MP2_Report_47

組員	郭芳妤	楊智堯
學號	111065531	111065542
分工	Trace code、報告、實作	Trace code、報告、實作

Part I. Trace code

> Starting from “threads/kernel.cc Kernel::ExecAll()”, “threads/thread.cc thread::Sleep()” until “machine/mipssim.cc Machine::Run()” is called for executing the first instruction from the user program.

1. 首先觀察 thread 的資料結構

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int *stackTop;       // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

  public:
    Thread(char* debugName, int threadID);    // initialize a Thread
    ~Thread();        // deallocate a Thread
          // NOTE -- thread being deleted
          // must not be running when delete
          // is called

    // basic thread operations

    void Fork(VoidFunctionPtr func, void *arg);
            // Make thread run (*func)(arg)
    void Yield();     // Relinquish the CPU if any
        // other thread is runnable
    void Sleep(bool finishing); // Put the thread to sleep and
        // relinquish the processor
    void Begin();   // Startup code for the thread
    void Finish();      // The thread is done executing
```

```
    void CheckOverflow();     // Check if thread stack has overflowed
    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
  char* getName() { return (name); }

  int getID() { return (ID); }
    void Print() { cout << name; }
    void SelfTest();    // test whether thread impl is working

  private:
    // some of the private data for this class is listed above

    int *stack;     // Bottom of the stack
        // NULL if this is the main thread
        // (If NULL, don't deallocate stack)
    ThreadStatus status;  // ready, running or blocked
    char* name;
  int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
            // Allocate a stack for thread.
        // Used internally by Fork()

// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs];  // user-level CPU register state

  public:
    void SaveUserState();   // save user-level register state
    void RestoreUserState();    // restore user-level register state

    AddrSpace *space;     // User code this thread is running.
};
```

- 一個正在執行 user program 的 thread 有兩個 CPU register sets，分別是執行 user code 和 kernel code 的 state，而 state 又包含了：program counter、processor register、execution stack。
- 每個 thread 除了 register sets，還有創建一個 AddrSpace。AddrSpace 是用來追蹤執行中的 user program 的資料結構。（詳見 userprog/addrspace.h）
- 根據 AddrSpace::Execute()，我們可以知道要使用一個 thread 來 run user program 需要以下幾個步驟（詳見 userprog/addrspace.cc/ `AddrSpace::Execute(char* fileName)`）：
  - initRegister()：set the initial register values
  - RestoreState()：load page table register
  - kernel->machine->Run(); jump to the user program

2. 從題幹要求的 `ExecAll() → Run()` 重新 trace 一次：

   a. threads/kernel.cc Kernel::ExecAll()

      目的：使用 Exec 依序 parsing 指令並作相應的設置

```
void Kernel::ExecAll()
{
  for (int i=1;i<=execfileNum;i++) {
    int a = Exec(execfile[i]);
  }
  currentThread->Finish();
}
```

   b. threads/kernel.cc Kernel::Exec()

      目的：針對每一個要執行的 program 開一個 thread，給他一個 AddrSpace（程式要 load 入這個 space 才能執行），透過 fork() create 出 process stack 以及把 thread 放入 ready queue

```
int Kernel::Exec(char* name)
{
  t[threadNum] = new Thread(name, threadNum);
  t[threadNum]->space = new AddrSpace();
  t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
  threadNum++;

  return threadNum-1;
}
```

   c. 使用 Fork() 這個 function 時，可以發現他調用的是 ForkExecute 這個函數，而這個函數的功能有二：

      i. Load a program into addr space from file（file = t→getName()），如果失敗則回傳

      ii. Load 成功後，則呼叫 addr space 中的 Execute 執行此 program

```
void ForkExecute(Thread *t)
{
  if ( !t->space->Load(t->getName()) ) {
      return;             // executable not found
    }
    t->space->Execute(t->getName());
}
```

- 呼叫 Machine:: Run() 來執行程式（接回 MP1）

