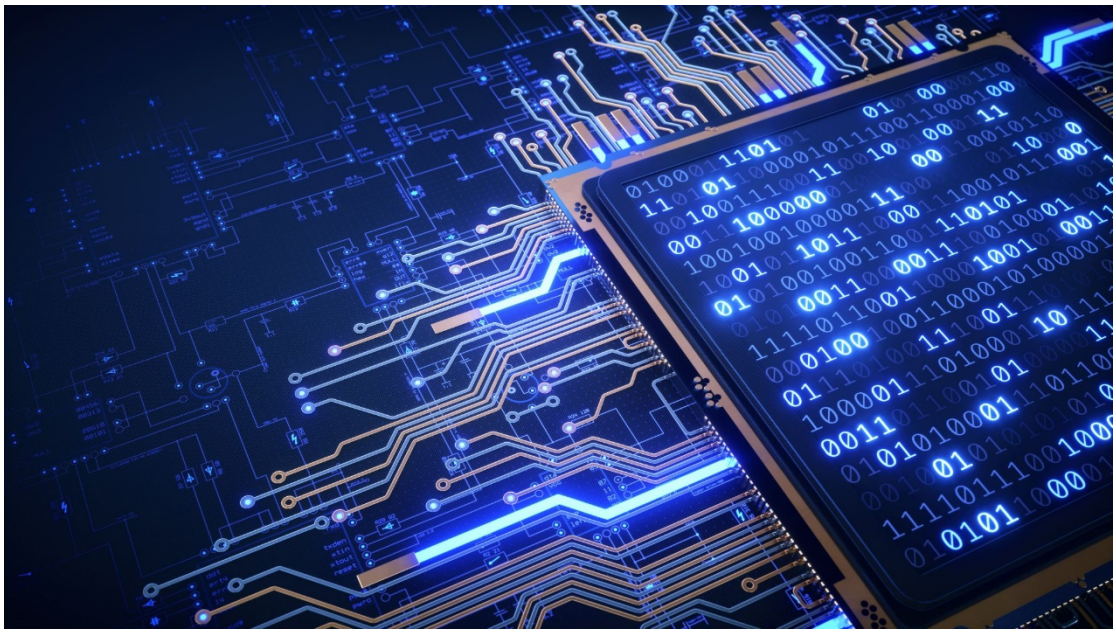
Operating System Mini Project Report

MP 2



TEAM 5

- 楊晨鍾：

  Code tracing, Implementation, Report

- 陳曦：

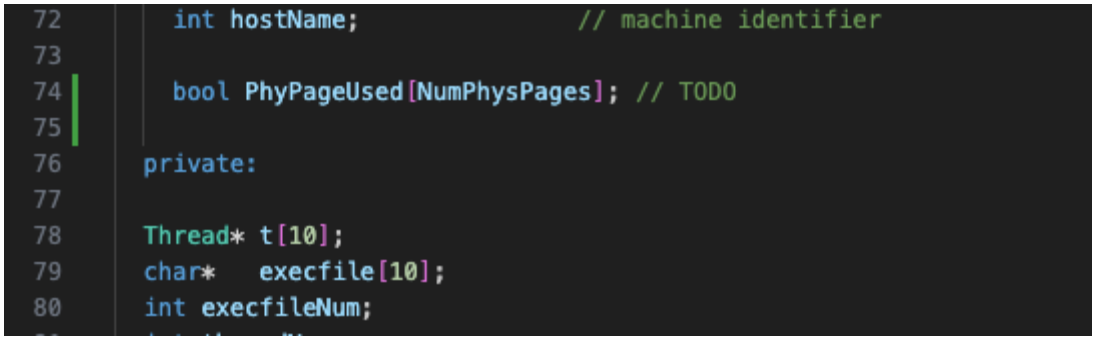
  Code tracing, Report

Nov 2023

1. Page table implementation

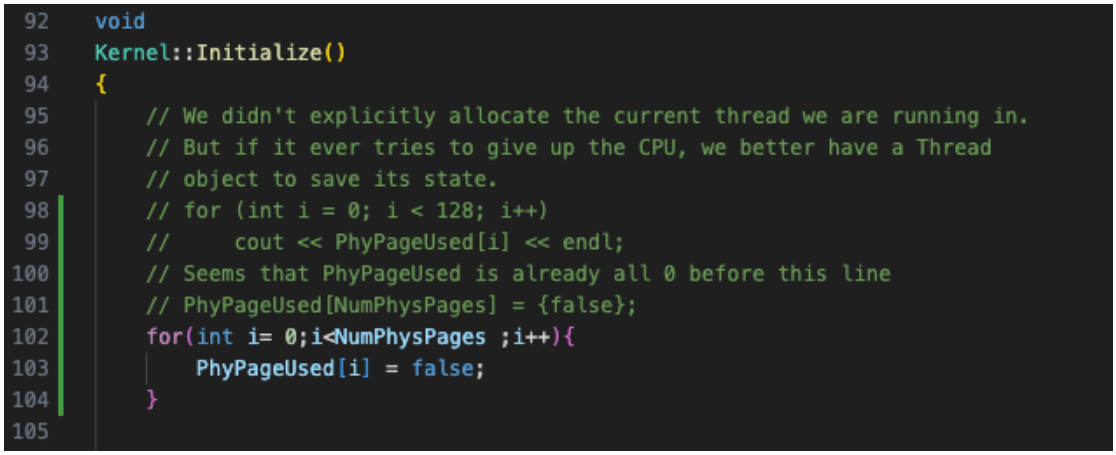
1.1 Record used physical memory

In kernel.h, we added an array to record whether the physical page is used or not.



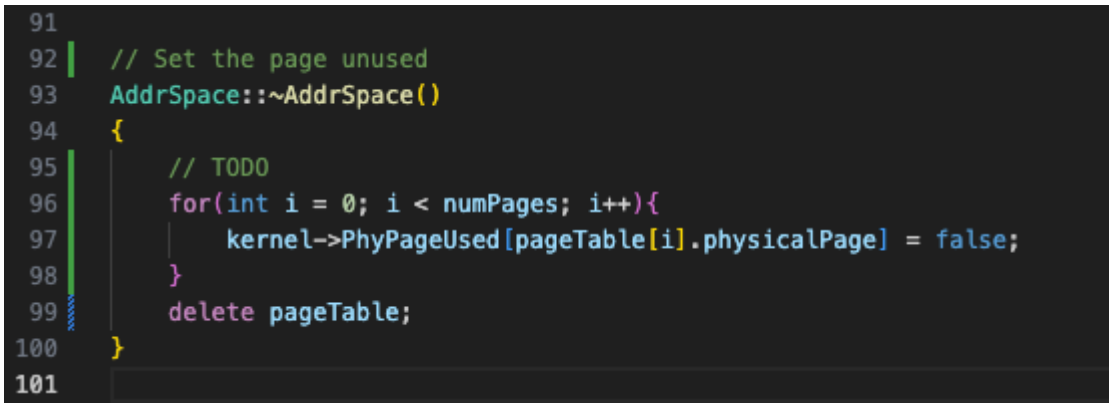
```
72      int hostName;               // machine identifier
73
74      bool PhyPageUsed[NumPhysPages]; // TODO
75
76    private:
77
78    Thread* t[10];
79    char*   execfile[10];
80    int execfileNum;
```

And in Kernel::Initialize(), in kernel.c, we initialize the array with value false.



```
92  void
93  Kernel::Initialize()
94  {
95      // We didn't explicitly allocate the current thread we are running in.
96      // But if it ever tries to give up the CPU, we better have a Thread
97      // object to save its state.
98      // for (int i = 0; i < 128; i++)
99      //     cout << PhyPageUsed[i] << endl;
100     // Seems that PhyPageUsed is already all 0 before this line
101     // PhyPageUsed[NumPhysPages] = {false};
102     for(int i= 0;i<NumPhysPages ;i++){
103         PhyPageUsed[i] = false;
104     }
105
```

