I. Trace Code

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

```
272  void Kernel::ExecAll() {
273      for (int i = 1; i <= execfileNum; i++) {
274          int a = Exec(execfile[i]);
275      }
276      currentThread->Finish();
277      // Kernel::Exec();
278  }
```

執行所有 execfile[](Exec())，完成後呼叫 Finish()。

Exec()

```
280  int Kernel::Exec(char *name) {
281      t[threadNum] = new Thread(name, threadNum);
282      t[threadNum]->setIsExec();
283      t[threadNum]->space = new AddrSpace();
284      t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
285      threadNum++;
286
287      return threadNum - 1;
288  }
```

接收傳入的 thread name 指標，建立新的 Thread 物件並回傳創建的 thread ID。
t 變數是 thread class 中定義的 private variable，紀錄 thread 相關變數。

呼叫 Thread()，傳入 thread name, ID，在函數中初始化 TCB 、machineState[]。
呼叫 setIsExec()，設置執行序為執行狀態。
呼叫 AddrSpace()，由於後續實作關係改為空函數(分配執行序新的地址物件)。
呼叫 Fork()，傳入 ForkExecute 地址(呼叫 ForkExecute)和 t[threadNum]作為參數。

Thread()

```
37  Thread::Thread(char *threadName, int threadID) {
38      ID = threadID;
39      name = threadName;
40      isExec = false;
41      stackTop = NULL;
42      stack = NULL;
43      status = JUST_CREATED;
44      for (int i = 0; i < MachineStateSize; i++) {
45          machineState[i] = NULL;  // not strictly necessary, since
46                                   // new thread ignores contents
47                                   // of machine registers
48      }
49      space = NULL;
50  }
```

setIsExec()

```
    void setIsExec() { this->isExec = true; }
```

AddrSpace()

```
    AddrSpace::AddrSpace() {}
```

ForkExecute()

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

查看 Thread space 中執行檔是否存在，load(呼叫 Addrspace::Load())成功->呼叫 Execute()。失敗則直接返回。

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);
    // calaulaten the executable need how many pages
    size = numPages * PageSize;
    // size is adjusted to the total memory allocation in bytes

    // TODO

    if(numPages > NumPhysPages){
        kernel->interrupt->setStatus(SystemMode);
        ExceptionHandler(MemoryLimitException);
        kernel->interrupt->setStatus(UserMode);
    }


    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);
```

主要目的是將 user program 從 file 載入 memory 中，先讀取檔案，再計算所需 page 數量，調整分配的 memory 大小，若所需 page 大於 physical page 能提供的數量，則產生 MemoryLimitException，處理 exception，最終，把資料存入 memory 中。

Execute()

```
300 void AddrSpace::Execute(char *fileName) {
301     kernel->currentThread->space = this;
302     this->InitRegisters();  // set the initial register values
303     this->RestoreState();   // load page table register
304     kernel->machine->Run();  // jump to the user progam
305     ASSERTNOTREACHED();  // machine->Run never returns;
306                          // the address space exits
307                          // by doing the syscall "exit"
308 }
```

執行 load 到 address space 中的 user program。設置了 program 運行所需的狀態，接著將控制權轉移到該程式。

為現在 thread 分配 address space，接著初始化 register 的值，並 load page table register，執行 Run()，將控制權交給 user，處理 memory 中 user program 指令。

InitRegisters()

```
    void AddrSpace::InitRegisters() {
        Machine *machine = kernel->machine;
        int i;
        for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister(i, 0);
        machine->WriteRegister(PCReg, 0);
        machine->WriteRegister(NextPCReg, 4);
        machine->WriteRegister(StackReg, numPages * PageSize - 16);
        DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize -
16);
    }
```

RestoreState()

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

Machine::Run()

