The Nachos Instructional OS

- Allow students to examine, modify and execute operating system software.
- A skeletal OS that supports threads and user-level processes, and simulates the general low-level facilities of typical machines, including interrupts, virtual memory and interrupt-driven device I/O.
- Difference between Nachos and a real OS: Nachos runs as a single Unix process, whereas real operating systems run on bare machines.
- Bad news: still complicated, over 9K lines of C++ code.
- Good news: Can understand its basic mechanisms by reading about 1-2K lines of code.

Project 0

- Copy the source code from `~cs170/src/nachos/code` to your home directory.
- Compile the source code using `gmake` on a Linux-based Intel machine.
- Run threads demo under the threads subdirectory (just run kernel test threads).
- Run user program demo under the threads subdirectory. Nachos reads a user binary program and executes it.
- Trace the threads demo using "nachos -d". You may try to find the main C++ functions involved in this printed trace and guess what they do.

Subdirectories in Nachos Code

- `machine` — Basic machine specification (MIPS simulator).
- `threads` — threads management (Project 1).
- `userprog` — binary code execution and system calls (Project 2).
- `vm` — virtual memory (empty, Project 3).
- `filesys` — file system (used in Projects 2/3)
- `test` — binary test code (short user test programs, Projects 2/3).
- `network` — networking protocol (not used in our projects).
- `bin` — utilities/tools (binary format conversion).
**Two modes of executions**

- **User mode (executing MIPS instructions)**
  - Nachos executes user-level processes by loading them into the simulator’s memory, initializing the simulator’s registers and then running the simulator.
  - User programs can only access the memory associated with the simulated machine.

- **Kernel mode.**
  - Kernel executes when Nachos first starts up and loads a user program.
  - or when a user-program executes an instruction that causes a hardware trap (e.g., illegal instruction, page fault, system call, etc.).

---

**Timer object**

Emulates a real time clock.

- Generate interrupts at regular or random intervals
- Then Nachos invokes the predefined clock event handling procedure.
- Supported operation:
  - `Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom)`
  
  Create a real-time clock that interrupts every `TimerTicks (100)` time units

- Software managed clock.
  - The clock advances one tick after every restored interrupt or after the MIPS simulator executes one instruction.
  - When the ready list is empty, fast-advance ticks until the next scheduled event happens.

---

**Machine object**

- Implement Nachos/MIPS machine.
- an instance created when Nachos starts up.

**Supported public variables:**
  - Registers: 40 registers.
  - Memory: Byte-addressable.
  - Virtual memory: use a single linear page table or a software-managed TLB.

**Supported operations:**
  - `Machine(bool debug)`
  - `Translate(int virtAddr, int* physAddr, int size, bool writing)`
  - `OneInstruction()`
  - `Run()`
  - `ReadRegister(int num)`
  - `WriteRegister(int num, int value)`
  - `ReadMem(int addr, int size, int* value)`
  - `WriteMem(int addr, int size, int value)`

---

**Interrupt object**

Simulates interrupts by maintaining an event queue together with a simulated clock.

- Supported operations:
  - `Schedule(VoidFunctionPtr handler, int arg, int when, IntType type)`
  
  Schedule a future event to take place at time “when”.

  - `SetLevel(IntStatus level)`
  
  Used to temporarily disable and re-enable interrupts for mutual exclusion purposes. Two levels are supported: `IntOn` and `IntOff`.

  - `OneTick()`
  
  - `CheckIDue(bool advanceClock)`
  
  Examines if some event should be serviced.

  - `Idle()`
  
  “advances” to the clock to the time of the next scheduled event
Console object

Simulates the behavior of a character-oriented CRT device
- Data can be written to the device one character at a time through the PutChar() routine.
- Input characters arrive one-at-a-time. They can be retrieved by GetChar().
- Supported operations:
  - `Console(char *readFile, char *writeFile, VoidFunctionPtr readAvail, VoidFunctionPtr writeDone, int callArg)`: Create a console instance. “readFile” is the Unix file of where the data is to be read from; if NULL, standard input is assumed.
  - `PutChar(char ch)`
  - `GetChar()`

Disk object

Simulates the behavior of a real disk.
- The disk has only a single platter, with multiple tracks (32).
- Each track contains the same number of sectors (32).
- Allow only one pending operation at a time.
- Contain a “track buffer” cache. Immediately after seeking to a new track, the disk starts reading sectors, placing them in the track buffer.
- Supported operations:
  - `Disk(char *name, VoidFunctionPtr callWhenDone, int callArg)`
  - `ReadRequest(int sectorNumber, char *data)`
  - `WriteRequest(int sectorNumber, char *data)`
  - `ComputeLatency(int newSector, bool writing)`

