

NACHOS PROJECT ASSIGNMENT 3

CS 370 – OPERATING SYSTEMS, LUMS

“THIS ASSIGNMENT HAS BEEN ADAPTED FROM CS162 COURSE @WASHINGTON-UNIVERSITY”

USERLEVEL PROGRAMS & MULTIPROGRAMMING

DUE DATE: REFER TO INDIVIDUAL TASKS*

* Slips not applicable on non-programming tasks.

‘YOU SHOULD READ THE PROJECT MECHANICS DOCUMENT BEFORE STARTING WORK ON THIS ASSIGNMENT.’

“NACHOS HAS BEEN DEVELOPED BY TOM ANDERSON, WE ONLY REUSE THE SOURCE”

-- All the submissions will be checked for plagiarism - do not resort to any unfair means --

1. System Calls (100)

Implement system call and exception handling. You must support all of the system calls defined in `syscall.h` (in the `userprog` directory), except for `thread fork` and `yield`, which can be implemented for extra credit (see #3). We have provided you an assembly-language routine, `start.s` (it can be found in the `test` directory in code), to provide a way of invoking a system call from a C routine (UNIX has something similar --try `"man syscall"`). Note: You'll need to do part 2 of this assignment in order to test out the `"exec"` and `"wait"` system calls.

Note that you will need to make Nachos kernel resilient from user program errors --there should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt).

Check the file `userprog/syscall.h` and you will find declarations for nachos system calls.

```
#define SC_Halt      0
#define SC_Exit      1
#define SC_Exec      2
#define SC_Join      3
#define SC_Create    4
#define SC_Open      5
#define SC_Read      6
#define SC_Write     7
#define SC_Close     8
#define SC_Fork      9
#define SC_Yield    10
```

Exceptions or interrupts are the way a user program take service from kernel, i.e kernel will run in the context of user program. When a user program is running in user mode, there is no way to directly go to kernel mode.

Rather every CPU provides special instructions to trap to Special places in memory, which contains interrupt/exception handlers not alterable by the user and control is returned to kernel code in kernel mode. This trap to kernel mode via system call is for security and consistency. When a system call occurs, a Syscall Exception is Raised.

Check the following red portion inside `Machine::OneInstruction()` function in `machine/mipsim.cc`

```
case OP_SYSCALL:
    RaiseException(SyscallException, 0);
    return;

case OP_XOR:
    registers[instr->rd] = registers[instr->rs] ^ registers[instr->rt];
    break;
```

Hint : Try to find why the `OP_XOR` case in has a "break" but `OP_SYSCALL` has a "return". You should be able to find it out by yourself, when you understand the different types of exceptions that may occur and associated actions. Consequently you will have to add some code (in each system call function) that may not execute in case of a System Call in the `OneInstruction()` function. Please see the `OneInstruction()` to understand what will not execute if there is return instead of break in case of a System Call.

Machine::RaiseException is in machine/machine.cc

RaiseException saves address of the exception creating instruction, changes to Kernel Mode and calls the ExceptionHandler (Interrupt Handlers for Intel 80X86 architectures), where control reappears as kernel mode. When returning from Handling the exception it again returns to UserMode again, as highlighted in violet in the code block below.

```
void Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);

    // ASSERT(interrupt->getStatus() == UserMode);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);           // finish anything in progress
    interrupt->setStatus(SystemMode);
    ExceptionHandler(which);     // interrupts are enabled at this point
    interrupt->setStatus(UserMode);
}
```

ExceptionHandler is in userprog/exception.cc Default implementation is simple and implements only SC_Halt, which just halts the machine. All other calls are signaled with False Assertion.

For this assignment, you have to fill this ExceptionHandler for all the system calls that are in syscall.h. You can make a big switch statement for this purpose and create separate functions for each type of system call. Call the corresponding function in the case of any system call. Something like this:

```
void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);
    int rv=0; // the return value from a syscall
    if (which == SyscallException) {
        switch (type) {
            default:
                DEBUG('a', "Unknown syscall - shutting down.\n");
            case SC_Halt:
                DEBUG('a', "Shutdown, initiated by user program.\n");
                interrupt->Halt();
                break;
            case SC_Create:
                DEBUG('a', "Create syscall.\n");
                Create_Syscall(machine->ReadRegister(4));
                break;
            case SC_Open:
                -----
        }
    }
}
```

Warning: Do not change any file in machine folder when programming. They are part of machine simulation and resembles the hardware. When we are to write an operating system, we can't change the hardware design, those are already fixed.

2. Enabling Multiprogramming (100)

Implement multiprogramming. The code we have given you is restricted to running one user program at a time. You will need to: (a) come up with a way of allocating physical memory frames so that multiple programs can be loaded into memory at once (cf. bitmap.h), (b) provide a way of copying data to/from the kernel from/to the user's virtual address space (now that the addresses the user program sees are not the same as the ones the kernel sees), and (c) add synchronization to the routines that create and initialize address spaces so that they can be accessed concurrently by multiple programs.