```
54 void Machine::Run() {
55     Instruction *instr = new Instruction;  // storage for decoded instruction
56     if (debug->IsEnabled('m')) {
57         cout << "Starting program in thread: " << kernel->currentThread->getName();
58         cout << ", at time: " << kernel->stats->totalTicks << "\n";
59     }
60     kernel->interrupt->setStatus(UserMode);
61     for (;;) {
62         DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
63                           << "== Tick " << kernel->stats->totalTicks << " ==");
64         OneInstruction(instr);
65         DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
66                           << "== Tick " << kernel->stats->totalTicks << " ==");
67
68         DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
69                           << "== Tick " << kernel->stats->totalTicks << " ==");
70         kernel->interrupt->OneTick();
71         DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
72                           << "== Tick " << kernel->stats->totalTicks << " ==");
73         if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
74             Debugger();
75     }
76 }
```

Machine::Run()會開始執行 load 的執行檔指令，並將 state 設定為 User mode。

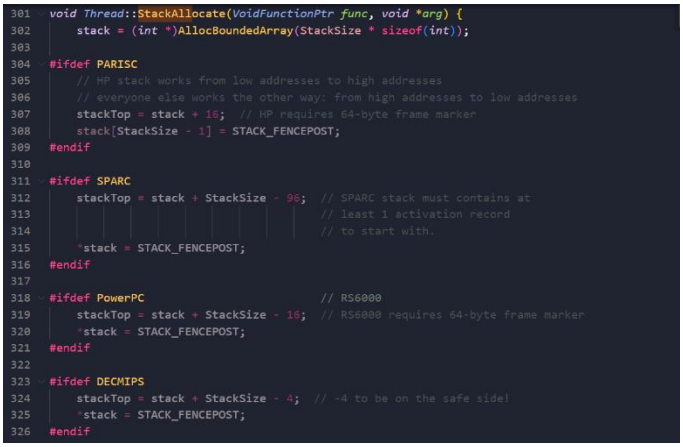
Fork()

```
91  void Thread::Fork(VoidFunctionPtr func, void *arg) {
92      Interrupt *interrupt = kernel->interrupt;
93      Scheduler *scheduler = kernel->scheduler;
94      IntStatus oldLevel;
95
96      DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " " << arg);
97      StackAllocate(func, arg);
98
99      oldLevel = interrupt->SetLevel(IntOff);
100     scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts
101                                   // are disabled!
102     (void)interrupt->SetLevel(oldLevel);
103 }
```

設定獲取 Kernel 物件參數，呼叫 StackAllocate()並傳入 PCState()、ArgState()參數(分別是 func, arg，用來設置新 allocate 的 thread 參數)。

將 interrupt 狀態設定為 IntOff，並保存狀態到 oldLevel，呼叫 ReadyToRun()將 thread 加入排程，完成後依照先前儲存的變數恢復狀態。

StackAllocate()



```
301 void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
302     stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
303
304 #ifdef PARISC
305     // HP stack works from low addresses to high addresses
306     // everyone else works the other way: from high addresses to low addresses
307     stackTop = stack + 16;  // HP requires 64-byte frame marker
308     stack[StackSize - 1] = STACK_FENCEPOST;
309 #endif
310
311 #ifdef SPARC
312     stackTop = stack + StackSize - 96;  // SPARC stack must contains at
313                                         // least 1 activation record
314                                         // to start with.
315     *stack = STACK_FENCEPOST;
316 #endif
317
318 #ifdef PowerPC                          // RS6000
319     stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
320     *stack = STACK_FENCEPOST;
321 #endif
322
323 #ifdef DECMIPS
324     stackTop = stack + StackSize - 4;  // -4 to be on the safe side!
325     *stack = STACK_FENCEPOST;
326 #endif
```

```
301 void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
328 #ifdef ALPHA
329     stackTop = stack + StackSize - 8;  // -8 to be on the safe side!
330     *stack = STACK_FENCEPOST;
331 #endif
332
333 #ifdef x86
334     // the x86 passes the return address on the stack.  In order for SWITCH()
335     // to go to ThreadRoot when we switch to this thread, the return addres
336     // used in SWITCH() must be the starting address of ThreadRoot.
337     stackTop = stack + StackSize - 4;  // -4 to be on the safe side!
338     *(--stackTop) = (int)ThreadRoot;
339     *stack = STACK_FENCEPOST;
340 #endif
341
342 #ifdef PARISC
343     machineState[PCState] = PLabelToAddr(ThreadRoot);
344     machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
345     machineState[InitialPCState] = PLabelToAddr(func);
346     machineState[InitialArgState] = arg;
347     machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
348 #else
349     machineState[PCState] = (void *)ThreadRoot;
350     machineState[StartupPCState] = (void *)ThreadBegin;
351     machineState[InitialPCState] = (void *)func;
352     machineState[InitialArgState] = (void *)arg;
353     machineState[WhenDonePCState] = (void *)ThreadFinish;
354 #endif
355 }
```

Allocate 一個 8192*sizeof(int)大小的空間位址作為 thread 的 stack 空間。

並根據不同系統架構設計了不同的堆疊方式。

最後會使用 machineState 這個數據結構保存最終的 CPU reg 等等 thread 參數。

ReadyToRun()