```
kernel->machine->Run();   // jump to the user progam
```

- 最後的 `ASSERTNOTREACHED();` 是因為 run 本身是個不會 return 的 function，只有使用 system call 的 exit 才有辦法讓 address space exit

---

Explain Function 1-1

▼ 1-1-a `Thread::Sleep (bool finishing)`

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Sleeping thread: " << name);
    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << ker
nel->stats->totalTicks);
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle();  // no one to run, wait for an interrupt
  }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

- 功能：
  1. 當 thread 已經完成，或者 thread 正在等待 synchronization 時，**交出手中的 CPU 資源**，並把這個 thread 的 status 設為 blocked。【註】：如果這個 thread 是在等待 synchronization，sync 完成後會由另一個 thread 再把此 thread 喚醒、並放入 ready queue，此 thread 即可被 re-scheduled。
  2. 如果 ready queue 中沒有待執行的 thread，讓 CPU 維持 Idle
- 這邊有一個重要的設定，即「interrupt 必須被 disabled」，這是為了保障執行的 atomicity：即確保在 ready queue 中的 thread 從被取出到 switch 到這個 thread 之間不會出現 interrupt 導致的中斷。trace code 之後可以發現從 Finish() 開始 interrupt 就會保持 disabled 的狀態直到下一次 Exec() 為止。

- 傳入參數：bool finishing（其實是給 `Scheduler::Run` 使用）
  - True：表示此 thread 已結束，如果是由 Finish() 調用 Sleep，則傳入參數就會是 True
  - False：表示此 thread 尚未結束，傳入 Run() 時，該 thread 就不會被標記要被 destroyed
- 細節說明：
  1. 檢查此刻的 thread 是否為 currentThread、檢查 interrupt 此時是否已經被 disabled（確保這個流程不被其他 event 打斷）

     ```
     ASSERT(this == kernel->currentThread);
     ASSERT(kernel->interrupt->getLevel() == IntOff);
     ```

  2. 由 scheduler 的 `FindNextToRun()` 尋找下一個需要執行的 thread。如果 Ready Queue 中沒有 NextThread 需要執行，則我們必需 idle CPU（call Interrupt::Idle()）直到下一次 I/O interrupt 出現時（I/O interrupt 會造成本來在 waiting 狀態的 job 進入 ready queue 中）

     ```
     while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
         kernel->interrupt->Idle();
     }
     ```

  3. 由 scheduler 的 Run() 執行下一個可以執行的 thread 時則執行之（Run 的程式細節請參考 1-4-b），可以注意到真正交出 CPU 的動作是發生在 RUN()

     ```
     // returns when it's time for us to run
     kernel->scheduler->Run(nextThread, finishing);
     ```

▼ 1-1-b `Thread::StackAllocate()`

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));

#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;  // HP requires 64-byte frame marker
```

```
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
          // least 1 activation record
          // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4; // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif

#ifdef ALPHA
    stackTop = stack + StackSize - 8; // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif


#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4; // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif

#ifdef PARISC
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

- 功能：**處理新 thread 的 stack 初始化設定，主要包含 (1) stack allocation 以及 (2) 初始化 stack frame**。stack allocation 確保記憶體有一塊空間可以讓這個 thread 存 local variable、return address 或調用 function call，而 stack

frame 讓 thread 能執行在 ThreadRoot，並確保 function 執行完以後會呼叫 Thread::Finish()

- 傳入參數：
  - VoidFunctionPtr func：procedure to forked
  - void *arg：傳入 procedure 的參數
- 細節說明：
  - allocate the stack：

    ```
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    ```

    而 `AllocBoundedArray` 實作於 libs/sysdep.cc:

    ```
    char *
    AllocBoundedArray(int size)
    {
    #ifdef NO_MPROT
        return new char[size];
    #else
        int pgSize = getpagesize();
        char *ptr = new char[pgSize * 2 + size];

        mprotect(ptr, pgSize, 0);
        mprotect(ptr + pgSize + size, pgSize, 0);
        return ptr + pgSize;
    #endif
    }
    ```

    主要實現以下功能：

    1. call `getpagesize()` 得知一個 page 的大小
    2. 創造 2*page + 傳入 size（在此指 StackSize） 的空間
    3. 將頭尾兩個 page 的範圍設為不可以 read, write, execute (rwx) 的空間：相當於在前後加入兩個 page 用來保護中間的 stack，目的是為了偵測有無發生 stack overflow
    4. 最後 return 的位置為實際可用的 stack 起始位置
  - 接著，根據架構不同會直行不同區塊，我們是使用 x86，因此會執行：

    ```
    stackTop = stack + StackSize - 4; // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    ```

```
*stack = STACK_FENCEPOST;

machineState[PCState] = (void*)ThreadRoot;
machineState[StartupPCState] = (void*)ThreadBegin;
machineState[InitialPCState] = (void*)func;
machineState[InitialArgState] = (void*)arg;
machineState[WhenDonePCState] = (void*)ThreadFinish;
```

- stack 的記憶體生長方向為 high→low，所以先將 stackTop 移動到這塊記憶體的最高位置
- 把 ThreadRoot 放上 stackTop
- 將 stack 的底部放上 STACK_FENCEPOST (0xdeadbeef，定義於 thread.cc)，目的是偵測有無 overflow，此值為 stack 邊界
- 設定初始的 register，確保 thread 在正確的參數、狀態下執行

▼ 1-1-c `Thread::Finish()`

