What is NachOS?

workstation simulator: the simulated workstation includes a MIPS processor, main memory, and a collection of devices including a timer, disk(s), a network interface, and input and output consoles.

operating system: the NachOS operating system manages the simulated workstation, and implements a set of system calls for user programs

user programs: NachOS user programs run on the simulated machine, and use services provided by the NachOS operating system

How does NachOS differ from a "real" OS?

- The NachOS operating system runs *beside* the simulated workstation, not *on* it. This means that the operating system and user programs (which run *on* the simulated workstation) do not share system resources.
- The NachOS operating system controls simulated devices through a set of abstract device interfaces. Instead of executing special I/O instructions or writing codes into device control registers, the operating system calls methods like `Disk::ReadRequest`.

NachOS Thread Facilities

Threads: new threads can be created, and threads can be destroyed. Each new thread executes a kernel procedure that is specified when the thread is created. (`threads/thread.*`)

Scheduling: a round-robin ready queue for threads (`threads/scheduler.*`)

Synchronization: semaphores, locks, and condition variables. These are integrated with the scheduler: blocked threads are kept off of the ready queue, unblocked threads are placed back onto the ready queue. (`threads/synch.*`)

Birth of a NachOS Process

- the creator does:
  1. update the process table
  2. create and initialize an address space (allocate physical memory, set up page table, load user program and data into allocated space)
  3. create a new thread and put it on the ready queue. The new thread executes the kernel function `ProcessStartup`.
- the `ProcessStartup` function does:
  1. Initialize the registers of the simulated machine (page table pointer, program counter, stack pointer, and general registers)
  2. Call `Machine::Run`. This call never returns.

`Machine::Run` starts simulation of the user program. *This corresponds to an exception return in a real system.* The thread is now simulating the execution of user program code. That is, it is in user mode.

System Calls

- to perform a system call, a user program executes a MIPS `syscall` instruction, as usual.
- to simulate the `syscall` instruction, the simulator's `Machine::Run` method (indirectly) calls the kernel's `ExceptionHandler` function. (`userprog/exception.cc`)
- `ExceptionHandler` performs any kernel operations that are needed to implement the system call.
- When `ExceptionHandler` returns, control goes back the `Machine::Run` and the user program simulation picks up from where it left off, just as in real life.

The call to `ExceptionHandler` is the switch from user mode to system mode. The return from `ExceptionHandler` to `Machine::Run` is the switch from system mode back to user mode.

Exceptions and Interrupts

Exceptions: Exceptions are handled in the same way as system calls. If an user program instruction causes an exception, the simulator (`Machine::Run`) calls `ExceptionHandler` so that it can be handled by the kernel

Interrupts:

