OS HW3

106030028 劉多聞
107030028 劉騏鋒
106030014 陳致瑋

Part 1 Trace Code

1. Load program
   a. UserProgKernel::Run()
      使用 for 迴圈將所有 execfile 一個個初始化，初始化出新的 thread control block 後配置一個 page table 給 thread，接著再進入 Thread::Fork(VoidFunctionPtr, void*)，fork 這個 thread。

   b. AddrSpace::AddrSpace()
      初始化 thread 的 page table，把” NumPhysPages”個 entries 初始化。pageTable 裡面含有的資訊主要是 virtualPage 和 physicalPage，兩者透過 pageTable 溝通。

   c. ThreadedKernel::Run()
      一般程式 main return 值時，所有程式會停止運行，但因為有多個 thread 在 os 裡面執行，所以要夠過 ThreadedKernel::Run()來防止”main.cc” return。

   d. Thread::Finish()
      當一個 thread 執行完畢後會進到 Finish()先處理 interrupt 的問題，在處理 thread 占用 cpu 的優先次序時是不能被 interrupt 的，所以要先把 interrupt 關掉。接下來，利用 ASSERT 檢查當前的 thread 是否與 kernel 裡的 Thread 一致。最後呼叫 Sleep 來進行 context switch。

   e. Thread::Sleep (bool finishing)
      前幾行的 ASSERT 和 DEBUG 是確保 OS 正確運行的檢查機制，接著把當前 thread 的 status 設為 BLOCKED，在利用 while 迴圈不斷檢查與尋找在 ready queue 裡面下一個要運行的 thread。若沒有下一個要跑的 thread，kernel 就會回到 IDLE()，反之則呼叫 Scheduler::Run()來執行下一個 thread。

   f. Scheduler::Run (Thread *nextThread, bool finishing)
      Scheduler::Run 的目的是做 context switch，將目前 thread 狀態記錄下來

並移除 running state，再把 next thread load 進來 running state。

1. 首先確認目前的 thread 是否完成，如果為 true 則刪掉 thread。
2. 再來將目前 thread 的 CPU registers 狀態記錄下來。
3. 檢查 thread control block 是否溢位。
4. 把 kernel 目前的 thread 從舊的改成新的。
5. 將新的 thread 狀態設為”running”。
6. 接著做 switch()。

g. SWITCH (thread *t1, thread *t2)
首先將要 switch 出去的舊 thread CPU registers 狀態存回 control block，接著再將新的 thread 的 control block 狀態資料重新讀回 CPU registers

1. 把 eax 的值存起來。
2. 讓 eax 指向 thread 1。
3. 將 ebx、ecx、edx、esi、edi、ebp、esp 值存回 control block。
4. ebx 讀取 eax 原來的值。
5. eax 原來的值存回 control block。
6. 讓 eax 取得返回的位置。
7. 讓_PC 取得 stack top。
8. 讓 eax 指向 thread 2。
9. 將 thread 2 control block 的 eax 值給 ebx。
10. 把 ebx 存起來。
11. 把 control block 的 ebx、ecx、edx、esi、edi、ebp、esp 從新讀回 CPU 暫存器。
12. 把_PC 值給 eax。
13. 把 eax 值給 4(%esp)。
14. 把剛剛 control block 裡的 EAX 給 eax。
15. 返回。

h. ThreadBegin()
這個 function 的功能主要是當一個新的 thread 要開始執行時進行依些檢查動作。ASSERT 和 DEBUG 是確保 OS 正確運行的檢查機制，再來夠過 CheckToBeDestroyed 把上一個執行完的 thread 移除，最後呼叫 Enable()，讓 interrupt 可以發生來切割 OS 執行時間。

i. ForkExecute (Thread *t)
執行新的 thread。

j. AddrSpace::Execute (char *fileName)
   1. 將”filename”檔案 load 進記憶體。
   2. 初始化暫存器的狀態。
   3. 把 page table 的起始位置跟 page table 大小讀進暫存器。

k. AddrSpace::Load (char *fileName)
   利用檔案操作物件把 fileName 打開並讀取，接著計算檔案 size 以及 page 的個數，因為會有 internal fragmentation 的問題，所以將 size 更新為 numPages * PageSize。最後將 code 及資料複製到記憶體並關掉檔案。

