MP2_report

109070025 林泓錩

1 Trace Code

The first part, we have to trace the code starting from "threads/kernel.cc Kernel::ExecAll()", but in the trace process, I found that before looking to the Kernel::ExecAll(), we need to look Kernel::Kernel() first.

- **Kernel::Kernel()**

```
Kernel::Kernel
 Interpret command line arguments in order to determine flags
 for the initialization (see also comments in main.cc)
```

The purpose of Kernel::Kernel() function is to interpret command line arguments in order to determine flags for the initialization , and as the comment hint, I also see the code in the main.cc.

```
kernel = new Kernel(argc, argv);

kernel->Initialize();
```

```
else if (strcmp(argv[i], "-e") == 0) {
  execfile[++execfileNum]= argv[++i];
  cout << execfile[execfileNum] << "\n";
```

In the main.cc, ther is a int main(int argc, char **argv) function. It will handle the command line arguments, and create a new Kernel. Back to the Kernel::Kernel(), it will help to translate the command line arguments, and then store the file that will be execute in the the array execfile[ ].

```
currentThread = new Thread("main", threadNum++);
currentThread->setStatus(RUNNING);

stats = new Statistics();          // collect statistics
interrupt = new Interrupt;         // start up interrupt handling
scheduler = new Scheduler();       // initialize the ready queue
```

Then int main() will call kernel->initialize, using the arguments that have already been handled to initialize the NachOS global data structure, such as Statistic(performance matrix), Interrupt(start up interrupt handling), Scheduler(initialize the ready queue), etc.. In the meanwhile, kernel->initialize will set the currentThread to main, and set its status "Running".

```
kernel->ExecAll();
```

After doing the initialization, it will back to the main() function, and call the Kernel::ExecAll() function.

- **Kernel::ExecAll()**

```
for (int i=1;i<=execfileNum;i++) {
    int a = Exec(execfile[i]);
}
```

Executin of the Thread: Call Kernel::Exec() to run through all the files in execfile[ ] array.

```
currentThread->Finish();
```

After finishing the for() loop, it will call Thread::Finish() to tell the scheduler to call the destructor to destruct the Thread once it is running in the context of a different thread.

- **Kernel::Exec()**

```
t[threadNum] = new Thread(name, threadNum);
```

First, it will create a thread and store it in the array of thread class.

```
t[threadNum]->space = new AddrSpace();
```

Then, using AddrSpace::AddrSpace to create a space for the thread, and store it in the space in t[ ], which it created in the first step.

```
t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
```

The last step, it will call fork() function to put thread into ready queue.

- **AddrSpace::AddrSpace()**

The purpose of AddrSpace::AddrSpace() is to create an address space to run a user program.

```
pageTable = new TranslationEntry[NumPhysPages];
```

To do that, it will create a pageTable to set up the translation from program memory to physical memory.

```
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;
}
```

Every element in the table is an object of TranslationEntry, which use variables to record vitualPage, physicalPage, valid,...etc..According to those variables,

we can know the status of a page.

As for the translation from program memory to physical memory, since the virtual page number is the same with the physical page number, for NachOs, virtual address equals to physical address, this is simple 1:1.

- **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);
}
```

Put in the VoidFunctionPtr for ForkExecute, which is a pointer to a function taking an arbitrary pointer argument and returning nothing, and the threadNum that would like to execute.

Then, it will call StackAllocate() to allocate a stack for the thread and set up machineState as preparation before running.

After that set the Setlevel to be IntOff, for ReadyToRun() assumes that interrupts are disabled, in the long run, call the ReadyToRun() to put the thread on the ready queue.

- **Thread::StackAllocate()**

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

The purpose of Thread::StackAllocate() is to allocate and initialize an execution stack. The stack is initialized with an initial stack frame for ThreadRoot, which enables interrupts, calls (*func)(arg), and calls Thread::Finish().

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

```
void ThreadRoot();

// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
```

```
static void ThreadFinish()    { kernel->currentThread->Finish(); }
static void ThreadBegin() { kernel->currentThread->Begin(); }
```

The address of ThreadRoot is made the return address of SWITCH() by the routine Thread::StackAllocate().
Set Begin() function in ThreadBegin as StartupPCstate, which will be called by ThreadRoot when a thread is about to begin executing the forked procedure.
Set ”func” as InitialPCstate, which will be called by ThreadRoot to executing the forked procedure, and ”arg” will be InitialArgState.
P.S. ”func”: void ForkExecute(Thread *t) ; ”arg”: the name of the execution thread.
Set Finish( ) function as WhenDonePCstate, which will be called by ThreadRoot when a thread is done.

- **Scheduler::ReadyToRun()**

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

Mark a thread as ready, but not running.
Put it on the ready list, for later scheduling onto the CPU.

- **Thread::Finish()**

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);                // invokes SWITCH
    // not reached
}
```

Thread::Finish() is called by ThreadRoot when a thread is done executing the forked procedure.
NOTE: 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.