Nachos Threads

- Nachos threads execute and share the same code (the Nachos source code), share the same global variables.
- The Nachos scheduler maintains a ready list, containing all threads that are ready to execute.
- Each thread is in one of four states: READY, RUNNING, BLOCKED, JUST_CREATED.
- Each thread object maintains a context block.
- Thread object supports the following operations:
  - `Thread(char *debugName)`: Create a thread.
  - `Fork(VoidFunctionPtr func, int arg)`: Let a thread execute a function.
  - `Yield()`: Suspend the calling thread and select a new one for execution.
  - `Sleep()`: Suspend the current thread, change its state to BLOCKED, and remove it from the ready list.
  - `Finish()`

Thread switching

- Switching involves suspending the current thread, saving its state, and then restoring the state of the new thread.
- Following code involved in execution: the old code, the new code, and the code that performs switching.
- Routine `Switch(oldThread, newThread)`: 
  - Save all registers in oldThread's context block.
  - Save the program address to be used when the old thread is resumed.
  - Load new values into the registers from the context block of the new thread.
  - Once the saved PC of the new thread is loaded, `Switch()` is no longer executing.
Scheduler object for thread scheduling

- A scheduler decides which thread to run next by scanning the ready list.
- The scheduler is invoked whenever the current thread gives up the CPU.
- The current Nachos scheduling policy is round-robin: new threads are appended to the end of the ready list, and the scheduler selects the front of the list.
- The Scheduler object has the following operations:
  - ReadyToRun(Thread *thread).
    Make thread ready to run and place it on the ready list.
  - Thread *FindNextToRun()
  - Run(Thread *nextThread)

When Nachos begins executing

After you type “nachos” under threads subdirectory:
- It is executing as a single Unix process.
- The main() calls initialize() to start up interrupt handling, create a scheduler for managing the ready queue.
- Then main() calls ThreadTest() (to be explained for Project 1).
- By default, Nachos executes a single thread. Nachos turns this single UNIX process into a single Nachos thread (executing main()).
- If other threads have been created and continue to exist, the Unix process continues executing Nachos until all threads have terminated.

The Semaphore object

for thread synchronization

- Disable and re-enable interrupts to achieve mutual exclusion (e.g., by calling Interrupt::SetLevel()).
- Operations for a Semaphore object:
  - Semaphore(char* debugName, int initialValue)
  - P(): Decrement the semaphore’s count, blocking the caller if the count is zero.
  - V(): Increment the semaphore’s count, releasing one thread if any are blocked waiting on the count.

Project 1: threads & synchronization

- Work under threads subdirectory.
- Modify ThreadTest() to do simple threads programming (spawning multiple threads).
- Implement locks and condition variables (missing from the file synch.cc).
- Design a few applications using the implemented synchronization primitives.

Implemented synchronization primitives will be used in Projects 2 and 3.
Project 1: Files involved

Key files for your programming.
- main.cc, threadtest.cc – a simple test of our thread routines.
- thread.h thread.cc – Nachos thread data structure and operations.
- scheduler.h scheduler.cc – The thread ready list.
- synch.h synch.cc – synchronization routines.

Other related files.
- synclist.h, synclist.cc – synchronized access to lists using locks/conditions (useful examples for your programming).
- list.h list.cc – generic list management.
- system.h, system.cc – Nachos startup/shutdown routines.
- utility.h utility.cc – some useful definitions and debugging routines.
- interrupt.h interrupt.cc – manage interrupts.

Running a user program

- Source code: under userprog subdirectory.
- The current Nachos can run a single MIPS binary (Noff format) e.g. type “nachos -x ..//test/halt”.
- A user program must be compiled using a cross-platform compiler that generates MIPS code.
- A Noff-format file contains (bin/off.h)
  1. the Noff header, describes the contents of the rest of the file
  2. executable code segment
  3. initialized data segment
  4. uninitialized data segment.
- Each segment has the following information:
  1. virtualAddr: virtual address that segment begins at.
  2. inFileAddr: Pointer within the Noff file where that section actually begins.
  3. The size (in bytes) of that segment.

User process for executing a program

- A Nachos thread is extended as a process to execute a program.
- Each process has its own address space consisting of
  1. Executable code.
  2. Stack space for local variables.
  3. Heap space for global variables and dynamically allocated memory (BSS/DATA).
- Instructions are executed sequentially within the process.
- A process owns some other objects, such as open file descriptors.
**User process creation**

Currently only execute a single user program.

- Create an address space.
- Zero out all of physical memory.
- Read the binary into physical memory and initialize data segment.
- Initialize the translation tables to do a one-to-one mapping between virtual and physical addresses.
- Zero all registers, setting PCReg and NextPCReg to 0 and 4 respectively.
- Set the stackpointer to the largest virtual address of the process (stack grows downward).

