CS342301: Operating System

MP3: CPU Scheduling

Team31

111060013 劉祐廷 111060019 黃子恩

	Contribution
劉祐廷	1. Trace code in Part I and write the report for it. 2. Implement code and write the report for part II. 3. Discuss and complete the entire report.
黃子恩	1. Trace code in Part I and write the report for it. 2. Implement code and write the report for part II. 3. Discuss and complete the entire report.

OS MP3

Trace Code

New → Ready

Kernel::ExecAll()

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

在 main thread 的時候，用迴圈呼叫 `Exec()` 去將每個 thread 造出來，接著結束 main thread。

Kernel::Exec(char*)

```
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;
}
```

`t[threadNum]->setIsExec();` 將建好的 thread 的 `isExec()` 設成 `true`。

Thread::Fork(VoidFunctionPtr, void*)

```
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);
    }
```

先用 `StackAllocate` 將空間開好，再呼叫 `ReadyToRun()` 將 thread 的 status 設定成 `READY`，並 append 到 ready queue 裡面。

Thread::StackAllocate(VoidFunctionPtr, void*)

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

在 `AllocBoundedArray()` 裡面，會將記憶體空間開好，並且在該記憶體前後加上一段不可被 access 的空白記憶體安全區，接著把 `stackTop` 設定成 `stack + StackSize - 4`，空出一個 int 的長度放 `STACK_FENCEPOST`，之後 check stack overflow 會用到。

Scheduler::ReadyToRun(Thread*)

```
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);
}
```

將 thread 的 status 設定成 `Ready`，並 append 到 ready queue 裡面。

Running → Ready

Machine::Run() / Interrupt::OneTick()

在 `Machine::Run()` 裡面會用無窮迴圈呼叫 `OneInstruction()` 和 `OneTick()` 模擬 CPU，在 `OneTick()` 裡面會檢查各種條件，如果要執行 context switch 的話會去呼叫 `Yield()`。

```
void Thread::Yield() {
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

呼叫 `FindNextToRun()` 找有沒有下一個 thread 可以做，如果有的話就會呼叫 `ReadyToRun(this)` 將目前正在執行的 thread status 從 `RUNNING` 改回 `READY`，之後呼叫 `Run()` 做 context switch 去執行下一條 thread。

Scheduler::FindNextToRun()

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

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

確認 ready queue 裡面有無 thread 等待被執行，有的話就把他從 ready queue 裡面拿出來。

Scheduler::Run(Thread*, bool)

這個 function 主要是在做 thread 交換執行的步驟，並將新的要執行的 thread status 設成 `RUNNING`。

```
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。

```
    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` (原本的 `oldThread`) 是否為 `NULL`，若不是 `NULL` 的話則用 delete 釋放記憶體。最後把一開始存在要執行的 thread register 裡面的 state 寫回 machine 的 register。

`oldThread->space->RestoreState();` 會把 page table 和 page 數量切換成要執行的 thread 的資料。

Running → Waiting

SynchConsoleOutput::PutChar(char)

```
void SynchConsoleOutput::PutChar(char ch) {
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

呼叫 `Acquire()` 取得鎖，以防在輸出的時候遇到被打斷的問題，接著呼叫 `PutChar()` 將輸出字這個動作放進 console output 的 waiting list 裡面後，呼叫 `P()` 請求並等待資源可用，最後做完要呼叫 `Release()` 將鎖釋放。

Semaphore::P()

```
void Semaphore::P() {
    DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {                // semaphore not available
        queue->Append(currentThread);   // so go to sleep
        currentThread->Sleep(FALSE);
    }
    value--;  // semaphore available, consume its value

    // re-enable interrupts
    (void)interrupt->SetLevel(oldLevel);
}
```

對 `Lock` 來說，`value` = 1 表示解鎖，`value` = 0 表示上鎖，因此可以看到在這裡當 `value` = 0 的時候必須呼叫 `Sleep()` 讓 thread 先去 wait，而對 `SynchConsoleOutput` 來說，`value` = 1 表示輸出完成 (用 callback function 讓輸出完成時 `value` 變成 1)，`value` = 0 表示還沒輸出完成，因此也是 `value` = 0 時要呼叫 `Sleep()`。

List<T>::Append(T)

將請求資源的 thread 放進該設備的 waiting list。

Thread::Sleep(bool)

`Sleep()` 會將當前等不到資源的 thread status 設定成 `BLOCKED` (waiting)，並且呼叫 `Scheduler::FindNextToRun()` 和 `Scheduler::Run(Thread*, bool)`，去找下一條 thread 或是 interrupt 來做。

Waiting → Ready

Semaphore::V()

```
void Semaphore::V() {
    DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;

    // re-enable interrupts
    (void)interrupt->SetLevel(oldLevel);
}
```

Interrupt 處理完，釋放資源並將 `value` 加一，把 waiting list 裡面的 thread pop 出來，呼叫 `ReadyToRun()` 將他放進 ready queue 並把該 thread 的 `status` 設成 `READY`。