```
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);        // invokes SWITCH
    // not reached
}
```

- 功能：**Thread 結束時進行終止**。
- 細節說明：
  - 因為會使用到 Sleep() 這個 function，為了確保執行的 atomicity，要先將 interrupt 設為 disabled，並檢查是否為當前的 thread

    ```
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    ```

  - 調用 Sleep（詳細解釋於 1-1-a）：將會 invoke switch，將 CPU 歸還並切換成別的 thread 執行

▼ 1-1-d `Thread::Fork()`

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " "
<< arg);
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    // ReadyToRun assumes that interrupts are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

- 功能：為 thread 調用的 function，使用 StackAllocate 這個 function 宣告與初始化此 thread 的 stack，並將 thread 放入 ready queue。
- 傳入參數：
  - VoidFunctionPtr func：為 procedure 要同步運行的程式
  - void *arg：傳入 procedure 的參數
- 細節說明：
  - 宣告與初始化 stack（詳見 1-1-b）

    ```
    StackAllocate(func, arg);
    ```

  - 將 Interrupt 關閉（因為 ReadyToRun assume interrupt disabled）

    ```
    oldLevel = interrupt->SetLevel(IntOff);
    ```

  - 將 Thread 標記為「ready to run」（但實際還沒有開始），將 thread 放入 ready list

    ```
    scheduler->ReadyToRun(this); // this 即此 thread
    ```

  - 將 interrupt 恢復為原本的狀態：

```
(void) interrupt->SetLevel(oldLevel);
```

Explain Function 1-2

▼ 1-2-a `AddrSpace::AddrSpace()`

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
  pageTable[i].virtualPage = i; // for now, virt page # = phys page #
  pageTable[i].physicalPage = i;
  pageTable[i].valid = TRUE;
  pageTable[i].use = FALSE;
  pageTable[i].dirty = FALSE;
  pageTable[i].readOnly = FALSE;
    }

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

- 功能：建造出 address space 讓 user program 可以執行在上面，並設定 logical memory 和 physical memory 的 translation 【註】：在這邊 physical:logical memory 為 1:1 的 mapping，因為初始 nachos 設定只有一個 unsegmented page table，而且一次只 run 一個 thread
- 傳入參數：無
- 細節說明：
  - create 一個 page Table：`TranslationEntry` 這個資料結構實作於 translate.h，表示 page table 或 TLB 當中的一個 entry，每一個 entry 代表一組 physical ↔ logical memory 的對應

    ```
    pageTable = new TranslationEntry[NumPhysPages];
    ```

  - 因為是模擬 1-1 mapping，所以直接用迴圈將 virtual page 和 physical page 設為相同的值，接著是針對 extra bits 做設置：
    - valid bit = TRUE：表示此 page 在 該 process 的 logical address space 中

        - use bit = FALSE：由 HW 設置 → page 是否被參考（reference）或更改過（初始值為 FALSE）
        - dirty bit = FALSE：由 HW 設置 → page 是否被更改過（初始值為 FALSE）
        - readOnly = FALSE：page 的內容是否允許被更改（初始值為 FALSE）

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

    - 清空記憶體：給予被分配到此記憶體空間的 thread 準備乾淨的記憶體空間

```
bzero(kernel->machine->mainMemory, MemorySize);
```

▼ 1-2-b `AddrSpace::Execute()`

```
void
AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

    this->InitRegisters();    // set the initial register values
    this->RestoreState();   // load page table register

    kernel->machine->Run();   // jump to the user progam

    ASSERTNOTREACHED();     // machine->Run never returns;
          // the address space exits
          // by doing the syscall "exit"
}
```

- 功能：使用當前的 thread 執行 user program（called by `kernel::ForkExecute`）
- 傳入參數：fileName（t→getName()）

- 細節說明：
  - 將 kernel 當前 thread 的 space 設為現在這個 address space

    ```
    kernel->currentThread->space = this;
    ```

  - 初始化 CPU register、將 page table register load：通常是設定 program counter 到 program 或其他 register 的 resetting，

    ```
    this->InitRegisters();    // set the initial register values
    ```

  - Restore Address space state：將當前 page table address load 入 table register 中，確保 MMU 能使用自己 process 的 page table（確保執行期間 virtual 和 physical 的轉換）

    ```
    this->RestoreState();   // load page table register
    ```

  - 呼叫 machine::Run() 來執行已經 load 到 address space 的 program

    ```
    kernel->machine->Run();   // jump to the user progam
    ```

▼ 1-2-c `AddrSpace::Load()`

```
bool
AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;

    if (executable == NULL) {
  cerr << "Unable to open file " << fileName << "\n";
  return FALSE;
    }

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
    (WordToHost(noffH.noffMagic) == NOFFMAGIC))
      SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

#ifdef RDATA
// how big is address space?
```

```
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
                                                // we need to increase the size
            // to leave room for the stack
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
      + UserStackSize;  // we need to increase the size
            // to leave room for the stack
#endif
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    ASSERT(numPages <= NumPhysPages);   // check we're not trying
            // to run anything too big --
            // at least until we have
            // virtual memory

    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
  DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
      noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
  DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
      noffH.initData.size, noffH.initData.inFileAddr);
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
  DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.siz
e);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
      noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif

    delete executable;      // close file
    return TRUE;      // success
}
```

- 功能：將 file 中的 program load 入 memory 中（在這個過程就會進行 virtual address 和 physical address 的 translation）

- 傳入參數：fileName
- 細節說明：
  - 先將檔案用 filesystem 中的 open() 開啟可執行檔

