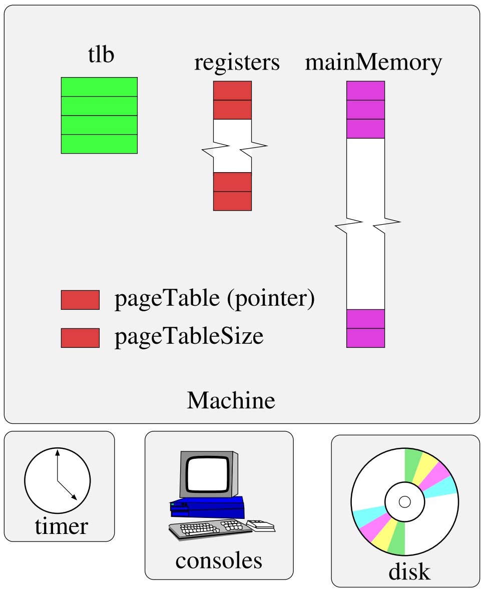
What is NachOS?

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

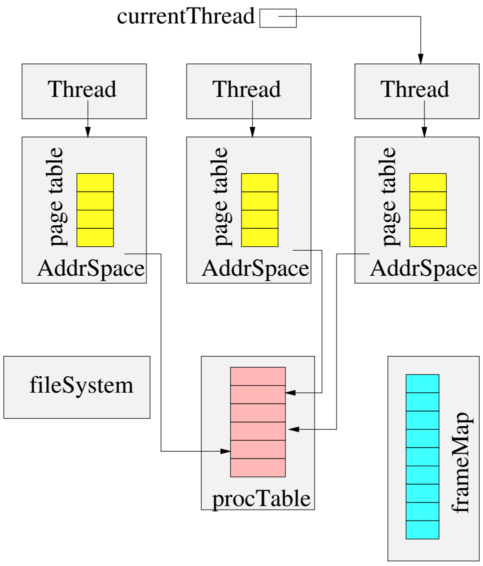
user programs: NachOS user programs run on the simulated machine, and use services provided by the NachOS operating system

The NachOS Machine Simulator



- registers include the program counter and stack pointer
- main memory consists of `NumPhysPages` frames, each of size `PageSize`
- some devices (e.g., network, second disk) are not shown
- simulator uses *either* the TLB or the `pageTable` and `pageTableSize` registers, but not both.

Some Components of The NachOS Kernel



- not all OS components are shown
- each NachOS process has an entry in `ProcTable`, a thread, and an address space
- address spaces are implemented by `AddrSpace` objects, which include a page table
- `frameMap` tracks which main memory frames are in use
- NachOS has two file system implementations

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`.

Review: MIPS Register Usage

```
R0,  $0  =
R2,  $2  =
R4,  $4  =
R5,  $5  =
R6,  $6  =
R7,  $7  =
R29, $29 =
R30, $30 =
R31, $31 =
```


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.

How a System Call Works (example C program)

```
/* call.c
 * Show how a function/syscall is made.
 */
#include "syscall.h"

int
main()
{


  Write("Hello World\n", 12, 1);
}
```


How a System Call Works (compiled program - part 1)

```
  .file   1 "call.c"
  .rdata
  .align  2
$LC0:
  .ascii  "Hello World\n\000"
  .text
  .align  2
  .globl  main
  .ent  main
```

How a System Call Works (compiled program - part 2)

```
main:
  .frame  $fp,24,$31
  .mask  0xc0000000,-4
  .fmask  0x00000000,0
  subu  $sp,$sp,24
  sw  $31,20($sp)
  sw  $fp,16($sp)
  move  $fp,$sp
  jal  __main
  la  $4,$LC0
  li  $5,0x0000000c
  li  $6,0x00000001
  jal  Write
```


How a System Call Works (compiled program - part 3)

```
$L1:
  .set  noreorder
  move  $sp,$fp
  lw  $31,20($sp)
  lw  $fp,16($sp)
  j  $31
  addu  $sp,$sp,24
  .set  reorder
  .end  main
```

How a System Call Works (call stub from start.s)

```
        .globl Write
        .ent    Write
Write:
        addiu $2,$0,SC_Write
        syscall
        j       $31
        .end Write
```