It will do SetLevel(Intoff) to disable interrupts because Sleep() assumes interrupts are disabled.
After that, it will call Thread::Sleep().

- **Thread::Sleep()**

  Relinquish the CPU, because the current thread has either finished or is blocked waiting on a synchronization variable (Semaphore, Lock, or Condition). In the latter case, eventually some thread will wake this thread up, and put it back on the ready queue, so that it can be re-scheduled.

```
status = BLOCKED;
```

  Set Thread::Status be BLOCKED to declare it is idle now, stop executing currentThread and find nextThread.

```
while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle();  // no one to run, wait for an interrupt
}
```

  Through the while loop to check whether there is thread on the ready queue, if not, call Interrupt::Idle to signify that we should idle the CPU until the next I/O interrupt occurs.

```
kernel->scheduler->Run(nextThread, finishing);
```

  Then, use scheduler->Run() to dextruct current thread, and execute nextThread.

- **Scheduler::Run()**

  Dispatch the CPU to nextThread. Save the state of the old thread, and load the state of the new thread, by calling the machine dependent context switch routine, SWITCH.

```
Thread *oldThread = kernel->currentThread;
```

  Define oldThread be the currentThread.

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

  If there is a need to delete currentThread, set toBeDestory to be the oldThread that we defined in the previous step.

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

If the oldThread->space is not NULL, it means that it is a user program(only user program will new AddrSpace()). Store the value of CPU register into the user register of this thread, and also store the value of page in machine into this thread.

```
oldThread->CheckOverflow();
```

Check that oldThread doesn't have stack overflow (oldThread->CheckOverflow()).

```
kernel->currentThread = nextThread;
nextThread->setStatus(RUNNING);
```

Set currentThread to nextThread, and set the status of Thread to RUNNING(original: BLOCK).

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

Call SWITCH to let CPU execute nextThread, and the value of CPU register will change from the value of oldThread to nextThread.

```
CheckToBeDestroyed();
```

Check whether the new Thread run successfully, then also can check whether there is a need to destory the oldThread, if yes, destory it. And, ForkExecute() will be executed in oldThread(after nextThread is switched).

- **Kernel::ForkExecute()**

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

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

Call AddrSpace::Load() to load a user program into memory from a file.
After that it call AddrSpace::Execute() with the name of thread as argument.

- **AddrSpace::Load()**
  Load a user program into memory from a file.

```
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);
```

Set an OpenFile object pointer named executable to point to the opened user program. Then read the header of the user program into our NoffHeader noffh by the ReadAt function, in order to get the segment sizes.

```
    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);
```

To know the size that we need for the program, need to add up the code size, initData size, uninitData size, and also the stack size. By dividing size by PageSize and rounding up, get the num of pages we need. By knowing the size and inFileAddr of each segment, we can now correctly load the object code of the user program into the desired physical address of memory, where from the code above is the same with the virtual address, but in the implement part, we calculate its physical address.

```
delete executable;
return TRUE;
```

After loading the the file into memory, close the file and return true.

- **AddrSpace::Execute()**

```
void
AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

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

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

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

Run a user program using the current thread.
Set Object AddrSpace as the space for currentThread.
Call InitRegisters() to set initial register values for the user-level register set.

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

Call RestoreState() to tell the machine where to find the page table.
In the end, call Run() jumping to the user program.

- **Trace Code Questions**

Q1: How does Nachos allocate the memory space for a new thread(process)?

A: When uniprogramming, we call addrspace::addrspace() and allocate the whole physical memory for a new thread ; When multiprogramming, we call addrspace::load() to allocate the physical memory space.

Q2: How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?

A: When addrspace::load(), using ReadAt() function to load binary code into memory.

Q3: How does Nachos create and manage the page table?

A: When addrspace::load() ,let pageTable = new TranslationEntry[NumPhysPages]; to create a new page table, and during each page, we manage the virtual page and physical page and some parameter(valid, use, dirty, readOnly) in the page table.

Q4: How does Nachos translate addresses?

A: In addrspace::load(), we use code_exception = Translate(noffH.code.virtualAddr, &addr, true); to translate virtual address into physical address. Generally speaking, we calculate page number and offset by vaddr/pagesize and vaddr % pagesize to get the right data in physical memory.

Q5: How Nachos initializes the machine status (registers, etc) before running a thread(process)?

A: In AddrSpace::Execute(), we call AddrSpace::InitRegisters() and AddrSpace::RestoreState() initializes some machine status (registers, etc) into registers.

Q6: Which object in Nachos acts the role of process control block?

A: In thread ' s class, we will record some information including threadstatus, registers$\cdots$, which is the role of process control block.

Q7: When and how does a thread get added into the Ready-ToRun queue of Nachos CPU scheduler?

A: In Thread::fork(), we call the scheduler->ReadyToRun(this) to mark the thread ' s status to READY and put the thread into readylist by readyList->Append(thread).

2 Implement

- First, I go to **machine.h** to add "MemoryLimitException" into the ExceptionType to handle the exception when there is insufficient memory for a thread.