We first need to write code for the system calls SC_Exec and SC_Exit assuming monoprogramming. But we cannot even test these right in that form, because calling SC_Exec means creating a user process from a user process which -- by definition is multiprogramming

Suppose from user application (e.g., Shell), we would like to invoke a new user program (e.g., sort)

```
SpaceId myProcess; //Address Space identifier of "myprocess"
```

```
myProcess = Exec("../test/sort");
```

And this will execute noff format binary file "sort" residing in test folder as a user process.

You may see the StartProcess function in userprog/progtest.c.

This function is called from .threads/main.cc when -x is specified as an option when running nachos.

Essentially, what we need to do in this part of the assignment is to port this code as a system call handler in /userprog/exception.cc.

StartProcess gets executable file name as parameter (actually the pointer). For StartProcess that pointer is inside kernel memory, but for SC_Exec the pointer is inside user memory. So we have to use machine->ReadMemory(..) to read that location.

MIPS System call passes parameters into registers r4, r5, r6, r7...

To get the address of the filename, we write

```
int buffadd = machine->ReadRegister(4); /* only one argument, so that's in R4 */
```

Next we read the file name into kernel space, using ReadMem.

Now we have the filename, and we can open it. Open it, as it is in StartProcess, calculate space needed, create a thread, allocate address space and load

the executable file into memory (Monoprogramming model like StartProcess) A thread is not created in Start Process, and in fact its not necessary for monoprogramming, but as we eventually are going to make it multiprogramming, creating a "execute-worker" thread for the process makes work easier for future.

```
OpenFile *executable = fileSystem->Open(filename);
```

```
AddrSpace *space;
```

```
space = new AddrSpace(executable);
```

```
Thread *t = new Thread(tname);
```

```
t->space = space;  
int processId = t->getId(); /* create this function by yourself */
```

We also need to fork the thread to create a kernel level stack (user level stack is created by AddrSpace) and also to schedule it, so that the newly created user process runs.

Check StartProcess function. There are some initialization tasks about the register set and machine state which needs to be configured and restored each time a user-process-executing-thread is scheduled or yielded when a new process is started and is ready to run. As StartProcess doesn't create new threads, those initializations are written in StartProcess. But here we are using a separate kernel thread to execute the process. So we need those initializations exactly when that thread is scheduled, and not before or not after.

Create a function inside exception.cc, say for example, processCreator, and fork the executor thread with that function. *t->Fork(processCreator,0);*

processCreator may look like the following function:

```
void processCreator(int arg)
{
    currentThread->space->InitRegisters();
    currentThread->space->RestoreState();      // load page table register
    machine->Run();                          // jump to the user program
    ASSERT(FALSE);                          // machine->Run never returns;
}
```

So when the thread *t* is forked with processCreator, *t* is scheduled. When *t* gets a chance to run as a kernel thread, it initializes its register set and states for the user process. After Machine::Run is executed, the user process runs.

MEMORY MANAGEMENT

- Several program needs to reside in main memory
- One program should not be able to access another program's area
- This memory management should be transparent to each user process, i.e memory should be managed entirely by operating system and hardware, user process should be invariant whether there is multiprogramming or monoprogramming

Here we use page table based memory management. Each process has its own address space. Each process understands and realizes addresses of its own address space only. The page table translates virtual addresses (address from a processes own address space) to physical address (actual address in physical memory accessible by kernel only).

The memory is divided into several fixed size pages.

Check `/machine/translate.cc` to see how nachos MIPS simulator translates the virtual address into physical addresses.

We need a page table for each process.

You need to work in the `AddrSpace` class.

The some important code is given on the next page let's try to understand it.

```

AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) &&
        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

    // how big is address space?
    size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
          + UserStackSize; // we need to increase the size
                          // to leave room for the stack
    numPages = divRoundUp(size, PageSize);
    size = numPages * PageSize;

    //print the number of pages needs to allocated for this user process
    // printf("number of pages = %ud\n", numPages);

    ASSERT(numPages <= NumPhysPages); // check we're not trying
                                     // to run anything too big --
                                     // at least until we have
                                     // virtual memory

    DEBUG('a', "Initializing address space, num pages %d, size %d\n",
          numPages, size);
    // first, set up the translation
    pageTable = new TranslationEntry[numPages];
    for (i = 0; i < numPages; i++) {
        pageTable[i].virtualPage = i; // for now, virtual page # = phys page #
        pageTable[i].physicalPage = i;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE; // if the code segment was entirely on
                                     // a separate page, we could set its
                                     // pages to be read-only
    }
}

```

RED MARKED LINES NEED SPECIAL ATTENTION AND CORRECTION FOR MULTIPROGRAMMING.