Running → Terminated

ExceptionHandler(ExceptionType) case SC_Exit

```
case SC_Exit:
    DEBUG(dbgAddr, "Program exit\n");
    val = kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
```

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
    }
```

這裡會呼叫 `Sleep(TRUE)`，在 `Sleep()` 裡面 `status` 會被設定成 `BLOCKED`，並且呼叫 `FindNextToRun()` 去找下一個要做的 thread，接著進入 `Scheduler::Run()` 將新的 thread 設定成 `RUNNING` 並將舊的 thread status destroy 掉。

Ready → Running

在 `FindNextToRun()` 裡面尋找下一個可用的 thread，進到 `Run()` 之後會將新的要執行的 thread status 設定成 `RUNNING`，並透過 `SWITCH()` 做 context switch。

SWITCH()

呼叫 `SWITCH()` 之後，在 `switch.S` 裡面會根據不同的 ISA 有不同的指令，目的是實際去切換 thread。

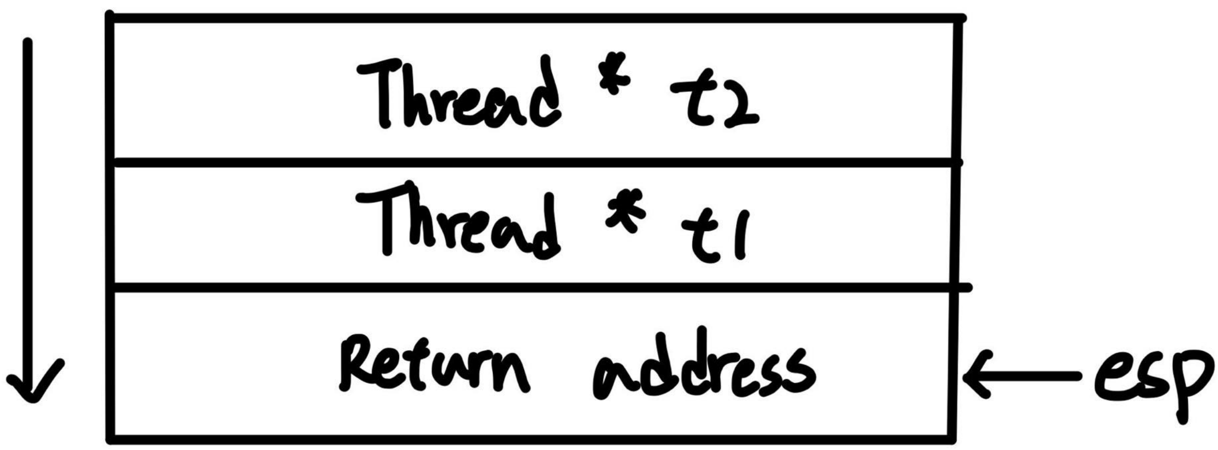
如果是切換到一條 new thread，那 `SWITCH()` 後會跳到新 thread 的 thread 接著呼叫 thread begin，接著做 `ForkExecute()`，從 `AddrSpace::Execute()` 裡面回到 `machine::Run()`。

如果是切換到一條從 running 被放回 ready queue 的 thread，則會把 `Scheduler::Run()` 裡面的 `SWITCH()` 後面的部分繼續做完，接著一層一層 return 回去 (-> `Yield()` -> `OneTick()` -> `Run()`)，回到 `machine::Run()` 的 for loop。

如果是切換到一條從 waiting 被放回 ready queue 的 thread，那麼把 `Scheduler::Run()` 裡面的 `SWITCH()` 後面的部分繼續做完後，會一步一步回到 `ExceptionHandler()` 繼續處理 exception，處理完後會跳回 `RaiseException()`，然後回到 `OneInstruction()` 再回到 `Machine::Run()` 的 for loop。

switch.S

一開始stack的狀態為



SWITCH

```
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
```

先將 `%eax` 的值存到 `_eax_save`，將 `%eax` 的值改為 `%esp + 4`，也就是 Thread t1。

