



國立清華大學  
NATIONAL TSING HUA UNIVERSITY

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER  
SCIENCE

CS342300 - SPRING 2023

## MP2 – Multi-Programming

*Annabella Putri Dirgo*  
109006238

# 1 Trace Code

## 1.1 threads/kernel.cc Kernel::ExecAll()

```
1 void Kernel::ExecAll(){
2     for (int i=1;i<=execfileNum;i++) {
3         int a = Exec(execfile[i]);
4     }
5     currentThread->Finish();
6 }
```

Listing 1: Our Target for Trace Code Kernell::ExecAll()

Before we start tracing the code, the foundation to execute this code is simply extract the data structure of Thread in NachOS which it can be found in the threads/thread.h. The class Thread is a function that makes a new thread with the name and ID given.

```
1 class Thread {
2     private:
3         int *stackTop;
4         void *machineState[MachineStateSize];
5
6     public:
7         Thread(char* debugName, int threadID);
8         ~Thread();
9         void Fork(VoidFunctionPtr func, void *arg);
10        void Yield();
11
12        void Sleep(bool finishing);
13        void Begin();
14        void Finish();
15        void CheckOverflow();
16        void setStatus(ThreadStatus st) { status = st; }
17        ThreadStatus getStatus() { return (status); }
18        char* getName() { return (name); }
19        int getID() { return (ID); }
20        void Print() { cout << name; }
21        void SelfTest();
22
23    private:
24        int *stack;
25        ThreadStatus status;
26        char* name;
27        int ID;
28        void StackAllocate(VoidFunctionPtr func, void *arg);
29        int userRegisters[NumTotalRegs];
30
31    public:
32        void SaveUserState();
33        void RestoreUserState();
34        AddrSpace *space;
35 };
```

In NachOS, a process may consist of address space which includes all the memory that are allowed by the process to reference, a single thread controls so that the CPU executes instructions sequentially within the process, and other object such as open file descriptors. Inside a thread there consist of a register sets. However, within this code the Thread class does not clearly specify its CPU register settings. Instead, the function machineState is an array of CPU registers except the stack pointer. This comprises user-level and kernel-level CPU register states. The userRegisters array includes just user-level CPU register status. SaveUserState() and RestoreUserState() save and restore the thread's user-level CPU register state respectively. Hence, other than having a register sets, Thread also consist of AddrSpace which in charge to keeping track of executing user programs.

After we acknowledge how the machine initialize the thread. The code tracing will starts from threads/kernel.cc with the method Kernel::Exec() which is used to start a new user level program.

```
1 int Kernel::Exec(char* name) {
2     t[threadNum] = new Thread(name, threadNum);
3     t[threadNum]->space = new AddrSpace();
4     t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
5     threadNum++;
6
7     return threadNum-1;
8 }
```

In line 3, the process starts with making a new `Thread` and pass the parameter `name` and an integer representing thread's ID `threadNum`. Next, on line 4, it creates a new address space for the new thread in the user level process named `AddrSpace`. Next, in line 5 they fork the new thread and this process will be run for N number of thread since they increment the thread number `threadNum ++`.

On a closer inspection, the function `Fork` involved two pointer: a pointer to a function that the new thread will be executed (`VoidFunctionPtr`) `&ForkExecute` and a pointer to a data structure which will be passed to the function as an argument `Fork ((void *)t[threadNum])`

- (`VoidFunctionPtr`) `&ForkExecute`

Earlier, we see how thread was made in process so we will continue its process of forking the thread whereupon we can found it in `threads/kernel.cc`. Now, `Fork` serves the function of allocating stack space for the new thread, initializes the registers (by saving the initial value's in the thread's context block), etc

```
1 void ForkExecute(Thread *t) {
2     if ( !t->space->Load(t->getName()) ) {
3         return;
4     }
5     t->space->Execute(t->getName());
6 }
```

In this code, we will load the executable into the memory and if this condition was not met then it will return nothing that means the executable was not found. Otherwise we will execute the program into the memory in `userprog/addrspace.cc` (`AddrSpace::Execute`).