ASSERT(numPages <= NumPhysPages)

This assertion should be on Number of Free pages remaining, rather than total number of pages.

pageTable = new TranslationEntry[numPages];

Fine, they have made the page table for us 😊

pageTable[i].physicalPage = i;

Here virtual page is no longer = physical page. Rather we need to find a free page to assign for that virtual page. Also we should update the free pages data structure so that the page just assigned is not assigned to some other process again until it's freed.

```

// zero out the entire address space, to zero the uninitialized data segment
// and the stack segment
    bzero(machine->mainMemory, size);

// then, copy in the code and data segments into memory
    if (noffH.code.size > 0) {
        DEBUG('a', "Initializing code segment, at 0x%x, size %d\n",
            noffH.code.virtualAddr, noffH.code.size);
        executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
            noffH.code.size, noffH.code.inFileAddr);
    }
    if (noffH.initData.size > 0) {
        DEBUG('a', "Initializing data segment, at 0x%x, size %d\n",
            noffH.initData.virtualAddr, noffH.initData.size);
        executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),
            noffH.initData.size, noffH.initData.inFileAddr);
    }
}

AddrSpace::~AddrSpace()
{
    delete pageTable;
}

```

// zero out the entire address space,

bzero(machine->mainMemory, size);

This is totally fatal here. We need to zero out the pages allocated for the process, not the entire machine->mainMemory. For monoprogramming full main memory is allocated to 1 program, so each time a new program is loaded, that memory needs to be initialized. But this is not the case for multiprogramming. So we should call bzero only in the loop, for each of the individual pages allocated.

```

executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),

                noffH.code.size, noffH.code.inFileAddr);

executable->ReadAt(&(machine->mainMemory[noffH.initData.virtualAddr]),

                noffH.initData.size, noffH.initData.inFileAddr);

```

Virtual address no longer equals to physical address. And we need translate noffH.code.virtualAddr to physical address explicitly here

delete pageTable;

yes, we will delete pageTable when address space destructor is called, but before doing that we need to update our free pages management data structure. i.e the pages that were allocated to the process are free now, and they should be updated to some data structure so that they can be used later.

Free pages can be tracked either by Bitmap class provided in Nachos.

During “nachos boot up” this free memory data structure needs to be initialized, and update is necessary

each time a new process is created or destroyed.

You will probably want to write some utilities like UNIX “cp” and “cat”, and use the “shell” provided in the “test” subdirectory to verify that system call handling and multiprogramming are working properly.

It is a long assignment, try to start early and write one system call each day.

3. Fork and Yield (Extra Credit)

Implement multithreaded user programs. Implement the thread fork and yield system calls, to allow a user program to fork a thread to call a routine in the same address space, and then ping pong between the threads. (Hint: you will need to change the kernel to allocate memory in the user address space for each thread stack.)

- THE END -

APPENDIX 1

The new code is spread over several directories. There are some new kernel files in “userprog”, there are a few additional machine simulation files in “machine”, and a stub file system in “filesys”. The user programs are in “test”, and utilities to generate a Nachos loadable executable are in “bin”. Since Nachos executes MIPS instructions (and there aren't very many MIPS machines left!), we also provide you a cross-compiler. The cross-compiler runs on Linux and compiles user programs into MIPS format.

You may find Narten's "road map" to Nachos helpful; Google it. Also, it is OK to change the constants in "machine.h", for example, to change the amount of physical memory, if that helps you design better test cases (you may choose not to). The files for this assignment include:

USERKERNEL.H, USERKERNEL.CC -- routines for booting and testing a multiprogramming kernel.

ADDRSPACE.H, ADDRSPACE.CC -- create an address space in which to run a user program, and load the program from disk.

SYSCALL.H -- the system call interface: kernel procedures that user programs can invoke.

EXCEPTION.CC -- the handler for system calls and other user-level exceptions, such as page faults. In the code we supply, only the “halt” system call is supported.

BITMAP.H, BITMAP.CC -- routines for manipulating bitmaps (this might be useful for keeping track of physical page frames)

FILESYS.H, OPENFILE.H (found in the filesys directory) -- a stub defining the Nachos file system routines. For this assignment, we have implemented the Nachos file system by directly making the corresponding calls to the UNIX file system; this is so that you need to debug only one thing at a time.

TRANSLATE.H, TRANSLATE.CC -- translation table routines. In the code we supply to run “halt”, we assume that every virtual address is the same as its physical address -- this restricts us to running one user program at a time. You will generalize this to allow multiple user programs to be run concurrently. We will not ask you to implement virtual memory support until in assignment 3; for now, every page must be in physical memory.

MACHINE.H, MACHINE.CC -- emulates the part of the machine that executes user programs: main memory, processor registers, etc.

MIPSSIM.H, MIPSSIM.CC -- emulates the integer instruction set of a MIPS R2/3000 processor.

CONSOLE.H, CONSOLE.CC -- emulates a terminal device using UNIX files. A terminal is (i) byte oriented, (ii) incoming bytes can be read and written at the same time, and (iii) bytes arrive asynchronously (as a result of user keystrokes), without being explicitly requested.

SYNCHCONSOLE.H, SYNCHCONSOLE.CC -- provides synchronized access to the console device.

Some text in this handout has been borrowed from Md Tanvir Al Amin's document on MULTIPROGRAMMING, PROCESS MANAGEMENT AND CONSOLE that may be found on this link:

<http://tanviramin.com/documents/nachos2.pdf>

Students may read this document for further understanding of this assignment in particular and NACHOS in general.