```
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage
```

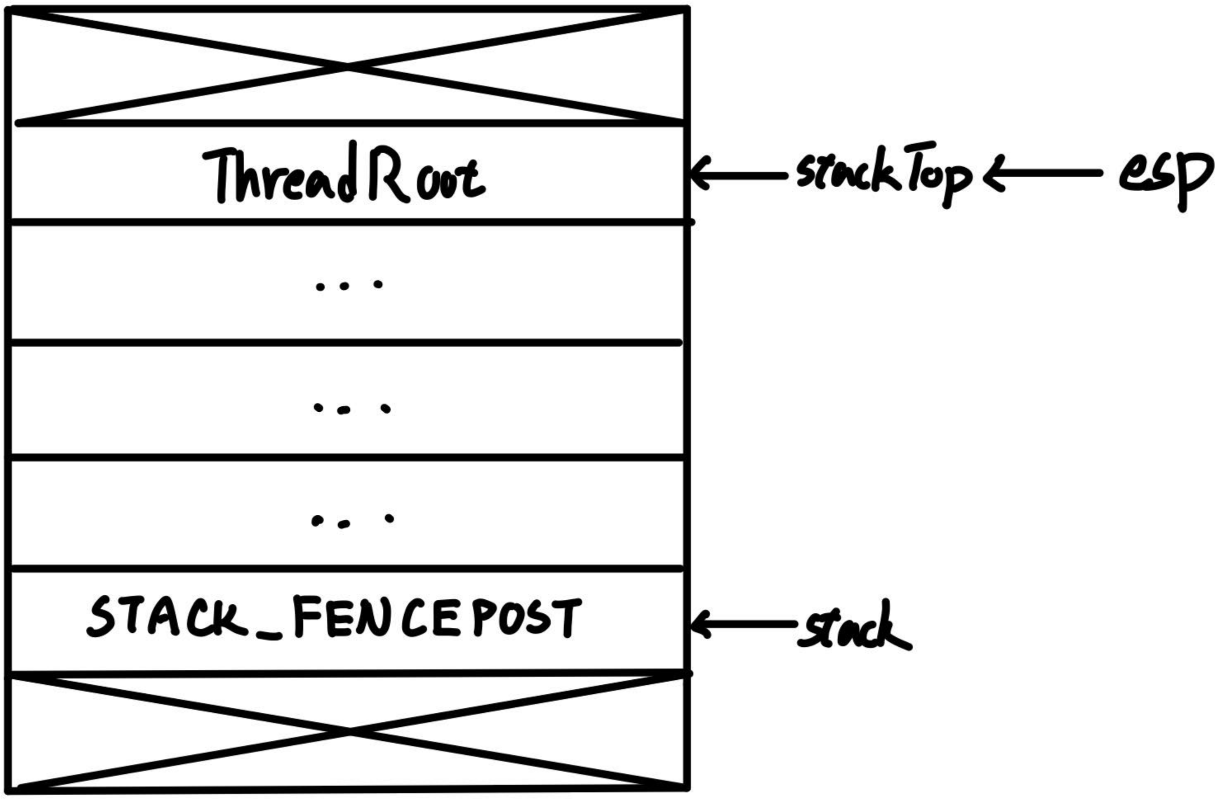
接著用 `%eax` 當作 temp register 將所有 thread 1 的 register 存入對應的 memory address。在`switch.h` 中，`_ESP` 為 0，所以改 `machineState[ESP]` 實際上是改 Thread class 裡的 stackTop。

```
        movl    8(%esp),%eax            # move pointer to t2 into eax
```

將 `%eax` 的值改為 `%esp + 8`，也就是 Thread t2

```
        movl    _EAX(%eax),%ebx         # get new value for eax into ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
```

接著將原本存放在 memory 裡面 thread 2 所有 register 的 state load 回來，並且把 return 到的位址放在 `%eax`。其中，`_ESP(%eax)` 實際上是Thread t2 的 stackTop，因此現在的 `%esp` 會指向這一塊記憶體




```
        movl    _PC(%eax),%eax          # restore return address into eax
        movl    %eax,4(%esp)            # copy over the ret address on the stack
        movl    _eax_save,%eax