2. Page Faults
   a. Machine::Run()
      1. 主要目的為模擬機器執行 user programs。
      2. 首先切換成 user mode 然後無限 for loop，逐條執行 userprogram 的程式指令，模擬 MIPS CPU 讀指令的樣子。
      3. 進 OneInstruction()，讀完一個指令後讓 clock tick。

   b. Machine::OneInstruction(Instruction *instr)
      如果 instruction fetch 成功則繼續。
      針對 instruction 的 opcode 做對應的 case 處裡動作。

   c. Machine::ReadMem(int addr, int size, int *value)
      進 Translate()尋找 physicalAddress，若沒找到 physicalAddress 代表 page fault 發生，進 RaiseException 做對應的處置，接著 return false。如果有成功找到記憶體位置，則根據要找的資料 size(word、half word、byte)做不同存取，最後 return true。

   d. Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
      1. 一開始判斷 virtual address 的 format 是否對的上 size，若錯誤則 return AddressErrorException。
      2. 如果有的是 page table 則
         如果 virtual page number 大於等於 pagetable 的 size，則 return AddressErrorException，若否，則檢查該 page number 的 valid bit 是否為真，若否，則代表出現 page fault 並 return PageFaultException。

         若以上 vpn 的檢查結果有效且無 page fault 則把該 page number 對應的真實記憶體 frame 位置及其他 translation 的狀態用”entry”記錄下來。

3. 如果有的是 TLB 則
   從 TLB 頭掃到尾，若有發現對應的 virtualpage 且 valid bit 為真，代表 TLB hit，用”entry”紀錄 translation 的所有狀態。若沒找到則代表 TLB miss，return TLB fault Exception。
4. 檢查如果有償是改寫 ready-only 的情形發生，則 return ReadOnlyException。
5. 將 translation 所記錄的真實記憶體 frame 位置給 pageFrame，並檢查是否為有效 frame number 如果超過真實 frame number 則 return BusErrorException。
6. 若到這裡都沒 raise exception 代表可以成功找到真實記憶體，把 use bit 改成 true(作為之後 page replacement 演算法參考資訊)，如果有做修改則設 dirty bit 為 true。
7. 將對應的 frame 做 offset 後得到真實記憶體位置並 return NoException。

e. Machine::RaiseException(ExceptionType which, int badVAddr)
   當 user program invoked system call 或是發生 exception 時會呼叫此 function，在 page fault 的情況為”發生 exception”。將有問題的 virtual address 存入 BadVAddrRegister，並切回 systemMode 處理該 exception，處理完後再回 userMode。

f. ExceptionHandler(ExceptionType which)
   根據 RaiseException 傳過來的 ExecutionType 做對應的處置，如 case BusErrorException、case ReadOnlyException、case AddressErrorException... 等。

3. Implement
   a. AddSpace::AddrSpace()

      修正原本程式碼的初始值

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (unsigned int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = 0;
    pageTable[i].physicalPage = 0;
    pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
//    bzero(kernel->machine->mainMemory, MemorySize);
}
```

      static void copyPage()

      把 page 從 virtual memory copy 進 page table 以及從 page table copy 回 virtual memory。做 context switch 時用。

```
static
void copyPage(OpenFile *dst, int dstPos, OpenFile *src, int srcPos, int size)
{
    char *tmpPage = new char[PageSize];
    int byteLeft = size;

    DEBUG(dbgAddr, "copyPage: dst=" << dst << ", dstPos=" << dstPos << ", s
    rc =" << src << ", srcPos=" << srcPos << ", size=" << size << "\n");

    while (byteLeft >= (int)PageSize) {
        // read PageSize bytes from src file to tmpPage buffer
            DEBUG(dbgAddr, "byteLeft = " << byteLeft << "\n");

            ASSERT(src->ReadAt(tmpPage, PageSize, srcPos) == PageSize);
        srcPos = srcPos + PageSize;

