Today's plan

- Signals
- File Systems
  - user interface
  - API
  - implementation

Unix Signals

- any process may set a signal handler for a given signal

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- the argument to the signal handler is the signal number, so the same signal handler can handle multiple signals
- the return value of `signal` is the old signal handler
- the signal handlers default to `SIG_DFL`, which:
  - aborts if the signal is HUP, INT, PIPE, ALARM, TERM, USR1, USR2 (and other non-Posix signals such as POLL)
  - aborts and dumps core if the signal is QUIT, ILL, ABRT, FPE, SEGV, etc
  - returns (ignoring the signal) if the signal is CHLD, URG, or others
- See `signal(7)` for details
- the signal handlers can also be set to SIG_IGN, which ignores the signal (except KILL and STOP cannot be blocked or ignored)
- "Unix" signal handling varies among systems, but typically the signal is blocked during execution of the signal handler

sigaction: another way of handling signals in Unix/Posix

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int,
           siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
}
int sigaction(int signum,
    const  struct sigaction  *act,
    struct sigaction *oldact);

int  sigprocmask(int  how,
     const  sigset_t *set,
     sigset_t *oldset);

int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

- `sigaction` is defined portably across Unix-like systems
- the second parameter (if non-null) is used to define a signal handler, either a sa_handler or a sa_sigaction, but not both
- the `sa_handler` can be set to SIG_DFL or SIG_IGN
- the mask specifies which signals should be blocked during execution of the handler
- the flags can be used to specify
  - whether we are specifying a sigaction or a handler
  - that certain signals should be ignored (e.g. child stop signals)
  - whether the signal handler should be permanently installed, or executed at most once
  - whether system calls should continue across a signal
  - whether further instances of the signal being handled should be deferred while the signal handler is executing
- `sigprocmask` lets us block or unblock the given signals
- `sigpending` returns raised signals that are currently blocked
- `sigsuspend` suspends the process after temporarily installing the given signal mask

Signal Handling and System Calls

- a process making a system call is suspended (in Minix, blocked on receive)
- what is the appropriate behavior of a long-running system call (e.g. read from a pipe or a socket) when its process gets a signal?
- some OSs think the system call should be ended with errno = EINTR, so the caller can (or must) call it again
- some OSs execute the signal handler while the main part of the process is blocked on the system call -- a system call return then has to make sure it is returning to the right place
- Linux and Solaris (at least) give a choice via the `SA_RESTART` flag (in `sigaction`), which if set, means slow system calls are automatically restarted
- Minix completes/aborts the system call, by sending it a message with result code EINTR

- how about other OSs?

Motivation for File Systems

- more storage than is available in a process's memory
- persistence of information across process termination and system crashes
- information sharing among processes
- solution: store information on disk in **named** units called **files**
- files are only deleted by explicit user action
- file system design must decide on
  - file/directory (folder) structure(s)
  - file names (length and structure)
  - what access is supported for files, including what protection is allowed and what operations are supported
  - how these operations are implemented given a persistent (block?) device
- first three are user interface issues, last is an implementation issue

File System Overview

- Memory Management for storage of variable-sized data
  - files are allowed to grow
  - storage is persistent
- Named, usually hierarchical reference to data

File Names

- a name is the way of referring to a persistent object (pointers are a kind of name, but not very user-friendly)
- file names are often limited in length, e.g. 8 or 255 characters
- some operating systems distinguish between upper- and lower-case names (e.g. Unix), others don't (e.g. Windows/ DOS)
- some operating systems support a single extension (following a period) and ascribe significance to it, others support arbitrary extensions (e.g. .tar.gz) and the operating system itself does not care about the extension

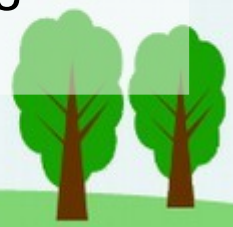
File Contents

- most common these days is the unstructured file consisting of an arbitrary-length sequence of bytes: Unix, Dos/Windows
- files could also be composed of a sequence of fixed-sized records, in which case read, write, and positioning operations access records, not bytes
- each record can also contain a distinct key (perhaps unique, perhaps not), and the file system may structure the file to make access by key efficient, e.g. as a tree or using an index (Indexed Sequential Access Method, ISAM, on IBM, then DEC/VMS)
- operating systems usually also offer versions of content-based (associative) access to files, which creates an index across an entire file system
  - `find`

File Types

- Unix: regular files, directories, device files including character special files and block special files
- ASCII (text) files contain 0 or more arbitrary-length lines, typically ended by a newline ('\n') in Unix, or a carriage-return + newline ("\r\n") in Windows/Dos

  "\r\n" is standard
- binary files usually have internal structure, but that structure is entirely defined by the program that creates them, e.g. MS-Word files or executable files
  - example: archive file has a collection of headers followed by object modules. Each header has the module name, date, owner, protection, and size.
  - zip files have their own structure
- windows file types defined by the extension, implies a default program to be used to access the file: easier to use, but less flexible



Accessing Files

- files are normally accessed sequentially:
  - an editor may read the entire file into memory before editing
  - a daemon reads its entire configuration file when it starts
  - a compiler reads and processes a file linearly, from start to end
  - a web server reads a file one (arbitrary sized) block at a time and sends the block
- some programs, especially data base or similar systems, need random access to files -- position can be specified as part of the `read` or `write` call, or by a separate `seek` operation
- many programs that need to be efficient now `mmap` files into their virtual memory, then let the operating system page them in and out as needed

File Attributes

- size (and maybe maximum size and/or record size, key offset in record)
- creation time, time of last (read, modify) access
- owner (and maybe group)
- permissions (read/write/execute/visible/system)
- has this file been archived? is it temporary? is it locked?