Some System Call Codes (from start.s)

```
/* system call codes */
#define SC_Halt         0
#define SC_Exit         1
#define SC_Exec         2
#define SC_Join         3
#define SC_Create       4
#define SC_Open         5
#define SC_Read         6
#define SC_Write        7
#define SC_Close        8
#define SC_Fork         9
#define SC_Yield        10
```

How a System Call Works (NachOS exception handler)

```
void ExceptionHandler(ExceptionType which)
{
  int type = kernel->machine->ReadRegister(2);

  kernel->currentThread->SaveUserState();
  kernel->currentThread->space->SaveState();

  switch(which) {
  case SyscallException:
    switch(type) {

    case SC_Write:
      vaddr = kernel->machine->ReadRegister(4);
      len = kernel->machine->ReadRegister(5);
      fileID = kernel->machine->ReadRegister(6);
      retval = WriteHandler(fileID, vaddr, len);
      break;
```


Exceptions and Interrupts

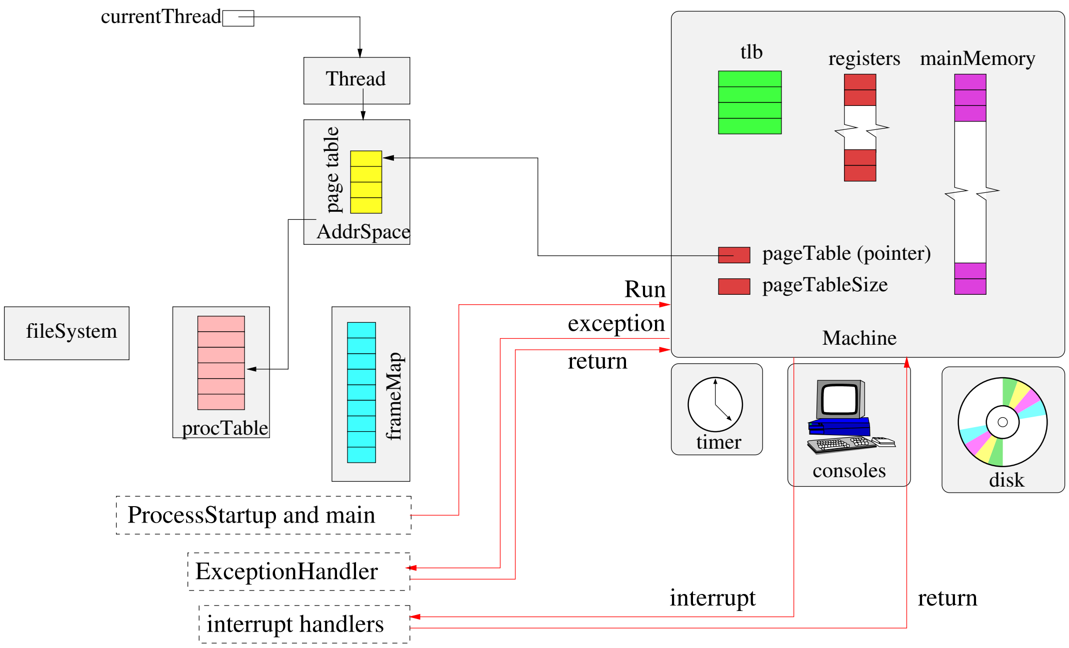
Exceptions: Exceptions are handled in the same way as system calls. If a user program instruction causes an exception, the simulator (`Machine::Run`) calls `ExceptionHandler` so that it can be handled by the kernel

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).