```
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,  //MP2
                     NumExceptionTypes
};
```

- Then, I add two public member in the **kernel.h**.
  One is "bool frameTable[128]". It is used to record whether the frame has already been allocate, if yes, it can't be aloocated again. Another is int NumFreeFrame. It is used to record how many physical frame left we can use.

```
bool frameTable[128];  //MP2 use to record whether the frame has already been allocated
                       // if yes, it can't be aloocated again.
int NumFreeFrame;      //MP2 use to record how many physical frame left we can use.
```

- Next, I go to the **Initialize()** in **kernel.cc** to initialize two public member.

```
frameTable[128] = {0};   //MP2
NumFreeFrame = NumPhysPages;   //MP2
```

- After that, I go to the **AddrSpace()** in **AddrSpace,cc** to Comment out the initialization in the constructer, for we only need create the size for each thread, so do not need to create the virtual page in the same size of the whole physical page).

```
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); */

}
```

- Then, I go to the **AddrSpace::Load(char *fileName)** to allocate page table. In the first step, I add a if condition in line 144 to let if there is a situation that the numPages bigger than the free frame we left(NumFreeFrame) the Exceptionhandler will run the ExceptionType I set in the first step (MemoryLimitException) to tell the user what goes wrong.

```
if( numPages>kernel->NumFreeFrame)
    ExceptionHandler(MemoryLimitException);
```

- After that, I go to construct the logic of PageTable Allocation. Because Page table will connect virtual page to the frame of physical memory, we need to know whether the frame is empty to connect. By doing so, we need to create a frame table to record whether the frame has already been used. Therefore, I create a for() loop to traverse the frame. If the frame==1, it means that this frame has already been used, then it will keep traversing till meet the frame==0, which means that the frame is able to be allocated to the page and record on the page table. By the way, we need to notice that when we allocate the frame, we should clean the frame (bzero) to confirm that the frame is completely new, then, alocate the frame to the thread.

```
pageTable = new TranslationEntry[numPages]; //for this user program
for( int i=0, page=0; i< numPages; i++)
{
    while(page < NumPhysPages && kernel->frameTable[page]==1) page++;
    if(page==NumPhysPages) ExceptionHandler(MemoryLimitException);
    for(int j = 0; j < NumPhysPages; j++)
    {
        if(kernel->frameTable[j] == 1){
            continue;
        }
        else{
            //frame can be used
            bzero(&kernel->machine->mainMemory[j*PageSize], PageSize);
            pageTable[i].virtualPage = i;
            pageTable[i].physicalPage = j;
            pageTable[i].valid = TRUE;
            pageTable[i].use = FALSE;
            pageTable[i].dirty = FALSE;
            pageTable[i].readOnly = FALSE;
            kernel -> frameTable[j] = 1;
            kernel->NumFreeFrame--;
            DEBUG(dbgAddr, "THREAD"<<kernel->currentThread->getName()<<" virtualPage"<<i<<"used");
            break;
        }
    }
```

- According to the original code, we imitiate and modify the code to count the page table. The thing we have to notice is that in the "ReadAt" function, we must change the virtualAddr to the physical address to let the noffH put code and data section from disk into the frame in the mainMemory.

