# Lecture 6 “Nachos”

- nachos overview
- directory structure
- nachos emulated machine
- nachos OS
- nachos scheduler
- nachos threads

## Nachos overview

- 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 filesys directories
    - We will not study networking, so the network directory will not be used
  - 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
- machine emulator is used for user programs only OS is executed on the host machine directly
- The OS and machine emulator run together as a single UNIX process

## Nachos distribution structure

Most of the subdirectories of `code` directory of Nachos distribution contains a separate independent source code for nachos with different features. Nachos is compiled When Nachos is compiled, an executable called `nachos` is produced in every subdirectory with corresponding features enabled.

Important directories:

- `threads` - implements threads interface: scheduling, context switch, synchronization
- `machine` - implements i/o, interrupts, address translation, console device, disk, etc. no binary is produced here.
- `filesys -` Nachos filesystem
- `userprog -` execution of user programs
- `vm` - virtual memory

## 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
- `switch.h, swtch.h -` machine dependent context switch routines in assembly

## 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 execution statistic

## 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 even, probably should be on some event waiting queue

## Nachos 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
- main( ) (in threads/main.cc)<br>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

## Scheduler’s code

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

void
Scheduler::ReadyToRun (Thread *thread)
{
  DEBUG('t', "Putting thread %s on ready list.\n",
  thread->getName());
  thread->setStatus(READY);
  readyList->Append((void *)thread);
}

Thread *
Scheduler::FindNextToRun ( )
{
   return (Thread *)readyList->Remove();
}
```

## Scheduler’s code (cont.)

```
void
Scheduler::Run (Thread *nextThread)
{
  Thread *oldThread = currentThread;

  oldThread->CheckOverflow();
  currentThread = nextThread;
  currentThread->setStatus(RUNNING);

  DEBUG('t', "Switching from thread \"%s\" to thread
  \"%s\"\n",
           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 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: `fork()`

```
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);
}
```

example:

```
Thread *t = new Thread("forked thread");
t->Fork(SimpleThread, 1)
```

## Manipulating threads: `yield()`

```
void Thread::Yield ()
{
  Thread *nextThread;

  IntStatus oldLevel = interrupt->SetLevel(IntOff);

  ASSERT(this == currentThread);
  DEBUG('t', "Yielding thread \"%s\"\n", getName());

  nextThread = scheduler->FindNextToRun();
  if (nextThread != NULL) {
           scheduler->ReadyToRun(this);
           scheduler->Run(nextThread);
  }
  (void) interrupt->SetLevel(oldLevel);
}
```

example:

```
currentThread->Yield();
```

# Manipulating threads: `sleep()`

```
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);
}
```

example:

```
currentThread->sleep();
```