[NACHOS] MULTIPROGRAMMING, PROCESS MANAGEMENT AND CONSOLE

January, 2008

Md Tanvir Al Amin

Nachos is an instructional operating system designed by Thomas Anderson at University of California, Berkeley. Originally written in C++ for MIPS, Nachos runs as a user-process on a host OS. Here, in this report we have discussed our work about adding multiprogramming facility and console support in nachos 3.4

1 [Nachos] Multiprogramming, Process management and console

… WHEN NACHOS REALLY LOOKS LIKE AN OPERATING SYSTEM

We can do meaningful operations with Nachos at kernel level, without even thinking of any user. That's exactly what we did for Threading and Synchronization assignment. Nachos was acting just as a thread package - Scheduling, Running and Switching between non preemptive kernel threads.

But, by definition, an operation system is to be a system organizer on behalf of the user, running in supervisor mode, directly communicating with hardware and presenting some level of abstraction [1][2].

1.1 SOME BACKGROUND

User Program and Its difference with Kernel

An user program is a sequence of instructions for a specific machine, which it can directly execute by loading in memory. Full instruction set of an architecture isn't available for a user program. To ensure safety, security and consistency of kernel data structures, certain memory locations and instructions are available to the kernel only, i.e. they are available only in supervisor mode.

Kernel runs in supervisor mode (or kernel mode) and as a manager for all resources available, it works on behalf of the user. Kernel loads one or many of those user programs in memory, allocates and de allocates memory, registers and frees up resources.

There are two extreme ends in OS design. One is monolithic approach, and the another is microkernel based. In monolithic design, entire kernel is loaded as a single big program in single address space. Each part of it can access other parts or variables and can call any function residing in the address space. If there is fatal bug in a single portion, the entire kernel is hooked up and we need to restart the machine. Monolithic kernel is fast, and if well written and bug free – is best suited for performance.

On the other end is Microkernel. Just most necessary functions are part of kernel – hence the name microkernel. And rest of the operations are written as services. Servers operate in user mode and one server can communicate with another via message passing. The good point is that - if any modules fail, the system doesn't hang and just restarting that service usually solves the problem. The bad point is that – The good point comes only after a penalty paid in performance. Windows NT initially was designed with

microkernel approach. Though it resulted in maintainable modular code, they eventually had to move most of the parts to kernel to make it a successful business product.

So we can understand, Kernel is what runs in supervisor mode, and all other is user programs [1]. Compilers, window managers, and utility programs packaged with a typical operating system are actually user programs though they are part of system software. So all the Linux distributions are basically same. If we don't alter the kernel code, they differ only in provided utility programs.

Transition of User mode and Kernel mode in Unix

After the system boots up, kernel does some management tasks, creates and initializes the data structures necessary. Unix based operating systems then create [3] Process 0 which runs in kernel mode. Process 0 forks and creates User mode Process called init(). Every other user programs are created by calling fork() copying parent's address space. Record of parent child relationships are kept inside several kernel data structures. [3]

When user process is let to run, the system is in user mode and kernel mode (or supervisor mode) is achieved only via traps or system calls [6]. For detailed study, please check out the associated references.

1.2 USING USER PROGRAMS IN DEFAULT IMPLEMENTATION IN NACHOS

Nachos starts up execution at main() function in threads/main.cc. There are blocks for THREADS, USER_PROGRAM, NETWORK, FILE_SYSTEM tests. Here we will work with user program tests. The default nachos implementation can just run a single user program specified in command line.

To test user programs we will run the "halt" program. It is in test directory. We have to enter userprog directory and run nachos like

```
cd userprog
```

```
./nachos -x ../test/halt
```

The halt.c program is written as :

```
#include "syscall.h"

int
main()
{
    Halt();
    /* not reached */
}
```

Commands for compiling this user program when nachos is compiled is included in test directory's Makefile, as

```
halt.o: halt.c
	$(CC) $(CFLAGS) -c halt.c
halt: halt.o start.o
	$(LD) $(LDFLAGS) start.o halt.o -o halt.coff
../bin/coff2noff halt.coff halt
```

If we modify only the user program, we need to recompile only the program, which can be done by executing command "gmake" while in test directory. If we execute gmake from nachos root entire source tree will be compiled according to the Makefile in each directory.

