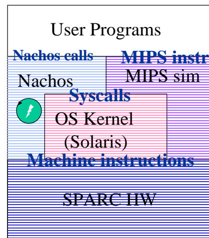
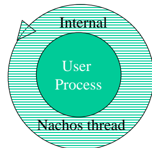
Introducing User Programs into Nachos





Conceptually:
Nachos thread encapsulates user program, remains the schedulable entity

Nachos Systems Call (Process)

userprog/syscall.h

- Spaceid Exec (char *name, int argc, char** argv, int pipectrl) - Creates a user process by
  - creating a new address space,
  - reading the executable file into it, and
  - creating a new internal thread (via Thread::Fork) to run it.
  - To start execution of the child process, the kernel sets up the CPU state for the new process and then calls Machine::Run to start the machine simulator executing the specified program's instructions in the context of the newly created child process.

Nachos Systems Call (Process)

userprog/syscall.h

- Exit (int status) - user process quits with status returned. The kernel handles an Exit system call by
  - destroying the process data structures and thread(s),
  - reclaiming any memory assigned to the process, and
  - arranging to return the exit status value as the result of the Join on this process, if any.
- Join (Spaceid pid) - called by a process (the joiner) to wait for the termination of the process (the joinee) whose SpaceId is given by the pid argument.
  - If the joinee is still active, then Join blocks until the joinee exits. When the joinee has exited, Join returns the joinee's exit status to the joiner.

StartProcess(char *filename)

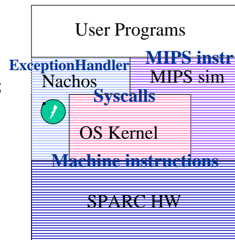
```
{
   OpenFile *executable = fileSystem->Open(filename);
   AddrSpace *space;
   if (executable == NULL) {
      printf("Unable to open file %s\n", filename);
      return;
   }
   space = new AddrSpace(executable);
   currentThread->space = space;
   delete executable;               // close file
   space->InitRegisters();          // set the initial register values
   space->RestoreState();           // load page table register
   machine->Run();                  // jump to the user progam
   ASSERT(FALSE);                   // machine->Run never returns;
                           // the address space exits
                           // by doing the syscall "exit"
}
```

–> Exec

ExceptionHandler(ExceptionType which)

```
{
   int type = machine->ReadRegister(2);

   if ((which == SyscallException) && (type == SC_Halt)) {
      DEBUG('a', "Shutdown, initiated
      by user program.\n");
      interrupt->Halt();
    } else {
      printf("Unexpected user mode
         exception %d %d\n", which, type);
      ASSERT(FALSE);
    }
}
```



Note: system call code must convert user-space addresses to Nachos machine addresses or kernel addresses before they can be dereferenced

AddrSpace::AddrSpace(OpenFile *executable)

```
{ ...
   executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
   if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic) ==
    NOFFMAGIC))   SwapHeader(&noffH);
   ASSERT(noffH.noffMagic == NOFFMAGIC);

// how big is address space?
   size = noffH.code.size + noffH.initData.size + noffH.uninitData.size +
    UserStackSize;     // we need to increase the size to leave room for the stack
   numPages = divRoundUp(size, PageSize);
   size = numPages * PageSize;
   ASSERT(numPages <= NumPhysPages);          // check we're not trying
                              // to run anything too big --

                              // at least until we have virtual memory
```

```
// first, set up the translation
   pageTable = new TranslationEntry[numPages];
   for (i = 0; i < numPages; i++) {
      pageTable[i].virtualPage = i;   // for now, virtual page # = phys page #
      pageTable[i].physicalPage = i;
      pageTable[i].valid = TRUE;
      pageTable[i].use = FALSE;
      pageTable[i].dirty = FALSE;
      pageTable[i].readOnly = FALSE;  // if the code segment was entirely on
                                      // a separate page, we could set its
                                      // pages to be read-only
   }

// zero out the entire address space, to zero the unitialized data segment
// and the stack segment
 bzero(machine->mainMemory, size);
```

```
// then, copy in the code and data segments into memory
   if (noffH.code.size > 0) {
      DEBUG('a', "Initializing code segment, at 0x%x, size %d \n",
                noffH.code.virtualAddr, noffH.code.size);
      executable->ReadAt(&(machine->mainMemory[noffH.code.virtualAddr]),
                noffH.code.size, noffH.code.inFileAddr);
   }
   if (noffH.initData.size > 0) {
      DEBUG('a', "Initializing data segment, at 0x%x, size %d \n",
                noffH.initData.virtualAddr, noffH.initData.size);
      executable->
      ReadAt(&(machine->mainMemory [noffH.initData.virtualAddr]),
                noffH.initData.size, noffH.initData.inFileAddr);
   }

}
```

Non-contiguous VM

Now:

- Need to know which frames are free
- Need to allocate non-contiguously and not based at zero

0:

pageTable

NumPhysPages - 1:

mainMemory

Nachos File System

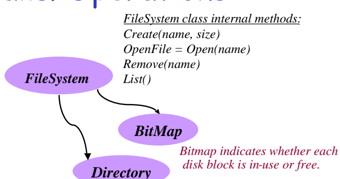
Nachos File Syscalls/Operations

```
Create("zot");

OpenFileId fd;
fd = Open("zot");
Close(fd);

char data[bufsize];
Write(data, count, fd);
Read(data, count, fd);
```

FileSystem class internal methods:
Create(name, size)
OpenFile = Open(name)
Remove(name)
List()



Bitmap indicates whether each disk block is in-use or free.

