Nachos

- Nachos is an instructional operating system developed at UC Berkeley
- Nachos consists of two main parts:
  - Operating system
    - This is the part of the code that you will study and modify
    - This code is in the threads, userprog, and network directories
    - We will not study user programs, so you can ignore files in the userprog directory
  - Machine emulator — simulates a (slightly old) MIPS CPU, registers, memory, timer (clock), console, disk drive, and network
    - You will study this code, but will not be allowed to modify it
    - This code is in the machine directory
- The OS and machine emulator run together as a single UNIX process

Preparing for the First Project

- Copy the files and compile Nachos
  - See “Getting Started” (online)
    - Threads version, then network version
- Start reading:
  - Read Nachos “Overview paper” (online)
  - Read Section 2 “Nachos Machine” and Section 3 “Nachos Threads” in Narten’s “A Road Map Through Nachos” (online)
  - Read about threads, synchronization, interrupts, and networking in Kalra’s “Salsa — An OS Tutorial” (online)
  - Start looking at the code in the threads, machine and network directories
  - Road Map plus printouts of all code are available in the MCS office for $4.50
- If you are not familiar with C++ or the gdb debugger, see the class web page

Preparing for the First Project (cont.)

- Compiling the code
  - Nachos source code is available in ~walker/pub
  - Read ~walker/pub/README
  - Decide where you want to work, so you can copy files from the appropriate directory into your account
    - ~walker/pub/nachos-3.4-hp
      - For HP workstations (aegis, intrepid)
      - Recommended
    - ~walker/pub/nachos-3.4-sparc
      - For Sun workstations (nimitz)
    - ~walker/pub/nachos-3.4-orig
      - The original, unmodified version
  - Read “Project 1 — Getting Started” on the class web page to find out how to copy the necessary files to your account, and compile an executable copy of Nachos into the threads directory

Nachos — The Emulated Machine

- Code is in the machine directory
  - machine.h, machine.cc — emulates the part of the machine that executes user programs: main memory, processor registers, etc.
  - mipssim.cc — emulates the integer instruction set of a MIPS R2/3000 CPU.
  - interrupt.h, interrupt.cc — manages enabling and disabling interrupts as part of the machine emulation.
  - timer.h, timer.cc — emulates a clock that periodically causes an interrupt to occur.
  - stats.h — collects interesting statistics.
Nachos - The Operating System

For now, we will mostly be concerned with code in the threads directory.

main.cc, threadtest.cc — a simple test of the thread routines.

system.h, system.cc — Nachos startup/shutdown routines.

thread.h, thread.cc — thread data structures and thread operations such as thread fork, thread sleep and thread finish.

scheduler.h, scheduler.cc — manages the list of threads that are ready to run.

list.h, list.cc — generic list management.

utility.h, utility.cc — some useful definitions and debugging routines.

Nachos Threads

As distributed, Nachos does not support multiple processes, only threads:

- All threads share / execute the same code (the Nachos source code)
- All threads share the same global variables (have to worry about synch.)

Threads can be in one of 4 states:

- JUST_CREATED — exists, has not stack, not ready yet
- READY — on the ready list, ready to run
- RUNNING — currently running (variable currentThread points to currently running thread)
- BLOCKED — waiting on some external event, probably should be on some event waiting queue

Scheduling in Nachos

The Nachos scheduler is non-preemptive FCFS — chooses next process when:

- Current thread calls Thread::Sleep() (to block (wait) on some event)
- Current thread calls Thread::Yield() to explicitly yield the CPU

main() (in threads/main.cc) calls Initialize() (in threads/system.cc)

- which starts scheduler, an instance of class Scheduler (defined in threads/scheduler.h and scheduler.cc)

Interesting functions:

- Mechanics of running a thread:
  - Scheduler::ReadyToRun() — puts a thread at the tail of the ready queue
  - Scheduler::FindNextToRun() — returns thread at the head of the ready queue
  - Scheduler::Run() — switches to thread

Scheduling in Nachos (cont.)