        // write PageSize bytes from tmpPage buffer to dest file
        ASSERT(dst->WriteAt(tmpPage, PageSize, dstPos) == PageSize);
        dstPos = dstPos + PageSize;

        byteLeft -= PageSize;
    }
    // copy the rest bytes
    if (byteLeft > 0) {
        // read byteLeft bytes from src file to tmpPage buffer
                DEBUG(dbgAddr, "byteLeft = " << byteLeft << "\n");
        ASSERT(src->ReadAt(tmpPage, byteLeft, srcPos) == byteLeft);
        srcPos = srcPos + byteLeft;

        // write byteLeft bytes from tmpPage buffer to dest file
        ASSERT(dst->WriteAt(tmpPage, byteLeft, dstPos) == byteLeft);
        dstPos = dstPos + byteLeft;
    }
}
```

static void clearPage():
創建新的 user stack 空間。

```
static
void clearPage(OpenFile *dst, int dstPos, int size)
{
    char *tmpPage = new char[PageSize];
    int byteLeft = size;

    DEBUG(dbgAddr, "clearPage: dst=" << dst << ", dstPos=" << dstPos
    << ", size=" << size << "\n");

    // clear page
    for (int i=0; i< PageSize; i++) tmpPage[i] = 0;

    while (byteLeft >= (int)PageSize) {
            DEBUG(dbgAddr, "byteLeft = " << byteLeft << "\n");

        // write PageSize bytes from tmpPage buffer to dest file
        ASSERT(dst->WriteAt(tmpPage, PageSize, dstPos) == PageSize);
        dstPos = dstPos + PageSize;

        byteLeft -= PageSize;
    }
    // copy the rest bytes
    if (byteLeft > 0) {
                DEBUG(dbgAddr, "byteLeft = " << byteLeft << "\n");
        // write byteLeft bytes from tmpPage buffer to dest file
        ASSERT(dst->WriteAt(tmpPage, byteLeft, dstPos) == byteLeft);
        dstPos = dstPos + byteLeft;
    }
}
```

In >> bool AddrSpace::Load():
創造 virtual memory 並記錄 virtual memory 指標

```
string vmName = (string)fileName;
vmName += "_VM";
if (kernel->fileSystem->Create((char *)vmName.c_str()) == TRUE) {
DEBUG(dbgAddr, "Create file " << vmName << " successful\n");
virSpace = kernel->fileSystem->Open((char *)vmName.c_str());
}

int vmPos;
int sfPos;
```

Compiler 會 compile 出三個 segment(code、initial data 與 uninitial data)並 load 進 virtual memory。
將三個 segment 從 virtual memory 搬到 pagetable。

```
if (noffH.code.size > 0) {
    DEBUG(dbgAddr, "Initializing code segment.");

// copy code segment
vmPos = noffH.code.virtualAddr;
sfPos = noffH.code.inFileAddr;
DEBUG(dbgAddr, "noffH.code.virtualAddr = " << noffH.code.virtualAddr << ", noffH.code.size = " << noffH.code.size);
DEBUG(dbgAddr, "vmPos = " << vmPos << ", sfPos = " << sfPos);
copyPage(virSpace, vmPos, executable, sfPos, noffH.code.size);

}
```

```
if (noffH.initData.size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");

// copy data segment
vmPos = noffH.initData.virtualAddr;
sfPos = noffH.initData.inFileAddr;
DEBUG(dbgAddr, "noffH.initData.virtualAddr = " << noffH.initData.virtualAddr << ", noffH.initData.size = " << noffH.initData.size);
DEBUG(dbgAddr, "vmPos = " << vmPos << ", sfPos = " << sfPos);
copyPage(virSpace, vmPos, executable, sfPos, noffH.initData.size);


}
```

```
if (noffH.uninitData.size > 0) {
    DEBUG(dbgAddr, "Un-initializing data segment.");

// copy uninit data segment if there is any
vmPos = noffH.uninitData.virtualAddr;
sfPos = noffH.uninitData.inFileAddr;
DEBUG(dbgAddr, "noffH.uninitData.virtualAddr = " << noffH.uninitData.virtualAddr
 << ", noffH.uninitData.size = " << noffH.uninitData.size);
DEBUG(dbgAddr, "vmPos = " << vmPos << ", sfPos = " << sfPos);
copyPage(virSpace, vmPos, executable, sfPos, noffH.uninitData.size);
}
```

程式在跑的時候會不停紀錄程式進行的 stack 狀態，定義這個 stack 的位置。