In the destructor of Addrspace, we added a for loop to maintain the availability of the physical pages.



```
91
92  // Set the page unused
93  AddrSpace::~AddrSpace()
94  {
95      // TODO
96      for(int i = 0; i < numPages; i++){
97          kernel->PhyPageUsed[pageTable[i].physicalPage] = false;
98      }
99      delete pageTable;
100 }
101
```

1.2 Allocate memory to the process

In the constructor of AddrSpace, we commented all the code since the original function give all the physical page to one program, which leads the program overwriting previous programs when running multiple programs.

In Addrspace::Load(), we use the size of space, numPages, needed by the program to allocate the memory accordingly.

```
152
153     // TODO
154     pageTable = new TranslationEntry[numPages];
155
156     for(unsigned int i = 0, idx = 0; i < numPages; i++) {
157         // Set the number of virtual page
158         pageTable[i].virtualPage = i;
159
160         // Find unused physical page
161         while(idx < NumPhysPages && kernel->PhyPageUsed[idx] == true) idx++;
162
163         // Out if memory
164         if(idx >= NumPhysPages){
165             // cout << "Out of Memory" << endl;
166             ExceptionHandler(MemoryLimitException);
167
168         }
169         kernel->PhyPageUsed[idx] = true;
170         bzero(&kernel->machine->mainMemory[idx * PageSize], PageSize);
171
172         pageTable[i].physicalPage = idx;
173         pageTable[i].valid = true;
174         pageTable[i].use = false;
175         pageTable[i].dirty = false;
176         pageTable[i].readOnly = false;
177     }
178     // Count how many physical page are available
179     int available_page = 0;
180     for(int i = 0;i<NumPhysPages;i++){
181         if(kernel->PhyPageUsed[i] == false) available_page++;
182     }
183     // cout << "Available pages = " << available_page << endl;
```

Note that in line 164, idx exceed 128 means there is no more space for the program, we call the ExceptionHandler to raise an exception under this circumstance.

Additionally, since the virtual addresses is no longer equal to the physical addresses, we need to translate from the virtual addresses to the physical ones while executing.

```
190
191     if (noffH.code.size > 0) {
192         DEBUG(dbgAddr, "Initializing code segment.");
193         DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
194
195         // Translate virtual address to physical address
196         unsigned int PAddr;
197         Translate(noffH.code.virtualAddr, &PAddr, 1);
198
199         executable->ReadAt( &(kernel->machine->mainMemory[PAddr]), //noffH.code.virtualAddr
200                             noffH.code.size,
201                             noffH.code.inFileAddr);
202     }
203     if (noffH.initData.size > 0) {
204         DEBUG(dbgAddr, "Initializing data segment.");
205         DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
206
207         // Translate virtual address to physical address
208         unsigned int PAddr;
209         Translate(noffH.initData.virtualAddr, &PAddr, 1);
210
211         executable->ReadAt( &(kernel->machine->mainMemory[PAddr]),//noffH.code.virtualAddr
212                             noffH.initData.size,
213                             noffH.initData.inFileAddr);
214     }
```

2. Code tracing

2.1 Function explanations

2.1.1 Thread::Sleep()

First, check this thread is not currently running and set the status to blocked. After that, keep calling **Idle()** till there's a thread to run. When there's one, call **Scheduler::Run()** to run the thread.

2.1.2 Thread::StackAllocate()

Allocate and initialize the stack for the process.
By setting machineState, it make sure the stack frame enables interrupts, run the process that is going to be forked (user program), and finish the thread by calling Thread::Finish() at the end.