        ret
```

將 Thread t2 的 Program Counter 放到 return address，並取回 `%eax` 的值。return 時從 `%esp` pop return address 到 PC，也就是跳轉到 ThreadRoot。

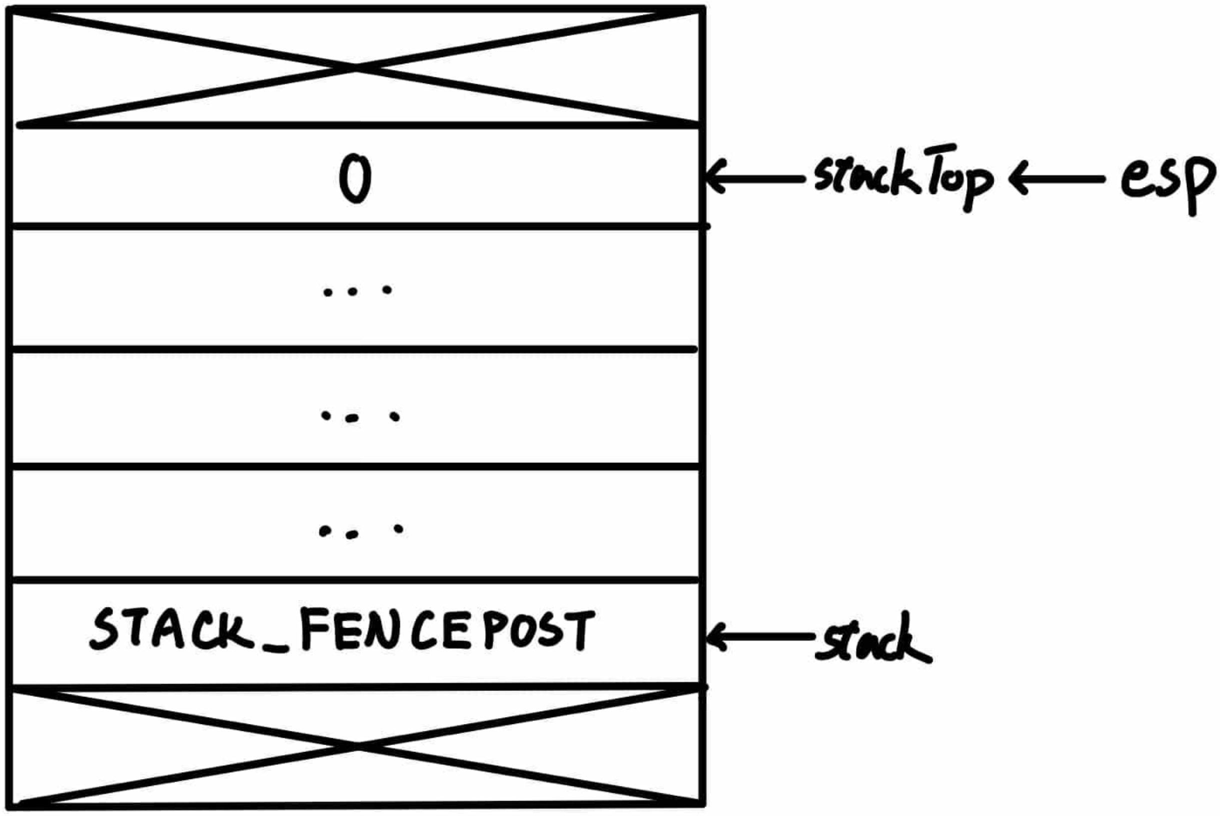
ThreadRoot

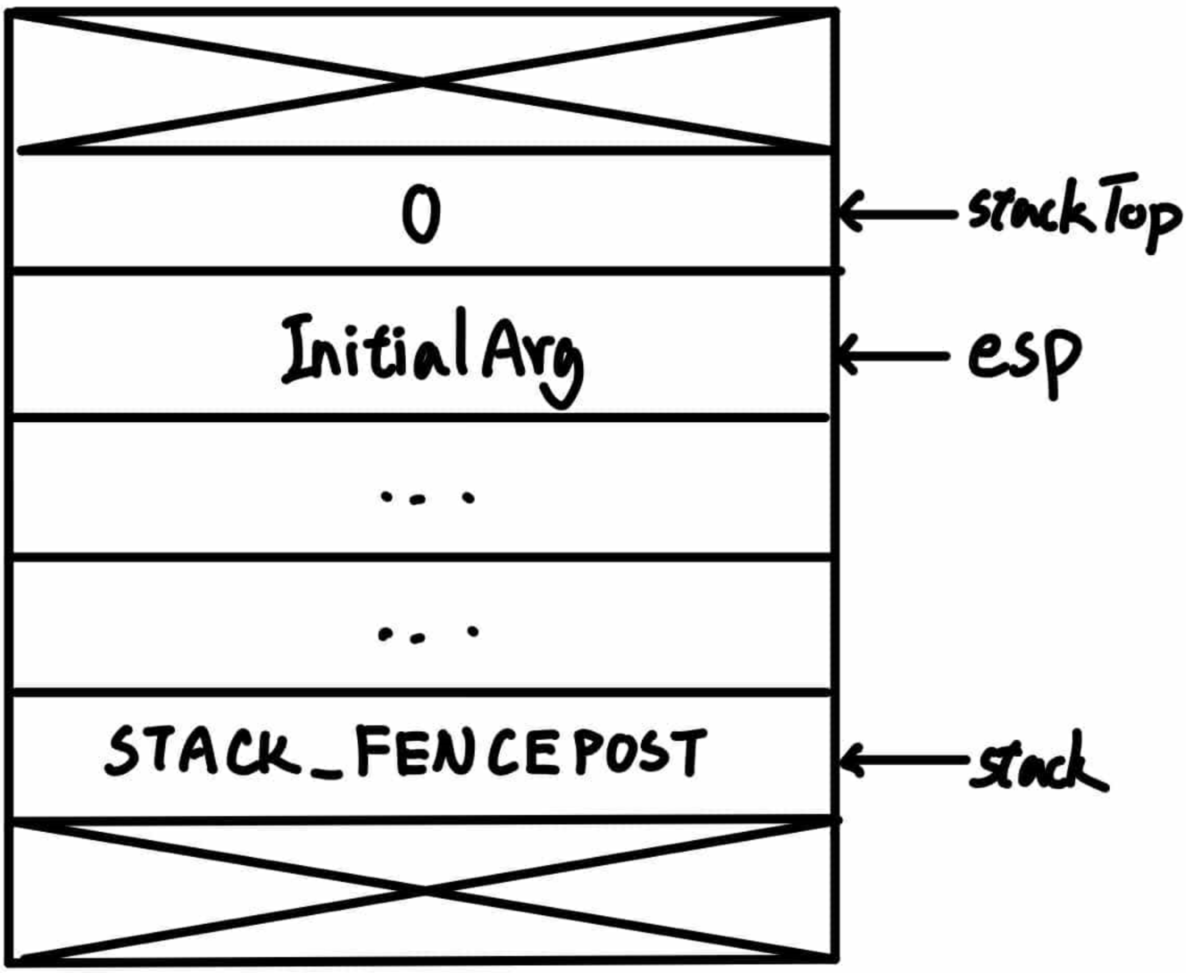
跳轉到 Thread t2 的 Program counter 上，Thread t2 `machineState[PC]` 是 ThreadRoot 的位址，所以現在會跑到 ThreadRoot 上執行。