```
54
55  void Scheduler::ReadyToRun(Thread *thread) {
56      ASSERT(kernel->interrupt->getLevel() == IntOff);
57      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
58      // cout << "Putting thread on ready list: " << thread->getName() << endl ;
59      thread->setStatus(READY);
60      readyList->Append(thread);
61  }
```

先確認 interrupt 的狀態是否是 IntOff(檢查 Fork()是否正確將 interrupt 關閉)，將 thread 狀態設置為 Ready 後放入 list 中等待排程。

Finish()

```
167  void Thread::Finish() {
168      (void)kernel->interrupt->SetLevel(IntOff);
169      ASSERT(this == kernel->currentThread);
170
171      DEBUG(dbgThread, "Finishing thread: " << name);
172      if (kernel->execExit && this->getIsExec()) {
173          kernel->execRunningNum--;
174          if (kernel->execRunningNum == 0) {
175              kernel->interrupt->Halt();
176          }
177      }
178      Sleep(TRUE);  // invokes SWITCH
179      // not reached
180  }
```

對應 Thread::Begin()，將 interrupt 設定成 IntOff 狀態後將判斷工作完成的 thread 刪除，若全部的 thread 都完成後則呼叫 Halt()結束 Process。呼叫 Sleep()將 thread 工作完成的 bool 值傳入 Run()，並完成 context switch。

Sleep()

```
236  void Thread::Sleep(bool finishing) {
237      Thread *nextThread;
238
239      ASSERT(this == kernel->currentThread);
240      ASSERT(kernel->interrupt->getLevel() == IntOff);
241
242      DEBUG(dbgThread, "Sleeping thread: " << name);
243      DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
244
245      status = BLOCKED;
246      // cout << "debug Thread::Sleep " << name << "wait for Idle\n";
247      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
248          kernel->interrupt->Idle();  // no one to run, wait for an interrupt
249      }
250      // returns when it's time for us to run
251      kernel->scheduler->Run(nextThread, finishing);
252  }
```

將當前 thread 的 state 設定為 block，並從等待序列中 schedule 一個新的 thread 執行(呼叫 FindNextToRun())若沒有則將 CPU mode 設置為 idle。
最後呼叫 Run()。

FindNextToRun()

```
Thread * Scheduler::FindNextToRun() {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

Scheduler::Run()

```
99  void Scheduler::Run(Thread *nextThread, bool finishing) {
100     Thread *oldThread = kernel->currentThread;
101     ASSERT(kernel->interrupt->getLevel() == IntOff);
102
103     if (finishing) {  // mark that we need to delete current thread
104         ASSERT(toBeDestroyed == NULL);
105         toBeDestroyed = oldThread;
106     }
107     if (oldThread->space != NULL) {  // if this thread is a user program,
108         oldThread->SaveUserState();  // save the user's CPU registers
109         oldThread->space->SaveState();
110     }
111
112     oldThread->CheckOverflow();  // check if the old thread
113                                  // had an undetected stack overflow
114
115     kernel->currentThread = nextThread;  // switch to the next thread
116     nextThread->setStatus(RUNNING);      // nextThread is now running
117     DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
118     // This is a machine-dependent assembly language routine defined
119     // in switch.s.  You may have to think
120     // a bit to figure out what happens after this, both from the point
121     // of view of the thread and from the perspective of the "outside world".
122     SWITCH(oldThread, nextThread);
123     // we're back, running oldThread
124     // interrupts are off when we return from switch!
125     ASSERT(kernel->interrupt->getLevel() == IntOff);
126     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
127     CheckToBeDestroyed();  // check if thread we were running
128                            // before this one has finished
129                            // and needs to be cleaned up
130     if (oldThread->space != NULL) {     // if there is an address space
131         oldThread->RestoreUserState();  // to restore, do it.
132         oldThread->space->RestoreState();
133     }
134 }
```