The default nachos implementation only takes an argument for the name of userprogram and runs it in a simple memory model and exits. It neither supports multiprogramming for user programs, nor have any process management or console functionality. Well, that is the task for Nachos assignment – 2.

> **Note :** The test directory contains many user programs. But with default implementation, only halt.c will work and the others will signal illegal operations, because those programs contains system calls and console writes which are not yet implemented. Only SC_Halt is implemented in userprog/exeception.cc

1.3 HOW NACHOS WORKS IN USER PROGRAM MODE

If –x is specified, main() function in /threads/main.cc looks for a user program in command line. If one is found StartProcess(….) is called to execute that file. If –c is found console test demo runs.

StartProcess is in userprog/progtest.cc

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

For ease of checking, important lines are shown in red-bold in following code blocks. Debug and routine task code lines are grayed.

```
void StartProcess(char *filename)
{
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
      printf("Unable to open file %s\n", filename);
      return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;              // close file

    space->InitRegisters();         // set the initial register values
    space->RestoreState();          // load page table register

    machine->Run();                 // jump to the user progam
    ASSERT(FALSE);                  // machine->Run never returns;
                                // the address space exits
                                // by doing the syscall "exit"
}
```

Machine::Run() is part of MIPS simulator.

When Machine::Run() is called, we actually have the user program running. (To be a purist, simulating in the MIPS environment created). Machine::Run() explicitly set the mode in UserMode by calling setStatus(UserMode). Then it takes and executes each instruction via OneInstruction, and signals a clock tick.

Machine::OneInstruction written in machine/mipssim.cc really does the work of a decode, execute, write back stage of a MIPS machine. Check that function to see how it actually does it.

In real system, this will be done by CPU, and clock tick will be occurred by the real time clock in hardware. Clock ticks generate a clock tick interrupt. So in real system, this will cause hardware trap and cause system to run interrupt handling code written in special area of memory. On the other hand in nachos this will cause the simulated hardware to run codes in Interrupt::OneTick().

```
void Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    if(DebugIsEnabled('m'))
        printf("Starting thread \"%s\" at time %d\n",
             currentThread->getName(), stats->totalTicks);
    interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
      interrupt->OneTick();
      if (singleStep && (runUntilTime <= stats->totalTicks))
        Debugger();
    }
}
```

```
void Interrupt::OneTick()
{
    MachineStatus old = status;

// advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
      stats->systemTicks += SystemTick;
    } else {                                // USER_PROGRAM
      stats->totalTicks += UserTick;
      stats->userTicks += UserTick;
    }
    DEBUG('i', "\n== Tick %d ==\n", stats->totalTicks);

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff);           // first, turn off interrupts
                              // (interrupt handlers run with
                              // interrupts disabled)
    while (CheckIfDue(FALSE))       // check for pending interrupts
      ;
    ChangeLevel(IntOff, IntOn);           // re-enable interrupts
    if (yieldOnReturn) {            // if the timer device handler asked
                              // for a context switch, ok to do it now
      yieldOnReturn = FALSE;
      status = SystemMode;          // yield is a kernel routine
      currentThread->Yield();
      status = old;
    }
}
```

As we can see where, there is a provision for Thread yield in Interrupt::OneTick. If –rs switch is specified, threads are set to be subject to random switching at defined points, then Interrupt::OneTick may switch a context. (Nachos don't have time slicing implemented).

If current Thread is not yielded control again returns inside Machine::Run(), it fetches another instruction according to PC via `registers[PCReg]` and executes it.

At the last of user program it calls (automatically added) the SC_Halt system call. This system call is meant to do necessary process management (to be added by the student) and clean up tasks when a process terminates.

1.4 NACHOS SYSTEM CALLS

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 on context of the user program. When a user program is running is user mode, there is no way to directly go to kernel mode. Rather every CPU provides special instructions to trap in 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 system is for security and consistency. When a system call occurs, a Syscall Exception is Raised in MIPS [7]. For details on the actual MIPS cpu mechanism refer to the *Computer Organization and Design* books by Henessy and Patterson.

