

Scheduler::Scheduler(SchedulerType type) {
        schedulerType = type;
        switch(type)
        {
            case RR:
                readyList = new List<Thread *> ;
                break;

            case /* scheduler type */ :
                readyList = new SortedList<Thread *>( /*Your Compare Method*/ );
                break;
        }
        toBeDestroyed = NULL;
} // Scheduler()
```

Trace Code

- threads/scheduler.cc – **int CompareMethod(Thread *a, Thread *b)**

  - **實作 5**：CompareMethod(Thread *a, Thread *b) 是自定義的 Function

    此 Function 的目的是根據排程方法進行 Thread 之間的比較

    不同的排程方法都要有屬於自己的 compare method。

    由於此函式為 int 型別，因此我們規定

    如果 Thread a 與 Thread b 經過比較後 b 的執行優先級大於 a

    則 return 1，反之 return -1。

Trace Code

- threads/scheduler.cc – **int CompareMethod(Thread *a, Thread *b)**
  - 條件判斷式會因為排程的不同而有所變化
  - 一個很簡單的框架如下

```
int MethodCompare(Thread *a, Thread *b) {
    if ( 條件判斷式 ) { return 1 ; }
    else { return -1 ; }
} // MethodCompare()
```

```
//----------------------------------------------------------------------
// Scheduler::Scheduler(SchedulerType type)
//----------------------------------------------------------------------

Scheduler::Scheduler(SchedulerType type) {
        schedulerType = type;
        switch(type)
        {
            case RR:
                readyList = new List<Thread *> ;
                break;

            case /* scheduler type */ :
                readyList = new SortedList<Thread *>( /*Your Compare Method*/ );
                break;
        }
        toBeDestroyed = NULL;
} // Scheduler()
```

Trace Code

- threads/main.cc - **int main(int argc, char **argv)**

```
DEBUG(dbgThread, "Entering main");

kernel = new Kernel(argc, argv);

// kernel->Initialize();

/* Lab2 - Scheduling - Start */

kernel->Initialize(scheType) ;

/* Lab2 - Scheduling - End */

CallOnUserAbort(Cleanup);  // if user hits ctl-C

// at this point, the kernel is ready to do something
// run some tests, if requested
if (threadTestFlag) {
    kernel->ThreadSelfTest();  // test threads and synchronization
    return 1 ;
}
```

Trace Code

- threads/kernel.cc - **Kernel::ThreadSelfTest()**

```
void Kernel::ThreadSelfTest() {
    Semaphore *semaphore;
    SynchList<int> *synchList;

    LibSelfTest();  // test library routines

    currentThread->SelfTest();  // test thread switching

    // test semaphore operation
    semaphore = new Semaphore("test", 0);
    semaphore->SelfTest();
    delete semaphore;

    // test locks, condition variables
    // using synchronized lists
    synchList = new SynchList<int>;
    synchList->SelfTest(9);
    delete synchList;
}
```

Shell script and Makefile

Shell script

- *$ bash build_nachos.sh* 做了什麼事
  - 打開 *code/test/build_nachos.sh* 可以看到以下指令

```
#!/bin/sh

# build nachos
cd ../build.linux
make clean > /dev/null 2>&1
make -j > /dev/null 2>&1
if [ $? -eq 0 ]; then
    echo "Build success"
else
    echo "Build failed"
    exit 1
fi
```

  - 這個 Shell Script 的用途為：切換目錄、清除上次 *make* 指令產生的檔案、執行 *make*。
  - *make* 會讀取 Makefile 並自動建立 program。

Shell script

- Message : Build success / Build failed

```
#!/bin/sh

# build nachos
cd ../build.linux
make clean > /dev/null 2>&1
make -j > /dev/null 2>&1
if [ $? -eq 0 ]; then
    echo "Build success"
else
    echo "Build failed"
    exit 1
fi
```

- */dev/null* 會將標準輸出 (Standard Output) 指向到 */dev/null*。
- *2>&1* 會將標準錯誤 (Standard Error) 也指向到同樣的地方。
- 隱藏所有輸出內容。

Shell script

- 如果你想要檢查你的 Error Message (When you compile error)

```
#!/bin/sh

# build nachos
cd ../build.linux
make clean > /dev/null 2>&1
make -j > /dev/null 2>&1
if [ $? -eq 0 ]; then
    echo "Build success"
else
    echo "Build failed"
    exit 1
fi
```

1. 刪掉 shell script 中的 > */dev/null 2>&1*

make clean > /dev/null 2>&1 #You don't need to change here

make -j

2. 到 *code/build.linux/* 手動執行 *make clean* 與 *make*。

Makefile

- Makefile：用於自動化編譯和建置軟體的工具。
  - 輸入 *make* 就能自動完成編譯。
  - 自動識別需要重新編譯的文件。

- 在 NachOS 中會用到兩個 Makefile
  1. */code/build.linux/Makefile*
  2. */code/test/Makefile*

/code/build.linux/Makefile ▸

```
CPP=/lib/cpp
CC = g++
LD = g++
AS = as
RM = /bin/rm

INCPATH = -I../network -I../filesys -I../userprog -I../threads -I../machine -I../lib -I-

PROGRAM = nachos

#
# Edit these lists as if you add files to the source directories.
# See the instructions at the top of the file for more information.
#

