Operating System

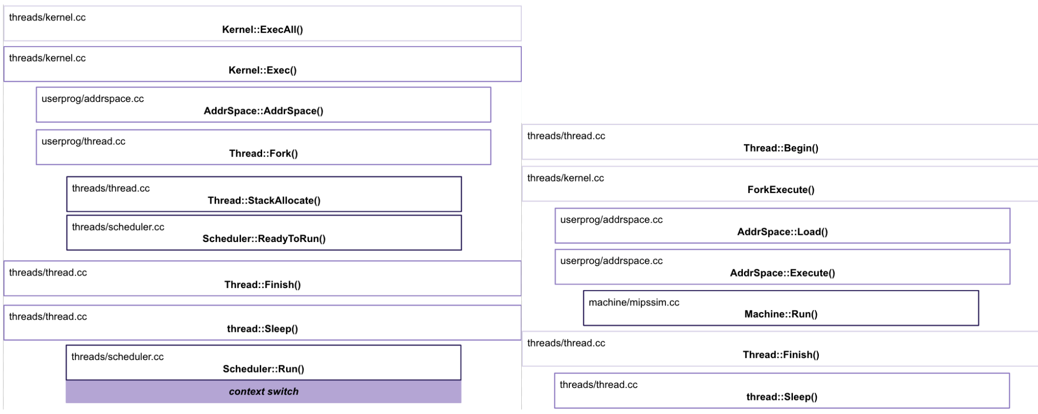
MP2: Multi-Programming

Team Member(Team 10): 林子謙 (112065517)、陳奕潔 (110000212)

Contributions: Both contributed to page table implementing and report writing.

Contents

1 Code Tracing



(a) Initialization (b) Executing a Thread

Figure 1: Code Flow

In this assignment, we begin tracing the code from `Kernel::ExecAll()`. The code flow is illustrated in Figure 1. First, we explain the `Kernel::Kernel()` function, then detail how the components involved in initializing and executing a thread work, to provide an understanding of how the system runs a single program.

1.1 Kernel::Kernel()

`Kernel::Kernel()` (shown in Figure 2) initializes system settings, including console devices, network reliability, and command-line flags for debugging, and other configurations. If the `-e` flag is specified (line 49), the system stores the file name into `execfile`, allowing it to access the program file the user wants to execute.

1.2 Initialization

1.2.1 Kernel::ExecAll()

`Kernel::ExecAll()`, in Figure 3, is called by the initial thread to execute each user program by invoking `Kernel::Exec()` for each program file store in `execfile`. After completing all executions, the thread calls `Thread::Finish()` to trigger a context switch.

```
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];
```

Figure 2: Kernel::Kernel()

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

Figure 3: Kernel::ExecAll()

```
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;
    /*
        cout << "Total threads number is " << execfileNum << endl;
        for (int n=1;n<=execfileNum;n++) {
                t[n] = new Thread(execfile[n]);
                t[n]->space = new AddrSpace();
                t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
                cout << "Thread " << execfile[n] << " is executing." << endl;
            }
            cout << "debug Kernel::Run finished.\n";
    */
    //  Thread *t1 = new Thread(execfile[1]);
    //  Thread *t1 = new Thread("../test/test1");
    //  Thread *t2 = new Thread("../test/test2");

    //    AddrSpace *halt = new AddrSpace();
    //  t1->space = new AddrSpace();
    //  t2->space = new AddrSpace();

    //    halt->Execute("../test/halt");
    //  t1->Fork((VoidFunctionPtr) &ForkExecute, (void *)t1);
    //  t2->Fork((VoidFunctionPtr) &ForkExecute, (void *)t2);

    //  currentThread->Finish();
    //    Kernel::Run();
    //  cout << "after ThreadedKernel:Run();" << endl;  // unreachable
}
```

Figure 4: Kernel::Exec()

1.2.2 Kernel::Exec()

`Kernel::Exec()` (shown in in Figure 4) creates a new thread, currently represented as a process, passes the function address and its arguments to `Fork()`, and initializes its thread (process) control block (a `Thread` object is a thread control block).