In this function, it will run a user program using the current thread on side note that it will also link the address space of the current thread with the caller. Foremost, it will initialize user-level CPU registers before jumping to user code. Then, it will initialize machine registers and write it directly into the machine and continue with the line of code `this->RestoreState()` where in here we set the PC to the start of the program, and the SP to the top of the stack and we also set the page table register to the page table we just created. Last, it will jump into user program and call `run()` to simulate program execution.

```
1 void AddrSpace::Execute(char* fileName) {
2     kernel->currentThread->space = this;
3     this->InitRegisters();
4     this->RestoreState();
5     kernel->machine->Run();
6     ASSERTNOTREACHED();
7 }
```

- `Fork(VoidFunctionPtr func, void *arg)`

In this function first, we get the kernel's interrupt and scheduler and save them in a pointers named `interrupt` and `scheduler`. Moreover, we will also create a variable to store the old level of interrupt. This process will continued by the function `StackAllocate(func, arg)` which it will allocate the stack for the thread and save it in the stack variable of the thread object. In `StackAllocate(func, arg)`, it will initializes various Kernel Registers. On line 10, we disable the interrupts and save the old level of interrupt in the `oldLevel` variable plus on line 11, we put the thread in the ready queue of the scheduler and assume that in here the interrupts are being disabled. Last, it set the level of interrupt to the old level.

```
1 void Thread::Fork(VoidFunctionPtr func, void *arg) {
2     Interrupt *interrupt = kernel->interrupt;
3     Scheduler *scheduler = kernel->scheduler;
4     IntStatus oldLevel;
5
6     DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg)
7     ;
8     StackAllocate(func, arg);
9
10    oldLevel = interrupt->SetLevel(IntOff);
11    scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
12                                // are disabled!
13    (void) interrupt->SetLevel(oldLevel);
14 }
```

By now, we can look back to the code,we understand means that the Main Thread `Kernel` executes `Exec` all the programs `Thread` to be executed in sequence. After executing all the programs `Thread`, call `Finish` to prepare to release the space of the `Thread`.

For a side note if we look at `Thread::Finish()`, the function `Thread::Sleep()` will be called in `Finish` to block the current `Thread`. Since we can't immediately de-allocate the thread data structure or the execution stack, because we're still running in the thread and we're still on the stack. Instead, we tell the scheduler to call the destructor, once it is running in the context of a different thread.

```
1 void Thread::Finish () {
2     (void) kernel->interrupt->SetLevel(IntOff);
3     ASSERT(this == kernel->currentThread);
4     DEBUG(dbgThread, "Finishing thread: " << name);
5     Sleep(TRUE);
6 }
```

## 1.2 threads/thread.cc Thread::Sleep()

```
1 void Thread::Sleep (bool finishing) {
2     Thread *nextThread;
3     ASSERT(this == kernel->currentThread);
4     ASSERT(kernel->interrupt->getLevel() == IntOff);
5
6     DEBUG(dbgThread, "Sleeping thread: " << name);
7     DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->
8         totalTicks);
9
10    status = BLOCKED;
11    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
12        kernel->interrupt->Idle();
13    }
14    kernel->scheduler->Run(nextThread, finishing);
15 }
```

Listing 2: Our Target for Trace Code `Thread::Sleep()`

Recall, in `Thread::Finish()` we call `Sleep(TRUE)` and this function call was located in `threads/thread.cc`. The function `Sleep` only accept boolean as its input parameter. On line 2, we make a local copy of the `nextThread` pointer, so that we can call `FindNextToRun()` with interrupts disabled. On line 3 and 4, `ASSERT` check if the current thread is the same as the thread that is running and also check if the interrupt is off.

Hence, the main idea of `Thread::Sleep` is on line 10. Once we have the thread to run, we put it on the ready list, and then call the scheduler, which will switch to it. Then we run a while loop to judge `kernel->scheduler->FindNextToRun()` to see if we are the last thread to finish. If a thread is available to run, we will call `Run()` method which in charge to switch execution to the next thread and at last it will the `finishing` parameter is passed on `Run()` to indicate whether it has finished or not.

If this situation did not happen, the while loop will enter `Idle Mode` by calling `idle()`, at this time it will judge whether there are no Interrupts and Threads to be executed, if not, the entire NachOS operation will come to an end or the NachOS will halt.

## 1.3 machine/mipssim.cc Machine::Run()

```
1 void Machine::Run() {
2     Instruction *instr = new Instruction;
3     if (debug->IsEnabled('m')) {
4         cout << "Starting program in thread: " << kernel->currentThread->getName();
5         cout << ", at time: " << kernel->stats->totalTicks << "\n";
6     }
7     kernel->interrupt->setStatus(UserMode);
8     for (;;) {
9         DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
10             << "==" Tick " << kernel->stats->totalTicks << " ==");
11         OneInstruction(instr);
12         DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction "
13             << "==" Tick " << kernel->stats->totalTicks << " ==");
14
15         DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
16             << "==" Tick " << kernel->stats->totalTicks << " ==");
17         kernel->interrupt->OneTick();
18         DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
19             << "==" Tick " << kernel->stats->totalTicks << " ==");
20         if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
21             Debugger();
22     }
23 }
```