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

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 8oX86 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);
}
```

So we learn, basically this is the way control switches from the program running in user mode to kernel. Kernel will run in the context of that program. A running user program can return to kernel due to hardware interrupts, software exceptions, errors. Clock ticks are also hardware exceptions. In real hardware when a clock tick occurs, the clock tick exception handler take some scheduling decisions about if a context switch is to be done or whether some program more eligible for the CPU is waiting.

A process is created or halted, a program asks for read or write operations, asks to spawn a thread or to join another thread, claims dynamic memory or resources or asks for some other operating system "assistance" – everything is done via system calls, and these provide ample opportunity for the kernel to take control back, do some tasks and return to user again.

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

```
void ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);

    if ((which == SyscallException) && (type == SC_Halt)) {
```

```
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
    } else {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
}
```

For Nachos-2 assignment we have to fill this ExceptionHandler up for at least 5 more exceptions.

Check the *Nachos Primer* document [4] for details about Nachos machine simulation.

> **Notes:** When in actual hardware, implementing system calls requires a control transfer which involves some sort of architecture specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the kernel so software simply needs to set up some register with the system call number they want and execute the software interrupt. For MIPS there is a SYSCALL instruction. With Intel CPU, we can trap to kernel mode via INT instruction. Pentium II provides fast system call interface via SYSENTER and SYSEXIT instructions.

1.5 NACHOS CONSOLE

Go to userprog directory, and start nachos as