1.2.3 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);
}
```

Figure 5: AddrSpace::AddrSpace()

`AddrSpace::AddrSpace()`, called by `Kernel::Exec()`, creates a page table for the thread and clears the entire main memory (see Figure 17).

The function uses `TranslationEntry` objects to map logical memory to physical memory. Since the system currently supports only uniprogramming, this mapping is one-to-one, allowing the entire main memory to be reset.

1.2.4 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);
}
```

Figure 6: Thread::Fork()

`Thread::Fork()`, also called by `Kernel::Exec()`, allocates and initializes a stack for the thread using `StackAllocate()`. Finally, it puts the thread on the ready queue with interrupts disabled. The function is shown in Figure 6.

1.2.5 Thread::StackAllocate()

`Thread::StackAllocate()`, called by `Thread::Fork()`, initializes the stack for the thread (see Figure 7). The function sets the stack top based on the hardware architecture. It then places `ThreadRoot`, `ThreadBegin`, the function pointer and its arguments, and `ThreadFinish` on the stack. `ThreadRoot` enables interrupts, allowing the thread to execute upon context switching and ensures the thread stops upon completion. Specifically, `ThreadBegin` and `ThreadFinish` are invoked by `ThreadRoot` to call `Thread::Begin()` and `Thread::Finish()`, respectively.

1.2.6 Scheduler::ReadyToRun()

`Scheduler::ReadyToRun()` (see Figure 8), also called by `Thread::Fork()`, sets the thread status to ready and places it on the ready queue after ensuring interrupts are disabled.

1.2.7 Thread::Finish()

After placing all threads on the ready queue, the system returns to `Kernel::ExecAll()` and calls `Thread::Finish()`. `Thread::Finish()`, shown in Figure 9, first disable interrupts for `Sleep()`. If all threads have finished, it deallocats the thread; otherwise, it calls `Sleep()` to perform a context switch, as the thread may still be needed by other threads.

1.2.8 Thread::Sleep()

`Thread::Sleep()` (see Figure 10) blocks the current thread and attempts to retrieve the first thread from the ready queue. If no threads are available, the system enables interrupts to allow other programs to execute. Otherwise, it calls `Run()` to perform a context switch.

1.2.9 Scheduler::Run()

`Scheduler::Run()`, shown in Figure 11, is to perform a context switch. For each context switching, the system will save CPU state and machine state for the space for each user program. If the current thread has finished, it will be mark as destroyed and the system executes the next thread and invokes `Thread::Begin()`

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

#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;  // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

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

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

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

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

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

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

Figure 7: Thread::StackAllocate()

```cpp
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);
}
```

Figure 8: Scheduler::ReadyToRun()

```
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
}
```

Figure 9: Thread::Finish()

```
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);
}
```

Figure 10: Thread::Sleep()

```cpp
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;
    }

    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

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

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

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

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

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

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

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

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

Figure 11: Scheduler::Run()

of the next thread. Otherwise, after context switching, this thread will restore its data by writing the register state into machine registers to continue executing.

In the initialization phase, once the main thread finishes, the system performs a context switch to the next thread, marking the end of initialization.

1.3 Executing a Thread

1.3.1 Thread::Begin()

```
void Thread::Begin() {
    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Beginning thread: " << name);

    kernel->scheduler->CheckToBeDestroyed();
    kernel->interrupt->Enable();
    if (kernel->execExit && this->getIsExec()) {
        kernel->execRunningNum++;
    }
}
```

Figure 12: Thread::Begin()

`Thread::Begin()` is called by `ThreadRoot` after a context switch (see Figure 12). The function first releases resources of the previous thread if it has finished by calling `Scheduler::CheckToBeDestroyed()`, then enables interrupts to support time-sharing. Finally, it updates the count of running threads in the system.

1.3.2 Scheduler::CheckToBeDestroyed()

`Scheduler::CheckToBeDestroyed()`, as Figure 13 shows, deletes deletes the finished thread by deallocating its stack.

1.3.3 ForkExecute()

`ForkExecute()`, shown in Figure 14, is used to execute forked threads. If the current thread's data has not been loaded, it calls `Load()`. The system then uses the thread ’ s stack, which stores the functions and their arguments, to execute the program.

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

Figure 13: Scheduler::CheckToBeDestroyed()

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

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

Figure 14: ForkExecute()

1.3.4 AddrSpace::Load()

`AddrSpace::Load()`, shown in Figure 15, loads the code and data of a user program into main memory by first segmenting and then paging.

It opens the program file and reads a special header, noffH, to get information about the program's segments. Using this header, it calculates the required memory size and the number of pages. If there is enough physical memory, it loads the code and data segments into memory using virtual address, where virtual addresses currently match physical addresses. Finally, it closes the file and returns `TRUE` to indicate successful loading.

1.3.5 AddrSpace::Execute()

`AddrSpace::Execute()`, in Figure 16, initializes the machine by calling `InitRegisters()` to set up the machine registers, including the stack register, the initial program counter, and the next program counter. It also specifies the page table location for the process using `RestoreState()` (in contrast to context switching, where it retrieves the machine state).

After this setup, it calls `machine->Run()` to begin executing the program (handling instruction fetching, etc.).

`Machine::Run()` continuously fetches and executes instructions one by one. If resources are insufficient, the current thread goes to sleep until resources become available. Execution continues until an `SC_Exit()` exception occurs, which trig-

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

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

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

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

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

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

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

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

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

Figure 15: AddrSpace::Load()