For the machine simulation to work, the `void Machine::Run()` function must be present as it facilitates program execution, handles interruptions, and includes debugging tools. The variable `instr` is used to keep track and store decoded instructions. Then, on line 3 we check if the debug flag is set to 'm' and print out the name of the thread and the time. On line 7, we set the status of the interrupt to `UserMode` and then we enter an infinite loop on line 8. Inside this infinite for loop, We then call the `OneInstruction` function to execute one instruction from the user-level program and then we call the `OneTick` function to increment the total number of ticks and usually an instruction assumes that the system will advance by one Clock.

On `OneInstruction`, whenever an exception or interrupt occurs, it execute the exception handler and return to `Run()`, which will re-invoke the function in a loop. If the state changes, we can restart instruction execution. The OS program must increment the PC to start execution at the instruction following a syscall. And at last, through `singleStep` and `Debugger()`, the code sets the interrupt state to `UserMode` and debug.

## 2 Implementation

Given the source code of `code/test/consoleIO_test1.c` and `code/test/consoleIO_test2.c`, one should anticipate to execute `consoleIO_test2.c` and get a series of ascending numbers 20, 21, 22, 23, 24, 25 and execute `consoleIO_test1.c` and get a sequence of dropping numbers 9, 8, 7, 6.



```
[os23s27@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e
consoleIO_test2
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

Figure 1: Correct results with multiprogramming

Yet, when we run the multi-programming testing command, the result of `consoleIO_test1.c` increases at the end. This is because Nachos uses a one to one mapping strategy, which means that only uni-programming is supported and we only have a single page table.

All processes will utilize the same page table and so map to the same physical page. If multi-programming is used, this technique must be altered. This is a concern with multi-programming since the processes will execute the identical code section. We must alter `NachOS-4.0_MP2/code/userprog/addrspace.cc` and `NachOS-4.0_MP2/code/userprog/addrspace.h`, which generate the page table for a process. The virtual and physical pages have the same number in the `AddrSpace` constructor. This must be changed when implementing multi-programming.

Inside class `AddrSpace`, I define a new static bool called `static bool UsedPhysicalPages[]` for tracking used physical pages. I defined the function in `private` because we don't want to expose the implementation details of `AddrSpace` to other classes.

```
1 class AddrSpace {
2     ...
3     private:
4         ...
5         // add
6         static bool UsedPhysicalPages[]; // track used physical pages
7 };
```

## 2.1 userprog/addrspace.cc

The routine `userprog/addrspace.cc` is a routines to manage address spaces. At first, the function statement

```
1 bool AddrSpace::UsedPhysicalPages[NumPhysPages] = {FALSE};
```

initializes a boolean array named `UsedPhysicalPages` with a size of `NumPhysPages` and it sets the value `FALSE` to all the elements of the array.

While using multiprogramming, it is necessary to maintain track of the physical page information for each process, which can be found in the code as `pageTable[i].physicalPage`. Each process is represented by an instance of `AddrSpace`. So, the page table allows the operating system to locate the correct physical page when a program is run. Because all processes in the original Nachos-4.0 start on the same physical page, multitasking produces unanticipated results. The page table is updated to reflect the location of the matching physical page when the process loads into memory.

Hence, I modify the function `AddrSpace::Load(char *fileName)` to search for the first unused physical page using a linear search.

```
1 pageTable = new TranslationEntry[numPages];
2 for (unsigned int i = 0; i < numPages; i++){
3     unsigned int j = 0;
4     pageTable[i].virtualPage = i;
5     while (j < NumPhysPages && AddrSpace::UsedPhysicalPages[j] == TRUE){
6         j++;
7     }
8     ASSERT(j < NumPhysPages);
9     pageTable[i].physicalPage = j;
10    AddrSpace::UsedPhysicalPages[j] = TRUE;
11    pageTable[i].valid = TRUE;
12    pageTable[i].use = FALSE;
13    pageTable[i].dirty = FALSE;
14    pageTable[i].readOnly = FALSE;
15    size = numPages * PageSize;
16 }
```

`TranslationEntry` is a class that contains the information about the page. It contains the virtual page number, the physical page number, the valid field, the use field, the dirty field, and the read only field. Hence, in this part of code we set a new `TranslationEntry` for each page in the page table. Using the for loop, we will run on all the pages and set the virtual page number to the index of the page in the page table. Then we will run on the `UsedPhysicalPages` array and find the first free page.

So, `ASSERT` is responsible to check if we have enough physical pages to run the program. If `j` is less than the number of physical pages, it means that we have a free page and we can run the program. Otherwise, we will print an error message and exit the program. After that we will set the physical page number to the index of the free page in the `UsedPhysicalPages` array. Then we will set the valid field to true, and the rest of the fields to false. After that we will set the size of the address space to the number of pages multiplied by the page size.

At runtime, we must determine the virtual memory address in order to locate the entry point. The relationship is as follows: the page offset is equal to `virtualAddress % PageSize`, where `PageSize` is the page base and `virtualAddress` is the virtual address. The main memory starting point (physical address) is equal to the sum of the page base and page offset.

