An Introduction to Nachos

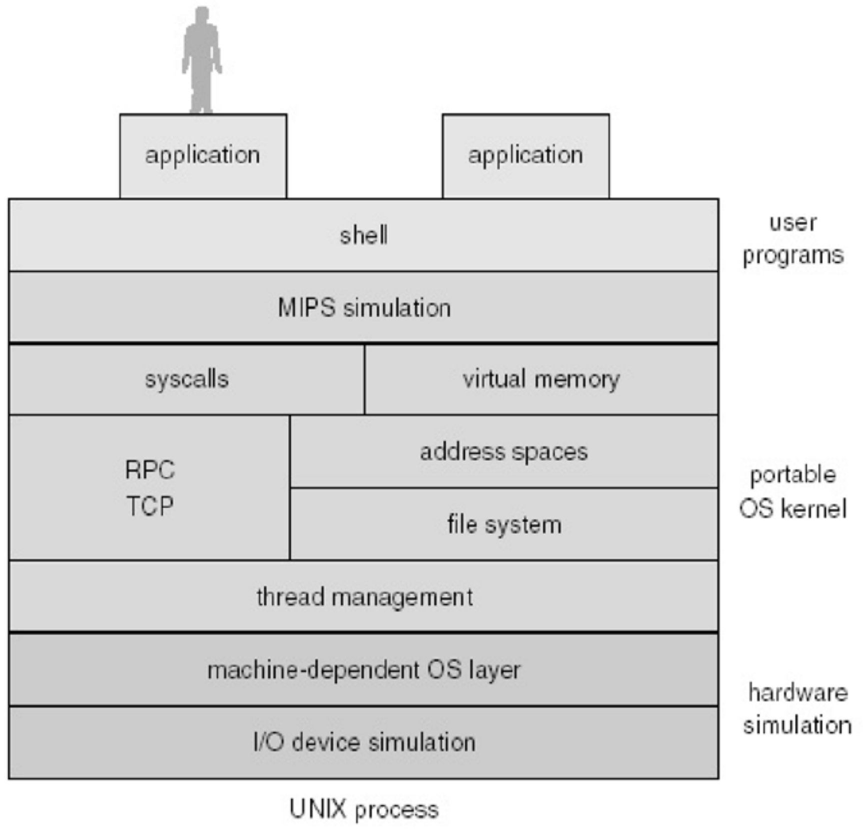
JIAYU LI

NachOS Directory and File Structure

- code/threads: Heart of the kernel scheduler, synch primitives, etc.
- code/filesys: Filesystem
- code/lib: Library routines
- code/machine: MIPS simulator and simulated hardware
- code/network: Networking
- code/test: test user programs in C. Need a cross-compiler
- code/userprog: Support for user-level processes (thread)

Nachos Architecture

- An instance of Nachos is a regular Unix process
- Processes (referred as “Threads” in Nachos) on Nachos are run by the MIPS emulator in code/machine/mipssim.cc and others
- “Kernel-level processes (or threads)” refer to processes running under Nachos OS
- “User-level processes (threads)” refer to processes running on Nachos. These programs are compiled in the code/test directory
- Each user-level process has a corresponding kernel-level process
- This means the user-level processes in Nachos has two-sets of registers and stacks (one under kernel-space and the other under user-level)



Nachos Architecture

- MIPS Simulator runs as a main event loop, invoked with Machine::Run (machine/mipssim.cc)
- Kernel code gets called from the simulator through (simulated) exceptions and interrupts (userprog/exception.cc)
- Interrupts cause the simulator to call the appropriate interrupt handler (machine/machine.cc)
- Exceptions and System Calls cause the simulator to call the interrupt handler (userprog/exception.cc)
- Returning from the interrupt handler or exception handler returns control to the simulator (userprog/exception.cc)

Nachos vs Real OS

- Nachos: OS runs on host machine; User threads run on machine simulator
- Real OS: OS and User processes run on same machine
- Nachos: **Hardware simulated**
  - ✓Interact with hardware by calling functions that eventually call underlying host OS library routines
- Real OS: **Hardware is real**
  - ✓Interact with hardware at a lower level Read/write device registers or issue special I/O instructions

Nachos vs Real OS

❖ **Nachos: Time is simulated (incremented at discrete points in real time, onetick(), interrupt.cc)**

✓Interrupts can’t happen immediately

✓Interrupts happen only in places where simulated time gets advanced. When simulated time advances, Nachos will check for interrupts

✓Simulated time advances when interrupts re-enabled, between user instructions, when nothing in the ready queue