```
if (UserStackSize > 0) {
    DEBUG(dbgAddr, "UserStackSize.");
// Reserve UserStack space
// assume our virtual space size is defined by VirtualSpaceSize
int vmPos = VirtualSpaceSize - UserStackSize;

DEBUG(dbgAddr, "vmPos = " << vmPos);
clearPage(virSpace, vmPos, UserStackSize);
}
```

void AddrSpace::SaveState():

儲存程式進行的 stack 狀態，用於 context switch

```
void AddrSpace::SaveState()
{
        pageTable=kernel->machine->pageTable;
        numPages=kernel->machine->pageTableSize;

    // Copy all dirty pages back to virtual space,
    // then clean the dirty flag
    for (int i=0; i < NumPhysPages; i++) {
            if (pageTable[i].dirty == TRUE) {
            unsigned int phyAddr = pageTable[i].physicalPage * PageSize;
            unsigned int virAddr = pageTable[i].virtualPage * PageSize;
            DEBUG(dbgAddr, "Save a dirty page " << pageTable[i].physicalPage <<
            " back to virtual page " << pageTable[i].virtualPage);
            DEBUG(dbgAddr, "Copy phyAddr: " << phyAddr << " to virAddr: " << virAddr);
            virSpace->WriteAt(&(kernel->machine->mainMemory[phyAddr]), PageSize, virAddr);
            // dirty page copied, set the dirty flag to FALSE
            pageTable[i].dirty = FALSE;
        }
    }
}
```

void AddrSpace::RestoreState():

把上一次 context switch 所記錄的狀態從新 load 回 page table。

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;

    // Copy back all valid pages from virtual space to physical memory
    for (int i=0; i < NumPhysPages; i++) {
        if (pageTable[i].valid == TRUE) {
        unsigned int phyAddr = pageTable[i].physicalPage * PageSize;
        unsigned int virAddr = pageTable[i].virtualPage * PageSize;
        DEBUG(dbgAddr, "restore physical page " << pageTable[i].physicalPage <<
        " from the virtual page " << pageTable[i].virtualPage);
        DEBUG(dbgAddr, "Copy to phyAddr: " << phyAddr << " from virAddr: " << virAddr);
        virSpace->ReadAt(&(kernel->machine->mainMemory[phyAddr]), PageSize, virAddr);
    }
    }
}
```

int AddrSpace::findPage2Use():
一開始要把 page 填進 page table 時，要檢查 page 欄位 valid 與否，若 invalid 則可填入，反之亦然。這個程式的尋找 empty page 的方式是對整個 page table 進行 linear search，若皆被填滿，則用 LRU 和 second chance 尋找 victim page 移出 circular queue。

判斷要置換的 page，若 dirty bit == true 則要寫回 virtual memory。

```
int AddrSpace::findPage2Use(){
    static int curPageIdx = 0;
    int result = 0;
    int i = curPageIdx;
    // find an invalid page to use
    do {
    if (pageTable[i].valid == FALSE) {
        // an invalid found
        // Next time, find from the next page
        result = i++;
        curPageIdx = i % NumPhysPages;
        return (result);
    }
    i = ++i % NumPhysPages;
    } while (i != curPageIdx);
    // No invalid page. Find an unused page to replace
    do {
    if (pageTable[i].use == FALSE) {
        // an unused page found
        // Next time, find from the next page
        result = i++;
        curPageIdx = i % NumPhysPages;
        return (result);
    }
    i = ++i % NumPhysPages;
    } while (i != curPageIdx);
    // no available page to use
    // just choose the current page to replace
    result = curPageIdx++;
    curPageIdx %= NumPhysPages;
    // if it is dirty, copy back to virtual space
    if (pageTable[result].dirty == TRUE) {
    unsigned int phyAddr = pageTable[result].physicalPage * PageSize;
    unsigned int virAddr = pageTable[result].virtualPage * PageSize;
    DEBUG(dbgAddr, "Get a dirty page " << result << " to use");
    DEBUG(dbgAddr, "Copy phyAddr: " << phyAddr << " to virAddr: " << virAddr);
    virSpace->WriteAt(&(kernel->machine->mainMemory[phyAddr]), PageSize, virAddr);
    }
    DEBUG(dbgAddr, "result = " << result << ", curPageIdx = " << curPageIdx);
    // return that page index
    return result;
}
```

