113-1 Operating System MP2

Group 45: 111062332 朱誼學, 111062333 高英耀

Contributions	**111062332**	**111062333**
Trace code	50%	50%
Implementation	50%	50%
Report	50%	50%
Explanation	We've done all our work in discord vc	

Implementation explanation

a. machine/machine.h:

```
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.

                    // start code
                     MemoryLimitException,

                     NumExceptionTypes
};
```

We add a new exceptionType MemoryLimitException to handle insufficient memory for a thread.

b. threads/kernal.h, kernal.cc

```
kernel.h:
    int frameTable[128]; // define
```

```
kernel.cc/Kernel::Initialize():
    for(int f = 0; f < 128; f++) // initialize
        frameTable[f] = 0;
```

We set a new array `frameTable` to record all the 128 frames (128 bytes each frame) is available or not.

c. userprog/addrspace.cc:

```
AddrSpace::AddrSpace() {
    // start code
    pageTable = new TranslationEntry[32];
    for (int i = 0; i < 32; i++) {
        pageTable[i].virtualPage = i;  // for now, virt page # = phys page #
        // pageTable[i].physicalPage = i;
        for(int f = 0; f < 128; f++){
            if(kernel->frameTable[f] == 0){
                pageTable[i].physicalPage = f; // assign the fth frame to this
page
                kernel->frameTable[f] = 1; // set the frame is occupied
                break;
            }
        }
        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);
}
```

In `AddrSpace::AddrSpace()`, instead of directly mapping the virtual page to the physical page, we search through the `frameTable` to find 32 available frames and map it to the virtual page table in this `AddrSpace` (this thread). Also, we cancel the operation of clear whole memory when a new `AddrSpace` is created.

```
AddrSpace::~AddrSpace() {
    // start code
    Machine *machine = kernel->machine;
    for(int i = 0; i < 32; i++){
        int frame;
        frame = pageTable[i].physicalPage;
        bzero(kernel->machine->mainMemory + frame*128, 128);
        kernel->frameTable[frame] = 0;
    }
    delete pageTable;
}
```

We move the operation of clear memory to `AddrSpace::~AddrSpace()`. We clear all frames recorded in `pageTable` and set the corresponding `frameTable` available.

```
bool AddrSpace::Load(char *fileName) {

    ...

    numPages = divRoundUp(size, PageSize);

    // start code
    if(numPages > 32){
        ExceptionHandler(MemoryLimitException);
    }

    ...

    // then, copy in the code and data segments into memory
    // Note: this code assumes that virtual address = physical address
    // start code
    if (noffH.code.size > 0) {
        DEBUG(dbgAddr, "Initializing code segment.");
        DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
        executable->ReadAt(
            &(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr >>
7].physicalPage * 128]),
            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[pageTable[noffH.initData.virtualAddr >>
7].physicalPage * 128]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }

    ...
}
```

In `AddrSpace::Load()`, when this thread requests memory more than its memory limit (we set 32 as the memory limit), then we call Exceptionhandler to handle MemoryLimitException.
And, we convert the virtual page number to physical page number when it accesses Main Memory with virtual address. Right shift the `virtualAddr` for 7 bits, we can get the virtual page number, then used it as the index of `pageTable`, we can get the physical page number (frame number).

Trace code explanation

a. threads/thread.cc:

```
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);
}
```

`Thread::Sleep()` will let the CPU stays IDLE, since the current thread has either finished or is blocked waiting a synchronization variable (this will be determined by `finishing`). If the current thread is waiting, it will eventually be waken up by a certain thread. After that, this thread will be put back into the ready queue. `kernel->scheduler->FindNextToRun()` will find this thread in the ready queue, and break the `while-loop`. Next, `kernel->scheduler->Run(nextThread, finishing)` will be called. In this function `SWITCH()` will be invoked, the detail about this function will be explained later.

```
void Thread::StackAllocate(VoidFunctionPtr func, void *arg);
```

`Thread::StackAllocate()` is called when a thread is forked, and it has two parts need to be emphasized.

```
stackTop = stack + StackSize - 4;  // -4 to be on the safe side!
*(--stackTop) = (int)ThreadRoot;
*stack = STACK_FENCEPOST;
```

This part of the function will handle the case `SWITCH()` is called and switch to this thread.
The return address of the `SWITCH()` will be the threadRoot, so the procedure can be run.