Summary of Machine/Kernel Interactions in NachOS



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.*`)

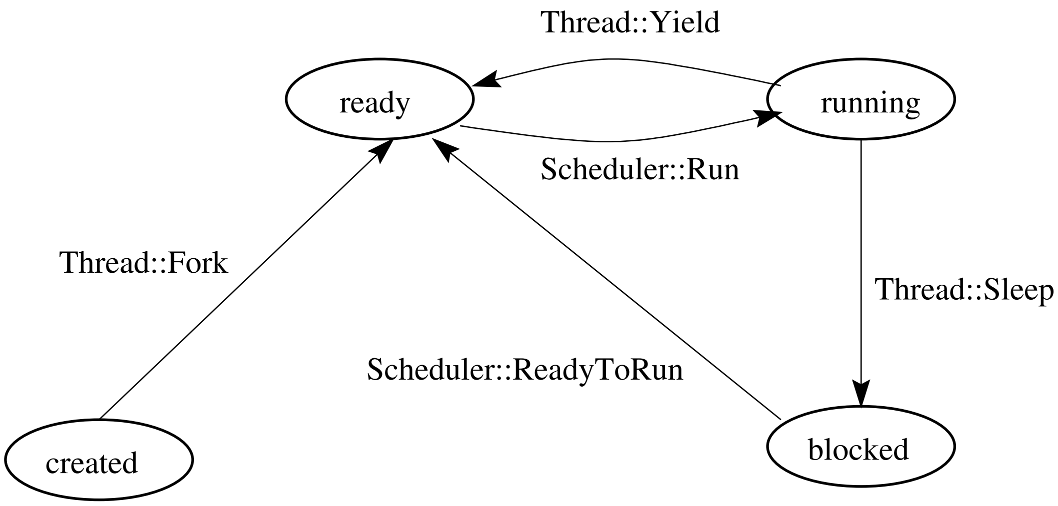
Scheduling: a round-robin ready queue for threads (`threads/scheduler.*`)

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.*`)

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



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.

Starting Up a User Program (__start)

```
/* In Nachos all user programs are linked
 * with start.s, it begins at virtual address 0
 */
  .globl __start
  .ent    __start
__start:
  jal     main
  move    $4,$0
  jal     Exit
  .end __start
```

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

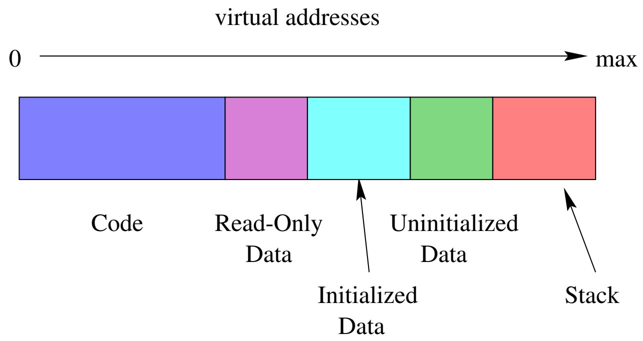
- `SynchConsoleInput::GetChar()` returns one character from the console, and causes the calling thread to *block* (until a character is available) if there are no available input characters.
- Implementation uses a single semaphore:
  - `SynchConsoleInput::GetChar()` does a `P()` before attempting to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

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.

C Code for Segments Example

```
#define N   (5)

unsigned int x = 0xdeadbeef;
int y = 0xbb;
const int blah = 0xff;
int data[N];

struct info {
  int x;
  int y;
};
```


C Code for Segments Example (cont'd)

```
main()
{
  int i;
  int j = 0xaa;
  int k;
  const int l = 0xee;
  char *str = "Hello World\n";

  for (i=0; i<N; i++) {
    data[i] = i;
  }
}
```

Coff2noff Output for Segments

```
Loading 4 sections:
".text"  filepos  52  (0x34)      mempos    0   (0x0)
         size 736 (0x2e0)
".rdata" filepos 788 (0x314)      mempos  768 (0x300)
         size  32 (0x20)
".data"  filepos 820 (0x334)      mempos  896 (0x380)
         size  16 (0x10)
".bss"   filepos  -1 (0xffffffff) mempos 1024 (0x400)
         size  20 (0x14)
<not in file>
```


Some Output from objdump

```
Contents of section .rdata:
0300 ff000000 48656c6c 6f20576f 726c640a
     ....Hello World.
0310 00000000 00000000 00000000 00000000
     ................
Contents of section .data:
0380 efbeadde bb000000 00000000 00000000
     ................
```

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 may seem unrealistic, however, a diskless workstation with network boot uses a similar mechanism.

