EECE 315 – UBC

Assignment 3

Overview

In this assignment you will focus on implementing **Multiprogramming** in Nachos.

Up to now, all the code you have written for Nachos has been part of the operating system kernel. In a real operating system the kernel not only uses its procedures internally, but also allows user-level programs to access some of its routines via system calls. An executing user program is a process. In this project you will modify Nachos to support multiple processes, using system calls to have processes request services from the kernel.

Since your kernel does not trust user programs to execute safely, the kernel and the (simulated) hardware will work together to protect the system from damage by malicious or buggy user programs. To this end, you will implement simple versions of key mechanisms found in real operating system kernels: virtual addressing, and protected system calls.

Virtual addressing prevents user processes from accessing kernel data structures or the memory of other programs; your kernel will use process page tables to safely allow multiple processes to reside in memory at the same time. With protected system calls and exceptions, all entries into the kernel funnel through a single kernel routine, "ExceptionHandler"; you will "bullet-proof" this routine so that buggy or malicious user programs cannot cause the kernel to crash or behave inappropriately. Finally, your kernel will use preemptive scheduling to share the simulated CPU fairly among the active user processes, so that no process can take over the system. All of these protection mechanisms require cooperation between the hardware and the operating system kernel software. Your implementation will be based on "hardware" support in the Nachos MIPS simulator, which resembles a real MIPS processor.

If all processes are created by other processes, then who creates the first user process? The operating system kernel creates this process itself as part of its initialization sequence. This is bootstrapping. You can "boot" the Nachos kernel by running nachos with the -x option (x for "execute"), giving the name of an initial program to run as the initial process. The Nachos release implements the -x option by calling StartProcess in progtest.c to handcraft the initial process and execute the initial program within it. The initial process may then create other processes, which may create other processes...and so on.

The first step is to read and understand the part of the system we have written for you. The code that you will have to look at and modify is spread over several

directories. There are some kernel files in "userprog", 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 with a cross-compiler. The cross-compiler runs on Linux and Solaris and compiles user programs into MIPS format.

The code we provide can run only a single user-level "C" program at a time. As a test case, we've provided you with a trivial user program, "halt". All "halt" does is to turn around and ask the operating system to shut the machine down. To run the "halt" program, make and then run Nachos in the "userprog" directory:

% cd userprog % ./nachos -x ../test/halt

Trace what happens as the user program gets loaded, runs, and invokes a system call.

The files for this assignment include:

- **progtest.cc** - test routines for running user programs.
- **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)
- **translate.h, translate.cc** -- translation table routines. In the code we supply, 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 project 3; for now, every page must be in physical memory.

HOW NACHOS WORKS IN USER PROGRAM MODE

StartProcess is called to execute the file passed as input to the Nachos.

This function opens the executable file, creates an address space for it, attaches the a ddress space with currentThread, initializes registers for the new user process a nd calls machine>Run() in machine/mipssim.cc. Creating the address space means allocating memory, specifying test, data and stack segments, loading instruction tex t in the text segment from NOFF format executable (NOFF = Nachos Object File Form at). For more details check the userprog/addrspace.cc file.

Task 1 Implement Multiprogramming

As discussed above Nachos support uni-programming model. You job is to enable Nachos to support multiprogramming. First, you will need basic facilities to load processes into the memory of the simulated machine. Spend a few minutes studying the AddrSpace class, and look at how the StartProcess procedure uses the AddrSpace class methods to create a new process, initialize its memory from an executable file, and start the calling thread running user code in the new process context. The current code for AddrSpace and StartProcess works OK, but it assumes that there is only one program/process running at a time (started with StartProcess from main via the nachos -x option), and that all of the machine's memory is allocated to that process. Your first job is to generalize this code to implement the Exec system call for the general case in which multiple processes are active simultaneously.

Implement a memory manager module to allow your kernel to allocate page frames of the simulated machine's memory for specific processes, and to keep track of which frames are free and which are in use (cf. bitmap.h). You might want to implement a **thread-safe** Table class that will help you with your implementation of AddrSpace. The Table class will store a collection of untyped object pointers indexed by integers in the range [0...size-1], and support the following methods:

```
/* Create a table to hold at most "size" entries. */

Table(int size)
/* Allocate a table slot for "object", returning the "index" of the     allocated entry;
otherwise, return -1 if no free slots are available. */

int Alloc(void *object)
/* Retrieve the object from table slot at "index", or NULL if that     slot has not been
allocated. */

void *Get(int index)
/* Free the table slot at index. */

void Release(int index)
```

Modify AddrSpace to allow multiple processes to be resident in the machine memory at the same time. The default AddrSpace constructor code assumes that all of the machine memory is free, and it loads the new process contiguously starting at page frame 0. You must modify this scheme to use your memory manager to allocate page frames for the new process, and load the process code and data into those allocated page frames (which are not typically contiguous). For now it is acceptable to fail and return an error (0) from Exec if there is not enough free total machine memory to load the executable file.

Test your code:

You cannot test your code until unless you complete Task 2 and Task 3, as you need to implement *exec* and *exit* functions in Nachos, so that user programs can start new processes. Only thing you can do here is to run the halt program as in previous assignment and make sure that your modified code works well with single programs.

NOTE: To test your code for Task 2 and Task 3, you need to finish Task 1 first, otherwise your output for Task 2 and Task 3 might be a deadlock.

Task 2: SC_Exit

After implementing the memory management, you can move on to the next step of multiprogramming: Implementing system call for exit.

Code to be implemented:
Exit() as a system call is handled as an exception and thus should be implemented in ***/userprog/exception.cc***. Currently, there is only one exception handled by default: ***SC_Halt***, which is corresponding to the system call: Halt(). You should implement the routines for Exit() whose corresponding type is***: SC_Exit***. Read how Halt() is processed and you should know how to pick out Exit() from all system call exceptions.