```
void AddrSpace::Execute(char *fileName) {
    kernel->currentThread->space = this;

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

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

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

Figure 16: AddrSpace::Execute()

gers `kernel->currentThread->Finish()`. The `Thread::Finish()` function has been explained above, so we will skip it here.

1.4 Questions in Spec

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

  In `Kernel::Exec()`, a new `AddrSpace` is assigned to the new thread. The `AddrSpace` constructor initializes a page table using `TranslationEntry` to map logical memory directly to physical memory, as the system currently supports only uniprogramming.

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

  As mention, the `AddrSpace` constructor initializes a page table with `TranslationEntry` objects, mapping logical memory directly to physical memory. It sets the valid field to `TRUE` and the use, dirty, and readOnly bits to `FALSE`.

  In `Kernel::Exec()`, `Thread::Fork()` is called with the function address and arguments. `Thread::Fork()` then calls `Thread::StackAllocate()`, which places the function pointer and arguments on the stack.

  When `ForkExecute()` is invoked, it calls `AddrSpace::Load()` to load the user binary code. `AddrSpace::Load()` uses a special header (`noffH`) to load each segment, mapping virtual addresses to physical memory since they are currently identical.

- **How does Nachos create and manage the page table?**

  As mention, in `Kernel::Exec()`, a new `AddrSpace` is created and stored in the `space` attribute of the new thread. The `AddrSpace` constructor initializes a page table using `TranslationEntry` to map logical memory directly to physical memory, as the system currently supports only uniprogramming. This setup allows the system to access the page table through `space`.

- **How does Nachos translate addresses?**

  When loading a program file into main memory, `AddrSpace::Load()` loads the user binary code using a special header (`noffH`) by segmenting and paging as mention above. `AddrSpace::Load()` uses virtual addresses, which currently match physical addresses, to load each segment into memory.

During execution, `Machine::OneInstruction()` calls `Machine::ReadMem()`, which in turn calls `Machine::Translate()`. `Machine::Translate()` uses the virtual address and page size to determine the virtual page number and the offset within the page. It accesses the `physicalPage` attribute in the page table entry to get the frame number, allowing it to compute the physical address.

- **How does Nachos initialize the machine status (registers, etc) before running a thread (process)?**

  When a thread is executed, `ForkExecute()` calls `AddrSpace::Execute()`. `AddrSpace::Execute()` initializes the machine by calling `InitRegisters()`, which sets up the machine registers, including the stack register, the initial program counter, and the next program counter.

- **Which object in Nachos acts the role of process control block?**

  `Thread`. Currently, each thread represents a single process. `Thread` objects include the thread ID, thread status, stack, page table, and other PCB-related information.

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

  A thread is added to the ready queue when it is forked in `Thread::Fork()`. `Scheduler::ReadyToRun()`, called by `Thread::Fork()`, sets the thread status to ready and places it on the ready queue.

  If a thread is waiting for resources, `Semaphore::V()` will call `Scheduler::ReadyToRun()` once the resources become available, adding the thread back to the ready queue.

2 Page Table Implementation

In order to support multiprogramming, we need to take the following steps:

1. Construct a page table with an appropriate number of entries for each process.

2. Check whether there are enough frames to accommodate the process. If so, assign the available frames to it.

3. Load the corresponding code into these frames.
4. Finally, if the process has completed its work and is to be terminated, delete the page table and return the frames to the kernel.

2.1 Construct a page table

For the following explanation, please refer to Figures 17 through 20.