LIB_H = ../lib/bitmap.h\
      ../lib/copyright.h\
      ../lib/debug.h\
      ../lib/hash.h\
      ../lib/libtest.h\
      ../lib/list.h\
      ../lib/sysdep.h\
      ../lib/utility.h

LIB_C = ../lib/bitmap.cc\
      ../lib/debug.cc\
      ../lib/hash.cc\
      ../lib/libtest.cc\
      ../lib/list.cc\
      ../lib/sysdep.cc

LIB_O = bitmap.o debug.o libtest.o sysdep.o
```

Makefile

- 解讀 Makefile

```
CFLAGS = -g -Wall $(INCPATH) $(DEFINES) $(HOSTCFLAGS) -DCHANGED -m32
LDFLAGS = -m32
CPP_AS_FLAGS= -m32
```

- CFLAGS (Compiler Flags) : 將參數傳遞給 C/C++ Compiler。

- LDFLAG (Linker Flags) :傳遞給 Linker 的選項。

- CPP_AS_FLAGS (C PreProcessor Assembler Flags) : 主要用於處理 Assembler 文件。

Makefile

- 解讀 Makefile

```
CPP=/lib/cpp
CC = g++
LD = g++
AS = as
RM = /bin/rm

INCPATH = -I../network -I../filesys -I../userprog -I../threads -I../machine -I../lib -I-

PROGRAM = nachos
```

- CPP (C PreProcessor)、CC (C Compiler)、LD (Linker)、AS (Assembler)、RM (Remove)。

- INCPATH (Include Path) : 告訴 Compiler 標頭檔 (Header File) 的位置。

- PROGRAM : 定義最終生成的可執行檔之檔名。

This is why you will enter *../build.linux/nachos -e fileIO_test1*

Makefile

- 解讀 Makefile

```
THREAD_S = ../threads/switch.s

HFILES = $(LIB_H) $(MACHINE_H) $(THREAD_H) $(USERPROG_H) $(FILESYS_H) $(NETWORK_H)
CFILES = $(LIB_C) $(MACHINE_C) $(THREAD_C) $(USERPROG_C) $(FILESYS_C) $(NETWORK_C)

C_OFILES = $(LIB_O) $(MACHINE_O) $(THREAD_O) $(USERPROG_O) $(FILESYS_O) $(NETWORK_O)

S_OFILES = switch.o
OFILES = $(C_OFILES) $(S_OFILES)

$(PROGRAM): $(OFILES)
	$(LD) $(OFILES) $(LDFLAGS) -o $(PROGRAM)

$(C_OFILES): %.o:
	$(CC) $(CFLAGS) -c $<
```

- 將不同資料夾的檔案進行分類 (*.h .cc .o* 等等)。

- 生成 PROGRAM。

```
LIB_H = ../lib/bitmap.h\
        ../lib/copyright.h\
        ../lib/debug.h\
        ../lib/hash.h\
        ../lib/libtest.h\
        ../lib/list.h\
        ../lib/sysdep.h\
        ../lib/utility.h

LIB_C = ../lib/bitmap.cc\
        ../lib/debug.cc\
        ../lib/hash.cc\
        ../lib/libtest.cc\
        ../lib/list.cc\
        ../lib/sysdep.cc

LIB_O = bitmap.o debug.o libtest.o sysdep.o

MACHINE_H = ../machine/callback.h\
        ../machine/interrupt.h\
        ../machine/stats.h\
        ../machine/timer.h\
        ../machine/console.h\
        ../machine/machine.h\
        ../machine/mipssim.h\
        ../machine/translate.h\
        ../machine/network.h\
        ../machine/disk.h
```

Makefile

- 解讀 Makefile

```
depend: $(CFILES) $(HFILES)
	$(CC) $(INCPATH) $(DEFINES) $(HOSTCFLAGS) -DCHANGED -M $(CFILES) > makedep
	@echo '/^# DO NOT DELETE THIS LINE/+1,$$d' >eddep
	@echo '$$r makedep' >>eddep
	@echo 'w' >>eddep
	@echo 'q' >>eddep
	ed - Makefile.dep < eddep
	rm eddep makedep
	@echo '# DEPENDENCIES MUST END AT END OF FILE' >> Makefile.dep
	@echo '# IF YOU PUT STUFF HERE IT WILL GO AWAY' >> Makefile.dep
	@echo '# see make depend above' >> Makefile.dep

clean:
	$(RM) -f $(OFILES)
```

- depend：利用 Make 文件的比較功能，在編譯時會檢查 depend file，如果有更改過則會重新編譯。

- clean : 要刪除哪些檔案。

Makefile

- 總結

- *build_nachos.sh* 是一個 Shell Script，它會切換目錄並執行該目錄下的 *Makefile*。

  Q : 哪個目錄？　　A : */code/build.linux/*

  換句話說，Shell Script 裡面的所有指令也可以透過手動輸入完成，但會花費更多時間。

- 執行 *make* 指令時就是在運行 *Makefile*，而 *Makefile* 就是在編譯使用者指定的檔案。

  Q1 : 編譯了哪些檔案？　A1 : 寫在 */code/build.linux/Makefile* 裡的檔案

  實際上，這個 Makefile 負責編譯所有檔案 (但除了 */code/test/*)，最終生成 */code/build.linux/nachos*。

  Q2 : 那 */code/test/* 路徑下的檔案會由誰負責編譯？

- *$ bash build_nachos_docker.sh* 就是在生成名為 nachos 的可執行檔，也就是 nachos。

END