❖ **Real OS: Time is continuous**

✓Kernel can be pre-empted anywhere where interrupts are enabled

Build Nachos

- Go to the build directory:
  - ✓cd nachos/code/build.linux
- Build NachOS:
  - ✓make clean, make depend, make
- Use ls command to check the executable file nachos (if you find, the compilation is successful)
- Run NachOS:
  - ✓./nachos –K (only run ThreadTest())
  - ✓./nachos –x compiled_userprogram_path (run user programs)

Build Nachos

- It does look like a cross-platform compilation issue. You can use the following command to install the related packages:
  - ✓ sudo apt install gcc-multilib g++-multilib

- If you've installed a version of gcc / g++ that doesn't ship by default you'll want to match the version as well:
  - ✓ sudo apt-get install gcc-4.8-multilib g++-4.8-multilib

Guide to reading the NACHOS source

- The Nachos MIPS Simulator
- Machine Components
  - ✓ Interrupt Management
- Threads & Scheduling
- System Calls and Exception Handling
- User-Level Processes
  - ✓ Process Creation
  - ✓ Creating a Noff Binary

The Nachos MIPS Simulator

- You will use the **simulated MIPS machine** to **execute test(user) programs**. Your nachos executable will contain a MIPS simulator that reads the test program executables as data and interprets them, simulating their execution on a real MIPS machine booted with your Nachos kernel. “It’s like a ship in a bottle.”

- The MIPS CPU Simulator

This simulator understands the format of **MIPS instructions** and the expected behavior of those instructions as defined by the MIPS architecture. When the MIPS simulator is executing a ‘‘user program’’ it simulates the behavior of a real MIPS CPU by executing a tight loop, fetching MIPS instructions from a simulated machine memory and ‘‘executing’’ them. (mipssim.cc)

Machine Components

- The Nachos/MIPS machine is implemented by the Machine object, an instance of which is created when Nachos first starts up. The Machine object exports a number of operations and public variables that the Nachos kernel accesses directly. In the following, we describe some of the important variables of the Machine object; describing their role helps explain what the simulated hardware does. (kernel.cc, machine.cc)

- The Nachos Machine object provides registers, physical memory, virtual memory support as well as operations to run the machine or examine its current state. When Nachos first starts up, it creates an instance of the Machine object and makes it available through the global variable machine. The following public variables are accessible to the Nachos kernel:

Machine Components

- **registers**

An array of 40 registers, which include such special registers as a stack pointer, a double register for multiplication results, **a program counter**, **a next program counter** (for branch delays), a register target for delayed loads, a value to be loaded on a delayed load, **and the bad virtual address after a translation fault**. The registers are number 0-39; see the file **machine.h** for symbolic names for the registers having special meaning (e.g., PCReg).

Although registers can be accessed directly via machine->registers[x], the Machine object provides special **ReadRegister()** and **WriteRegister()** routines for this purpose. (exception.cc)

- **mainMemory**

Memory corresponding to physical address x can be accessed in Nachos at machine->mainMemory[x]. By default, the Nachos MIPS machine has 31 pages of physical memory. The actual number of pages used is controlled by the **NumPhysPages** variable in **machine.h**.

Machine Components

- At this point, we know enough about the Machine object to explain how it executes arbitrary user programs. First, we load the program's instructions into the machine's physical memory (e.g, the machine->mainMemory variable). Next, we initialize the machine's page tables and registers. Finally we invoke machine->Run(), which begins the fetch-execute cycle for the machine. (addrspace.cc)

The Machine object provides the following operations:

- **Machine(bool debug)**

  The Machine constructor takes a single argument debug. When debug is TRUE, the MIPS simulator executes instructions in single step mode, invoking the debugger after each instruction is executed. The debugger allows one to interactively examine machine state to verify (for instance) that registers or memory contain expected values.

Machine Components

- **ExceptionType Translate(int virtAddr, int* physAddr, int size, bool writing)**

converts virtual address virtAddr into its corresponding physical address physAddr. Translate examines the machine‘s translation tables in order to perform the translation. When successful, Translate returns the corresponding physical address in physAddr. Otherwise, it returns a code indicating the reason for the failure (e.g., page fault, protection violation, etc.) Whenever a translation fails, the MIPS simulator invokes the Nachos routine RaiseException to deal with the problem. RaiseException is responsible for handling all hardware trap conditions. When RaiseException returns, the Nachos Machine assumes that the condition has been corrected an resumes its fetch-execute cycle. (translate.cc)

Machine Components

- Note that from a user-level process's perspective, traps take place in the same way; a trap handler is invoked to deal with the problem. However, from the Nachos perspective, RaiseException is called via a normal procedure call by the MIPS simulator.