```
_ThreadRoot:
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg
        call    *StartupPC
        call    *InitialPC
        call    *WhenDonePC

        # NOT REACHED
        movl    %ebp,%esp
        popl    %ebp
        ret
```

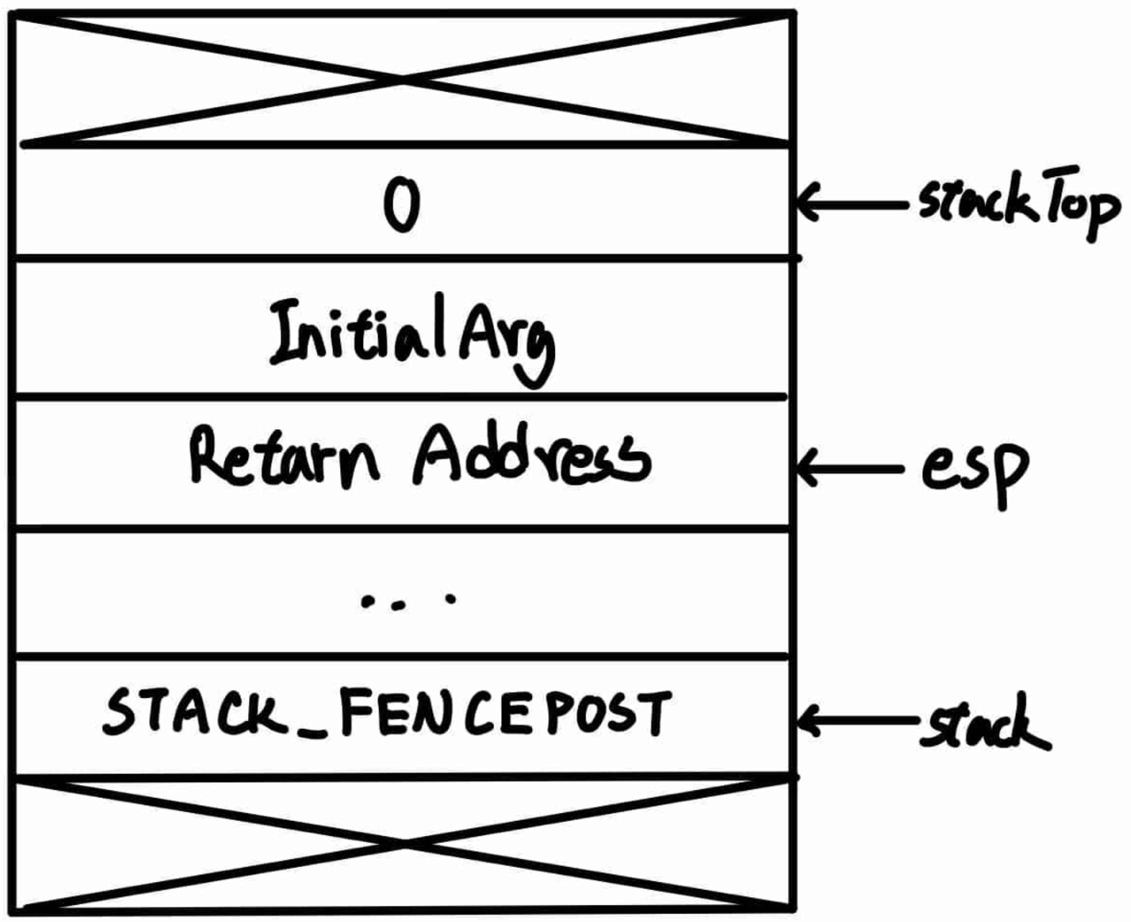
`pushl %ebp`， `%ebp` 的初始值為 0，然後將 `%esp` 的值指到 `%ebp`, 接著 `pushl InitialArg`。





Call `StartupPC`，會 push 下一個指令位址，`StartupPC` 在 `threads/switch.h`，定義為 `%ecx`。在前面 `SWITCH` 中，`%ecx` 的值為 Thread t2 的 `machineState[ECX]`。而 Thread t2 的`machineState[ECX]` 的值在

`StackAllocate()` 中有設定 `machineState[StartupPCState] = (void *)ThreadBegin;`，所以現在會呼叫 `ThreadBegin`：`static void ThreadBegin() { kernel->currentThread->Begin(); }`。`kernel->currentThread` 現在為 Thread t2，所以會呼叫 Thread t2 的 `Begin()`。在執行`Thread::Begin()` 中：`kernel->scheduler->CheckToBeDestroyed()`， Thread t1 在此被砍掉。



最後，call `InitialPC` 以及 call `WhenDonePC`

Implementation

lib/debug.h

按照 spec 要求新增一個 `'z'` tag