```
OpenFile *executable = kernel->fileSystem->Open(fileName);
NoffHeader noffH;
unsigned int size;
```

  - 讀取 file 的 noff header：包含 code, data segment 等資訊

```
executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
```

  - 除錯 & 切換位元組順序（檢查是否為正確 noff format）

```
if ((noffH.noffMagic != NOFFMAGIC) &&
    (WordToHost(noffH.noffMagic) == NOFFMAGIC))
      SwapHeader(&noffH);
ASSERT(noffH.noffMagic == NOFFMAGIC);
```

    NOFFMAGIC 為一串定義好的位址（0xbadfad），用來標示 nachos object code file，如果發現 file 的 noffMagic 並非 NOFFMAGIC。另由於 Nachos 模擬的處理器是 big-endian，所以 host 的處理器為 little-endian，則使用 SwapHeader 將位元組順序做交換

  - 計算 address space 大小

```
#ifdef RDATA
// how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.siz
e + noffH.uninitData.size + UserStackSize;
 // we need to increase the size to leave room for the stack
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
+ UserStackSize;  // we need to increase the size
            // to leave room for the stack
#endif
```

  - 計算程式所需要的 page 數量，並確認 page 數量不超過實際記憶體的大小

```
numPages = divRoundUp(size, PageSize);
size = numPages * PageSize;
ASSERT(numPages <= NumPhysPages);
```

- 將 code、data segment 複製到 memory 中，最後關閉檔案，成功即回傳 TRUE

```
if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
  DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
      noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG(dbgAddr, "Initializing data segment.");
  DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.siz
e);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
      noffH.initData.size, noffH.initData.inFileAddr);
    }

#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        DEBUG(dbgAddr, "Initializing read only data segment.");
  DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonly
Data.size);
        executable->ReadAt(
    &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
      noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif
```

Explain Function 1-3

▼ 1-3-a `Kernel::Kernel()`

```
Kernel::Kernel(int argc, char **argv)
{
    randomSlice = FALSE;
    debugUserProg = FALSE;
    consoleIn = NULL;          // default is stdin
    consoleOut = NULL;         // default is stdout
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
    reliability = 1;            // network reliability, default is 1.0
```

```
    hostName = 0;               // machine id, also UNIX socket name
                                // 0 is the default machine id
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
        ASSERT(i + 1 < argc);
        RandomInit(atoi(argv[i + 1]));// initialize pseudo-random
      // number generator
        randomSlice = TRUE;
        i++;
        } else if (strcmp(argv[i], "-s") == 0) {
            debugUserProg = TRUE;
    } else if (strcmp(argv[i], "-e") == 0) {
          execfile[++execfileNum]= argv[++i];
      cout << execfile[execfileNum] << "\n";
    } else if (strcmp(argv[i], "-ci") == 0) {
        ASSERT(i + 1 < argc);
        consoleIn = argv[i + 1];
        i++;
    } else if (strcmp(argv[i], "-co") == 0) {
        ASSERT(i + 1 < argc);
        consoleOut = argv[i + 1];
        i++;
#ifndef FILESYS_STUB
    } else if (strcmp(argv[i], "-f") == 0) {
        formatFlag = TRUE;
#endif
        } else if (strcmp(argv[i], "-n") == 0) {
            ASSERT(i + 1 < argc);   // next argument is float
            reliability = atof(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-m") == 0) {
            ASSERT(i + 1 < argc);   // next argument is int
            hostName = atoi(argv[i + 1]);
            i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            cout << "Partial usage: nachos [-rs randomSeed]\n";
        cout << "Partial usage: nachos [-s]\n";
            cout << "Partial usage: nachos [-ci consoleIn] [-co consoleOut]\n";
#ifndef FILESYS_STUB
        cout << "Partial usage: nachos [-nf]\n";
#endif
            cout << "Partial usage: nachos [-n #] [-m #]\n";
    }
    }
}
```

- 功能：**parsing command line 的指令，並根據指令作出相應設定**。舉例來說，若 command line 指令為 nachos -e halt，execfile = [”nachos”, “-d”, “halt”]，而 halt 及會被放入 execfile 中，而 execfileNum（要被執行的 file 個數）也會隨之 +1
- 傳入參數：argc（參數個數）、argv（參數 array）

▼ 1-3-b `Kernel::ExecAll()`

```
void Kernel::ExecAll()
{
  for (int i=1;i<=execfileNum;i++) {
    int a = Exec(execfile[i]);
  }
  currentThread->Finish();
  //Kernel::Exec();
}
```

- 功能：將放入 execfile 內的檔案一一執行
- 細節說明：使用 for loop 呼叫 Exec() 執行 execfile 內的檔案（Exec 詳細內容請見下方 1-3-c），回傳的 a 為此刻 thread 的總數，並在執行結束後使用 Finish 結束 thread 的執行。

▼ 1-3-c `Kernel::Exec()`