檢查 interrupt 狀態(IntOff)，呼叫 CheckOverflow()確認 oldthread 是否使用超過 allocate 空間；調用 SWITCH 完成 context switch(硬體)。

若 oldthread 已完成(finishing == True)，SWITCH()返回後 CheckToBeDestroyed()會檢查並刪除 oldthread；否則將呼叫 RestoreUserState()、RestoreState()，回復之前儲存的狀態(SaveUserState()、SaveState())

CheckOverflow()

```
void Thread::CheckOverflow() {
    if (stack != NULL) {
#ifdef HPUX  // Stacks grow upward on the Snakes
        ASSERT(stack[StackSize - 1] == STACK_FENCEPOST);
#else
        ASSERT(*stack == STACK_FENCEPOST);
#endif
    }
}
```

CheckToBeDestroyed()

```
void Scheduler::CheckToBeDestroyed() {
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
        toBeDestroyed = NULL;
    }
}
```

➢ 補充圖檔

A. Kernel.h : Kernel class

```
class Kernel {
  public:
    Kernel(int argc, char **argv);
    // Interpret command line arguments
    ~Kernel();  // deallocate the kernel
    void Initialize();  // initialize the kernel -- separated
                        // from constructor because
                        // refers to "kernel" as a global
    // TODO
    int Find_Frame();  // To find available frame
    void ExecAll();
    int Exec(char *name);
    void ThreadSelfTest();  // self test of threads and synchronization
    void ConsoleTest();  // interactive console self test
    void NetworkTest();  // interactive 2-machine network test
    Thread *getThread(int threadID) { return t[threadID]; }
    void PrintInt(int number);
    int CreateFile(char *filename);  // fileSystem call
    OpenFileId OpenFile(char *name);                    // fileSystem call
    int WriteFile(char *buffer, int size, OpenFileId id);  // fileSystem call
    int ReadFile(char *buffer, int size, OpenFileId id);   // fileSystem call
    int CloseFile(OpenFileId id);                       // fileSystem call
    // These are public for notational convenience; really,
    // they're global variables used everywhere.

    Thread *currentThread;  // the thread holding the CPU
    Scheduler *scheduler;   // the ready list
    Interrupt *interrupt;   // interrupt status
    Statistics *stats;      // performance metrics
    Alarm *alarm;           // the software alarm clock
    Machine *machine;       // the simulated CPU
    SynchConsoleInput *synchConsoleIn;
    SynchConsoleOutput *synchConsoleOut;
    SynchDisk *synchDisk;
    FileSystem *fileSystem;
    PostOfficeInput *postOfficeIn;
    PostOfficeOutput *postOfficeOut;
    bool execExit;      // exit if all threads are finished
    int execRunningNum;  // number of running threads
    int hostName;  // machine identifier
    // TODO
    int *frame_Table_status;  // a pointer which point to an array store the frame is empty or not

  private:
    Thread *t[10];
    char *execfile[10];
    int execfileNum;
    int threadNum;
    bool randomSlice;    // enable pseudo-random time slicing
    bool debugUserProg;  // single step user program
    double reliability;  // likelihood messages are dropped
    char *consoleIn;     // file to read console input from
    char *consoleOut;    // file to send console output to
#ifndef FILESYS_STUB
    bool formatFlag;  // format the disk if this is true
#endif
};
```

定義 kernel class 物件。

B. Thread.h Thread class

```
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
    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;
    bool isExec;  // Is this thread an user executable thread
    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:
    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 setIsExec() { this->isExec = true; }
    bool getIsExec() { return (isExec); }
    void Print() { cout << name; }
    void SelfTest();  // test whether thread impl is working
    void SaveUserState();      // save user-level register state
    void RestoreUserState();  // restore user-level register state
    AddrSpace *space;         // User code this thread is running.
};
```

TCB 實作(thread ID, name, isEcex, StackAllocate…)