The NachOS File System: disk.h

```
#define SectorSize      128 // bytes
#define SectorsPerTrack 32
#define NumTracks       64  // per disk
#define NumSectors (SectorsPerTrack * NumTracks)
                  // 32 * 64 = 2048
```

Disk Size = 2048 sectors * 128 bytes = 256 KB

The NachOS File System: filesys.h

```
#define FreeMapSector     0
#define DirectorySector   1

#define FreeMapFileSize (NumSectors / BitsInByte)
                        // 2048 / 8 = 256

#define NumDirEntries     10
#define DirectoryFileSize
   (sizeof(DirectoryEntry) * NumDirEntries)
   // 20 * 10 = 200 (1.5625 sectors)
```


The NachOS File System: filesys.h (cont'd)

```
class FileSystem {
        ...
private:
  // Bit map of free disk blocks,
  // represented as a file
  OpenFile* freeMapFile;

  // "Root" directory -- list of
  // file names, represented as a file
  OpenFile* directoryFile;
};
```

FreeMap file has 2048 entries and occupies 2 sectors (256 bytes), plus one sector for its header. Directory file has 10 entries, which requires 200 bytes (2 sectors), plus one sector for its header.

The NachOS File System: filehdr.h

```
#define NumDirect
  ((SectorSize - 2 * sizeof(int))
   / sizeof(int))
#define MaxFileSize
  (NumDirect * SectorSize)

class FileHeader {
  ...
private:
  int numBytes;
  int numSectors;              // data sectors
  int dataSectors[NumDirect];  // sector numbers
...
```


The NachOS File System: filehdr.h (cont'd)

- `FileHeader` fits in one sector = 128 bytes
- first two fields (`numBytes` and `numSectors`) use 8 bytes
- 120 bytes are left for block pointers
- each block pointer requires 4 bytes, so

$$\texttt{NumDirect} = \frac{128 - 2 * 4}{4} = 30$$

- maximum file size is:

$$MaxFileSize = \texttt{NumDirect} * \texttt{SectorSize} = 30 * 128 = 3840$$

The NachOS File System: directory.h

```
#define FileNameMaxLen 9
// for simplicity, we assume
// file names are <= 9 characters long

class DirectoryEntry {
public:
  bool inUse;
  int sector;
  char name[FileNameMaxLen + 1];
  // Text name for file, with +1 for
  // the trailing '\0'
};
```

4 bytes for `inUse`, 4 bytes for `sector`, 10 bytes for `name`.

File System Command Line Example

```
mobey 1% nachos -f
Machine halting!

Ticks: total 34510, idle 34500,
system 10, user 0
Disk I/O: reads 3, writes 5
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

File System Command Line Example (part 2)

```
mobey 2% nachos -D
Bit map file header:
FileHeader contents.  File size: 128.
File blocks:
2
File contents:
\1f\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```


File System Command Line Example (part 3)

```
Directory file header:
FileHeader contents.  File size: 200.
File blocks:
3 4
File contents:
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
Bitmap set:
0, 1, 2, 3, 4,
Directory contents:
Machine halting!
```

File System Command Line Example (part 4)

```
mobey 3% cat > File1
Hello
mobey 4% cat > File2
World
mobey 5%  nachos -cp File1 File1
Machine halting!

mobey 6% nachos -cp File2 File2
Machine halting!
```


File System Command Line Example (part 5)

```
mobey 7% nachos -D
Bit map file header:
FileHeader contents.  File size: 128.
File blocks:
2
File contents:
\ff\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

File System Command Line Example

```
Directory file header:
FileHeader contents.  File size: 200.
File blocks:
3 4
File contents:
\1\0\0\0\5\0\0\0File1\0\0\0\0\0\0\0\1\0\0\0
\7\0\0\0File2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```


File System Command Line Example (part 6)

```
Bitmap set:
0, 1, 2, 3, 4, 5, 6, 7, 8,
Directory contents:
Name: File1, Sector: 5
FileHeader contents.  File size: 6.
File blocks:
6
File contents:
Hello\a
Name: File2, Sector: 7
FileHeader contents.  File size: 6.
File blocks:
8
File contents:
World\a

Machine halting!
```