```
./nachos -c
```

Nachos console demo (ConsoleTest function) will start, which just prints what user have written.

ConsoleTest function is in userprog/progtest.cc. It shows how to implement and synchronize a shared console, when there are multiple processes.

1.6 TASKS TO PERFORM

Default Nachos executes user programs only using a simple memory model (check addrspace.cc), and only one program can run at a time. We have to alter address space creation in such way so that multiprogramming is supported, and several user programs can reside in the provided memory and one program can't access other program's memory area (memory protection). Each user program will use its address space as if it is the only program running in a Flat model (i.e. Memory management will be done by Nachos, not the user program). In short we have to implement page table based virtual memory scheme [1] to support multiprogramming.

User processes are meant to have process Id. One user process can create another user process, so there is a tree hierarchy. Parent can also wait for the child. We have to add support for process management, so that process parent child relationships are maintained in a data structure. Proper actions are taken for the cases parent exits before child, parent never calls join or parent calls join and child is already exited and so on. So we need to add classes and related functions so that **process management** is done when a user program created, exits and Joins other program.

To implement a fault free console for user i/o, we have to write synchronized console class. The ConsoleTest will provide clues for a synchronized console.

We have to write system call handlers so that a **new user program can be created** (similar to fork in linux, but not like the thread Fork of nachos), one process can **wait** for another and user program is supported with **Read** and **Write** operations. So in exception.cc, we have to write system calls handlers for **SC_Exec, SC_Exit, SC_Write, SC_Read, SC_Join** so that Multiprogramming, Console I/O and Process Management is supported. Check Nachos-2 assignment [8] description for detailed specification.

In our implementation we also implemented **PageFaultException** handler so that swapping of memory pages in and out of swap-space was also supported.

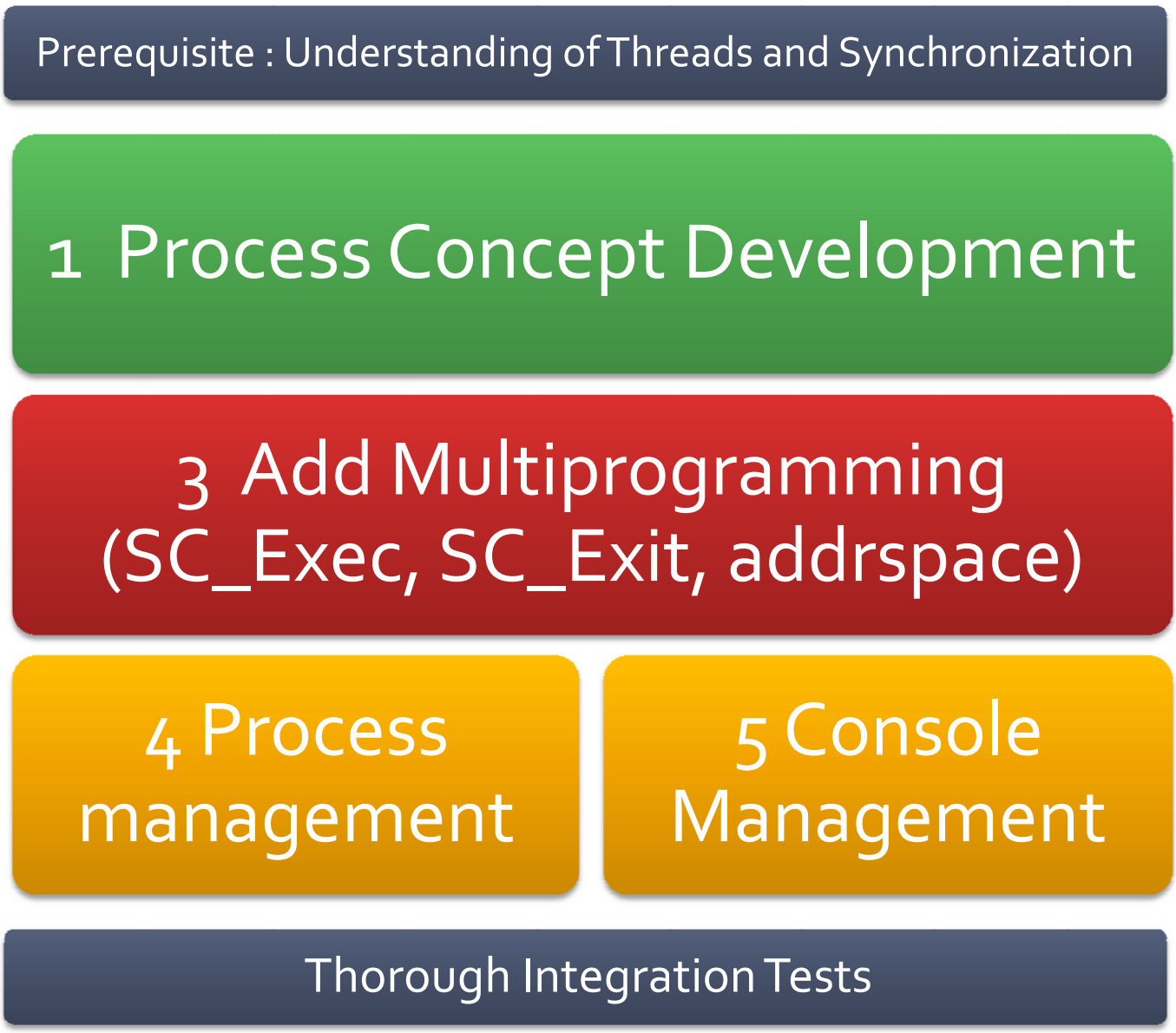
To complete the project successfully, one needs to have successfully implemented Nachos-1, **Threads and Synchronization Primitives**. In this concept development stage, you are highly encouraged to study the source codes starting from **main.cc** to **system.cc, interrupt.cc, scheduler.cc, machine.cc, mipssim.cc, list.cc, bitmap.cc, synchlist.cc, exception.cc, addrspace.cc, progtest.cc, synch.cc, thread.cc, translate.cc**. You will also need **console.cc** later.

> **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 Tracking the Project

… DEPENDENCY OF TASKS, WHICH ONE TO DO WHEN

When doing this project we had to work iteratively or like spiral. Things are interrelated, so one task wasn't complete in single pass. Here our process model is described. It may not be the best model to follow, but it worked fine for us. This whole report also follows the following sequence. The numbers on left are associated sections.



3 Add Multiprogramming

Default nachos provides example of running use program through monoprogramming [Refer to sec 1.3]

To create new process from another process, we need to invoke system calls.

SC_Exec is the system call to start a new user process.

SC_Exit is the system call to exit a user process.

In section 3.1 and 3.2 we at first write codes for the system calls SC_Exec and SC_Exit assuming monoprogramming. But we can't 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. In section 3.3 we add a typical memory management for multiprogramming so that several user process can reside in main memory – hence complete the multiprogramming part.

3.1 SC_EXEC

Use Case

From user program we want to write like

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

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

Idea

The basic step of works needed for this is described in /userprog/progtest.cc, in function StartProcess (refer to section 1.3 of this document). This function is called from .threads/main.cc when –x is specified as an option when running nachos. So what we actually have to do in this section is to port this code as a system call handler in /userprog/exception.cc.

So start like this (inside /userprog/exception.cc).