2.1.3 Thread::Finish()

First, it disables the interrupt, then call Thread::Sleep(), which will invoke SWITCH to replace the old thread with new thread, i.e. finish the old thread.

2.1.4 Thread::Fork()

First allocate a stack by Thread::StackAllocate(), which initializes the stack.
Then, put the thread on the ready queue by Scheduler::ReadyToRun()

2.1.5 AddrSpace::AddrSpace()

In the original NachOS implementation before MP2, this function creates page table for a program and then initial each page's use, valid, and dirty, etc. (virtual page = physical page) Then, zero entire address space.
In our implementation, since we let AddrSpace::Load() take up this part, the AddrSpace::AddrSpace() now does nothing.

2.1.6 AddrSpace::Execute()

Let the current thread run the user code by Machine::Run() after properly setting the registers.

2.1.7 AddrSpace::Load()

In the original NachOS implementation before MP2, this function opens the file, read the file and store the data in a structure NoffHeader. And then if the noffMagic is not equal to the original constant, swap it.
Calculate the size by adding each part in the noffH and size of user stack, and then count how many pages is needed. If the pages needed are over number of physical pages, abort. And then copies the code and data into memory so that they can execute. i.e., load the user program into memory from the opened file.

After our implementation, this function now handles not only the works above but also handles the work of AddrSpace::AddrSpace(), and checks whether there is sufficient memory for the process along to support multiprogramming. If there is not sufficient space, call the ExceptionHandler to raise an exception.

2.1.8 Kernel::Kernel()

Check all the arguments and execute the corresponding action. E.g. When getting option -e, set execfile and execfileNum according to the arguments.

2.1.9 Kernel::ExecAll()

Call Kernel::Exec() for each execfile to execute all the files. After executing all the files, set the current thread finish.

2.1.10 Kernel::Exec()

First, create a new thread, then get the space for user code by AddrSpace.
After that, fork the process with Thread::Fork(), with Kernel::ForkExecute().

2.1.11 Kernel::ForkExecute()

Load the user code to the memory by AddrSpace::Load() and execute it by AddrSpace::Execute().

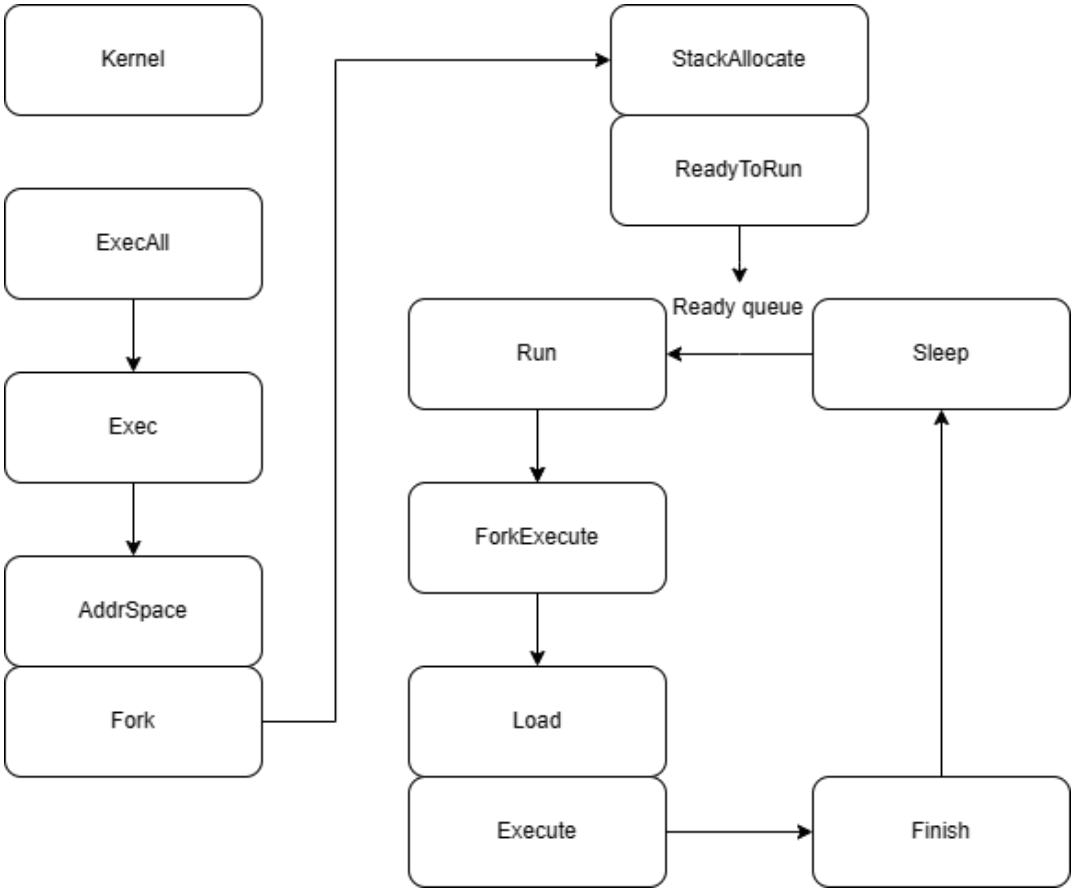
2.1.12 Scheduler::ReadyToRun()