```
int Kernel::Exec(char* name)
{
  t[threadNum] = new Thread(name, threadNum);
  t[threadNum]->space = new AddrSpace();
  t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
  threadNum++;

  return threadNum-1;
}
```

- 功能：初始化 thread 的 control block，並開一個 address space 給這個 thread，最後接回 1-1-d 的 Fork() 函式，宣告與初始化這個 thread 並放入 ready queue 中。
- 傳入參數：傳入 Thread() 這個函式的 name，為欲執行的 filename，用於 debug

▼ 1-3-d `Kernel::ForkExecute()`

```
void ForkExecute(Thread *t)
{
  if (!t->space->Load(t->getName())) {
      return;             // executable not found
    }
    t->space->Execute(t->getName());
}
```

- 功能：
  - Load a program into address space from file（file = t→getName()）
  - Load 成功後，則呼叫 address space 中的 Execute 執行此 program
- 傳入參數：thread
- 細節說明：
  - 把 program 從 file中 load 入 thread 的 address space，失敗（找不到可執行檔）則 return 掉

```
if (!t->space->Load(t->getName())) {
      return;             // executable not found
 }
```

  - 在 address space call Execute 執行 user program

```
t->space->Execute(t->getName());
```

Explain Function 1-4

▼ 1-4-a `Scheduler::ReadyToRun()`

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

- 功能：將 thread 放入 ready list，並標記為 Ready（call by Thread::Fork）
- 傳入參數：欲作標記的 thread
- 細節說明：
  - 確認 interrupt 已被 disabled

```
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

- 將 thread status 設為 READY（thread status 總共有三種：ready / running / blocked，定義於 thread.h）

```
thread->setStatus(READY);
```

- 將 thread 放入 ready list

```
readyList->Append(thread);
```

▼ 1-4-b `Scheduler::Run()`

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    // 紀錄當前的 thread，標示為 old thread
    Thread *oldThread = kernel->currentThread;
    // 檢查 interrupte 是否 disabled（原子性）
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    // 檢查傳入參數 finishing，如果為 true（表示 thread 已經執行完）
    // 將 toBeDestroyed 設為 oldThread
    if (finishing) {  // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
   toBeDestroyed = oldThread;
    }

    // 如果 thread 此時在執行 user program，則存入 user's registers
    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();   // save the user's CPU registers
        oldThread->space->SaveState();
    }

    // 確認是否有 undetected stack overflow
    oldThread->CheckOverflow();       // check if the old thread
              // had an undetected stack overflow

    // kernel 將「當前 thread 指標」指向 nextThread，並將狀態設為 running
    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << next
Thread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".
    // 將 oldThread 替換為 nextThread
    SWITCH(oldThread, nextThread);
```

```
    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();   // check if thread we were running
          // before this one has finished
          // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();     // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

- 功能：執行 context switch，包括保存當前 thread 資訊、將欲執行的 thread 改為 running 狀態
- 傳入參數：欲切換執行的 thread、當前 thread 是否已經 finish
- 細節說明：
  - 紀錄當前的 thread：標示為 old thread，並檢查 interrupte 是否 disabled（原子性）

    ```
    Thread *oldThread = kernel->currentThread;
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    ```

  - 檢查傳入參數 finishing，如果為 true（表示 thread 已經執行完），將 toBeDestroyed 設為 oldThread（表示 oldThread 要被 destruct 掉）

    ```
    if (finishing) {  // mark that we need to delete current thread
             ASSERT(toBeDestroyed == NULL);
       toBeDestroyed = oldThread;
    }
    ```

  - 保存當前 thread 狀態（存入 user register），並確認是否有 undetected stack overflow

    ```
     if (oldThread->space != NULL) {  // if this thread is a user program,
            oldThread->SaveUserState();   // save the user's CPU registers
            oldThread->space->SaveState();
    ```

```
    }
 oldThread->CheckOverflow();
```

- 進行 thread 切換：將 kernel 的 currentThread 換成新的 thread，並將這個 thread 的狀態改為 running，最後執行 thread 切換

```
kernel->currentThread = nextThread;  // switch to the next thread
nextThread->setStatus(RUNNING);      // nextThread is now running
SWITCH(oldThread, nextThread);
```

- 等新的 thread 執行完，確認是否切回原本的 thread

```
CheckToBeDestroyed();   // check if thread we were running
      // before this one has finished
      // and needs to be cleaned up

if (oldThread->space != NULL) {     // if there is an address space
    oldThread->RestoreUserState();     // to restore, do it.
    oldThread->space->RestoreState();
}
```

> Explain how NachOS creates a thread (process), load it into memory and place it into the
> scheduling queue as requested in the Trace code part.
> Your explanation on the functions along the code path should at least cover answer for the questions below:

1. How does Nachos allocate the memory space for a new thread(process)?
   - Nachos 在 thread 建立後（透過 `Thread()` 這個 function ），這個 thread 會透過 `AddrSpace()` 分配此 thread 可以使用的記憶體空間。
   - 整體流程：

```
t[threadNum] = new Thread(name, threadNum);
```

```
t[threadNum]->space = new AddrSpace();
```

2. How does Nachos initialize the memory content of a thread(process), including loading
   the user binary code in the memory?