Unlike the original implementation in NachOS, multiprogramming requires constructing the page table based on the size of each process. Therefore, we first comment out the code in `AddrSpace::AddrSpace()` (As shown in Figure 17) and create a page table constructor that takes `numPages` as an argument (As shown in Figure 18 and Figure 19), which will be invoked in `AddrSpace::Load()` (As shown in Figure 20, line 164.). (Since we need to know the size of the process to construct the page table, which is only available in `AddrSpace::Load()`.)

```
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;  // modify in Load()
    //     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);
}
```

Figure 17: Comment out AddrSpace::AddrSpace()

2.2 Assign the available frames

Before calling `AddrSpace::ConstructPageTable()`, we need to check the number of free frames to ensure that the process has enough space in memory. To track free frames, we add `int frameTable[NumPhysPages]` to record empty frames to the `Kernel` class in `kernel.h`. In this array, the index represents the frame number of a physical page, and if a frame is available, its value is set to 0 (and vice versa). Additionally, we add `int NumFreeFrame`, which represents the number of free frames in memory, too. Please refer to Figure 21 and Figure 22.

In addition, we add a new exception type, `MemoryLimitException`, in `machine.h` and set up a corresponding exception handler in `exception.cc` to handle situations where there is not enough space for the process (On line 153, Figure 20). You may refer to Figure 23 and Figure 24 for detail.

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

    void ConstructPageTable(int numPages);

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

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

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

    // Translate virtual address _vaddr_
    // to physical address _paddr_. _mode_
    // is 0 for Read, 1 for Write.
    ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);

  private:
    TranslationEntry *pageTable;  // Assume linear page table translation
                                  // for now!
    unsigned int numPages;        // Number of pages in the virtual
                                  // address space

    void InitRegisters();  // Initialize user-level CPU registers,
                           // before jumping to user code
};
```

Figure 18: AddrSpace::ConstructPageTable() Declaration

```
void AddrSpace::ConstructPageTable(int numPages) {
    pageTable = new TranslationEntry[numPages];
    for (int i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i;   // modify in Load()
        pageTable[i].physicalPage = i;  // modify in Load()
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }
}
```

Figure 19: AddrSpace::ConstructPageTable()

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

    if (executable == NULL) {⋯

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

#ifdef RDATA⋯
#else⋯
#endif
    // DEBUG(dbgThread, "OwO " << noffH.code.size << ", " << noffH.initData.size⋯
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    // Check #FreeFrames and #UsedPages
    if (kernel->NumFreeFrame < numPages) ExceptionHandler(MemoryLimitException);

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

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

    // Construct page table then find empty frames to give to the process
    kernel->currentThread->space->ConstructPageTable(numPages);
    for (int i = 0, cnt = 0; i < NumPhysPages && cnt < numPages; i++) {
        if (kernel->frameTable[i] == 0) {
            kernel->frameTable[i] = 1;
            kernel->NumFreeFrame--;
```

Figure 20: Invokes ConstructPageTable in AddrSpace::Load()

```cpp
class Kernel {

    int hostName;   // machine identifier
    int frameTable[NumPhysPages];
    int NumFreeFrame;

  private:
```

Figure 21: `int frameTable[NumPhysPages]`, `int NumFreeFrame` Declaration

```
Kernel::Kernel(int argc, char **argv) {
    randomSlice = FALSE;
    debugUserProg = FALSE;
    execExit = FALSE;
    consoleIn = NULL;   // default is stdin
    consoleOut = NULL;  // default is stdout
    frameTable[NumPhysPages] = {0};
    NumFreeFrame = NumPhysPages;
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
    reliability = 1;  // network reliability, default is
```

Figure 22: `int frameTable[NumPhysPages]`, `int NumFreeFrame` Initialization

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

Figure 23: New exception type: `MemoryLimitException`

```
void ExceptionHandler(ExceptionType which) {
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception "
                      << which << " type: " << type << ", "
                      << kernel->stats->totalTicks);
    switch (which) {
        case MemoryLimitException:
            cerr << "Unexpected user mode exception " << (int)which << "\n";
            ASSERT(false);
```

Figure 24: Handling `MemoryLimitException` in exception handler.