Set the status to ready and append the thread to the ready list.

2.1.13 Scheduler::Run()

Save current thread's state (to old thread), then set current thread as next thread and set its status to running.
After that, call SWITCH() and restore the old thread's state if needed at the end.

2.2 Code path explanation



2.2.1 Thread creation

After Kernel::Kernel() set the execfile and execfilNum, Kernel::ExecAll will call Kernel::Exec() to execute the file, this will create a new instance of the process control block, the Thread object.
Right after this, AddrSpace::AddrSpace() will allocate the memory space for the thread will creating the page table and set it up, and the thread will be forked by Thread::Fork().
During this, the stack will be allocated and initialized by Thread::StackAllocate().

2.2.2 Load the thread into memory

Kernel::ForkExecute() will load the user code by AddrSpace::Load() and execute it by AddrSpace::Execute().

2.2.3 Place the thread into the scheduling queue

In Thread::Fork(), Scheduler::ReadyToRun() will put the thread into the ready list.

2.3 Answers to the questions

The explanation above should almost cover all these questions' answer, for supplementation, we answer these questions directly here.

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

Answer: AddrSpace::AddrSpace(), called by Kernel::Exec(), this will release all the memory space to the thread.

After MP2 implementation, it's now handled by AddrSpace::Load()

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

Answer: Thread::StackAllocate()

2.3.3 How does Nachos create and manage the page table?

Answer: AddrSpace::AddrSpace(), this will set up all the properties of the page, e.g. valid, dirty … .
After MP2 implementation, it's now handled by AddrSpace::Load()
AddrSpace::~ AddrSpace() will maintain the table by set the unused page to false.

2.3.4 How does Nachos translate addresses?

Answer: Machine::Translate(), calculate the virtual page number and offset, then find the corresponding physical page by the page table.

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

Answer: AddrSpace::Execute() initializes the registers, and Thread::StackAllocate() inits the machine states, including PC states and argument states.

2.3.6 Which object in Nachos acts the role of process control block

Answer: The Thread class.

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

Answer: After allocate and initialize stack (which is done by Thread::StackAllocate(), and a thread is added into the queue by Scheduler::ReadyToRun().

2.3.8 Please look at the following code from urserprog/exception.cc and answer the question:

```
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 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.)

Answer: If every virtual page storing the message point back to continuous physical pages with the same sequence, the message will be output without error. Otherwise, since the message isn't stored in continuous memory, it won't be output properly, i.e., an error will occur.