Unix I/O overview

* I/O allows a computer system to interact with other systems
* CPU is attached to the main memory of the system via some kind of memory bus
* Some devices are connected to the system via a general I/O bus
* A linux file is a sequence of m bytes
* All I/O devices are represented as files
* Even the kernel is represented as a file
* Check l**ecture 16b slide 3** for deets

Opening files

* Opening a file informs the kernel that user is ready to access that file
* Returns a small identifying integer file descriptor
* Each process created by a linux shell begins life with three open files associated with a terminal:
  + 0: standard input (stdin)
  + 1: standard output (stdout)
  + 2: standard error (stderr)

Closing files

* Closing a file informs the kernel that user is finished accessing that file
* Moral: always check return codes, even for seemingly benign functions such as close()

Reading files

* Reading a file copies bytes from the current file position to memory, and then updates file position
* Returns number of bytes read from file fd into buf
  + Return type ssize\_t is signed integer
  + Nbytes < 0 indicates that an error occurred
  + Short counts (nbytes < sizeof(buf)) are possible and are not errors

Writing files

* Writing a file copies bytes from meory to the current file position, and the nupdates current file position
* Returns number of bytes written from buf to file fd
  + Nbytes < 0 indicates that an error occurred
  + As with reads, short counts are possible and are not errors

Simple unix I/O example

* Copying stin to stdout, one bytes at a time: check **lect 16b slide 8**

On short counts

* Short counts can occur in these situations:
  + Encountering (end-of-file) EOF on reads
  + Reading text lines from a terminal
  + Reading and writing network sockets
* Short counts never occur in these situations:
  + Reading from disk files (except for EOF)
  + Writing to disk files
* Best practice is to always allow for short counts

Buffered I/O

* Applications often read/write one character at a time
  + Getc, putc, ungetc
  + Gets, fgets
    - Reads a line of text one char at a time, stopping at a newline
* Implementing as unix I/O calls expensive
  + Read and write require unix kernel calls
  + > 10,000 clock cycles
* Solution: buffered read
  + Use unix read to grab block of bytes
  + User input functions take one byte at a time fom buffer
  + Refill buffer when empty

The Robust I/O (RIO) package

* RIO is a set of wrappers that provide efficient and robust I/O in apps, such as network programs that are subject to short counts
  + Not a built-in package but very convenient
* RIO provides two different kinds of functions
  + Unbuffered input and output of binary data
    - Rio\_readn and rio-WRITEN
  + Buffered input of text lines and binary data
    - Rio\_readlineb and rio\_readnb
    - Buffered RIO routines are thread-safe and can be interlaced arbitrarily on the same descriptor
  + Available on course website: csapp.c and csapp.h

Unbuffered RIO input and output

* Same interface as unix read and write
* Especially useful for transferring data on network sockets
  + Rio\_readn returns short count only if it encounters EOF
    - Only use it when you know how many bytes to read
  + Rio\_writen never returns a short count
  + Calls to rio\_readn and rio)\_writen can be interleaved arbitrarily on the same descriptor

Buffered RIO input functions

* Efficiently read text lines and binary data from a file partially cached in an internal memory buffer
  + Rio\_readlineb reads a text line of up to maxlen bytes from file fd and stores the line in usrbuf
    - Especially useful for reading text lines from network sockets
  + Stopping conditions:
    - Maxlen bytes read, EOF encountered, newline encountered
  + Rio\_readnb reads up to n bytes from file fd
  + Stopping conditions:
    - Maxlen bytes read, EOF encountered
  + Calls to rio\_readlnineb and rio\_readnb can be interleaved arbitrarily on the same descriptor
    - Warning: don’t interleave with calls to rio\_readn

RIO example

* Copying the lines of a text file from standard input to standard output
* **Check slide 16**

File metadata

* Metadata is data about data, in this case file data
* Per-file metadata maintained by kernel
  + Accessed by users with the stat and fstat functions

How the unix kernel represents open files

* Two descriptors referencing two distinct open files
  + Descriptor 1 (stdout) points to terminal
  + Descriptor 4 points to open disk file
  + Check slide 19

File sharing

* Two distinct descriptors sharing the same disk file through two distinct open file table entries
  + Ex: calling open twice with the same filename argument
* How processes share files: fork
  + A child process inherits its parent’s open files
    - Note: situation unchanged by exec functions (use fcntl to change)
  + Before fork call: check slide 21 for diagram
  + After fork call: check slide 22 for diagram
    - Child’s table is same as parent’s, and +1 to each refcnt

I/O redirection

* Question: how does a shell implement I/O redirection?
  + $> ls > foo.txt
* Answer: by calling the dup2 (oldfd, newfd) function
  + Copies (per-process) descriptor table entry oldfd to entry newfd
* **I/O redirection example: check slides 25-26**
  + Step 1: open file to which stdout should be redirected
    - Happens in child executing shell code, before exec
  + Step 2: call dup2 (4, 1)
    - Because fd = 1 (stdout) to refer to disk file pointed at by fd = 4