```
1 if (noffH.code.size > 0) {
2     DEBUG(dbgAddr, "Initializing code segment.");
3     DEBUG(dbgAddr, noffH.code.virtualAddr << " ", " << noffH.code.size);
4     unsigned int virtualAddress = noffH.code.virtualAddr;
5     unsigned int physicalAddress = pageTable[virtualAddress/PageSize].physicalPage *
6     PageSize + virtualAddress % PageSize;
7     executable->ReadAt(
8     &(kernel->machine->mainMemory[physicalAddress]),
9     noffH.code.size, noffH.code.inFileAddr);
10 }
```

and we will also do the same for code segment

```
1 if (noffH.initData.size > 0) {
2     DEBUG(dbgAddr, "Initializing data segment.");
3     DEBUG(dbgAddr, noffH.initData.virtualAddr << " ", " << noffH.initData.size);
4     unsigned int virtualAddress = noffH.code.virtualAddr;
5     unsigned int physicalAddress = pageTable[noffH.initData.virtualAddr/PageSize].
6     physicalPage * PageSize + virtualAddress % PageSize;
7     executable->ReadAt(
8     &(kernel->machine->mainMemory[physicalAddress]),
9     noffH.initData.size, noffH.initData.inFileAddr);
10 }
```

## 2.2 Handle Memory Limit Exception

At last, `MemoryLimitException` is thrown and the application is terminated if the sum of the necessary space exceeds the available space. Address space availability must be verified during file-based user program loading in order to properly handle the `MemoryLimitException`.

First, I declare an exception `MemoryLimitException` in `machine.h`

```
1 enum ExceptionType { NoException,
2   SyscallException,
3   PageFaultException,
4   ReadOnlyException,
5   BusErrorException,
6   AddressErrorException,
7   OverflowException,
8   IllegalInstrException,
9   MemoryLimitException,
10  NumExceptionTypes
11 };
```


and implement it inside `addrspace.cc`. In this code we check if the number of pages is bigger than the number of physical pages, if so we will delete the executable and call the `ExceptionHandler` to handle `MemoryLimitException`. And we will return false to terminate the program.

```
1 if (numPages > NumPhysPages) {
2     delete executable;
3     ExceptionHandler(MemoryLimitException);
4     return FALSE;
5 }
```

Hence to test it, I tried to modify `consoleIO_test1` and gave it a large number of array for input

```
1 #include "syscall.h"
2 int arr[4028];
3 int main() {
4     int n;
5     for (n=9; n>5; n--) {
6         PrintInt(n);
7     }
8     return 0;
9     //Halt();
10 }
```

and the result show that it has successfully handle the exception



```
[os23s27@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e
consoleIO_test2
consoleIO_test2
Unexpected user mode exception 8
Assertion failed: line 201 file ../userprog/exception.cc
Aborted
```

Figure 2: Correctly handle the exception about insufficient memory



3 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

3.1 How does Nachos allocate the memory space for a new thread(process)?

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

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

The execution of a program is a crucial step in the operation of NachOS. There are a number of stages to this procedure, which may be generally categorized as follows:

A new thread is established and initialized briefly at the outset. Then, a new addressing context is introduced here. To start fresh, `bzero()` is called from the address space constructor. Once the code to be executed has been loaded into the thread, the thread will eventually call `Fork`.

```
1 int Kernel::Exec(char* name) {
2     t[threadNum] = new Thread(name, threadNum);
3     t[threadNum]->space = new AddrSpace();
4     t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
5     threadNum++;
6
7     return threadNum-1;
8 }
9
10 void Thread::Fork(VoidFunctionPtr func, void *arg) {
11     Interrupt *interrupt = kernel->interrupt;
12     Scheduler *scheduler = kernel->scheduler;
13     IntStatus oldLevel;
14
15     StackAllocate(func, arg);
16
17     oldLevel = interrupt->SetLevel(IntOff);
18     scheduler->ReadyToRun(this);
19     (void) interrupt->SetLevel(oldLevel);
20 }
```

The program that will be run at that moment is sent to `Fork` via the `funcPtr` of `ForkExecute`. To begin setting up thread stacks, the `StackAllocate` function is then used. By executing the instruction `machineState[InitialPCState] = (void*)func;`, the original `funcPtr` is set as the code to be run by the program counter in the future. e.