Scheduler::Scheduler()
{
    readyList = new List;
}

void
Scheduler::ReadyToRun(Thread *thread)
{
    DEBUG("t",
        "Putting thread %s on ready list.
        thread->getName());
        thread->setStatus(READY);
        readyList->Append((void *)thread);
    }

Thread *
Scheduler::FindNextToRun()
{
    return (Thread *)readyList->Remove();
}
Scheduling in Nachos (cont.)

```cpp
void
Scheduler::Run (Thread *nextThread)
{
    Thread *oldThread = currentThread;
    oldThread->CheckOverflow();
    currentThread = nextThread;
    currentThread->setStatus(RUNNING);
    DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"",
          oldThread->getName(),
          nextThread->getName());
    SWITCH(oldThread, nextThread);
    DEBUG('t', "Now in thread \"%s\"
          \n          currentThread->getName());
    if (threadToBeDestroyed != NULL) {
        delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }
}
```

Working with a Non-Preemptive Scheduler

- The Nachos scheduler is non-preemptive
  FCFS — chooses next process when:
  - Current thread calls Thread::Sleep( ) (to block (wait) on some event)
  - Current thread calls Thread::Yield( ) to explicitly yield the CPU

- Some interesting functions:
  - Thread::Fork( ) — create a new thread to run a specified function with a single argument, and put it on the ready queue
  - Thread::Yield( ) — if there are other threads waiting to run, suspend this thread and run another
  - Thread::Sleep( ) — this thread is waiting on some event, so suspend it, and hope someone else wakes it up later
  - Thread::Finish( ) — terminate the currently running thread

Manipulating Threads in Nachos

```cpp
void
Thread::Fork(VoidFunctionPtr func, int arg)
{
    DEBUG('t', "Forking thread \"%s\" with
          func = 0x%x, arg = %d\n",
          name, (int) func, arg);
    StackAllocate(func, arg);
    IntStatus oldLevel = interrupt->
        SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

```cpp
void
Thread::Yield()
{
    Thread *nextThread;
    IntStatus oldLevel = interrupt->
        SetLevel(IntOff);
    ASSERT(this == currentThread);
    DEBUG('t', "Yielding thread \"%s\"
          \n          currentThread->getName());
    nextThread = scheduler->
        FindNextToRun();
    if (nextThread != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
```
Manipulating Threads in Nachos (cont.)

```cpp
void Thread::Sleep ()
{
    Thread *nextThread;

    ASSERT(this == currentThread);
    ASSERT(interrupt->getLevel() == IntOff);
    DEBUG('t', "Sleeping thread \"%s\"\n",
         getName());

    status = BLOCKED;
    while ((nextThread = scheduler->
            FindNextToRun()) == NULL)
        interrupt->Idle();

    scheduler->Run(nextThread);
}
```

Semaphores in Nachos

- The class Semaphore is defined in `threads/synch.h` and `synch.cc`
  - The classes Lock and Condition are also defined, but their member functions are empty (implementation left as exercise)
- Interesting functions:
  - Semaphores:
    - Semaphore::Semaphore( ) — creates a semaphore with specified name & value
    - Semaphore::P( ) — semaphore wait
    - Semaphore::V( ) — semaphore signal
  - Locks:
    - Lock::Acquire( )
    - Lock::Release( )
  - Condition variables:
    - Condition::Wait( )
    - Condition::Signal( )

Networking in Nachos

- Low-level emulation of the physical network is defined in `machine/network.h` and `network.cc`
  - Provides ordered, unreliable, fixed-size packet delivery to other Nachos machines
  - Packets can be dropped (user-controllable), but are never corrupted
- High-level protocols for communication between multiple Nachos machines are defined in `network/post.h` and `post.cc`
  - An instance of class PostOffice manages a set of MailBoxes for each machine
    - PostOffice::Send( ) sends a message to a specific machine and mailbox
    - PostOffice::Receive( ) retrieves a message, or waits if none is available
  - Could provide reliable delivery of arbitrary-size messages, but currently does **not** (see Spring’97 AOS Project 1)