COSC 3750

Input/Output

Kim Buckner

University of Wyoming

Feb. 03, 2022

Last time

- Gcc
- Make

GDB

GDB

- The GNU debugger.
- This allows you to step through a program a statement at time.
- Allows you to monitor variables .
- Probe memory locations.
- Check pointers.

The GOOD

- When you cannot solve the problem any other way.
- If you get a core dump, can examine it.
- Can trace execution into function calls.
- Can set breakpoints and watchpoints.

The BAD

- Takes experience to use it effectively.
- Can be extremely confusing.
- Documentation is somewhat confusing/sparse.
- Suggest reading the GDB manual and the info document.
- The man page helps but is necessarily brief.

The UGLY

- Until you get some experience, it is very confusing.
- It can be a time sink.
- Many people spend more time with gdb than needed.
- The printf statement is still usually the best debugging tool.

Suggestion

- Take the time to play with it.
- Overall, as good or better than Visual Studio's debugger.
- Do not try to figure it out if you only have 3 hours until the assignment/project is due.
- The user Manual is available at

https://www.gnu.org/software/gdb/documentation/

Brief GDB Tutorial

I/O

Program I/O

- All input and output defaults to file-like operations.
- There are at least 3 ways to handle process I/O:
  - accessing files in the file system,
  - accessing the network, and
  - memory mapping.

Files

- Probably the most familiar.
- There are some concepts we need to understand.
- The first is access type:
  - read,
  - write,
  - read/write,
  - append, and
  - random.

How managed?

- By the O/S of course.
- Linux/UNIX uses an "inode."
- This is a data structure that we can, in part, access.
- It provides status of a file, permissions, etc.
- This is an O/S construct that the user is usually not allowed to modify directly.

Access – read

- This is a sequential operation.
- Managed by maintenance of a "read position indicator."
- This is maintained in a data structure created when the file is opened.
- This data structure is a library construct.
- It contains information and buffers (as needed).

(more . . .)

- Reading from a file is simple:
- Create a process buffer.
- Open the file, then
  - call a function that copies data into that buffer,
  - process the data, and
  - repeat.

(more . . .)

- Data is accessed in-order throughout the file.
- A “marker” is returned when at the end.
- That last is subject to interpretation.

Return values

- Cannot stress this enough.
- Save and check return values!!
- They are often the only way to detect certain conditions
- They are not always enough, but they are a start.
- We are striving for "robust" programs.

Access – write

- Similar to read:
- Open the file then
  - create a process buffer,
  - call a function that copies data from that buffer into the file,
  - repeat,
- Data is written in-order throughout the file.
- This also uses a "position indicator."
- **All** original data is LOST

Access – read/write

- A combination of the other two.
- Reads are sequential, from the beginning of the file.
- Writes are sequential, at the write position indicator.
- Usually a separate position indicator for each.

Access – append

- The original file is intact.
- Writes are sequential, always at the end of the file.
- This is "safe."

Access – random

- This is a matter of management.
- The others are based on how the file is opened.
- For this one, additional functions are used to move the file pointers.
- The particular function depends on how the file was opened.

(more . . .)

- If the file was opened for appending, all writes occur at the end, regardless.
- Otherwise, the reads and writes occur at the file position indicator.
- Moving the position indicator changes where the read or write occurs.

¶

Opening files

- First, you have to have a path.
- File paths are based on the directory in which the process is executing.
- Functionality like “cd” is available.
- Of course, the process has to have permissions.

(more . . .)

- Two sets of functions:
  - one provides buffering (recommended) and
  - the other does not which can be useful in some situations but is usually NOT.
- Buffered functions: fopen(), fread(), fwrite(), fseek().
- Unbuffered functions: open(), read(), write(), lseek(), fcntl(), etc.

File handles

- The unbuffered functions use small integer values to refer to open files.
- The *open()* command returns this integer when the operation is successful.
- The others require this value to access the file.
- This value is sometimes referred to as a "file handle" but more usually called a "file descriptor."

File pointers

- The buffered functions require a pointer to a *FILE* structure.
- This is returned by *fopen()* when the operation is successful.
- This pointer object contains the buffering and other data about the opened file.
- DO NOT modify this object directly.

```
#include<stdio.h>
#include<errno.h>

  FILE *infile;
  infile=fopen("input_text","r");
  if(infile == NULL) {
    perror("input_text");
    return 1; }
```

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>

  int fd;
  fd=open("input_text",O_RDONLY);
  if(fd < 0) {
    perror("input_text");
    return 1;}
```

Automatic files

- Well, not really, but. . . .
- The shell opens 3 files (usually) for each process that it starts.
- These are cleverly called “standard input”, “standard output”, and “standard error.”
- **stdin** is initially connected to the keyboard (tty).

(more . . .)

- **stdout** is initially connected to the "display."
- **stderr**, also called the "diagnostic output," is also connected to the display.
- All of these can be changed by redirection.
- May also be changed inside the process.

Differences

- **stdin** is read-only. Attempts to write will result in nothing happening. Usually NOT an error.
- **stdout** is write-only, attempts to read fail but not an error. It is buffered.
- **stderr** like **stdout** but unbuffered.

Accessing

- **stdin**, **stdout**, and **stderr** are actually identifiers of type *FILE* pointers.
- Do not have to be declared, they are supplied in the library.
- If using *read()* or *write()* you need a file descriptor.

(more . . .)

- Older code supported those names as integers: 0, 1, and 2.
- Portable code (what you write!) uses STDIN_FILENO, STDOUT_FILENO, and STDERR_STDERR. (defined in unistd.h)
- Last, if you open it, close it. Do not rely on the compiler or *exit()* functions!!