C. Thread.h ThreadStatus 變數

```
60   enum ThreadStatus { JUST_CREATED,
61                       RUNNING,
62                       READY,
63                       BLOCKED,
64                       ZOMBIE };
```

備註：stacksize = 8*1024 (in words)

D. Unility.h Fork 參數

```
typedef void (*VoidFunctionPtr)(void *arg);
typedef void (*VoidNoArgFunctionPtr)();
```

E. Lib/sysdep.cc AllocBoundedArray

```
196  char *
197  AllocBoundedArray(int size) {
198  #ifdef NO_MPROT
199      return new char[size];
200  #else
201      int pgSize = getpagesize();
202      char *ptr = new char[pgSize * 2 + size];
203
204      mprotect(ptr, pgSize, 0);
205      mprotect(ptr + pgSize + size, pgSize, 0);
206      return ptr + pgSize;
207  #endif
208  }
```

F. Lib/list.cc Append()

```
69   template <class T>
70   void List<T>::Append(T item) {
71       ListElement<T> *element = new ListElement<T>(item);
72
73       ASSERT(!IsInList(item));
74       if (IsEmpty()) {  // list is empty
75           first = element;
76           last = element;
77       } else {  // else put it after last
78           last->next = element;
79           last = element;
80       }
81       numInList++;
82       ASSERT(IsInList(item));
83   }
```

G. Thread.cc SaveUserState()

Thread 離開 user mode 轉為 kernel mode 時，保存 thread 的 CPU 狀態。

```
368  void Thread::SaveUserState() {
369      for (int i = 0; i < NumTotalRegs; i++)
370          userRegisters[i] = kernel->machine->ReadRegister(i);
371  }
```

H. AddrSpace.cc SaveState()

```
276  //----------------------------------------------------------------------
277  // AddrSpace::SaveState
278  //  On a context switch, save any machine state, specific
279  //  to this address space, that needs saving.
280  //
281  //  For now, don't need to save anything!
282  //----------------------------------------------------------------------
283
284  void AddrSpace::SaveState() {}
```

I. Thread.cc CheckOverflow()

```
26   // this is put at the top of the execution stack, for
     detecting stack overflows
27   const int STACK_FENCEPOST = 0xdedbeef;
```

J. Scheduler.cc CheckToBeDestroyed()

```
152 void Scheduler::CheckToBeDestroyed() {
153     if (toBeDestroyed != NULL) {
154         delete toBeDestroyed;
155         toBeDestroyed = NULL;
156     }
157 }
```

K. Thread.cc RestoreUserState()

```
382 void Thread::RestoreUserState() {
383     for (int i = 0; i < NumTotalRegs; i++)
384         kernel->machine->WriteRegister(i, userRegisters[i]);
385 }
```

L. AddrSpace.cc RestoreState()

```
294 void AddrSpace::RestoreState()
295 {
296     kernel->machine->pageTable = pageTable;
297     kernel->machine->pageTableSize = numPages;
298 }
```

M. Kernel::Kernel() 解讀 command line argument，設定相對應的 flag。

ASSERT(i + 1 < argc): 確保 argv[i+1]存在，防止讀取到 command line argument 之外的位置。

- **-rs**：以 argv[i+1]的值為 seed，初始化隨機數生成器，並啟用 randomslice。
- **-s**：啟用 debugUserProg，進入 debug 模式。
- **-e**：將 argv 內下個參數指定為欲執行的檔案名稱，並存入 execfile 陣列中。
- **-ee**：啟用 execExit，當所有 thread 執行完畢後，program 應該退出。
- **-ci**：將 consoleIn 設為 argv[i+1]的內容。例如 program -ci input.txt，會將 consoleIn 設為 "input.txt"，就可以通過 consoleIn 到指定檔案讀取資料。
- **-co**：將 consoleOut 設為 argv[i+1]的內容。
- **-f**：啟用 formatFlag，格式化文件系統。
- **-n**：將 argv[i+1]的值作為可靠度。(argv[i+1]轉換成 float 的值應屬於 0~1)
- **-m**：將 argv[i+1]的值作為 hostName(機器 ID)。
- **-u**：顯示基本 command line argument 的說明。

II. Implement page table in Nachos