OS HW02 GROUP 21

Part 1:Trace Code

1. Explain function

1. threads/thread.cc

a) Thread::Sleep()

This function is invoked when the current thread has either finished its execution or is waiting on a synchronization variable.

It updates the thread's status to BLOCKED, indicating that it should not be scheduled for execution. The function then searches for another thread that can run. If no runnable threads are available, it places the CPU in idle mode until an interrupt occurs. Once a suitable thread is found, it switches execution to that thread.

```
void Thread::Sleep (bool finishing)
{
    Thread *nextThread;
    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

b) Thread::StackAllocate()

This function allocates an execution stack of a specified size. It then initializes the stack according to the architecture of the machine.

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
}
```

c) Thread::Finish()

Called by the ThreadRoot when the forked procedure has completed, this function performs the necessary cleanup for the thread.

This function first prevents the thread from being interrupted during its termination process. It then invokes the “Sleep” function, passing “TRUE” as an argument to indicate that this thread is finished and should not be awakened again.

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

d) Thread::Fork()

This function allocates a new stack for the thread to hold local variables and return addresses. Since it modifies the shared data structure of the "ready queue" by placing the newly created thread there, interrupts are disabled during this process to prevent any interruptions.

```
void Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;
    StackAllocate(func, arg);
    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

2. userprog/addrspace.cc

a) AddrSpace::AddrSpace()

It creates an address space for executing a user program and establishes the mapping between logical and physical memory. This address space is associated with a page table that facilitates memory translation. Initially, it sets up a one-to-one mapping, where each virtual page number corresponds directly to a physical frame number. Memory protection bits are configured for each entry in the page table. Finally, the entire address space is cleared, ensuring that no residual data from previous processes remains in the main memory.

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = i;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }
    bzero(kernel->machine->mainMemory, MemorySize);
}
```

b) AddrSpace::Execute()

This function prepares the execution environment for a user program by the current thread with the address space, initializing CPU registers, setting up the memory translation mechanism, and finally starting the program's execution by invoking “machine->Run” function.

```
void AddrSpace::Execute(char* fileName)
{
    kernel->currentThread->space = this;
    this->InitRegisters();
    this->RestoreState();
    kernel->machine->Run();
}
```

c) **AddrSpace::Load()**

This function is responsible for loading a user program into memory from a specified file. First, it opens the specified executable file. Next, the function reads the header of the executable into a structure called noffH, which contains important information about the program's segments. Then, it calculates the required size of the address space and determines the number of pages needed for the program. After that, it copies the code segment, the initialized data segment, and, if applicable, the read-only data segment from the executable file into their appropriate locations in the main memory. Finally, once all necessary segments are loaded, the function deletes the executable object to close the file properly.

```
bool AddrSpace::Load(char *fileName)
{
    OpenFile *executable = kernel->fileSystem->Open(fileName);
    NoffHeader noffH;
    unsigned int size;
    // Calculate the size... //
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    if (noffH.code.size > 0) {
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }
    if (noffH.readonlyData.size > 0) {
        executable->ReadAt(
            &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr]),
            noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
    }
    delete executable;
    return TRUE;
}
```

3. **threads/kernel.cc**

a) **Kernel::Kernel()**

Initializes the kernel-level data structures and handles command-line arguments to configure the kernel environment. For memory management, it sets up the "execfile" array and the "execfileNum" counter.

When the -e flag is provided, the file names to be executed are stored in the "execfile" array, and the count of these files is updated in "execfileNum."

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

b) **Kernel::ExecAll()**

Executes all programs stored in the "execfile" array by invoking the "Exec" function.

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

c) **Kernel::Exec()**

Creates a new thread for the specified executable file, allocates an address space for the user program, and then forks the thread to execute it using the ForkExecute() function.

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
}
```

d) **Kernal::ForkExecute()**

Loads the executable associated with the given thread into memory and then executes the program.

```
void ForkExecute(Thread *t)
{
    if ( !t->space->Load(t->getName()) ) { return; }
    t->space->Execute(t->getName());
}
```

4. **Threads/scheduler.cc**

a) **Scheduler::ReadyToRun()**

It marks a thread's status as "ready" and adds it to the ready queue, making it eligible for execution when the scheduler selects the next thread to run on the CPU.

```
void Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

b) Scheduler::Run()

This function manages the context switch between threads. It begins by saving the state of the currently running thread. First, it checks whether the thread is marked for deletion after the switch. Then, if the thread is associated with a user program, it saves the user's CPU registers and the state of the thread's address space. It also checks for any stack overflow. Next, it switches to the next thread using the “SWITCH” routine. Once this switch is complete, the function returns to the context of the old thread, with interrupts disabled. Finally, it checks if the old thread needs to be cleaned up and, if necessary, restores the state of the user's CPU registers and the address space for that thread.

```
void Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    if (finishing) {
        toBeDestroyed = oldThread;
    }
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }
    oldThread->CheckOverflow();
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

Part 2:Implementation

1. Detail of your implementation

a) addrspace.h

Define the FrameTable class to keep track of which frames are free. FindFreeFrame will return the index of a free frame, while ReleaseFrame frees up a specified frame.