File Operations

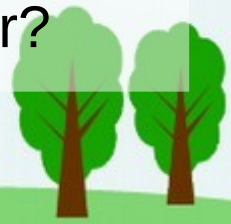
- create, delete (unlink)
- open, close
- read, write, append, seek
- get attributes, set attributes
- rename

Directories (Folders)

- directory (folder) systems are used to structure files
- directories are usually hierarchical when strictly looked at (e.g. by a program), but have some escapes to allow loops (e.g. Unix parent directory ".." and symbolic links, Windows "My computer" and other links)
- often each user has his/her own directory, e.g. /usr/loginname or /home/loginname or /home/mount/loginname
- a directory is usually a special kind of file which may or may not support specific regular file operations, but usually supports operations such as create, delete, open, close, read (one record at a time), rename (a subfile), link (a subfile), unlink (a subfile)
- Unix has hard links (multiple hard links to a file, possibly with different names, are equivalent) and soft links
  - hard links must be to files on the same device
  - when the last hard link to a file is unlinked, the file is removed
    - file attributes include a reference count of links
- a file may have any number of soft links (essentially, the name of the hard link stored in the soft link file), which can work across devices
  - soft links become invalid if the underlying file is removed or renamed

Directory Structures

- very simple systems might only have a single-level (top-level) directory
- most systems allow almost arbitrary depth
- a file name is generally interpreted either relative to a per-process **current (working) directory**, or as an **absolute** path name (beginning with "/" in Unix-like systems)
- some operating systems allow a process to set the root of its file system (**chroot**), which may provide more security if the process is engaging in risky activities (i.e. listening on the network)
- directories are stored in (special) files, usually containing at least the names of the subfiles, pointers to the data, and either the attributes or pointers to the attributes
- which is more efficient: to store all the attributes in the directory, or to have a pointer to the attributes in the directory? Which is simpler?



File System Implementation: which blocks belong to which file?

- contiguous allocation is simple and makes sequential or random access fast, but also requires knowing the file size when allocating and may give fragmentation such that the space is available but not usable (until defragmentation)
- linked list allocation makes sequential access fast, but random access is very slow and blocks end up containing a number n of bytes that is not a power of two.
- linked list allocation in memory keeps a global table with one pointer per block (table is in memory while running, frequently backed up on disk): sequential and random access is fast, block sizes are a power of two, but the memory table may be large (4GB disk with blocks of size 16KB and 4-byte block pointers requires 1MB of RAM)
- **i-node** (index node) allocation keeps a per-file table of blocks, sequentially ordered, and stored on disk until the file is opened. The inode may also store the file attributes, and some of the entries may point to **indirect blocks** which contain more pointers. Fast for sequential and random access, does not use a lot of memory or disk, does not cause fragmentation
- the last two algorithms could be combined, with an inode containing just a pointer to the part of the in-memory table where the file begins

Unix Inodes

- keeping track of which blocks of data belong to which file:
- some versions of Unix used i-nodes with 13 block addresses per inode
- this makes the inode constant size and such that it fits in a block
- if more than 10 data blocks are needed, the 11th pointer is a single indirect block, i.e. points to a block containing addresses of data blocks
- if this is not sufficient, the 12th pointer is a double indirect block, containing the address of a block which contains addresses of blocks of addresses of data blocks
- the last address is a triple-indirect block
- if each indirect block holds the addresses of up to 64 other blocks of 512 bytes each, what is the maximum file size (in bytes) in this system?

Pseudo File Systems

- a device file is really a pseudo file -- it does not correspond to space on disk
- a file system can implement arbitrary access to internal data structures
  - simplest example is the RAM disk
  - a file system can also be used to support access to kernel-internal data structures, e.g. in linux the /proc and sysfs file systems, and in many unix-like systems `/dev/mem` and `/dev/kmem`
- for example, to access the USB on a specific (old) Linux system I had to mount /proc/bus/usb

Implementing Directories

- fixed-size directory entries are simple, but either
  - are too limited in the length of the file name that is supported, or
  - waste too much space
- MS-Dos uses fixed-size (32-byte) directory entries with 8 bytes for the file name and 3 for the extension, 10 unused bytes, and space for size, time, and date
- Unix stores the inode and the filename in variable-sized entries:
  - management of directory entry deletion is more complicated, unless
  - space is defragmented after deletion, which makes deletion from big directories very slow
- Unix directories are stored in files, i.e. inodes point to the blocks on disk holding the directory entries

Management of Disk Space

- similar to (virtual) memory management, e.g block size selection, keeping track of free blocks
- must select a block size
- smaller block sizes waste less space:
  - small files leave most of a block unused
  - even large files typically leave unused about 1/2 of the last block
- larger blocks are faster (for accessing large files or large directories), since a single seek results in reading or writing more data
- median file size reported as 1K (1984) or 12K-15K (1997), so a disk block should not be dramatically larger than this
- Linux tries to allocate blocks nearly sequentially (i.e. as close as possible) to try to improve time to access a large file without taking more space than needed
- the Berkeley Fast File System uses small blocks for small files and large blocks for large files, which is very efficient but more complicated
- free blocks are kept track of as either linked lists or bitmaps