   在 1-2-a 的 `AddrSpace::AddrSpace()` 和 1-2-c 的 `AddrSpace::Load()` ，Nachos 會執行記憶體初始化以及把 binary code load 進 memory 的動作。

   - AddrSpace() 中會 create 該 process 的 Page table 並將被分配的記憶體（在原始 Nachos 設定上就是全部記憶體）做清零，確保 program 有乾淨的空間可以使用

     ```
     bzero(kernel->machine->mainMemory, MemorySize);
     ```

   - Load() 會計算可執行檔的大小，配置相對應數量的 page，並將可執行檔中的 data, code, readonlyData (if any) load 進 Memory 中

3. How does Nachos create and manage the page table?

   a. Create：在 1-2-a 的 `AddrSpace::AddrSpace()` 使用 TranslationEntry 這個資料結構建構有 NumPhyPages 個 Page 的 page table（實際上其實不需要這麼多，只是 nachos 初始設定上是 uniprogramming 所以會把全部 frame 都送給單一個 process 使用，這在實作時也會進行修改）

      ```
      pageTable = new TranslationEntry[NumPhysPages];
      ```

   b. manage：在 create page table 以後，`AddrSpace::AddrSpace()` 會用一個 for loop 去設定這個 page table 裡面的所有 page，執行相關 bit 的設定（valid bit, use bit, dirty bit 以及 Readonly bit）

4. How does Nachos translate addresses?

   a. 在原本的 code 中，因為 Nachos 是直接開讓 physical memory 和 logical memory 有 1:1 mapping，換言之，我們其實不用處理 translate address 的問題。

   b. 但考量到實際需要轉換的情形（比如 MP2 的實作），Nachos 必須要在每次收到 virtual address 時都能正確地轉換成 physical address，才能讀到正確的資料或正確地取值。可以觀察到 Machine 在利用 OneInstruction() 模擬執行 user program 時，裡面有一段：

      ```
      if (!ReadMem(registers[PCReg], 4, &raw)) return; // exception occurred
      ```

在 ReadMem() 中會 call Machine::Translate()：這一步就是在檢查 address 的轉換是否正確，如果在這一步 translate 的位置出問題（比如取到 invalid page 或者寫到 readonly page），則程式會執行相應的 Exception Handle

```
bool
Machine::ReadMem(int addr, int size, int *value)
{
    int data;
    ExceptionType exception;
    int physicalAddress;

    DEBUG(dbgAddr, "Reading VA " << addr << ", size " << size);

    exception = Translate(addr, &physicalAddress, size, FALSE);
...
```

5. How Nachos initializes the machine status (registers, etc) before running a thread(process)

   在 1-2-b `AddrSpace::Execute()` 可以看到 thread 的 addrSpace 會先初始化 CPU register 並 load 進 page table register 以後才開始執行程式

```
this->InitRegisters();    // set the initial register values
this->RestoreState();   // load page table register

kernel->machine->Run();   // jump to the user progam
```

6. Which object in Nachos acts the role of process control block

   可以從 Thread 這個 Class 中看出它是 process control block。其中包括 thread operation function 以及保存並維護 thread (process) 的必要資訊（如 register 等）

```
class Thread {
  private:
    int *stackTop;       // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

  public:
    Thread(char* debugName, int threadID);    // initialize a Thread
    ~Thread();        // deallocate a Thread
          // NOTE -- thread being deleted
          // must not be running when delete
          // is called

    // basic thread operations
    void Fork(VoidFunctionPtr func, void *arg);
```

```
            // Make thread run (*func)(arg)
    void Yield();     // Relinquish the CPU if any
        // other thread is runnable
    void Sleep(bool finishing); // Put the thread to sleep and
        // relinquish the processor
    void Begin();   // Startup code for the thread
    void Finish();      // The thread is done executing

    void CheckOverflow();     // Check if thread stack has overflowed
    void setStatus(ThreadStatus st) { status = st; }
    ThreadStatus getStatus() { return (status); }
  char* getName() { return (name); }

  int getID() { return (ID); }
    void Print() { cout << name; }
    void SelfTest();    // test whether thread impl is working

  private:
    // some of the private data for this class is listed above

    int *stack;     // Bottom of the stack
        // NULL if this is the main thread
        // (If NULL, don't deallocate stack)
    ThreadStatus status;  // ready, running or blocked
    char* name;
  int   ID;
    void StackAllocate(VoidFunctionPtr func, void *arg);
            // Allocate a stack for thread.
        // Used internally by Fork()

// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs];  // user-level CPU register state

  public:
    void SaveUserState();   // save user-level register state
    void RestoreUserState();    // restore user-level register state

    AddrSpace *space;     // User code this thread is running.
};
```