```
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);*/
    int code_page_num = divRoundUp(noffH.code.size, PageSize);  // # of pages that are used to store the "code segment"
    for (int page_idx = 0; page_idx < code_page_num; page_idx++) {
        // v_addr_temp is the virtual address of the start of a page
        // p_addr_temp is the physical address of the v_addr_temp
        unsigned int v_addr_temp = noffH.code.virtualAddr + page_idx * PageSize;
        unsigned int p_addr;
        ExceptionType e;
        if (NoException != (e = Translate(v_addr_temp, &p_addr, false /*should not chage executing file*/)))
            ExceptionHandler(e);

        int disk_addr = noffH.code.inFileAddr + page_idx * PageSize;  // the actuall address of the page we're loading
        int reading_size = (page_idx + 1 != code_page_num) ? PageSize /*not last page*/ : noffH.code.size % PageSize /*

        executable->ReadAt(
            &(kernel->machine->mainMemory[p_addr]),
            reading_size, disk_addr);
    }
}
```

```
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);*/

    // cout << "Initializing data segment." << noffH.initData.virtualAddr << ", " << noffH.initData.size << endl;

    // we should read the data in [virtualAddr, v_end_addr) == [virtualAddr, v_end_addr - 1]
    unsigned int v_end_addr = noffH.initData.virtualAddr + noffH.initData.size;
    // v_addr_temp is the virtual address of the start of a page
    // Initialize with the beggingng of initData
    unsigned int v_addr_temp = noffH.initData.virtualAddr;
    while (v_addr_temp < v_end_addr) {
        // p_addr_temp is the physical address of the v_addr_temp
        unsigned int p_addr;
        ExceptionType e;
        if (NoException != (e = Translate(v_addr_temp, &p_addr, false /*should not chage executing file*/)))
            ExceptionHandler(e);

        // disk_addr is the actuall address of the page we're loading in current iteration
        int disk_addr = noffH.initData.inFileAddr;  // we assume that v_addr_temp is following the code_segement

        int v_page_num = (v_addr_temp / PageSize) + 1;  // The number(not index) of virtual page that the current address are store in
        int reading_size = (v_page_num * PageSize < v_end_addr) ? v_page_num * PageSize - v_addr_temp : /*the last page*/ v_end_addr - v_addr_temp;
        executable->ReadAt(
            &(kernel->machine->mainMemory[p_addr]),
            reading_size, disk_addr);

        v_addr_temp += reading_size;
    }
}
```

```
if (noffH.readonlyData.size > 0) {
    DEBUG(dbgAddr, "Initializing read only data segment.");
DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);

    int total, curOffset = 0;
    for(total = noffH.readonlyData.size, curOffset = 0; total > 0; total-=PageSize, curOffset +=PageSize){
        unsigned int p = divRoundUp(noffH.readonlyData.size, PageSize);
        for(int i=0; i<p; i++){
            pageTable[noffH.readonlyData.virtualAddr/PageSize + i].readOnly = TRUE;
        }

        unsigned int loadSize;
        if(total < PageSize) loadSize = total;
        else loadSize = PageSize;

        unsigned int p_addr;
        ExceptionType e;
        if(NoException != (e = Translate(noffH.readonlyData.virtualAddr, &p_addr, 0)))
            ExceptionHandler(e);

        executable->ReadAt(
        &(kernel->machine->mainMemory[p_addr]),
        loadSize, noffH.readonlyData.inFileAddr + curOffset);

    }
}
```

- Last, I go to **AddrSpace()** to free the frameTable and increase the NumFreeFrame when the AddrSpace() is called.

```
AddrSpace::~AddrSpace()
{
   for(int i=0; i<numPages; i++){
    kernel->frameTable[pageTable[i].physicalPage] = 0; //free frame
    kernel->NumFreeFrame++;
   }
   DEBUG(dbgAddr, "Fre frame used by thread: "<<kernel->currentThread->getName());
   delete []pageTable;
}
```

- the test result:

```
[os23s47@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e consoleIO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

3 Difficults & FeedBack

(1) When I was tracing the code, it still cost me a lot of time like the last time. Especially, the part of FOrk() and ForkExecute() because it use a lot of pointer. It takes me some time to understand.

(2) When I was doing the implement, to finish the part "load a page at a time, and repeatedly load until the whole program finishes loading." relly takes a lot of time. At first, I only can load whole pages at a time, through many times attempt, I found to use the function divRoundup to count the pages needed, and used for() and while() to complete the page allocation.

(3) I still not very understand the function of noffMagic in noffH, by searching the data, I know that the noffH means "NachOs Object File Format Headers".

(4) The last thing is whether can offer the test file to let us test whether the MemoryLimitException is running promptly. ConsoleIO test1 and test2 can't test the function, so I hope there will be a test file to test it.Thanks!!!