void AddrSpace::PageFaulthandler

exceptionHandler 如果接收到 interrupt，就會用這個 function 把要找的 page load 進 pagetable。Load 進 pagetable 之後會把剛進來的 page 初始化。

```
void AddrSpace::pageFaultHandler()
{
    int i, virtAddr;

    DEBUG(dbgAddr, "enter my page fault handler\n");

    // The virtual address acused the page fault is saved in
    // machine's registers[BadVAddrReg]
    virtAddr = kernel->machine->ReadRegister(BadVAddrReg);
    DEBUG(dbgAddr, "virtual address: " << virtAddr);

    // 1. find a page to use
    i = findPage2Use();
    DEBUG(dbgAddr, "find page to use = " << i);

    // 2. copy virtual page to the physical memory page.
    virtAddr = virtAddr / PageSize;
    DEBUG(dbgAddr, "copy virtual page: " << virtAddr << " to phy page: " << i <<
    " from virtual page address: " << virtAddr*PageSize << ", to mainMemory["<<i*PageSize<<"]");
    virSpace->ReadAt(&(kernel->machine->mainMemory[i*PageSize]), PageSize, virtAddr*PageSize);

    // 3.set the virtual page number and physical page number in the pageTable
    //   set valid = TRUE, use = FALSE, dirty = FALSE, readonly = ?
    pageTable[i].virtualPage = virtAddr;
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
}
```

b. AddrSpace.h

將 UserStackSize 乘以 5 倍，以解決程式執行中 stack memory 不夠用的問題。

```
//#define UserStackSize     1024        // increase this as necessary!
#define UserStackSize       (1024*5)    // merge program need bigger stack

// my Virtual space size
#define VirtualSpaceSize    (128 * 128 + UserStackSize)
```

增加 pagefaulthandler function

```
class AddrSpace {
  public:
    AddrSpace();            // Creat
    ~AddrSpace();           // De-al

    void Execute(char *fileName);
                    // stored in the

    void SaveState();           // S
    void RestoreState();        // i

    // my page fault handler
    void pageFaultHandler();
```

增加 OpenFile *virSpace 和 int findPage2Use

```
private:
  TranslationEntry *pageTable;    // Assume linear page table translation
                  // for now!
  unsigned int numPages;      // Number of pages in the virtual
                  // address space

  OpenFile *virSpace;         // Our Virtual Space file

  bool Load(char *fileName);      // Load the program into memory
                  // return false if not found

  void InitRegisters();       // Initialize user-level CPU registers,
                  // before jumping to user code

  int findPage2Use();
```

c. exception.cc

處理 page fault interrput，呼叫 pagefaulthandler，並計算 pagefault 次數

```
case PageFaultException:
    // pageFaulException happened, go to our page fault handler
    DEBUG(dbgAddr, "PageFaultException\n");
    kernel->currentThread->space->pageFaultHandler();
    // increase the number of page fault exceptions
    kernel->stats->numPageFaults++;
    return;
default:
    cerr << "Unexpected user mode exception" << which << "\n";
    break;
```

d. translate.cc

用 linear 查詢 page table，若 page match 且 valid bit == true，會把 pagetable index 記錄下來。若找不到所要的 page 會 return pageFaultException。

```
if (tlb == NULL) {      // => page table => vpn is index into table
int i;
// find out if the virtual address is resident in physical memory
for (i=0; i<NumPhysPages; i++) {
    if (pageTable[i].virtualPage == vpn && pageTable[i].valid) {
        // virtual address is resident in physical memory
        DEBUG(dbgAddr, "translate memory hit page# " << i);
        break;
    }
}

if (i == NumPhysPages) {
    // virtual address is not resident in physical memory
        DEBUG(dbgAddr, "Invalid virtual address: " << virtAddr);
    // page fualt exception happened
        return PageFaultException;
}

    DEBUG(dbgAddr, "virtual address is resident in in pageTable[" << i << "]\n");
entry = &pageTable[i];

// original codes

if (vpn >= pageTableSize) {
    DEBUG(dbgAddr, "Illegal virtual page # " << virtAddr);
    return AddressErrorException;
} else if (!pageTable[vpn].valid) {
    DEBUG(dbgAddr, "Invalid virtual page # " << virtAddr);
    return PageFaultException;
}
    DEBUG(dbgAddr, "check pageTable[" << vpn << "]\n");
entry = &pageTable[vpn];
```