To run multiple processes (Project 2), Nachos should

- provide the physical memory management;
- set up a proper address translation table;
- save/restore address-space related state during process switching (AddrSpace::SaveUserState() and AddrSpace::RestoreUserState() are called).

---

**System Calls & Exception Handling**

User programs invoke system calls by executing the MIPS “syscall” instruction

- “syscall” generates a hardware trap into the Nachos kernel.
- A trap is implemented by invoking RaiseException() with arguments indicating the trap cause.
- RaiseException() calls ExceptionHandler() to take care of the specific problem.
- The system call number is stored in Register 2 and the return address is in Register 31.

Assembly code for Halt():

```
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j $31
.end Halt
```

---

**Address translation with linear page tables**

- A virtual address is split into page number and page offset components
- `machine \rightarrow pageTable` points to the page table to be used.
- `machine \rightarrow pageTableSize` – the actual size of the page table.
- Process switching requires to set this pointer properly for each user.
- Physical page 0 starts at `machine \rightarrow mainMemory`.
- Each page has 128-bytes.
- The actual number of physical pages is NumPhysPages (32).

---

**When you type “nachos -x halt”**

- The main thread starts by running function StartProcess() in file progtest.cc. This thread is used to run halt binary.
- StartProcess() allocates a new address space and loads the halt binary.
- It also initializes registers and sets up the page table.
- Call Machine::Run() to execute the halt binary using the MIPS emulator.
- The halt binary invokes the system call Halt(), which causes a trap back to the Nachos kernel via functions RaiseException() and ExceptionHandler().
- The exception handler determines that a Halt() system call was requested from user mode, and it halts Nachos.
Project 2: Multiprogramming & System Calls

- Modify source code under userprog subdirectory.
- The crossplatform compiler is ~cs170/destation-ultrix/bin/gcc.
- This compiler on x86 machines produces a.out with the coff format. Use utility coff2off to convert it as Noff.
- Check the makefile under test subdirectory on how to use gcc and coff2off.

System calls to be implemented:
- Multiprogramming: Fork(), Yield(), Exit(), Exec() and Join().
- File and console I/O: Creat(), Open(), Read(), Write(), and Close().

Nachos File System

Two versions (check openfile.h and filesys.h):
1. A “stub” version is simply a front-end to the Unix filesystem (used in our projects).
2. Implemented on top of a raw disk.

Function layers (from top to down):
- The FileSystem object: directory management (create, delete, open and close files).
- The OpenFile object: access an individual file (file seek, read, and write).
- The SynchDisk object: synchronized access to a Disk, blocking callers until a requested operation actually completes. Support safe disk operations of concurrent threads.
- The Disk object: a crude interface for initiating I/O operations for individual sectors.

Project 2: Files involved

Key files for your programming:
- progtst.c - test routines to run user code.
- addrspace.h addrspac.c - create an address space and load the program from disk.
- syscall.h - the system call interface.
- exception.c - the handler for system calls and other user-level exceptions such as page faults.
- filesys.h, openfile.h console.h - interface to the Nachos file system and console.

Other related files:
- bitmap.h bitmap.c - manipulate bitmaps (useful for keeping track of physical page frames).
- translate.h, translate.c - translation tables.
- machine.h, machine.c - emulates main memory, processor, etc.
- mipsim.c - emulates MIPS R2/3000 instructions.
- console.c - emulates a terminal using UNIX files.

The FileSystem object

Supported operations:
- FileSystem(bool format).
  It is called once at “boot” time. For the stub version, this does nothing; the Unix file system is used instead.
- Create(char *name, int initialSize).
  Create a Nachos disk file. “initialSize” is ignored in the stub version; Unix doesn’t require it.
- OpenFile *Open(char *name).
  Assume both read and write access. In the stub version, Open simply opens the Unix file for read/write access.
The OpenFile object

Manages the details of accessing the contents of a specific file.

- OpenFile(int sector)
  Create an OpenFile object which contains a file control info (used in FileSystem::Open). A file internally is identified by a section number, not a name string.
- ReadAt(char *buffer, int numBytes, int FilePosition)
- Read(char *buffer, int numBytes)
- WriteAt(char *buffer, int numBytes, int position)
- Write(char *buffer, int numBytes)
- Length(). Return the file size.

Project 3: Virtual Memory

- Work on vm subdirectory.
- Create/manage a backing store (a file called SWAP using the OpenFile class).
- Implement a page fault handler and a page replacement policy.
- Test under various conditions:
  - One process with an address space larger than physical memory.
  - Concurrent processes with combined address space larger than physical memory.