```
1 void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
2 {
3     stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
4     ...
5     machineState[PCState] = (void*)ThreadRoot;
6     machineState[StartupPCState] = (void*)ThreadBegin;
7     machineState[InitialPCState] = (void*)func;
8     machineState[InitialArgState] = (void*)arg;
9     machineState[WhenDonePCState] = (void*)ThreadFinish;
10    ...
11 }
```

Disabling interrupts and doing a preliminary initialization of the thread are done now. The thread is scheduled for future execution by the CPU after `scheduler->ReadyToRun(this)` places it in the ready queue.

To determine which thread should be processed next, the CPU scheduler will check the current value of the program counter and load it from the ready queue. In order to run a program on NachOS, it is necessary to do numerous tasks, such as starting a new thread, allocating memory, and setting up the stack. This is accomplished by loading the program into memory and inserting it into the ready queue using `Fork`.

3.4 How does Nachos create and manage the page table?

The kernel in Nachos is responsible for generating and maintaining the page table. The page table is responsible for translating between a process's virtual addresses and the corresponding memory locations. The `TranslationEntry` and `TranslationEntry *pageTable` types will be defined in `translate.h` and



addrspace.h, respectively, during implementation. Later addrspace.cc updates involving the Load function can make use of pageTable to perform Virtual Memory-related processing and translation. Moreover, we will clear up the Page Table once a process terminates. The kernel frees up the RAM that was allocated to the process's page table. The procedure also frees up any physical pages that were set aside for it.

```

1  ExceptionType Machine::Translate(int virtAddr, int* physAddr, int size, bool writing)
2  {
3      unsigned int vpn, offset;
4      TranslationEntry *entry;
5      unsigned int pageFrame;
6
7      vpn = (unsigned) virtAddr / PageSize; /* virtual page number */
8      offset = (unsigned) virtAddr % PageSize; /* offset within the page */
9      ...
10     entry = &pageTable[vpn]; /* translate using pageTable */
11     ...
12     pageFrame = entry->physicalPage;
13     ...
14     *physAddr = pageFrame * PageSize + offset;
15     return NoException;
16 }

```

### 3.5 How does Nachos translate addresses?

The translate is define in addrspace.h and machine.h respectively. When a process accesses a virtual address, the hardware translates the virtual address to a physical address using the page table. The page table is stored in memory and managed by the kernel.

```

1  class TranslationEntry {
2  public:
3      int virtualPage;
4      int physicalPage;
5      ...
6  };

```

### 3.6 Which object in NachOS acts the role of process control block

The object that NachOS acts the role of process control block (PCB) are:

- Program Counter: holds the current value of the program counter
- Process State: represent the current status of the process
- Thread State: indicates the current state of the thread
- Thread ID: identify and manage threads in the system
- CPU register
- memory management info (e.g. base / limit register)

```

1  class Thread {
2  private:
3      int *stackTop; /* the current stack pointer */
4      void *machineState[MachineStateSize]; /* all registers except for
5                                              * stackTop */
6
7      int *stack; /* bottom of the stack */
8      ThreadStatus status; /* ready, running or blocked */
9      char* name;
10     int ID;
11     int userRegisters[NumTotalRegs]; /* user-level CPU register state */
12     ...
13 public:
14     AddrSpace *space; /* user code this thread is running */
15     ...
16 };

```

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

The thread will be called in `thread.cc` in `Thread::Fork` where `Scheduler *scheduler = kernel->scheduler;` will be executed. This line of code will put the Thread that has allocated resources into the Ready Queue for future CPU scheduling execution

```
1  void Thread::Fork(VoidFunctionPtr func, void *arg)
2  {
3      Interrupt *interrupt = kernel->interrupt;
4      Scheduler *scheduler = kernel->scheduler;
5      IntStatus oldLevel;
6
7      DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
8      StackAllocate(func, arg);
9
10     oldLevel = interrupt->SetLevel(IntOff);
11     scheduler->ReadyToRun(this); // ReadyToRun assumes that interrupts
12     // are disabled!
13     (void) interrupt->SetLevel(oldLevel);
14 }
```