e. Makefile

```
all: halt shell matmult sort test1 test2 bubble merge quick
```

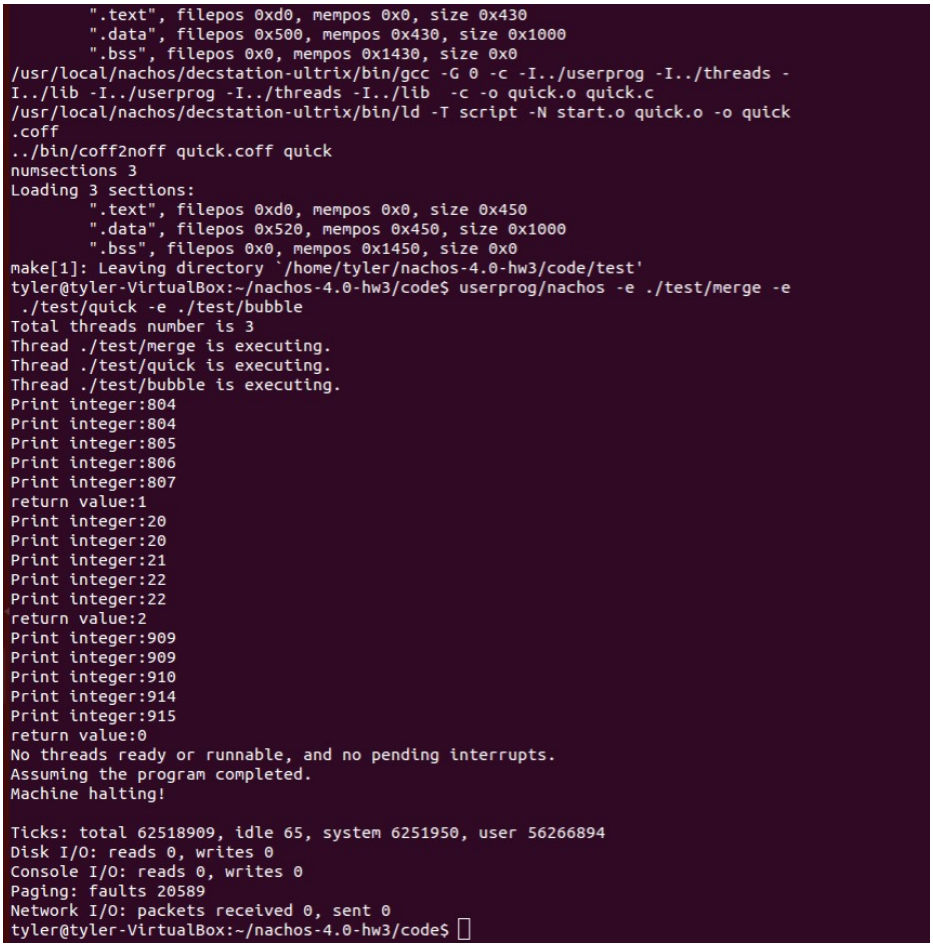
```
bubble: bubble.o start.o
    $(LD) $(LDFLAGS) start.o bubble.o -o bubble.coff
    ../bin/coff2noff bubble.coff bubble

merge: merge.o start.o
    $(LD) $(LDFLAGS) start.o merge.o -o merge.coff
    ../bin/coff2noff merge.coff merge

quick: quick.o start.o
    $(LD) $(LDFLAGS) start.o quick.o -o quick.coff
    ../bin/coff2noff quick.coff quick
```

執行 merge bubble quick 這三種 sort 方式

f. Results



分工: 劉騏鋒 60% 劉多聞 40%