```
...
const char dbgSys = 'u';      // systemcall
const char dbgTraCode = 'c';
const char dbgZ = 'z'; // TODO add 'z' flag for testing
...
```

threads/alarm.cc

實作 aging 的功能，並且輸出 debug message，其中 `UpdatePriority()` 的用處是當 thread priority 提升到超過門檻時要進到上一層 ready queue 裡面，實作細節會在後續內容提到。

```
void Alarm::CallBack() {
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    Thread* thread = NULL;

    // TODO : Implement aging and DEBUG output [C]
    for (int i = 0; i < kernel->getThreadNum(); i++) {
        thread = kernel->getThread(i);
        if (thread == NULL) continue;
        int oldLevel = thread->getPriorityLevel();
        if (thread->getStatus() == READY) {

            // 更新 ready time
            thread->setReadyTime(thread->getReadyTime() + 100);

            // 提升 priority
            if (kernel->stats->totalTicks - thread->getReadyTime() >= 1500) {
                int oldPriority = thread->getPriority();
                thread->setPriority(thread->getPriority() + 10);
                if (thread->getPriority() > 149) {
                    thread->setPriority(149);
                }
                int newPriority = thread->getPriority();
                DEBUG(dbgZ, "[C] Tick [" << kernel->stats->totalTicks << "]:
Thread [" << thread->getID() << "] changes its priority from [" << oldPriority <<
"] to [" << newPriority << "]");

                // 提升後重置 ready time
                thread->setReadyTime(kernel->stats->totalTicks);

                // 修正層級
                if (oldLevel != thread->getPriorityLevel()) {
                    kernel->scheduler->UpdatePriority(thread, oldLevel, thread-
>getPriorityLevel());
                }
            }
        }
    }
    ...
}
```

實作 preempt 的功能，透過在特定條件呼叫 `interrupt->YieldOnReturn()` 來達成。

```
void Alarm::CallBack() {
    ...
    int runningThreadLevel = kernel->currentThread->getPriorityLevel();

    // TODO : Implement preemption
    // 第一層的 preempt
    if (!kernel->scheduler->getReadyList(1)->IsEmpty() && status != IdleMode) {
```

```
        if (runningThreadLevel != 1) {
            interrupt->YieldOnReturn();
        } else if (kernel->currentThread->getRemainingBurstTime() > kernel-
>scheduler->getReadyList(1)->Front()->getRemainingBurstTime()) {
            interrupt->YieldOnReturn();
        }
    }
    // 第二層 preempt 第三層
    else if (runningThreadLevel == 3 && !kernel->scheduler->getReadyList(2)-
>IsEmpty() && status != IdleMode) {
        interrupt->YieldOnReturn();
    }
    // 第三層的 round-robin
    else if (runningThreadLevel == 3 && status != IdleMode) {
        interrupt->YieldOnReturn();
    }
}
```

threads/kernel.h

宣告一些會用到的東西。

```
class Kernel {
    public:
    ...
    int getThreadNum() { return threadNum; } // TODO for adding ready time

    private:
    ...
    int priority[10]; // TODO for storing priority input
    ...
};
```

threads/kernel.cc

實作讀取 `-ep` 的功能。

```
Kernel::Kernel(int argc, char **argv) {
    ...
    for (int i = 1; i < argc; i++) {
        ...
        } else if (strcmp(argv[i], "-ep") == 0){ //TODO handle -ep flag
            execfile[++execfileNum] = argv[++i];
            priority[execfileNum] = atoi(argv[++i]);
        } else if (strcmp(argv[i], "-ee") == 0) {
        ...
    }
}
```

修正 main thread 的建構子。

```
void Kernel::Initialize() {
    ...
    currentThread = new Thread("main", threadNum++, 149); // TODO modify thread
constructor
    ...
}
```

threads/thread.h

定義不同層 queue 的界線。

```
...
// TODO define some constant
#define L3_max_priority 49
#define L2_max_priority 99
#define L1_max_priority 149
...
```

宣告一些用來記錄 priority 和 ticks 的變數以及他們的 method。

```
class Thread {
    ...
    public:
    ...