- **OneInstruction()**

  does the actual work of executing an instruction. It fetches the current instruction address from the PC register, fetches it from memory, decodes it, and finally executes it. Any addresses referenced as part of the fetch/execute cycle (including the instruction address given by PCReg) are translated into physical addresses via the Translate() routine before physical memory is actually accessed.

Machine Components

- **Run()**

“turns on” the MIPS machine, initiating the fetch-execute cycle. This routine should only be called after machine registers and memory have been properly initialized. It simply enters an infinite fetch-execute loop. The main loop in Run does three things: 1) it invokes OneInstruction to actually execute one instruction, and 3) it increments a simulated clock after each instruction.

- **int ReadRegister(int num)**

fetches the value stored in register num.

- **void WriteRegister(int num, int value)**

places value into register num.

Machine Components

- **bool ReadMem(int addr, int size, int* value)**

  Retrieves 1, 2, or 4 bytes of memory at virtual address addr. Note that addr is the virtual address of the currently executing user-level program; ReadMem invokes Translate before it accesses physical memory. ReadMem is used (for instance) when dereferencing arguments to system calls. (translate.cc)

- **bool WriteMem(int addr, int size, int value)**

  writes 1, 2, or 4 bytes of value into memory at virtual address addr.

Interrupt Management

- Nachos simulates interrupts by maintaining an event queue together with a simulated clock. As the clock ticks, the event queue is examined to find events scheduled to take place now. The clock is maintained entirely in software and ticks under the following conditions: (interrupt.cc)

  - ✓ Every time interrupts are restored (and the restored interrupt mask has interrupts enabled), the clock advances one tick. Nachos code frequently disables and restores interrupts for mutual exclusion purposes by making explicit calls to **interrupt::SetLevel()**.
  - ✓ Whenever the MIPS simulator **executes one instruction**, the clock advances one tick.
  - ✓ Whenever **the ready list is empty**, the clock advances.
  - ✓ Whenever the clock advances, the event queue is examined and any pending interrupt events are serviced by invoking the procedure associated with the timer event (e.g., the interrupt service routine). All interrupt service routines are run with interrupts disabled, and the interrupt service routine may not re-enable them.

Interrupt Management

All routines related to interrupt management are provided by the Interrupt object.

- **void Schedule(VoidFunctionPtr handler, int arg, int when, IntType type)** schedules a future event to take place at time when. When it is time for the scheduled event to take place, Nachos calls the routine handler with the single argument arg.
- **IntStatus SetLevel(IntStatus level)** Change the interrupt mask to level, returning the previous value. This routine is used to temporarily disable and re-enable interrupts for mutual exclusion purposes. Only two interrupt levels are supported: IntOn and IntOff.
- **OneTick()** advances the clock one tick and services any pending requests (by calling CheckIfDue). It is called from machine::Run() after each user-level instruction is executed, as well as by SetLevel when the interrupts are restored.
- **bool CheckIfDue(bool advanceClock)** examines the event queue for events that need servicing now. If it finds any, it services them. It is invoked in such places as OneTick.
- **Idle()** `advances'' to the clock to the time of the next scheduled event. It is called by the scheduler (actually Sleep()) when there are no more threads on the ready list and we want to ``fast-forward'' the time.

Threads & Scheduling

- **Threads** that are ready to run are kept on the **ready list**. A thread is in the READY state only if it has all the resources it needs.

- **The scheduler decides which thread to run next**. The scheduler is invoked whenever the current thread wishes to give up the CPU. For example, the current thread may have initiated an I/O operation and must wait for it to complete before executing further.

- The Nachos scheduling policy is simple: threads reside on a single, unprioritized ready list, and threads are selected in a round-robin fashion. That is, threads are always appended to the end of the ready list, and the scheduler always selects the thread at the front of the list.

Threads & Scheduling

- Scheduling is handled by routines in the Scheduler object:

- **void ReadyToRun(Thread *thread):**

  Make thread ready to run and place it on the ready list. Note that ReadyToRun doesn't actually start running the thread; it simply changes its state to READY and places it on the ready list. The thread won't start executing until later, when the scheduler chooses it. ReadyToRun is invoked, for example, by Thread::Fork() after a new thread has been created.

- **Thread *FindNextToRun():**

  Select a ready thread and return it). FindNextToRun simply returns the thread at the front of the ready list.

- **void Run(Thread *nextThread):**

  Do the dirty work of suspending the current thread and switching to the new one. Note that it is the currently running thread that calls Run(). A thread calls this routine when it no longer wishes to execute.

Threads & Scheduling

❖**Run()** does the following:

✓Before actually switching to the new thread, check to see if the current thread overflowed its stack.