A single 10-entry directory stores names and disk locations for all currently existing files.

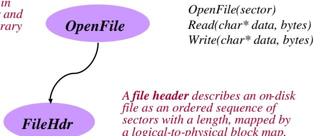
FileSystem data structures reside on-disk, with a *copy* in memory.

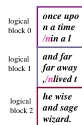
Limitations:
1. small, fixed-size files and directories
2. single disk with a single directory
3. stream files only: no seek syscall
4. file size is specified at creation time
5. no access control, etc.

Representing A File in Nachos

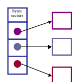
*An **OpenFile** represents a file in active use, with a seek pointer and read/write primitives for arbitrary byte ranges.*

OpenFile(sector)
Read(char data, bytes)*
Write(char data, bytes)*





*A **file header** describes an on-disk file as an ordered sequence of sectors with a length, mapped by a logical-to-physical block map.*



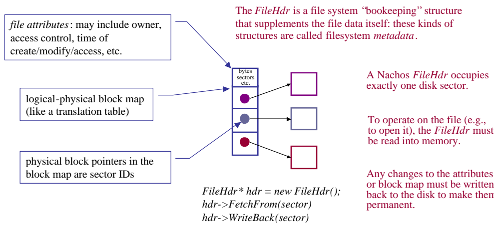
Allocate(...,filesize)
length = FileLength()
sector = ByteToSector(offset)

OpenFile ofd = filesys->Open("tale");*
ofd->Read(data, 10) gives 'once upon '
ofd->Read(data, 10) gives 'a time /in '

File Metadata

On disk, each file is represented by a *FileHdr* structure.

The *FileHdr* object is an in-memory *copy* of this structure.



The *FileHdr* is a file system "bookeeping" structure that supplements the file data itself: these kinds of structures are called filesystem *metadata*.

A Nachos *FileHdr* occupies exactly one disk sector.

To operate on the file (e.g., to open it), the *FileHdr* must be read into memory.

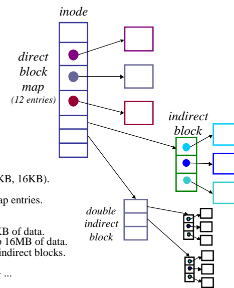
Any changes to the attributes or block map must be written back to the disk to make them permanent.

```
FileHdr* hdr = new FileHdr();
hdr->FetchFrom(sector)
hdr->WriteBack(sector)
```

Representing Large Files

The Nachos *FileHdr* occupies exactly one disk sector, limiting the maximum file size.

sector size = 128 bytes
120 bytes of block map = 30 entries
each entry maps a 128-byte sector
max file size = 3840 bytes



In Unix, the *FileHdr* (called an index-node or *inode*) represents large files using a hierarchical block map.

Each file system block is a clump of sectors (4KB, 8KB, 16KB).
Inodes are 128 bytes, packed into blocks.
Each inode has 68 bytes of attributes and 15 block map entries.

suppose block size = 8KB
12 direct block map entries in the inode can map 96KB of data.
One indirect block (referenced by the inode) can map 16MB of data.
One double indirect block pointer in inode maps 2K indirect blocks.

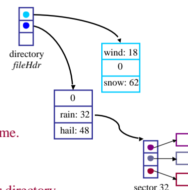
maximum file size is 96KB + 16MB + (2K*16MB) + ...

Nachos Directories

A *directory* is a set of file names, supporting lookup by symbolic name.

Each directory is a file containing a set of mappings from *name->FileHdr*.

Directory(entries)
sector = Find(name)
Add(name, sector)
Remove(name)



In Nachos, each directory entry is a fixed-size slot with space for a *FileNameMaxLen* byte name.

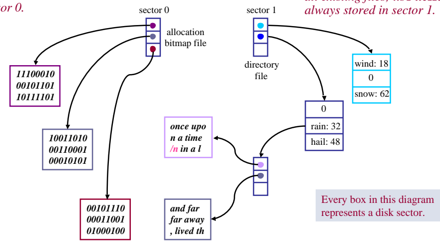
Entries or slots are found by a linear scan.

A directory entry may hold a pointer to another directory, forming a hierarchical name space.

A Nachos Filesystem On Disk

*An allocation bitmap file maintains free/allocated state of each physical block; its **FileHdr** is always stored in sector 0.*

*A directory maintains the name->FileHdr mappings for all existing files; its **FileHdr** is always stored in sector 1.*



Nachos File System Classes