Finally, we identify available frames by iterating through `frameTable[NumPhysPages]` with a `for` loop and record them in the page table. As shown in Figure 25. It is worth noting that we should also clean up the frame assigned to this process to avoid using incorrect data (Figure 25, line 171.).

```
164    kernel->currentThread->space->ConstructPageTable(numPages);
165    for (int i = 0, cnt = 0; i < NumPhysPages && cnt < numPages; i++) {
166        if (kernel->frameTable[i] == 0) {
167            kernel->frameTable[i] = 1;
168            kernel->NumFreeFrame--;
169            kernel->currentThread->space->pageTable[cnt].physicalPage = i;
170            // zero out the entire address space
171            bzero(&kernel->machine->mainMemory[i * PageSize], PageSize);
172            cnt++;
173        }
174    }
```

Figure 25: Finding free frames withing `frameTable[NumPhysPages]`

2.3 Load the code

For those segments that are not read only, you may refer to Figure 26 and Figure 27 for more implementation details.

To place the code into the correct frame, we first determine its page number and offset, then calculate its physical address (You may refer to Figure 26, from line 179 to 184, and Figure 27, from line 196 to 202.). In this implementation, we use `ReadAt()` to read the code into memory. Here, the code is divided into `code` and `initData`, and the method for loading them are the same.

```
177    if (noffH.code.size > 0) {
178        // Calculate physical address to place code
179        int PhysAddr_base = kernel->currentThread->space
180                            ->pageTable[noffH.code.virtualAddr / PageSize]
181                            .physicalPage *
182                        PageSize;
183        int PhysAddr_offset = noffH.code.virtualAddr % PageSize;
184        int physAddr = PhysAddr_base + PhysAddr_offset;
185
186        DEBUG(dbgThread, "Initializing code segment.");
187        DEBUG(dbgThread, "virtualAddr: " << noffH.code.virtualAddr
188                                     << ", size: " << noffH.code.size);
189
190        executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),
191                          noffH.code.size, noffH.code.inFileAddr);
192        DEBUG(dbgThread, "Finish code segment initialization.");
193    }
```

Figure 26: Loading code into corresponding free frame.

For the segments that place read only data, we need to change the `readOnly` variable to "True". As shown in Figure 28.

```
if (noffH.initData.size > 0) {
    // Calculate physical address to place initData
    int PhysAddr_base =
        kernel->currentThread->space
            ->pageTable[noffH.initData.virtualAddr / PageSize]
            .physicalPage *
        PageSize;
    int PhysAddr_offset = noffH.initData.virtualAddr % PageSize;
    int physAddr = PhysAddr_base + PhysAddr_offset;

    DEBUG(dbgThread, "Initializing data segment.");
    DEBUG(dbgThread, "virtualAddr: " << noffH.initData.virtualAddr
                                     << ", size: " << noffH.initData.size);
    executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),
                       noffH.initData.size, noffH.initData.inFileAddr);
    // May overwrite
    DEBUG(dbgThread, "Finish code data initialization.");
}
```

Figure 27: Loading initial data into corresponding free frame.

```
#ifdef RDATA
    if (noffH.readonlyData.size > 0) {
        int PhysAddr_base =
            kernel->currentThread->space
                ->pageTable[noffH.readonlyData.virtualAddr / PageSize]
                .physicalPage *
            PageSize;
        int PhysAddr_offset = noffH.readonlyData.virtualAddr % PageSize;
        int physAddr = PhysAddr_base + PhysAddr_offset;

        kernel->currentThread->space
            ->pageTable[noffH.readonlyData.virtualAddr / PageSize]
            .readOnly = TRUE;

        DEBUG(dbgAddr, "Initializing read only data segment.");
        DEBUG(dbgThread,
              "virtualAddr: " << noffH.readonlyData.virtualAddr
                              << ", size: " << noffH.readonlyData.size);
        executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),
                           noffH.readonlyData.size,
                           noffH.readonlyData.inFileAddr);
    }
```

Figure 28: Loading read only data into corresponding free frame.

2.4 Return the frames

Once the process completes its work, we need to delete the page table. There are a few things to keep in mind. First, `frameTable[NumPhysPages]` needs to be updated back to 0. Second, `NumFreeFrame` should be incremented to reflect the freed frames. Figure 29 shows our implementation of `AddrSpace::~AddrSpace()`.

```
AddrSpace::~AddrSpace() {
    for (int i = 0; i < numPages; ++i) {
        kernel->frameTable[kernel->currentThread->space->pageTable[i]
                            .physicalPage] = 0;
        kernel->NumFreeFrame++;
    }

    delete pageTable;
}
```

Figure 29: Returning frames to the kernel.