```
Void
ExceptionHandler(ExceptionType which)
{
    int type = machine->ReadRegister(2);


    if (which == SyscallException)
    {
```

```
        switch(type)
        {
                case SC_Halt:
                {
                        DEBUG('a', "Shutdown, initiated by user program.\n");
                        interrupt->Halt();
                        break;
                }

                case SC_Exec:
                {

                        /* write code for handling Exec */
                        /* ----------------------------*/

                        /* be careful to on and off interrupts at proper places */

                        /* ----------------------------*/

                        /* return the process id for the newly created process, return value
                        is to write at R2 */

                        Machine->WriteRegister(2, processed);



                        /* routine task – do at last -- generally manipulate PCReg,
                        PrevPCReg, NextPCReg so that they point to proper place*/

                        int pc;
                        pc=machine->ReadRegister(PCReg);
                        machine->WriteRegister(PrevPCReg,pc);
                        pc=machine->ReadRegister(NextPCReg);
                        machine->WriteRegister(PCReg,pc);
                        pc += 4;
                        machine->WriteRegister(NextPCReg,pc);

                }

        }
    }
}
```

Guide : How to write code for handling Exec

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.

```
char *filename = new char[100];

//find a proper place to free this allocation

if(!machine->ReadMem(buffadd,1,&ch))return;
i=0;

while( ch!=0 )
{
    filename[i]=(char)ch;
    bufadd+=1;
    i++;
    if(!machine->ReadMem(bufadd,1,&ch))return;
}
filename[i]=(char)0;
/* now filename contains the file */
```

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 StartProcess, 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 have 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 configured and restored each time a user-process-executing-thread is scheduled or yield. 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 can look like the following function, which actually does some of the initialization tasks required (check what is in StartProcess and which of those are moved to processCreator)

```
void processCreator(int arg)

{

    currentThread->space->InitRegisters();
    currentThread->space->RestoreState();       // load page table register

    machine->Run();                 // jump to the user progam

    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 (the mips simulation).

Now the basic code for StartProcess is done. We don't bother about process id, until process management is done. For the time being just keep a provision of id in Threads, (may be with a static variable to keep track of threads created) and return that as process id.

Be sure to on and off interrupts at proper places. Be sure to write the routine task codes updating PCReg, NextPCReg and PrevPCReg.

3.2 SC_EXIT

Use Case

From user program we will write like Exit(1);

And this system call will cause operating system to do the cleanup tasks and mange internal data structures.

Guide

For the purporse of this section, we just need to finish the current thread, currentThread->Finish(). Be sure that the clean up tasks are done properly.

3.3 MEMORY MANAGEMENT

What we have done so far, is just part of a big job. We haven't yet done anything about managing memory.

- ➔ 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 a multiprogramming or monoprogramming.

Page Table

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.

At /machine/machine.h check

```
#define PageSize  SectorSize
#define NumPhysPages   128
```

vpn#	ppn#
1	1
2	3
3	4
4	9

Process #1 Page Table

vpn#	ppn#
1	2
2	7

Process #2 Page Table

vpn#	ppn#
1	5
2	6
3	8
4	10
5	11
6	12

Process #3 Page Table

1	Process 1 Area
2	Process 2 Area
3	Process 1 Area
4	Process 1 Area
5	Process 3 Area
6	Process 3 Area
7	Process 2 Area
8	Process 3 Area
9	Process 1 Area
10	Process 3 Area
11	Process 3 Area
12	Process 3 Area
13	Unallocated
14	Unallocated

Physical Memory

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.

Check /userprog/addrspace.cc and /userprog/addrspace.h. An address space was created inside SC_Exec handler.

`AddrSpace::AddrSpace(OpenFile *executable)`

`find`

```
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
    }
```

```
// zero out the entire address space, to zero the unitialized 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;
}
```

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 its freed.

```
// 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 addres no longer equals to physical address. And we need to translate `noffH.code.virtualAddr` to physical address explicitly here. We may need to write our own translation function because translate.cc translate function works on machine->pageTable, which is the pageTable of the current process, but here we are working with a different process.

