Race Conditions Defined

1. Every data structure defines *invariant* conditions.
   - defines the space of possible *legal states* of the structure
   - defines what it means for the structure to be “well-formed”

2. Operations depend on and preserve the invariants.
   - The invariant must hold when the operation begins.
   - The operation may temporarily violate the invariant.
   - The operation restores the invariant before it completes.

3. Arbitrarily interleaved operations violate invariants.
   - Rudely interrupted operations leave a mess behind for others.

4. Therefore we must constrain the set of possible schedules.

Threads in Nachos
**Administrative**

Install and build Nachos if you haven’t already.

http://www.cs.duke.edu/~anderson/110/building.html

“I need more space to build Nachos.”

Space will be provided soon.

Until then, only build in the *threads* directory (needs 1.2 MB).

**Goodies:**

http://www.cs.duke.edu/~anderson/110/globalset.html

http://www.cs.duke.edu/~anderson/110/nachos_code

**Today’s slides:**

http://www.cs.duke.edu/~anderson/110/nachosthreads.html

---

**Operating Systems: The Big Picture**

The operating system (OS) is the interface between the user and the hardware.

An OS implements a sort of *virtual machine* that is easier to program than the raw hardware.
Nachos is designed to look, feel, and crash like a “real” OS. Both the Nachos “OS” and test programs run together as an ordinary process on an ordinary Unix system (Solaris).

Nachos runs real user programs on a simulated machine. MIPS simulator in Nachos executes real user programs. The real OS is treated as part of the underlying hardware.
Nachos Projects

- Labs 1-3: concurrency and synchronization
  - race conditions with processes and threads
  - implementing/using synchronization for safe concurrent code
- Lab 4: protected kernel with multiprogramming
  - OS kernel with system calls, memory allocation, virtual address translation, protection
- Lab 5: I/O and inter-process communication
- Labs 6-7: virtual memory
  - page faults and demand loading
  - page replacement and page cache management

Overview of Nachos Labs 1-3

In the thread assignments, you build and test the kernel’s internal primitives for processes and synchronization.

- Think of your program as a “real” kernel doing “real basic” things.
  - boot and initialize
    - run main(), parse arguments, initialize machine state
  - run a few tests
    - create processes (threads) and execute kernel code in their contexts
  - shut down
- ...runs native, directly on the host machine.
- Kernel only; no user programs (until Lab 4)
The Nachos Thread Library

The Nachos library implements concurrent threads.

- no special support needed from the kernel (use any Unix)
- thread creation and context switch are fast (no syscall)
- defines its own thread model and scheduling policies
- library threads are sometimes called coroutines

```
while(1) {
    t = get next ready thread;
    scheduler->Run(t);
}
```

A Nachos Thread

```
t = new Thread(name);
t->Fork(MyFunc, arg);
currentThread->Yield();
currentThread->Sleep();
```
Blocking or Sleeping

- An executing thread may request some resource or action that causes it to block or sleep awaiting some event.
  - passage of a specific amount of time (a pause request)
  - completion of I/O to a slow device (e.g., keyboard or disk)
  - release of some needed resource (e.g., memory)
  - In Nachos, threads block by calling `Thread::Sleep`.

- A sleeping thread cannot run until the event occurs.

- The blocked thread is awakened when the event occurs.
  - E.g., `Wakeup` or Nachos `Scheduler::ReadyToRun(Thread* t)`

- In an OS, processes may sleep while executing in the kernel to handle a system call or fault.

Nachos Thread States and Transitions

- States:
  - running
  - ready
  - blocked

- Transitions:
  - `Thread::Sleep` (voluntary)
  - `Scheduler::Run`
  - `Scheduler::ReadyToRun`
  - `Thread::Yield` (voluntary or involuntary)
The Nachos Scheduler

The core of Nachos is the Scheduler class:

- one global shared scheduler object
- pool of ready threads (the ready list)

```c
new = scheduler->FindNextToRun(); /* get next ready thread */
scheduler->Run(new); /* run it */
```

Run calls `SWITCH(currentThread, new)` to suspend current thread and pass control to new thread.

---

A Nachos Context Switch

```c
/*
 * Save context of the calling thread (old), restore registers of
 * the next thread to run (new), and return in context of new.
 */
switch/MIPS (old, new) {
old->stackTop = SP;
save RA in old->MachineState[PC];
save callee registers in old->MachineState;

restore callee registers from new->MachineState
RA = new->MachineState[PC];
SP = new->stackTop;

return (to RA).
}
```
Questions about Nachos Context Switches

- Can I trace the stack of a thread that has switched out and is not active? What will I see?  
  (a thread in the READY or BLOCKED state)
- What happens on the first switch into a newly forked thread?  
  \texttt{Thread::Fork} (actually \texttt{StackAllocate}) sets up for first switch.
- What happens when the thread’s main procedure returns?
- When do we delete a dying thread’s stack?
- How does Nachos know what the current thread is?

Threads and Data

Races can occur when threads operate on shared data.

When is data shared?

- local variables are \textit{private}  
  addressed off the stack, e.g., as @\texttt{(SP+offset)} 
  not shared, since each thread has its own stack pointer (SP) 
  never export (pass/share/store) a pointer to a local variable
- global variables and static objects are \textit{shared}  
  addressed with absolute addresses
- dynamic objects or other heap data are \textit{shared}  
  allocated from global storage with \texttt{new/delete}, or \texttt{malloc/free}
Debugging with Threads

Lab 1: demonstrate races with the Nachos List class.

- Some will result in a crash; know how to analyze them.
  - `gdb nachos` [or `ddd`]
    - `(gdb) run program_arguments`
      - Program received signal SIGSEGV, Segmentation Fault.
      - 0x10954 in `function_name(function_args)` at `file.c:line_number`
    - `(gdb) where`
    - Caution: `gdb [ddd]` is not Nachos-thread aware!
      - A context switch will change the stack pointer. Before stepping:
        - `(gdb) break SWITCH`