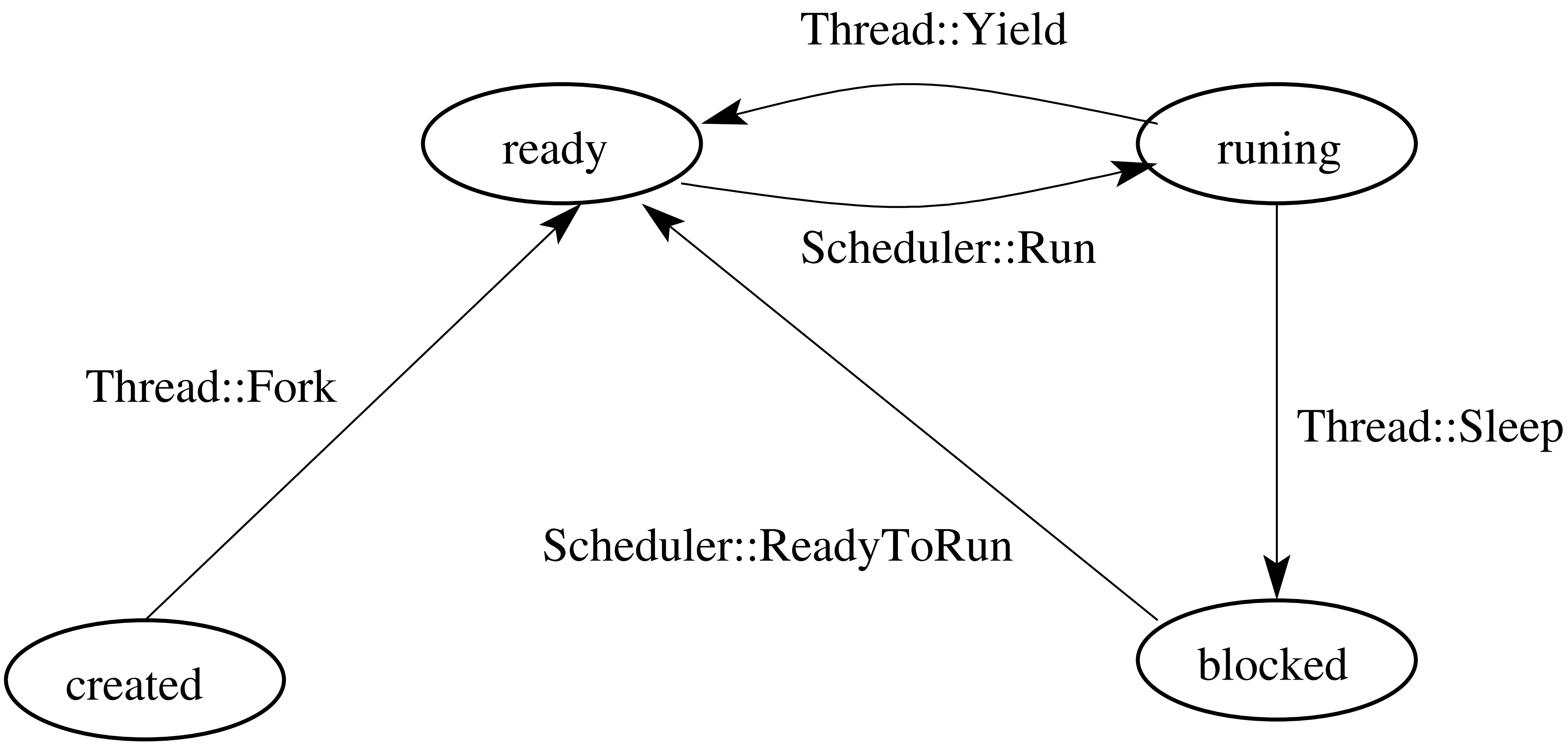
- The simulator keeps track of the simulation time at which device interrupts are supposed to occur.
- After simulating each user instruction, the simulator advances simulation time and determines whether interrupts are pending from any devices.
- If so, the simulator (`Machine::Run`) calls the kernel's handler for that interrupt before executing the next instruction.
- When the kernel's handler returns, the simulation continues executing instructions.

The kernel has a handler function for each type of interrupt (timer, disk, console input, console output, network).

Context Switches in NachOS

- The user context of a thread can be saved in the `thread` object.
- The thread's user context includes the values in the registers of the simulated machine, including the program counter and the stack pointer.
- When switching from one thread to another, the kernel:
  - saves the old thread's user context
  - restores the new thread's user context
- When the new thread returns to user mode, its own user context is in the simulated machine's registers.

