Administrivia

- **Lab 1 (due Thursday) Errata:**
  - Grab new grading script from `$cs240c/updates/lab1-grade.sh`
  - Few differences in new version of Bochs (check email)

Roadmap

- **Today:**
  - Quick review of UNIX (substituting current API)
  - Discuss Multics paper
  - Discuss UNIX as a reaction to multics
- **Next time:**
  - Discuss original UNIX developers' reaction to what UNIX became

PDP-11 Virtual Memory

- **PDP-11 was hardware for first version of UNIX with multiprogramming**
- **64K virtual memory, 8K pages**
- **8 Instruction page translations, 8 Data page translations**
- **Swap 16 machine registers on each context switch**

I/O through the file system

- **Applications "open" files/devices by name**
  - I/O happens through open files
- `int open(char *path, int flags, ...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**
- **Historical note: Needed `creat` to create files**

Error returns

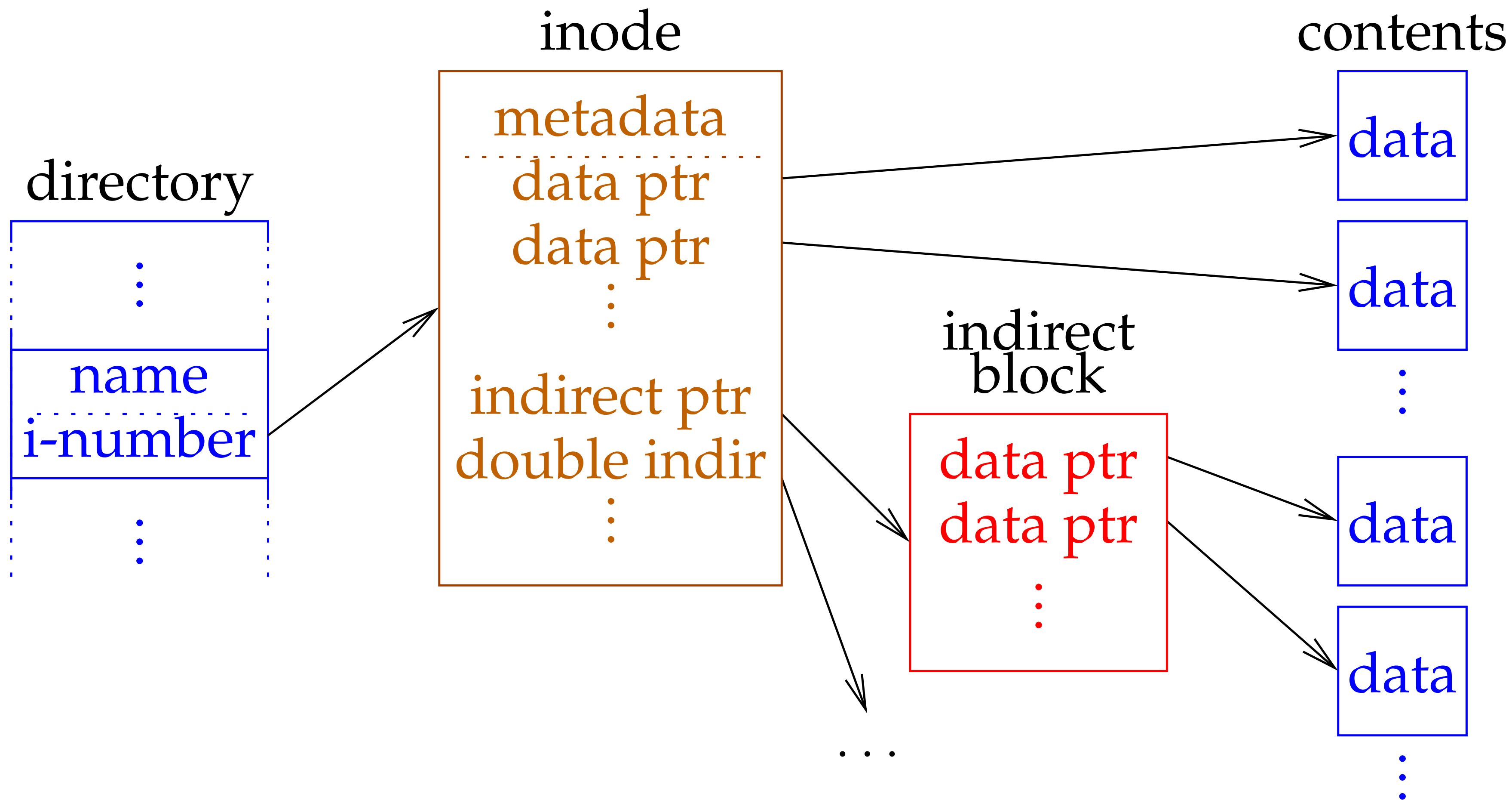
- **What if** `open` **fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int `errno`
- `#include <sys/errno.h>` **for possible values**
  - 2 = `ENOENT` "No such file or directory"
  - 13 = `EACCES` "Permission Denied"
- `perror` **function prints human-readable message**
  - `perror ("initfile");`