```
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.

Keeping Track of Free Pages and Allocation of Pages

Free pages can be tracked either by Bitmap class provided in Nachos, or with Freelist array.

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.

4 Process Management

Unix maintains a parent child hierarchy of processes. We are to maintain parent child relationship of processes in Nachos. One process can join another process. For simplicity, here, only a parent can wait for its child.

In this chapter we :

1. Create the process class and update thread class.
2. Revisit SC_Exec and thereby complete it.
3. Add system call SC_Join and revisit SC_Exit.

4.1 PROCESS CLASS

We need to have a process class for each of the process.

Here is the interface of our process class.

```
#ifndef PROCESS_H_
#define PROCESS_H_

#ifdef USER_PROGRAM

#include "list.h"

class Thread;

enum ProcessStatus { PROCESS_JUST_CREATED, PROCESS_RUNNING, PROCESS_READY,
PROCESS_BLOCKED, PROCESS_ZOMBIE };


class Process{
private:

      int processId;
      Process *parent;
      Thread *container;
      int nChildren;   // number of child
      int exitCode;
      ProcessStatus status;

      List *children;
      List *waitqueue;
```

```
public:
      Process(Thread *myExecutor,Process *myParent);
      Process *getParent() { return parent;}
      Thread *getThread() {return container;}
      int numberOfChildren(){return nChildren;}
      ProcessStatus getStatus() {return status;}
      void setStatus(ProcessStatus st) {status = st;}

      void addChild(Process *myChild);
      void wakeUpJoiner();
      void exit(int ec);
      void addJoiner(Process *joiner);
      void deathOfChild(Process *p);

      int getId() {return processId;}

      int getExitCode(){return exitCode;}
      void dumpChildInfo();
      ~Process();
};

#endif

#endif /*PROCESS_H_*/
```

Inside thead.h we update the following (red marked):

```
#ifdef USER_PROGRAM
// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs];    // user-level CPU register state
    Process *userProcess;

  public:
    void SaveUserState();        // save user-level register state
    void RestoreUserState();            // restore user-level register state

    AddrSpace *space;                   // User code this thread is running.

    void setProcess(Process *);
    Process *getProcess(void) {return userProcess;}
#endif
```

The thread class also have a unique ID generator, which creates ID for each thread creator keeping track via a static class variable., and we used same ID for process, and returned as SpaceID for SC_Exec system call.

4.2 REVISIT SC_EXEC

A new process object is created inside handler of SC_Exec. (Red marked)
To keep track of the process hierarchy, each process object is linked with its parent via `Process *parent` pointer in process class, and parent's information is passed when the child is created.

Each process also have a child list. So parent process also attaches the child with itself (green marked).

```
Process *parentProcess=currentThread->getProcess();
Process *myProcess = new Process(t,parentProcess);
parentProcess->addChild(myProcess);
```

Now the question is, how do we understand, currentThread is the executor of the parent of the process we are going to create ? very easy, because this code is inside the system call SC_Exec, and when user invokes this system call from some user process, that process is the parent, and executor thread of that process will actually come to the system call handler, create the new process, create the new thread and do other jobs.

Inside SC_Exit, we need the parent to de-attach its child, change the childs state to zombie if parent hasn't exited yet, check for orphaned children when a process exits and other clean up tasks. But as these are highly correlated with SC_Join, we discuss these at next section.

We also have a provision for translation between process object and process ID easily. So we have a translation table defined in /threads/system.cc. Check /threads/system.cc and find that global objects are defined in this file. Inside system.cc we have

```
#ifdef USER_PROGRAM
      SynchConsole *synchConsole; // visit this later
      List *processTranslationTable;
      Process *initProcess;
      Lock *pttLock;
#endif
```

A lock is needed for the processTranslationTable, because it is a shared object.

