OS MP2

Trace Code

Kernel::Kernel()

File: `threads/thread.cc`

> path: main() → Kernel::Kernel()

```
Kernel::Kernel(int argc, char **argv) {
    randomSlice = FALSE;
    debugUserProg = FALSE;
    execExit = 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], "-ee") == 0) {
            // Added by @dasbd72
            // To end the program after all the threads are done
            execExit = TRUE;
        } ...
    }
}
```

在 `main.cc` 中會呼叫 `Kernel` 的建構子，使用從 command line 讀取到的 `argc` `argv` 當參數創建 kernel。`randomSlice` 設定成 `FALSE` 表示兩次 context switch 之間的時間固定， `execExit` 設為 `TRUE` 時，執行完所有 threads 系統會自動停止運作。`for` 迴圈內會處理從 command line 拿到的各項參數。

---

void Kernel::ExecAll()

File: `threads/kernel.cc`

> path: Kernel::ExecAll()

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

把從指令讀進來的所有檔名都跑一次，跑完後使用 `Finish()` 結束 thread。

---

int Kernel::Exec()

File: `threads/kernel.cc`

> path: Kernel::ExecAll() → Kernel::Exec()

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

    return threadNum - 1;
}
```

這裡主要是在做建構 thread 的前置作業並維護 threads table `t`，會回傳 thread 的數量。

Thread::Thread()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → Thread::Thread()

```
Thread::Thread(char *threadName, int threadID) {
    ID = threadID;
    name = threadName;
    isExec = false;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
        machineState[i] = NULL;  // not strictly necessary, since
```

```
                                        // new thread ignores contents
                                        // of machine registers
    }
    space = NULL;
}
```

這是 Thread 的建構子，其中 `isExec` 預設為 `false` 表示還沒被執行，而 `stack` 預設為 `NULL` 表示尚未 allocate stack。隨後的 `t[threadNum]->setIsExec();` 會再將這個 thread 設為正在執行。

AddrSpace::AddrSpace()

File: `userprog/addrspace.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → 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);
}
```

`t[threadNum]->space = new AddrSpace();` 會開一個跟 physical memory size 一樣大的 virtual memory, 並新建一個 page table，並用 bzero 將空間的值都初始化成 0。

void Thread::Fork()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → 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);
    }
```

`Fork()` 會呼叫 `StackAllocate()` 去建構 stack，之後會把 interrupt 關掉，並呼叫 `ReadyToRun()` 把這個 thread 放進 ready queue 等待被執行。

void Thread::StackAllocate()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → Thread::Fork() → Thread::StackAllocate()

```
void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
    stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
#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
    machineState[PCState] = (void *)ThreadRoot;
    machineState[StartupPCState] = (void *)ThreadBegin;
    machineState[InitialPCState] = (void *)func;
    machineState[InitialArgState] = (void *)arg;
    machineState[WhenDonePCState] = (void *)ThreadFinish;
}
```

這裡會呼叫 `AllocBoundedArray()` 來創建一個stack，最後將所有參數放進 `machineState[]` 對應的 register 裡面。(`Switch.S` 會用到)

char *AllocBoundedArray()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → Thread::Fork() →Thread::StackAllocate() → AllocBoundedArray()

```
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
}
```

若不需要記憶體保護機制的話，`AllocBoundedArray()` 會直接開一個要求大小的記憶體空間必回傳。若需要記憶體保護的話，`AllocBoundedArray()` 會再要求的 stack 兩端加上一段距離的空白記憶體，並使用 `mprotect()` 來確保那兩段空白的記憶體不能被 access。

void Scheduler::ReadyToRun()

File: `threads/scheduler.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → Thread::Fork() → Scheduler::ReadyToRun()

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

把該條 thread 狀態設定成 `READY` 並且 append 到 `readyList` 的最後面。

void ForkExecute()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → ForkExecute()

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

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

`Fork()` 的第一項參數 `ForkExecute` 會把 thread 對應到的 files load 進 memory 裡面，並確保有成功 load 之後再呼叫 `t->space->Execute(t->getName());` 執行 thread。

bool AddrSpace::Load()

File: `userprog/addrspace.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → ForkExecute() → 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;
    }
```

一開始會用 `kernel->fileSystem->Open()` 去開檔案，若無法開啟則回傳 `FALSE` 表示 load file 失敗。

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

若有成功開啟檔案的話，則呼叫 `executable->ReadAt()` 去讀取開啟的 file Noff header 的部分。`noffH.noffMagic != NOFFMAGIC` 是用來檢查目前與本地的 byte order (little endian, big endian) 是否一致，不一致的話則會用 `WordToHost()` 去調整成對應的型式，並且再確認一次是否相符，接著使用 `SwapHeader()` 將 header 的 byte order 換成正確的格式，最後用 `ASSERT()` 再確保一次增加可靠度。

```
#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
```

計算所需空間 `size` ，需要加上 `UserStackSize` 留空間給 stack，預設大小是 1024。

```
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;
    ASSERT(numPages <= NumPhysPages);
    DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);
```

`numPages = divRoundUp(size, PageSize);` 計算 pages 的數量，`divRoundUp()` 是取上界，算完 pages 的數量後將 size 調整為 `PageSize` 的整數倍，並且檢查 Pages 數量是否超過 `NumPhysPages`，防止 load 到過大的程式。

```
    // 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.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
#endif
```

