# MP2: Multi-Programming

## Cover page

- 徐竣霆： Trace code, Implementation, Report.

- 江承紘： Trace code, Implementation, Report.

## Trace code - function explanation (in traversal order)

### 1. `threads/kernel.cc` `Kernel::Kernel()`

This constructor interprets command line arguments to determine flags for the initialization.

### 2. `threads/kernel.cc` `Kernel::ExecAll()`

This function is the starting point of all the applications in NachOS. It also runs in a thread and calls `Exec()` for each `execfile[i]` to create a new thread for each of them. After all the threads have finished executing, it calls `Finish()` on the current thread itself.

### 3. `threads/kernel.cc` `Kernel::Exec()`

This function creates a thread, `t[threadNum]` , and a memory space for it, which is currently to be executed. Then it packages `ForkExecute()` and `t[threadNum]` to `Thread::Fork()` to continue execution. `threadNum` represents which thread is currently executed. Finally, it returns the current `threadNum` after finishing `Thread::Fork()` .

### 4. `userprog/addrspace.cc` `AddrSpace::AddrSpace()`

This constructor creates an address space to run a user program. It sets up the translation from program memory to physical memory.

## 5. `threads/kernel.cc` `Kernel::ForkExecute()`

This function loads the current thread by `AddrSpace::Load()` and executes it by `AddrSpace::Execute()`.

## 6. `userprog/addrspace.cc` `AddrSpace::Load()`

This function loads a user program into memory from a file.

## 7. `userprog/addrspace.cc` `AddrSpace::Execute()`

This function runs a user program using the current thread. Note that the thread in this function is assumed to have already been loaded into the address space.

## 8. `threads/thread.cc` `Thread::Fork()`

This function invokes `(*func)(arg)`, allowing the caller and callee to execute concurrently. It contains the following three steps:

- Allocate a stack.
- Initialize the stack so that a call to SWITCH(context switch) will cause it to run the procedure.
- Put the thread on the ready queue.

## 9. `threads/thread.cc` `Thread::StackAllocate()`

This function allocates and initializes an execution stack. The stack is initialized with an initial stack frame for ThreadRoot, which:

- Enables interrupts
- Calls `(*func)(arg)`
- Calls `Thread::Finish()`

`func` is the procedure to be forked, and `arg` is the parameter to be passed to the procedure.

## 10. **`threads/scheduler.cc` `Scheduler::ReadyToRun()`**

This function marks a thread as "ready", but not "running". It puts the thread on the ready list, for later scheduling onto the CPU. `thread` is the thread to be put on the ready list.

## 11. **`threads/thread.cc` `Thread::Finish()`**

This function is called by ThreadRoot when a thread is done executing the forked procedure. We can't immediately de-allocate the thread data structure or the execution stack, because we're still running in the thread and we are still on the stack! Instead, we tell the scheduler to call the destructor, once it is running in the context of a different thread. We disable interrupts, because `Sleep()` assumes interrupts are disabled.

## 12. **`threads/thread.cc` `Thread::Sleep()`**

This function relinquishes 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.

If there are no threads on the ready queue, that means we have no thread to run. "Interrupt::Idle" is called to signify that we should idle the CPU until the next I/O interrupt occurs (the only thing that could cause a thread to become ready to run).

We assume interrupts are already disabled because it is called from the synchronization routines which must disable interrupts for atomicity. We need interrupts off so that there can't be a time slice between pulling the first thread off the ready list and switching to it.

## 13. **`threads/scheduler.cc` `Scheduler::Run()`**

Finally, this function dispatches 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.

Note: we assume the state of the previously running thread has already been changed from running to blocked or ready (depending).

The global variable `kernel->currentThread` becomes `nextThread`.

`nextThread` is the thread to be put into the CPU. `finishing` is set if the current thread is to be deleted once we're no longer running on its stack(when the next thread starts running)

## Trace code - problems

- How does Nachos allocate the memory space for a new thread(process)?
  - It calls `Kernel::Exec()` and to allocate spaces for a new thread.
- How does Nachos initialize the memory content of a thread(process), including loading the user binary code in the memory?
  - It calls `bzero(kernel->machine->mainMemory, MemorySize)` in `AddrSpace::AddrSpace()` to zero out the specific entire address space and gives it to the new thread.
- How does Nachos create and manage the page table?
  - In `AddrSpace::AddrSpace()`, it calls `pageTable = new TranslationEntry[NumPhysPages]` to create and manage the page table.
- How does Nachos translate addresses?
  - It calls `Machine::Translate()` in `machine/translate.cc` to translate a virtual address into a physical address. Note that the physical address can be calculated by `*physAddr = pageFrame * PageSize + offset;`
- How Nachos initializes the machine status (registers, etc) before running a thread(process)
  - In `AddrSpace::Execute()`, it runs `this->InitRegisters()` to set the initial register values, and `this->RestoreState()` to load page table register.