We create these objects inside initialize function in /threads/system.cc

```
#ifdef USER_PROGRAM
    machine = new Machine(debugUserProg); // this must come first
    // <group 9>
    synchConsole = new  SynchConsole (NULL, NULL); // visit this later
    processTranslationTable = new List;
    pttLock= new Lock("Process Translation Table Lock");
    // </group 9>
```

```
#endif
```

Inside the system call handler (SC_Exec) we update these data structures :

```
pttLock->Acquire();
     processTranslationTable->SortedInsert((void *)myProcess,myProcess->getId());
pttLock->Release();
```

And Inside SC_Exit we update these also. But as SC_Exit is highly correlated with SC_Join, we discuss it there.

4.3 ADD SC_JOIN

Parent process also can call join on child.

The use case may be like :

```
newProc = Exec ( "../test/gchild" ) ;
for(i=0;i<10;i++)
      j += i;
Join ( newProc ) ;
```

In this case, the parent process will be blocked until newProc exits.

There are some complicacies which needs to be settled carefully.

- ➔ Parent calls join, but child is already finished.
- ➔ Parent exits, but there are child process running, who will be the parent now ?

In Our Solution, For the first case, the SC_Join system call will just return. Logic behind this is, parent process needs to be blocked until child exits. As child is already exited, we can think about it a "delta" (very small) block.

The second case arrives, because child management data structures were created by the parent, the need to be freed by some appropriate "Authority".

When child finishes, but parent is still running, we render the child as Zombie. Zombie state is necessary because the parent needs an entry for clean up tasks, return values when it may call join etc etc. If child process along with their data structures are deleted as soon as they exits, we can't implement join.

Thatsa why we keep in Process.h

```
enum ProcessStatus { PROCESS_JUST_CREATED, PROCESS_RUNNING, PROCESS_READY,
PROCESS_BLOCKED, PROCESS_ZOMBIE };
```

System call handler for SC_Join is simple.

1. It at first acquires the lock for Process Translation Table and searches the joineeID there and gets Process pointer for that ID.
2. It then checks whether calling with the joineeID is valid.
3. Assuming valid, it checks whether the joinee already exited (Zombie State)
    a. If zombie, then it gets the exit code stored in the process structure for that child
    b. If not zombie, then current thread sleeps (current thread is the thread for the calling process)
4. On invalid calls, -1 is returned.

Not to mention here that the system call handler at first does the routine task (setting correct values PCReg, NextPCReg, PrevPCReg)

Now that we kept a Zombie state. Zombie state means the child is functionally dead, but it has still the entry in its parent's child list. Now when this entry will be deleted ? Easy, when the parent exits.

So we now check the handler for SC_Exit

1. The process is marked as zombie
2. If there are joiner for the process, it is awaken. (interrupts off)
3. Now we need to care about the children it has For all child c in child_list
    i. If c is a zombie process (this child exited before the parent and now we are accessing its undeleted entries), delete it from processTranslation table and also delete the process object. (but don't delete it from the linked list right now)

    ii. If c is not a zombie process, it is still running. We assign a dummy process name initProcess , which is just a process and does nothing.

4. Delete from child list linked list all the zombie childs.
5. If the current process's parent is initProcess, it means this process lost its parent previously and now has a dummy as a parent, which is not going to exit. So in that case the process itself deletes its data structures.

5 Console Management

Sorry this chapter will be available on next iteration.

References

[1] Modern Operating Systems, *Andrew S Tanenbaum*

[2] Operating System Concepts, *Silberchatz, Galvin, Gagne*

[3] The Design of Unix Operating System *(Chapter 6,7) Maurice J Bach*

[4] The Nachos Primer, *Matthew Peters, Robert Hill, Shyam Pather*

[5] A Roadmap to Nachos, *Thomas Narten*

[6] Understanding the Linux Kernel, Section 1.6. *Daniel P. Bovet, Marco Cesati*

[7] Computer Organization and Design, *Patterson, Henessy.*

[8] Nachos -2 Assignment Description, *Md. Sohrab Hossain, Khaled Mahmud Shahriar, Md. Moazzem Hossain, Chowdhury Sayeed Hayder,* Department of Computer Science and Engineering, BUET.

[9] Nachos-1 Assignment Report Group-9 , *Sukarna Barua, Tanvir Al Amin Popel* (Level 3 Term 2, July 2007 Term, Department of Computer Science and Engineering, BUET.