You are asked to print to the terminal the exit status which is the only argument of Exit() for the current thread who is going to end. And you should print out the number of spare pages spare **before** the current thread actually ends.
In the entry of all processes: **StartProcess()** in ***/userprog/progtest.cc***, add code to print out the total number of memory pages you have in your design. You should see different number of spare pages when you exit the thread.

Test your code:
In directory: ***/code/userprog/,*** run following command:
./nachos –x ../test/array
You can go the read the code for array in ***/code/test/array.c*** to understand what the code is doing, and you are encouraged to read the **Makefile** in /code/test to see how to compile a program for nachos
The exit status should display the sum of all the elements in the array. The sum is 1128. Code your exit handler to print this out.

Hints:

- If you want, you can go to */test/start.s* to have a general idea about how a function call is transformed into a system exception. And if you go to

/machine/mipssim.cc to read the code of *Run() and OneInstruction()*, you will understand why we should implement the code in ***/userprog/exception.cc***.

- You can find one available data structure for your memory tracker (memory management) in***:/userprog/bitmap.h***. If you decide to use it, don't forget to include the header in your code.
- For system calls, maybe you need to maneuver the **system interrupts** at the correct places to make sure system call routines won't be interrupted.
- Think about how to ***gracefully end a thread***, you shouldn't "delete" the thread directly since the running code is still in the "stack", as a member variable, of that thread. Direct deletion may cause the system to hang or halt totally (sure you can try and see the result, this is just my guess). **Find a nice way to end the thread in "thread.h" and you should delete your "space**" (unlike "stack", this is memory space for the user program running in this thread) in there.
- In general, the routine should include: **interrupt disable/enable, get exit status, get number of spare pages, and end the thread nicely**.

Task 3: SC_Exec

You will implement the routine for ***SC_Exec*** to enable your nachos to start a new thread when the running code wants to.

Code to be implemented

Currently, you should be able to keep a memory tracker (memory management) to track the uses of the physic pages. In the same way, you need to implement a tracker to record active processes. For each active process, you also need to assign a unique ID for it. You can add a member variable in thread class named as "**int spaceId**", and you will assign it in your ***SC_Exec*** routine. Your process tracker should be implemented globally, either as a global variable or heap object. It should be able to assign a unique spaceId to a new process when it is created. And it should mark the spaceId available when the corresponding thread exits.

In ***SC_Exec*** routine, you need to: copy the name of the executable from the "user memory space" (the virtual one) of the caller process to the "kernel memory space" (the space you are currently in when the PC is in system call routine). The address of the name string is stored in "register 4". Prepare for the called process in a similar way as ***StartProcess()*** does (**BUT NOT EXACTLY THE SAME**).

You will need to let the **PC increment by 4 manually**, it seems that nachos' virtual machine doesn't take care of that.

All in all, you need to modify ***thread.h*** and thread.cc to (1) add spaceId as a member variable, (2) find the right place to clear user space, (3) find the right "end" function

for finishing a thread. And you need to go to ***exception.cc*** ***SC_Exec*** part to (1) copy the name of the executable (2) open it as *StartProcess()* does, (3) prepare user memory space for the program, (4) assign a spaceId for that created new process and run the program in it, (5) let the **PC increment by 4**. And you should modify ***SC_Exit*** part to ***release the assigned spaceId*** of the thread which is going to end.

Don't forget interrupt disable/enable

Besides, you are asked to print following message to your terminal when a thread ends: **Exit status**, the **name of the thread exits** (use getName(), in your code, you should assign a name for your new thread), **number of spare memory pages**, **number of spare processes**.

Test your code:
In directory: ***/code/userprog/,*** run following command:
./nachos –x ../test/exectest
exectest will call exittest at runtime and initiate a new thread to run exittest. The spaceId of the thread for exittest will be returned to exectest and be used as the exit status of exectest. The correct result will show the exit status for "*main*" is *<a spaceId your processes tracker assigns to the thread of* ***exittest***>. The exit status for exittest should be 123.
If your implementation of memory management is incorrect, for example the original mono-programming way, you will result in a deadlock and you will need to terminate your terminal

Hints:

- Create a function named "***processorCreator()***" to wrap: **InitRegisters(), RestoreState(), machine->Run().** Once you have such a function to take care of the initialization of the new user program, you can use **thread::fork()** to insert the thread into the schedule list.
- You should avoid calling scheduler functions directly here.
- Don't forget to increment the PC
- You can set a maximum number of processes and use it to create your process tracker, and you just need to return the number of maximum processes – the number of active processes to get the number of spare processes (Or use bitmap functions).
- Read the comments in the top of ***execption.cc*** about the ***arguments and registers***.
- Read data in virtual memory through ***machine::ReadMem(),*** and you should be able to find a serial of functions to access registers and user space memory (virtual memory) in ***machine.h***. You will use them correctly to finish this task. Besides, you should read the **MACROs defined in machine.h** which is related to PC register.

- Don't modify the code in /machine, they are for the "machine" and you won't be able to modify your machine to install a operating system.

Evaluations

Task 1

Correct Implementation	50%

Task 2

Correct output for Exit	20%

Task 3

Correct output for Exec	30%

Submission Guidelines

You need to submit all the files that you will be modifying for this assignment:

Create and submit the zip file as an attachment to the eece315term2@gmail.com with the subject as follows:
"Assignment3: <Group No.>"

Once you submit the assignment, you will receive an automated reply from the system. You need to show that reply to the TA before they could assess your assignment in the lab, to make sure that you have submitted your assignment correctly.