再來會用 `ReadAt()` 將 code 和 data segments load 進 memory。

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

最後關閉 files 並且回傳 `True` 表示成功將程式 load 進 memory。

void AddrSpace::Execute()

File: `userprog/addrspace.cc`

> path: Kernel::ExecAll() → Kernel::Exec() → ForkExecute() → 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();
    }
```

`kernel->currentThread->space = this;` 會確保 kernel 有讀取到對的 page table，理論上其他地方的 code 沒有問題的話，這一行可以被省略，但是 kernel 必須確保不能有任何錯誤，因此會加上這行來增加 kernel 的可靠性。接著會將原本放在 state register 的 state restore 回來，呼叫 `kernel->machine->Run()` 開始執行這個 thread。

---

void Thread::Finish()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Thread::Finish()

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    if (kernel->execExit && this->getIsExec()) {
        kernel->execRunningNum--;
        if (kernel->execRunningNum == 0) {
            kernel->interrupt->Halt();
        }
    }
    Sleep(TRUE);  // invokes SWITCH
    // not reached
}
```

一開始先將 interrupt 關掉，並且確保 kernel 有跑在目前的 thread 上面。 `kernel->execExit` 為 `true` 的話表示所有 threads 執行完後會把該 program 停止。當所有 program 停止時，則呼叫 `kernel->interrupt->Halt()` 讓系統停止。但若還有其他 threads 或 program 要跑，則會呼叫 `Sleep()` 凍結這個 thread 以讓出 CPU 的資源。

void Thread::Sleep()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Thread::Finish() → Thread::Sleep()

```
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 << ", "
          << kernel->stats->totalTicks);

    status = BLOCKED;
    // cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    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);
}
```

當 thread 因為某些原因需要等待的話，會執行這個 function 讓出 CPU 資源。一開始的兩個 `ASSERT` 會確保要睡眠的 thread 是目前這條 thread， 並且 interrupt 是被關閉的。`status = BLOCKED;` 會將此 thread 設為不能被調用。接著使用 `while` 呼叫 `FindNextRun()` 去尋找有沒有其他 thread 可以被執行，若沒有則會呼叫 `kernel->interrupt->Idle()` 讓 CPU 進入閒置狀態，若有找到則會呼叫 `kernel->scheduler->Run()` 去執行下一個 thread。

Thread* Scheduler::FindNextToRun()

File: `threads/thread.cc`

> path: Kernel::ExecAll() → Thread::Finish() → Thread::Sleep() → Scheduler::FindNextToRun()

```
Thread* Scheduler::FindNextToRun() {
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

如果 `readyList` 裡面沒有待執行的 thread，則回傳 NULL，若有的話則會呼叫 `readyList->RemoveFront()` 回傳下一個要被執行的 thread 並且從 `readyList` 把他 pop 掉。

void Interrupt::Idle()

File: `machine/interrupt.cc`

> path: Kernel::ExecAll() → Thread::Finish() → Thread::Sleep() → Interrupt::Idle()

```
void Interrupt::Idle() {
    status = IdleMode;
    if (CheckIfDue(TRUE)) {  // check for any pending interrupts
        status = SystemMode;
```

```
            return;  // return in case there's now a runnable thread
        }
        DEBUG(dbgInt, "Machine idle.  No interrupts to do.");
        cout << "No threads ready or runnable, and no pending interrupts.\n";
        cout << "Assuming the program completed.\n";
        Halt();
    }
```

若沒有 interrupt 且沒有 thread 可以被執行，則 `Idle()` 會假設程式執行完成並呼叫 `Halt()` 停止運作。

void Scheduler::Run()

File: `threads/scheduler.cc`

> path: Kernel::ExecAll() → Thread::Finish() → Thread::Sleep() → Scheduler::Run()

這個 function 主要是在做 thread 交換執行的步驟。

```
void Scheduler::Run()Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;

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

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

一開始一樣會先確保 interrupt 被關閉了，若傳進來的 `finishing` 為 `True`，則把舊的 thread 記在 `toBeDestroyed`。

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

    oldThread->CheckOverflow();  // check if the old thread
                                 // had an undetected stack overflow
```

接著會將當前舊 thread 的 state 存到 thread 自己的 register 以便後續繼續恢復 state 時使用。`oldThread->CheckOverflow();` 會檢查舊 thread 有無 overflow (x86 會檢查 `stack` 是否指到 `STACK_FENCEPOST`)。

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

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

```
    SWITCH(oldThread, nextThread);

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

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

`kernel->currentThread = nextThread;` `nextThread->setStatus(RUNNING);` 會將 kernel 自己記錄的資料更新到執行新的 thread 的狀態，更新後去呼叫 `SWITCH()` 執行實際的 thread 切換。`SWITCH()` return 回來之後，代表切換回舊的的 thread。

```
    CheckToBeDestroyed();

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

`CheckToBeDestroyed()` 會檢查 `toBeDestroyed` 是否為 `NULL` 若有的話則用 delete 釋放記憶體。最後把一開始存在舊 thread register 裡面的 state 寫回 machine 的 register。

SWITCH()

File: `threads/switch.S`

> path: Kernel::ExecAll() → Thread::Finish() → Thread::Sleep() → Scheduler::Run() → SWITCH()

呼叫 `SWITCH()` 之後，在 switch.S 裡面會根據不同的 ISA 有不同的指令，不果大致上都是將舊 thread 的資料存回記憶體，並將新 thread 的資料 load 進 memory。

Implement page table in NachOS