    → "`initfile: No such file or directory`"

Original UNIX File System

- **Each FS breaks partition into three regions:**
  - Superblock (parameters of file system, free ptr)
  - *i-list* – table of metadata (i-nodes) for all files
  - File and directory data blocks
  - All composed of 512-bytes blocks
- **Directories very much like ordinary files**
  - Except user code can't directly write them
- **Free blocks kept in a linked list**
- **Today: Many optimizations, but still based on i-nodes**

Inodes



Device nodes

- **File namespace also gives access to some devices**
  - Open what looks like a file, to gain access to device
- **Examples (on my machine, others will vary):**
  - `/dev/null` – reads like EOF, writes like a data sink
  - `/dev/zero` – reads like an infinite stream of 0 bytes
  - `/dev/tty` – reads from or writes to current terminal
  - `/dev/rwd0c` – access raw disk sectors
  - `/dev/rcd0c` – CD-ROM device
  - `/dev/audio` – send audio samples to sound card
  - `/dev/wsmouse` – mouse
  - `/dev/bpf` – lets you snoop packets on the network

Permissions

- **Not every process can open every file**
- **Each process has a set of credentials**
  - User ID (typically 32-bit number, unique per login account)
  - Group ID, group list (32-bit numbers)
- **Files have permissions, too. E.g.,:**
  - (Link count = 1), User ID is 0, group ID 7

```
-r-xr-xr-x  1 0  7  79 Apr 14 10:32 /usr/bin/true
```

- **Three sets of "rwx" bits, for user, group, and other**
  - read/write/execute on normal files
  - on directories, "x" means traverse (cd or access any file)
  - on dirs, must have "w" to create, rename, or delete files

Unix root user

- **Unix user ID 0 is privileged "root" user**
  - Can perform most system calls without access checks
  - E.g., open any file
  - Can change owner of files
  - Can Change its own UID or group list
- **Not to be confused with privileged kernel**
  - Kernel runs with CPU in special "privileged" mode
  - Allows access to special instructions, I/O registers, etc.
  - root-owned processes are still just regular user processes

Example: Unix login process

- **Login process runs with UID 0 (root)**
- **Asks for username and password**
  - Checks against system password file
  - Keeps asking until valid password supplied
- **Once password matches**
  - Look up numeric UID and GIDs in system files
  - Set the GID list
  - Set the UID (this drops privileges)
  - Execute the user's shell

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
  - Guarantee that file contents is stably on disk

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout, printf` in ANSI C)
  - 2 – "standard error" (`stderr, perror` in ANSI C)
  - Normally all three attached to terminal

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new proc. in "parent"
  - Returns 0 in "child"
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error
- **Historical note: before waitpid/wait, more complex messaging primitive used**

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
- `int execvp (char *prog, char **argv);`
  - Search `PATH` for prog
  - Use current environment
- `int execlp (char *prog, char *arg, ...);`
  - List arguments one at a time, finish with `NULL`

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
  - Sets *close on exec* flag if `val` = 1, clears if `val` = 0
  - Makes file descriptor non-inheritable by spawned programs

Example: run prog w. /dev/null stdin

```
if (!(pid = fork ())) {
  int fd = open ("/dev/null", O_RDONLY);
  if (fd > 0) {
    dup2 (fd, 0);
    close (fd);
  }
  execlp ("prog", "prog", "arg1", NULL);
  perror ("prog");
  _exit (1);
}
waitpid (pid, &stat, 0);
printf ("prog exited %snormally\n", stat ? "ab" : "");
```

[note: no error checking here]

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Writes to `fds[1]` will be read on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`, or if blocked
    - Fails with EPIPE

Example multics segments

```
0,0,5  >sl1>hcs_         Gate into ring 0
1,1,5  >sl1>ms_          Gate into ring 1
1,5,5  >sss>ls           Standard system command
4,4,4  >udd>m>vv>fred    Random user's program
```