    // TODO define some getter and setter
    int getPriority(){ return priority; }
    void setPriority(int p){ priority = p; }

    float getRemainingBurstTime(){ return remainingBurstTime; }
    void setRemainingBurstTime(float time){ remainingBurstTime = time; }

    int getStartBurstTime() { return startBurstTime; }
    void setStartBurstTime(int time) { startBurstTime = time; }

    int getAccumulatedBurstTime() { return accumulatedBurstTime; }
    void setAccumulatedBurstTime(int time) { accumulatedBurstTime = time; }

    float getPredictedBurstTime() { return predictedBurstTime; }
    void setPredictedBurstTime(float time) { predictedBurstTime = time; }

    int getReadyTime() { return readyTime; }
    void setReadyTime(int time) { readyTime = time; }
```

```
    int getPriorityLevel() {
        if (priority <= L3_max_priority) {
            return 3;
        } else if (priority <= L2_max_priority) {
            return 2;
        } else if (priority <= L1_max_priority) {
            return 1;
        } else {
            DEBUG(dbgThread, "priority error, ID = " << this->getID());
            return 0;
        }
    }

    private:
    ...
    int priority;                // thread 優先度
    float remainingBurstTime;    // thread 剩餘 burst time
    int startBurstTime;          // thread 單次 burst start time
    int accumulatedBurstTime;    // thread 累積 burst time
    float predictedBurstTime;    // thread 預測 burst time
    int readyTime;               // thread ready time
    ...
};
```

threads/thread.cc

修正 `Thread()` 的內容。

```
//TODO modify the constructor
Thread::Thread(char *threadName, int threadID, int p) {
    ...
    priority = p;
    startBurstTime = 0;
    accumulatedBurstTime = 0;
    remainingBurstTime = 0;
    predictedBurstTime = 0;
    readyTime = 0;
    ...
}
```

只有要進 waiting 或是結束的時候才會經過這裡，因此在這裡計算 `predictedBurstTime` 和 `accumulatedBurstTime`，而從 waiting 重新被喚醒時，則需要將 `accumulatedBurstTime` 設成 0，因此在 `kernel->scheduler->Run(nextThread, finishing);` 後面寫了一行 `accumulatedBurstTime = 0;`。

```
void Thread::Sleep(bool finishing) {
    ...
    status = BLOCKED;
```

```
    // TODO update approximate burst time and DEBUG output [D]
    accumulatedBurstTime += kernel->stats->totalTicks - startBurstTime;
    float oldPredictedBurstTime = predictedBurstTime;
    predictedBurstTime = accumulatedBurstTime * 0.5 + predictedBurstTime * 0.5;
    DEBUG(dbgZ, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
this->getID() << "] update approximate burst time, from [" <<
oldPredictedBurstTime << "], add [" << accumulatedBurstTime << "], to [" <<
predictedBurstTime << "]")

    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);
    accumulatedBurstTime = 0; // TODO set accumulatedBurstTime to 0
}
```

threads/scheduler.h

定義在 `alarm.cc` 裡面用到的 `UpdatePriority()` 、 `getReadyList()`，還有不同層級的 ready queue。

```
class Scheduler {
    public:
    ...
    // TODO implement aging and rise level
    void UpdatePriority(Thread * thread, int oldLevel, int newLevel);
    // TODO getReadyList
    List<Thread*>* getReadyList(int level);

    private:
    // TODO declare readyList1, readyList2, readyList3
    SortedList<Thread*> * readyList1;
    SortedList<Thread*> * readyList2;
    List<Thread*>* readyList3;
    ...
};
```

threads/scheduler.cc

建構三層 ready queue，按照 spec 要求撰寫 compare function，並傳進 `SortedList` 的建構子。

```
...
// TODO compare function
int cmpReadyList1(Thread *a, Thread *b) {
    if (a->getRemainingBurstTime() != b->getRemainingBurstTime()) {
        return a->getRemainingBurstTime() - b->getRemainingBurstTime();
    }
    return a->getID() - b->getID();
}
```

```
int cmpReadyList2(Thread *a, Thread *b) {
    if (a->getPriority() != b->getPriority()) {
        return b->getPriority() - a->getPriority();
    }
    return a->getID() - b->getID();
}

Scheduler::Scheduler() {
    // TODO initialize readyList1, readyList2, readyList3
    readyList1 = new SortedList<Thread*>(cmpReadyList1);
    readyList2 = new SortedList<Thread*>(cmpReadyList2);
    readyList3 = new List<Thread*>;
    toBeDestroyed = NULL;
    switchNum = 0;
}