✓Change the state of newly selected thread to RUNNING. Nachos assumes that the calling routine (e.g. the current thread) has already changed its state to something else, (READY, BLOCKED, etc.) before calling Run().

✓Actually switch to the next thread by invoking Switch(). After Switch returns, we are now executing as the new thread.

✓If the previous thread is terminating itself (as indicated by the threadToBeDestroyed variable), kill it now (after Switch()). It is important to understand that it is actually another thread that physically terminates the one that called Finish().

System Calls and Exception Handling

- User programs invoke system calls by executing the MIPS “syscall” instruction, which generates a hardware trap into the Nachos kernel. The Nachos/MIPS simulator implements traps by invoking the Routine RaiseException(), passing it a arguments indicating the exact cause of the trap. RaiseException, in turn, calls ExceptionHandler to take care of the specific problem. ExceptionHandler is passed a single argument indicating the precise cause of the trap.

- The “syscall” instruction indicates a system call is requested, but doesn't indicate which system call to perform. By convention, user programs place the code indicating the particular system call desired in **register r2** before executing the “syscall” instruction. **Additional arguments to the system call (when appropriate) can be found in registers r4-r7**. Function (and system call) return values are expected to be in **register r2 on return**.

Nachos Process (Thread) Creation

- Nachos processes are formed by creating an address space, allocating physical memory for the address space, loading the contents of the executable into physical memory, initializing registers and address translation tables, and then invoking *machine::Run()* to start execution. *Run()* simply "turns on" the simulated MIPS machine, having it enter an infinite loop that executes instructions one at a time).

- When support for multiple user processes has been added, two other Nachos routines are necessary for thread switching. Whenever the current thread is suspended (e.g., preempted or put to sleep), the scheduler invokes the routine *AddrSpace::SaveUserState()*, in order to properly save address-space related. This becomes necessary when using virtual memory; when switching from one process to another, a new set of address translation tables needs to be loaded. The Nachos scheduler calls *SaveUserState()* whenever it is about to preempt one thread and switch to another. Likewise, before switching to a new thread, the Nachos scheduler invokes *AddrSpace::RestoreUserState*. *RestoreUserState()* insures that the proper address translation tables are loaded before execution resumes.

Creating a Noff Binary

- Nachos is capable of executing a program containing MIPS instructions. For example, C programs in the test directory are compiled using gcc on a MIPS machine to create “.o” files. To create an a.out binary file, the loader prepends the instructions in test/start.s before the code of the user program. File start.s contains initialization code that needs to be executed before the user's main program. Specifically, the very first instruction in start.s calls the user-supplied main routine, whereas the second instruction invokes the Nachos Exit system call, insuring that user processes terminate properly when their main program returns. In addition, start.s contains stub modules for invoking system calls.

Multiprogramming

The Nachos Fork and Exec routines have completely different semantic from the Unix system calls. In particular, the Nachos system calls do the following:

- **Exec** Creates a new address space, reads a binary program into it, and then creates a new thread (via Thread::Fork) to run it.
- **Fork** Creates a new thread of control executing in an existing address space.

When a Nachos program issues an Exec system call, the parent process is executing the code associated with the Exec call. At some point during the Exec call, the parent will need to invoke Thread::Fork to create the new thread that executes the child process.

Multiprogramming

- When the MIPS simulator traps to Nachos via the RaiseException routine, the program counter (PCReg) points point to the instruction that caused the trap. That is, the PCReg will not have been updated to point to the next instruction.

- On the other hand, when a user program invokes a system call via the “syscall” instruction, re-executing that instruction after returning from the system call leads to an infinite system call loop. Thus, as part of executing a system call, the exception handler code in Nachos needs to update PCReg to point to the instruction following the “syscall” instruction. The following code updates the program counter. In particular, not properly updating NextPCReg can lead to improper execution. (exception.cc)

```
pc = machine->ReadRegister(PCReg);

machine->WriteRegister(PrevPCReg, pc);

pc = machine->ReadRegister(NextPCReg);

machine->WriteRegister(PCReg, pc);

pc += 4;

machine->WriteRegister(NextPCReg, pc);
```

Reference

- http://people.cs.uchicago.edu/~odonnell/OData/Courses/CS230/NACHOS/reading-code.html
- https://users.cs.duke.edu/~narten/110/nachos/main/main.html
- https://users.cs.duke.edu/~chase/nachos-guide/guide/nachos.htm#_Toc535602509
- http://www.sci.brooklyn.cuny.edu/~jniu/teaching/csc33200/files/0922-NachosOverview.pdf
- https://homes.cs.washington.edu/~tom/nachos/
- https://cseweb.ucsd.edu/classes/sp09/cse120/projects/nachos.pdf