7. When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?

   當 kernel::Exec() create 好 thread、並且也準備好這個 thread 的 address space 以後，就可以透過 1-1-d `Thread::Fork()` allocate process 的 stack 並將此 thread 放入 ready queue

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
```

```
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    // ReadyToRun assumes that interrupts are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

8. Please look at the following code from urserprog/exception.cc and answer the question:

```
case SC_MSG:
    DEBUG(dbgSys, "Message received.\n");
    val = kernel->machine->ReadRegister(4);
    {
    char *msg = &(kernel->machine->mainMemory[val]);
    cout << msg << endl;
    }
    SysHalt();
    ASSERTNOTREACHED();
```

   According to the code above, please explain under what circumstances an error will occur if the message size is larger than one page and why? (Hint: Consider the relationship between physical pages and virtual pages.)

   因為我們是使用 virtual memory 進行紀錄，但實際上會需要轉換成 physical memory 以後才能讀取 memory，所以如果一個 message 超過一個 page size，雖然在 virtual 的位置上連續，但有可能 map 到 physical address 時是不連續的 frame，此時直接用 &(kernel->machine->mainMemory[val]) 有可能會取到不屬於自己 process 的 memory 範圍，就會產生 segmentation fault 或者讀到錯誤的值。

Part II. Implementation

> Modify its memory management code to make NachOS support

Verification 2-1

```
[os23team47@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

Implementation：

在尚未更改 file 以前，我們的 memory management 僅支援 uniprogramming，這是因為我們在使用 `AddrSpace::AddrSpace()` 為 user program 開 address space 時，所開的 page Table 大小直接是 physical memory 大小（single unsegmented table），等於一隻 user program 就佔滿了所有記憶體空間，但其實一隻程式並不需要用到這麼多空間。

- 觀察原本的 AddrSpace()：可以發現是強行開滿所有記憶體空間

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i; // for now, virt page # = phys page #
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
  }

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

因此我們的實作方式是改為在 Load 的時候再去處理分配 address space 的問題：

- 先在 addrspace.h 中觀察 AddrSpace 這個 class 結構，並在 public 多宣告兩個變數：
    1. usedPhyPag：紀錄 memory 的使用狀況（檢查這個 frame 是否已經被使用，初始值為 false）
    2. NumFreePages：紀錄此刻還可以使用的 page number（frame number）

```
class AddrSpace {
  public:
    AddrSpace();      // Create an address space.
    ~AddrSpace();     // De-allocate an address space

    bool Load(char *fileName);    // Load a program into addr space from
                                        // a file
          // return false if not found

    void Execute(char *fileName);               // Run a program
          // assumes the program has already
                                        // been loaded

    void SaveState();     // Save/restore address space-specific
    void RestoreState();    // info on a context switch

    // Translate virtual address _vaddr_
    // to physical address _paddr_. _mode_
    // is 0 for Read, 1 for Write.
    ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mod
e);
    static bool usedPhyPag[NumPhysPages];
    static int NumFreePages;
  private:
    TranslationEntry *pageTable;  // Assume linear page table translation
          // for now!
    unsigned int numPages;    // Number of pages in the virtual
          // address space
    void InitRegisters();   // Initialize user-level CPU registers,
          // before jumping to user code

};
```

- 接著，到 addrspcae.cc 中宣告這兩個變數的初始值：

```
bool AddrSpace::usedPhyPag[NumPhysPages] = {false};
int AddrSpace::NumFreePages = NumPhysPages;
```

- 由於我們希望是「知道 program 具體需要幾個 page（frame）」後再進行 page table 分配，因此我們把這個 create address space 的動作 delay 到 Load() 時再做，所以我們先把原本的 addr space 先註解掉：

```
AddrSpace::AddrSpace()
{
    // pageTable = new TranslationEntry[NumPhysPages];
    // for (int i = 0; i < NumPhysPages; i++) {
  // pageTable[i].virtualPage = i;  // for now, virt page # = phys page #
  // pageTable[i].physicalPage = i;
  // pageTable[i].valid = TRUE;
```

```
  // pageTable[i].use = FALSE;
  // pageTable[i].dirty = FALSE;
  // pageTable[i].readOnly = FALSE;
    // }

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

- 接著可以看到 Load() 中有一段是透過 size 計算 program 需要的 page 數量：

```
#ifdef RDATA
// how big is address space?
    size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
           noffH.uninitData.size + UserStackSize;
#else
// how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
      + UserStackSize;
#endif

numPages = divRoundUp(size, PageSize); // 計算所需要的 page 數量
```