Scheduler::~Scheduler() {
    // TODO de-allocate readyList1, readyList2, readyList3
    delete readyList1;
    delete readyList2;
    delete readyList3;
}
...
```

計算 thread 的各種 time。

- **RUNNING -> READY** 累加 `accumulatedBurstTime`，更新 `remainingBurstTime`，重設 `readyTime` 為 `kernel->stats->totalTicks`。
- **BLOCKED -> READY** 將 `remainingBurstTime` 重設為 `predictedBurstTime`，重設 `readyTime` 為 `kernel->stats->totalTicks`。
- **JUST_CREATED -> READY** 將 `remainingBurstTime` 設置為 `predictedBurstTime`，將 `accumulatedBurstTime` 和 `readyTime` 設置為 `kernel->stats->totalTicks`。

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


    // TODO update accumulate burst time, update remaining burst time
    if (prevStatus == RUNNING) {
        int singleBurstTime = kernel->stats->totalTicks - thread-
>getStartBurstTime();
        thread->setAccumulatedBurstTime(thread->getAccumulatedBurstTime() +
singleBurstTime);
        thread->setRemainingBurstTime(thread->getRemainingBurstTime() -
singleBurstTime);
        thread->setReadyTime(kernel->stats->totalTicks);
    } else if (prevStatus == BLOCKED) {
        thread->setRemainingBurstTime(thread->getPredictedBurstTime());
```

```
        thread->setReadyTime(kernel->stats->totalTicks);
    } else if (prevStatus == JUST_CREATED) {
        thread->setRemainingBurstTime(thread->getPredictedBurstTime());
        thread->setAccumulatedBurstTime(0);
        thread->setReadyTime(kernel->stats->totalTicks);
    } else {
        DEBUG(dbgThread, "error status, ID = " << thread->getID());
    }

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

按照 priority 將 thread 塞進 ready queue 裡面，並且輸出 debug message。

```
void Scheduler::ReadyToRun(Thread *thread, ThreadStatus prevStatus) {
    ...
    thread->setStatus(READY);

    // TODO insert thread into readyList and DEBUG output [A]
    if(thread->getPriority() <= L3_max_priority) {
        DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[" << 3 << "]");
        readyList3->Append(thread);
    } else if (thread->getPriority() <= L2_max_priority) {
        DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[" << 2 << "]");
        readyList2->Insert(thread);
    } else if (thread->getPriority() <= L1_max_priority) {
        DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[" << 1 << "]");
        readyList1->Insert(thread);
    } else {
        DEBUG(dbgThread, "priority error, ID = " << thread->getID());
    }
}
```

按照層級尋找可以被執行的 thread，有的話就回傳 Thread*，沒有的話就回傳 NULL。

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

    // TODO Remove the first thread from the ready list and DEBUG output [B]
    Thread* rtThread = NULL;

    if (!readyList1->IsEmpty()) {
        rtThread = readyList1->RemoveFront();
```

```
        DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
rtThread->getID() << "] is removed from queue L[" << 1 << "]");
    } else if(!readyList2->IsEmpty()){
        rtThread = readyList2->RemoveFront();
        DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
rtThread->getID() << "] is removed from queue L[" << 2 << "]");
    } else if(!readyList3->IsEmpty()){
        rtThread = readyList3->RemoveFront();
        DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
rtThread->getID() << "] is removed from queue L[" << 3 << "]");
    }

    return rtThread;
}
```

在要被 switch 前輸出 debug message，並重置要跑的那條 thread 的 `startBurstTime`。

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
    ...
    // TODO DEBUG output [E]
    DEBUG(dbgZ, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
nextThread->getID() << "] is now selected for execution, thread [" << oldThread-
>getID() << "] is replaced, and it has executed [" << oldThread-
>getAccumulatedBurstTime() << "] ticks")

    // TODO record the single burst start time
    nextThread->setStartBurstTime(kernel->stats->totalTicks);

    SWITCH(oldThread, nextThread);
    ...
}
```

實作 `alarm.cc` 裡面有用到的 `UpdatePriority()` 和 `getReadyList()` 這兩個 function。

```
// TODO implement aging and rise level
void Scheduler::UpdatePriority(Thread * thread, int oldLevel, int newLevel) {
    if (oldLevel == newLevel) {
        return;
    }

    if (oldLevel == 1) {
        readyList1->Remove(thread);
    } else if (oldLevel == 2) {
        readyList2->Remove(thread);
    } else if (oldLevel == 3) {
        readyList3->Remove(thread);
    } else {
```

```
        DEBUG(dbgThread, "priority error, ID = " << thread->getID());
    }

    if (newLevel == 1) {
        readyList1->Insert(thread);
    } else if (newLevel == 2) {
        readyList2->Insert(thread);
    } else if (newLevel == 3) {
        readyList3->Append(thread);
    } else {
        DEBUG(dbgThread, "priority error, ID = " << thread->getID());
    }
}

List<Thread*>* Scheduler::getReadyList(int level) {
    if (level == 1) {
        return readyList1;
    } else if (level == 2) {
        return readyList2;
    } else if (level == 3) {
        return readyList3;
    }
}
```