- Which object in Nachos acts the role of process control block?
  - The `Thread()` class. Since it contains the thread status, ID, name, etc. of a thread.
- When and how does a thread get added into the ReadyToRun queue of Nachos CPU scheduler?
  - In `Thread::Fork()`, after it calls `StackAllocate(func, arg)` to allocate spaces and disabled interrupts, calling `scheduler->ReadyToRun(this)` to trigger the function in `threads/scheduler.cc`. Then it calls `thread->setStatus(READY)` to set the status to ready, and `readyList->Append(thread)` to append the thread to the queue.

```
case SC_MSG:
    DEBUG(dbgSys, "Message received.\n");
    val = kernel->machine->ReadRegister(4);
    {
        char *msg = &(kernel->machine->mainMemory[val]);
        cout << msg << endl;
    }
    SysHalt();
    ASSERTNOTREACHED();
    break;
```

- According to the code from userprog/exception.cc above, please explain under what circumstances an error will occur if the message size is larger than one page and why? (Hint: Consider the relationship between physical pages and virtual pages.)
  - If the message size is larger than one page and at least one of the pages is not in the page table, the program will raise a PageFaultException.

## Implementation Explanation

### `machine/machine.h` `enum ExceptionType`

Add `MemoryLimitException`.

### `threads/kernel.h` `class Kernel`

Create a PhysPageTable[NumPhysPages] table to save the physical page -> virtual page relationship.

### `threads/kernel.cc` `Kernel::Initialize()`

Initialize all PhysPageTable entries as unused(-1).

### `userprog/addrspace.cc` `findEmptyPhyAddress()`

Find the unused physical table, get the entry for future use.

### `userprog/addrspace.cc` `Addrspace::AddrSpace()`

To support multiprogramming, we have to know the number of Page(numPages) needed. So, instead of allocating address space at `Addrspace::AddrSpace()` , we allocate the space at `Addrspace::Load()` .

### `userprog/addrspace.cc` `Addrspace::~AddrSpace()`

Deallocate the physical pages used in this program, so that they can be used by other programs.

### `userprog/exception.cc` `ExceptionHandler(ExceptionType which)`

Since we call the `Addrspace::Translate()` to translate the virtual address to physical address, we have to handle the Exception raised in `Addrspace::Translate()` . For NoException, it shouldn't raise ASSERTNOTREACHED(), shouldn't abort.

### `userprog/addrspace.cc` `Addrspace::Load()`

```
pageTable = new TranslationEntry[numPages];
for (int i = 0; i < numPages; i++) {
    int j = findEmptyPhyAddress();
    if(j == -1) ExceptionHandler(MemoryLimitException);
    kernel->PhysPageTable[j] = i;
    pageTable[i].virtualPage = i;
    pageTable[i].physicalPage = j;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    // zero out the segment address space
    bzero(kernel->machine->mainMemory + j * PageSize, PageSize);
}
```

Allocate the physical pages for this program, if findEmptyPhyAddress() cannot find available physical page, raise MLE.

```
if (noffH.code.size > 0) {
    DEBUG(dbgAddr, "Initializing code segment.");
    unsigned int physAddr;
    ExceptionHandler(this->Translate(noffH.code.virtualAddr, &physAddr, 1));
    DEBUG(dbgAddr, physAddr << ", " << noffH.code.size);
    executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");
    unsigned int physAddr;
    ExceptionHandler(this->Translate(noffH.initData.virtualAddr, &physAddr, 1));
    DEBUG(dbgAddr, physAddr << ", " << noffH.initData.size);
    executable->ReadAt(&(kernel->machine->mainMemory[physAddr]),noffH.initData.size, noffH.initData.inFileAddr);
}

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

Make use of the `Addrspace::Translate()` to translate the virtual Address to physical Address, use `ReadAt()` to get the code and initData to memory.

## `userprog/addrspace.cc` `AddrSpace::SaveState()`

To support the context switch, save the pageTable and pageTableSize. It's actually the inversion of `AddrSpace::RestoreState()` .

# Execution



## consoleIO_test1 and consoleIO_test2

```
[os23team66@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e console
IO_test2
consoleIO_test1
consoleIO_test2
9
8
7
6
1return value:0
5
16
17
18
19
return value:0
```

### consoleIO_test1, consoleIO_test2 and consoleIO_test3

```
[os23team66@localhost test]$ ../build.linux/nachos -e consoleIO_test1 -e console
IO_test2 -e consoleIO_test3
consoleIO_test1
consoleIO_test2
consoleIO_test3
9return value:0
Unexpected user mode exception 8
Assertion failed: line 213 file ../userprog/exception.cc
Aborted
[os23team66@localhost test]$
```

consoleIO_test1 and consoleIO_test2 will use 12 pages each, and by setting consoleIO_test3 use 105 pages, the total used page will exceed 128, hence raise MLE.

### consoleIO_test4

```
[os23team66@localhost test]$ ../build.linux/nachos -e consoleIO_test4
consoleIO_test4
Unexpected user mode exception 8
Assertion failed: line 213 file ../userprog/exception.cc
Aborted
[os23team66@localhost test]$
```

By setting the page needed of consoleIO_test4 exceed 128, raise MLE.