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

This part of the function is used to handle a thread which is forked, putting all the necessary values into registers.

```
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
}
```

`Thread::Finish()` will terminate the current thread and called `sleep()` with `finishing = TRUE`.

```
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);
}
```

`Thread::Fork()` forks a new thread by allocate a stack, initialize the stack so that a call to `SWITCH()` will cause it to run the procedure. Last, put the thread on the ready queue.

b. userprog/addrspace.cc (Functions in this section are already modified to fit the requirement)

```
AddrSpace::AddrSpace() {
    // code start
    pageTable = new TranslationEntry[32];
    for (int i = 0; i < 32; i++) {
        pageTable[i].virtualPage = i;  // for now, virt page # = phys page #
        // pageTable[i].physicalPage = i;
        for(int f = 0; f < 128; f++){
            if(kernel->frameTable[f] == 0){
                pageTable[i].physicalPage = f; // assign the fth frame to this
page
                kernel->frameTable[f] = 1; // set the frame is occupied
                break;
            }
        }
        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);
}
```

`AddrSpace::AddrSpace()` is used to handle memeory management by using page table. In the constuctor, it will initialize the page table and the related attributes.

```
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 program

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

This function will run user program to execute the file using current thread. Before running, not only the registers should be initialized, but also should called `RestoreState` to load the page table of current thread.

```
bool AddrSpace::Load(char *fileName);
```

This function loads the program into memeory.

c. threads/kernel.cc

```
Kernel::Kernel(int argc, char **argv);
```

This constuctor will interpret command line argument to determine flags for initialization.

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

`Kernel::ExecAll()` is called in `main.cc` (driver code) to run all the user programs mentioned in command line argument.

```
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;
}
```

`Kernel::Exec()` is the actual function constructing execution.

1. create a new thread
2. create page table
3. call `Fork()` to run this program on this thread

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

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

`ForkExecute()` will be passed to thread by `Fork()`, and this function will load the program into the memory and call `AddrSpace::Execute()` to run the user program.

d. threads/scheduler.cc

```
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);
}
```

`Scheduler::ReadyToRun()` will set the thread's state to `READY` and push into ready queue.

```
void Scheduler::Run(Thread *nextThread, bool finishing)
```

`Scheduler::Run()` is called in `Thread::Sleep()` when there is a thread in ready queue.
Following is the explanation of the most important part of this function.

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

CheckToBeDestroyed();
```

The first function of `Scheduler::Run()` is that `finishing` will be sent into the function by `Thread::Sleep()`, and if `finishing` is true, the original thread is already terminated, the thread will be deleted in `CheckToBeDestroyed()`.

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

SWITCH(oldThread, nextThread);
```

Next, `Scheduler::Run()` will switch the current thread to `nextThread` and call `SWITCH()`.
`SWITCH()` is defined with assembly code, it will load all the necessary values of the new thread into the registers and jump to the threadRoot of the new thread. Note that, when the new thread terminates, `SWITCH()` will also return, so it will come back to this thread. Then we check whether the old thread should be destroyed, or should continue running. In the latter case, we should restore the page table.

Questions

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

Ans:
In Nachos, when a new thread or process is created, threads will construct an `AddrSpace`.

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

`AddrSpace` object representing the virtual address space allocated to the thread or process.
`AddrSpace::AddrSpace()` creates an address space to run a user program. Also set up the translation from program memory to physical memory.

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

Ans:
Nachos calls the function `Thread::StackAllocate()` to set up the initial state of the stack and registers.
Then Nachos calls function `AddrSpace::Load()` to initialize the address space, this function opens and loads the binary executable file, then copying code, data, and any other necessary information from the executable into the process's memory.

> How does Nachos create and manage the page table?

Ans:
The page table in Nachos is created and managed by the `AddrSpace` class. The page table is an array of `TranslationEntry`. The `TranslationEntry` class stores information like the physical page number, validity, and use and readOnly bits, so this table supports virtual memory.

> How does Nachos translate addresses?

```
pageTable[virtualAddr >> 7].physicalPage
```

Ans:
Nachos uses the page table to translate virtual addresses to physical addresses. Before we modify the code to fit MP2 requirement, virtual addresses were equal to physical addresses. After that, we find the physical addresses in the page table with the virtual page number.

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

Ans:

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

Nachos calls the function `Thread::StackAllocate()` to set up registers. It puts some function that helps initialize the process to registers.

Setting `machineState[PCState]` to `ThreadRoot` initializes the program counter to start executing the `ThreadRoot` function when the thread begins running. It would help initialize the stack, and call functions in other `machineState`.
`StartupPCState` is set to `ThreadBegin`, which is another function that helps in starting the thread execution. It deallocates the previously running thread if it finishes and also enables interrupts.
`InitialPCState` is set to func, which would load the code file.
`InitialArgState` holds the argument arg to pass to func when it starts running.
`WhenDonePCState` is set to ThreadFinish, which is the function to be called when func completes its execution. This function puts this thread to Sleep helps the kernel switch to another thread.

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

Ans:
The `Thread` class acts as a process control block. It has the stack pointer of the process, and `machineState` to record registers, a pointer to the `AddrSpace` of the process and record the status of the process.

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

Ans:
When a thread is newly created (Fork) or it is rebooted from the other status, then Nachos will call `Scheduler::ReadyToRun()` to put the thread into the ReadyToRun queue.