```
struct FrameEntry {
    bool isFree;
};
class FrameTable {
public:
    FrameTable();
    int FindFreeFrame();
    void ReleaseFrame(int frame);
private:
    FrameEntry frames[NumPhysPages];
};
```

b) addrspace.cc

i) FrameTable

First, inialize the frame table when it is created. Then, implement FindFreeFrame, which iterates through the frame table to find a free frame, and ReleaseFrame, which sets the given frame as free.

```
FrameTable::FrameTable() {
    for (int i = 0; i < NumPhysPages; i++) {
        frames[i].isFree = true;
    }
}
int FrameTable::FindFreeFrame() {
    for (int i = 0; i < NumPhysPages; i++) {
        if (frames[i].isFree) {
            frames[i].isFree = false;
            return i;
        }
    }
    return -1;
}
void FrameTable::ReleaseFrame(int frame) {
    frames[frame].isFree = true;
}
```

ii) AddrSpace::AddrSpace()

When a page table is created, initialize it without setting the physical pages. The physical pages will be assigned during the loading process.

```
AddrSpace::AddrSpace()
{
    // TODO
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = -1;
    pageTable[i].valid = FALSE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }
    bzero(kernel->machine->mainMemory, MemorySize);
}
```

iii) AddrSpace::~AddrSpace()

Release the frame associated with each valid page when deleting the page table.

```
AddrSpace::~AddrSpace()
{
    // TODO
    for (int i = 0; i < NumPhysPages; i++) {
        if (pageTable[i].valid) {
            kernel->frameTable->ReleaseFrame(pageTable[i].physicalPage);
        }
    }
    delete pageTable;
}
```

iv) **AddrSpace::Load(char *fileName)**

When loading an address space, find free frames for the page table. By doing so, we can map the pages to non-contiguous physical frames.

```
// TODO
for (unsigned int i = 0; i < numPages; i++) {
    int physicalFrame = kernel->frameTable->FindFreeFrame();
    pageTable[i].physicalPage = physicalFrame;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
}
unsigned int physicalAddr;
```

Load the code segment into physical memory using the virtual address. First, use a while loop to load the code in segments, as physical frames is non-contiguous, we need to load segment by segment. Second, for each segment, calculate its size and load the code using the translated address, size, and file position.

```
if (noffH.code.size > 0) {
        // TODO
    unsigned int unReadSize = noffH.code.size;
    unsigned int chunkStart = noffH.code.virtualAddr;
    unsigned int inFilePosition = 0;
    while (unReadSize > 0) {
        unsigned int virtualPage = chunkStart / PageSize;
        unsigned int chunkSize = (virtualPage + 1) * PageSize - chunkStart;
        if (chunkSize > unReadSize) chunkSize = unReadSize;
        Translate(chunkStart, &physicalAddr, 1);
        executable->ReadAt(&(kernel->machine->mainMemory[physicalAddr]),
                           chunkSize, noffH.code.inFileAddr+inFilePosition);
        unReadSize -= chunkSize;
        chunkStart += chunkSize;
        inFilePosition += chunkSize;
    }
}
```

Use the same process used to load the code segment to load the initialized data.

```
if (noffH.initData.size > 0) {
    // TODO
    unsigned int unReadSize = noffH.initData.size;
    unsigned int chunkStart = noffH.initData.virtualAddr;
    unsigned int inFilePosition = 0;
    while (unReadSize > 0) {
        unsigned int virtualPage = chunkStart / PageSize;
        unsigned int chunkSize = (virtualPage + 1) * PageSize - chunkStart;
        if (chunkSize > unReadSize) chunkSize = unReadSize;
        Translate(chunkStart, &physicalAddr, 1);
        executable->ReadAt(&(kernel->machine->mainMemory[physicalAddr]),
                           chunkSize,
                           noffH.initData.inFileAddr+inFilePosition);
        unReadSize -= chunkSize;
        chunkStart += chunkSize;
        inFilePosition += chunkSize;
    }
}
```

Use the same process to load the read-only data, setting the 'readOnly' to true in the page table.

```
if (noffH.readonlyData.size > 0) {
    // TODO
    unsigned int unReadSize = noffH.readonlyData.size;
    unsigned int chunkStart = noffH.readonlyData.virtualAddr;
    unsigned int inFilePosition = 0;
    while (unReadSize > 0) {
        unsigned int virtualPage = chunkStart / PageSize;
        unsigned int chunkSize = (virtualPage + 1) * PageSize - chunkStart;
        if (chunkSize > unReadSize) chunkSize = unReadSize;
        pageTable[virtualPage].readOnly = TRUE;
        Translate(chunkStart, &physicalAddr, 1);
        executable->ReadAt(&(kernel->machine->mainMemory[physicalAddr]),
                    chunkSize, noffH.readonlyData.inFileAddr +inFilePosition);
        unReadSize -= chunkSize;
        chunkStart += chunkSize;
        inFilePosition += chunkSize;
    }
}
```

c) **kernel.h**

Create a frame table.

```
FrameTable *frameTable;
```

d) **kernel.cc**

i) **Kernel::Initialize()**

Using the frame table define in addrspace.

```
frameTable = new FrameTable();
```

ii) **Kernel::~Kernel()**

When destructor function for the kernel is called, delete the frame table.

```
delete frameTable;
```

Part 3:Contribution

1. **Describe details and percentage of each member’s contribution.**

   313551047 陳以瑄, contribution (50%):

   Trace code

   111550150 俞祺譯, contribution (50%):

   Implementation