NachOS Thread Scheduling

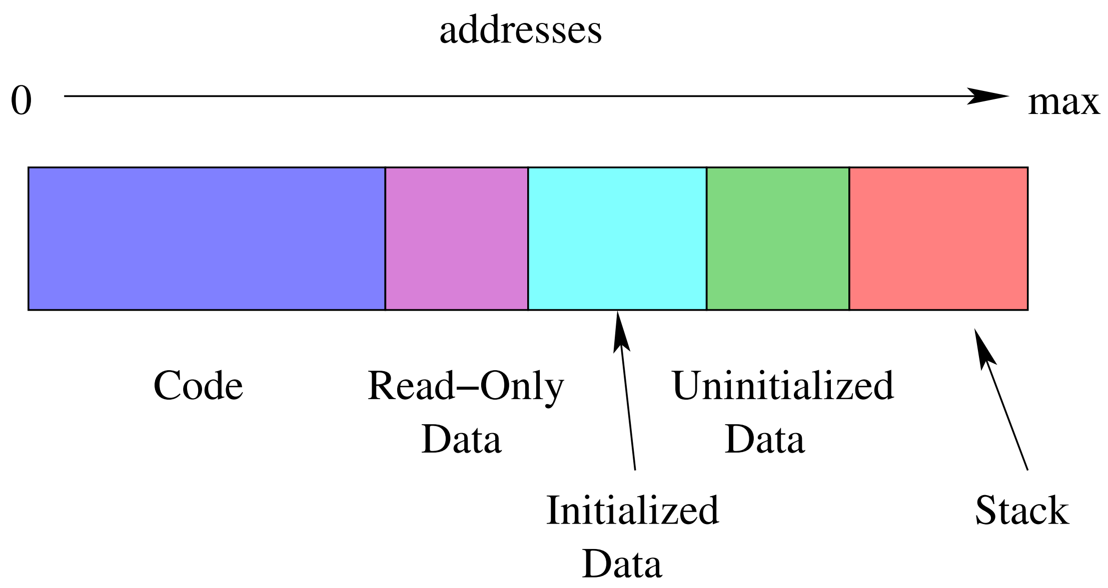


Address Spaces

- One `AddrSpace` object per NachOS process.
- `AddrSpace` maintains the process page table, and provides methods for reading and writing data from virtual addresses.
- NachOS page table entry:

```
class TranslationEntry
  public:
    int virtualPage;  // page number
    int physicalPage; // frame number
    bool valid;       // is this entry valid?
    bool readOnly;    // is page read-only?
    bool use;         // used by replacement alg
    bool dirty;       // used by replacement alg
;
```

Address Space Layout



- Size of each segment except stack is specified in noff file
- Code, read-only data and initialized data segments are initialized from the noff file. Remaining segments are initially zero-filled.
- Segments are page aligned.

NachOS Workstation Devices

- like many real devices, the NachOS workstation's simulated devices are *asynchronous*, which means that they use interrupts to notify the kernel that a requested operation has been completed, or that a new operation is possible. For example:
  - the input console (keyboard) generates an interrupt each time a new input character is available
  - the output console (display) can only output one character at a time. It generates an interrupt when it is ready to accept another character for output.
  - the disk accepts one read/write request at a time. It generates an interrupt when the request has been completed.
- the kernel implements *synchronous* interfaces to each of these devices
  - implemented using the synchronization primitives
  - synchronous interfaces are much easier for the rest of the kernel to use than the asynchronous interfaces. Use them!

Example: Synchronous Input Console

- `SynchConsoleInput::GetChar()` returns one character from the console, and causes the calling thread to *block* (until a character is available) if there are no available input characters.
- Implementation uses a single semaphore:
  - `SynchConsoleInput::GetChar()` does a `P()` before attempting to read a character from the input console.
  - Input console interrupt handler does a `V()`

The NachOS Stub File System

- NachOS has two file system implementations.
  - The real file system has very limited functionality. Files are stored on the workstation's simulated disk.
  - The "stub" file system stores files outside of the simulated machine, in the file system of the machine on which NachOS is running. Magic!
- Until Asst 3, the "stub" file system is used. This is why a file that is created by a NachOS user program appears on the machine on which NachOS is running. This is also why NachOS user programs can be stored in files on host machine, and not on the simulated workstation.
- The "stub" file system is a temporary, unrealistic convenience.