- 我們把分配 frame 的動作延遲到 numPages 計算出來才做，並使用我們宣告的 usedPhyPag 去檢查當前可用的 frame 在哪裡，從可用的 frame 開始進行分配。
  - 先使用 numPages create 出一個 pageTable
  - 接著在設定 pageTable 的 for 迴圈中，每次要分配 frame 以前，都先用 while loop 確認此刻找到的 frame 是 non-used frame（即 usedPhyPag[curr_idx] == false），才放心進行分配
  - 找到 free frame 時，將 `AddrSpace::usedPhyPag[curr_idx]` 設定為 true 以告知此 OS 此 frame 已經是被分配到的狀態，並將 `AddrSpace::NumFreePages` 減一
  - 接著清空這一塊記憶體
  - 設定 page table bit

```
// 分配記憶體
pageTable = new TranslationEntry[numPages];
  for(unsigned int i = 0, j = 0; i < numPages; i++){
      pageTable[i].virtualPage = i;
      while(j < NumPhysPages && AddrSpace::usedPhyPag[j] == true) j++;
      AddrSpace::usedPhyPag[j] = true;
      AddrSpace::NumFreePages--;
      // 清空即將分配的 page
      bzero(&kernel->machine->mainMemory[j*PageSize], PageSize);
      pageTable[i].physicalPage = j;
```

```
        pageTable[i].valid = true;
        pageTable[i].use = false;
        pageTable[i].dirty = false;
        pageTable[i].readOnly = false;
    }
```

- 接著，我們就要去更改 Load 下方 copy data 和 code 的位置，這邊以 code 為例，原本的讀法為（這是因為在原本的設定裡，virtual memory 跟 physical memory 位置相同，所以在讀取的時後不用考慮 page Table 的位置轉換）：

```
if (noffH.code.size > 0) {
      executable->ReadAt(
     &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
        noffH.code.size, noffH.code.inFileAddr);
  }
```

  我們要將讀取記憶體的位置改為：

```
if (noffH.code.size > 0) {
      executable->ReadAt(
     &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physi
calPage * PageSize + (noffH.code.virtualAddr%PageSize)]),
        noffH.code.size, noffH.code.inFileAddr);
  }
```

  可以看到放入位置變成 mainMemory 中的 `pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize + (noffH.code.virtualAddr%PageSize)`

  這是 mapping 過後的位置，計算方式為：

  1. virtual address / PageSize = 第幾個 page，當作是查找 PageTable 的 index
  2. 取到 PageTable 中相對應的 physical Page 後，乘上每個 Page 的大小（128，設定於 machine.h 中），得到在該 physical Page 的記憶體位置
  3. 用 virtual address % PageSize 計算出原本在 virtual page 內中的偏移，把相同的值加上計算出來的 physical page 位置，以便取得在 physical page 內，真正存放 code 的位置

- 以上為 code 的實作，接著在 initData 和 readonlyData 也如法炮製即可
- 做完以上步驟我們就已經基本完成 multiprogramming 的記憶體分配，接著要做的事情就是在執行結束時釋放掉 physical frame，後面的程式才能使用：

```
AddrSpace::~AddrSpace()
{
   for(int i = 0; i < numPages; i++){
    AddrSpace::usedPhyPag[pageTable[i].physicalPage] = false;
    AddrSpace::NumFreePages++;
   }
   delete pageTable;
}
```

Verification 2-2 MemoryLimitException：handle frame 不足的情形

```
[os23team47@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test3
consoleIO_test1
consoleIO_test3
9Unexpected user mode exception 8
Assertion failed: line 209 file ../userprog/exception.cc
Aborted
```

- 從 2-1 紀錄的 NumFreePage 可以得知一個 program 大小為 12 個 page，因此我們為了模擬 frame 不足而無法 multiprogramming 的問題，先在 machine.h 裡面改 NumPhysPages = 14（可以執行一個 thread、但無法同時執行兩個 thread 的情形）
- 因應題目要求，先在 machine.h 的 ExceptionType 中新增一個 MemoryLimitException：

```
enum ExceptionType { NoException,           // Everything ok!
         SyscallException,      // A program executed a system call.
         PageFaultException,    // No valid translation found
         ReadOnlyException,     // Write attempted to page marked
              // "read-only"
         BusErrorException,     // Translation resulted in an
              // invalid physical address
         AddressErrorException, // Unaligned reference or one that
              // was beyond the end of the
              // address space
         OverflowException,     // Integer overflow in add or sub.
         IllegalInstrException, // Unimplemented or reserved instr.
       MemoryLimitException, // 新增這個！

         NumExceptionTypes
};
```

- 接著，我們要在 Load() 時去檢查當需要的 #page 已經大於剩下的 free frame 時，就呼叫 Exception Handler，並在 exception 以前先 free resources：

```
if (numPages > AddrSpace::NumFreePages){
  this->~AddrSpace();
  ExceptionHandler(MemoryLimitException);
}
```