Logging and debugging

Stefan D. Bruda

CS 464/564, Fall 2023

What (and when) to log

- The most obvious reason to log stuff: debugging
  - Testing before release/submission
  - Maintenance (a lot of effort goes into that in the real world)
- While debugging, the server is usually attached to a terminal and prints messages to that terminal (remember the `-d` switch)
- Other reasons for logging (during normal operation):
  - load monitoring and resource management
  - security (e.g., log any connection attempt from unauthorized users, or log any suddenly terminated connection)
- Typically, when this kind of logging happens the server is detached and prints those messages to something else than a terminal (e.g., to a file)

Logging, take one: debugging

- The debugging phase is the hardest of them all if the program is complex enough
  - Even more complicated when debugging servers!
  - These beasts are usually multiprocess or multithreaded, so good luck with your favourite debugging GUI!
- Debugging techniques most suitable for servers:
  - Code inspection
  - Verbose output
- Don't forget to save the “working” program in a safe place before debugging
  - If you find yourself barking up the wrong tree (or if you manage to mess up your program in the process), you can then start all over again

Verbose log/output

- In debugging mode, you may want to convince your server to print out messages whenever something interesting (from a debugging point of view!) happens
  - E.g., when a read or write starts, you may want to print the number of reads/writes taking place simultaneously
  - Or you may want to print the command received from the client and the answer that was sent back
- This helps you isolate the problem
- Then, in the piece of code you deemed responsible for your problem,
  - Print out various data to see where it goes bad
  - Use print statements to isolate the error more tightly
  - Do inspect the code; proper indentation does help a lot
  - Keep around the print statements for critical data when you attempt to solve the problem (i.e., when you modify the buggy code)
- Just don't forget to suppress such messages during normal operation!

Verbose output (cont'd)

- Exaggeration is always bad; do control the verbosity of your program
- Sometimes, it is enough just to keep the server attached and printing to the terminal
- When you debug the access control to files, you do not need the messages that refer to, say, the client-server communication
- You may want to have a series of command line switches instead of only one
  - E.g., `-d` to keep the server attached, `-v comm` to produce debugging messages for the communication part, etc

Command-line switches (example)

```
const int DEBUG_COMM = 0; const int DEBUG_FILE = 1; const int DEBUG_DELAY = 2;
int debugs[3] = {0,0,0};

int main (...) {
  extern char *optarg;  extern int optind;
  int copt;
  int detach = 1;  // Detach by default
  while ((copt = getopt (argc,argv,"v:dD")) != -1) {
    switch ((char)copt) {
    case 'd':  detach = 0; break;
    case 'D':  debugs[DEBUG_DELAY] = 1; break;
    case 'v':
      if (strcmp(optarg,"all") == 0)
        debugs[DEBUG_COMM] = debugs[DEBUG_FILE] = 1;
      else if (strcmp(optarg,"comm") == 0)
        debugs[DEBUG_COMM] = 1;
      else if (strcmp(optarg,"file") == 0)
        debugs[DEBUG_FILE] = 1;
    }
  }
  // Options processed, get rid of them:
  argc -= (optind - 1);
  argv += (optind - 1);
}
```

```
int write_excl(...) {
...
  if (debugs[DEBUG_DELAY]) {
   sleep(5);
  } /* DEBUG_DELAY */
...
}
```


How to log (normal operation)

- The easier logging method is by output redirection
- If you open appropriate files on descriptors 1 and 2, logging is a breeze
  - Just print out messages to one of the output streams
  - They will go in the appropriate places since you already set the file descriptors to suit your needs
- There are however major differences between printing to a terminal and printing to a file: the buffer size and the synchronization policy
  - When printing to a terminal, you usually get away without flushing the output
  - Terminal buffers are small
  - Usually, when a newline is received, the output stream gets flushed and thus your message appears on the screen

File buffers

- When you "print" to a file, you have to flush
- Without flushing, there is absolutely no guarantee about the time it takes for your message to actually get written on disk
- If your server crashes, there is a very good chance that your last (or even all) messages are lost
- The reason for such a behaviour is the fundamental difference between the hard disk and the video memory
  - File manipulation usually means transfer of large amount of data
  - The hard disk is a slow device
  - It drains by comparison a large amount of power
  - It does make a lot of sense to wait for a large amount of data to arrive before writing it to disk, so that the OS can optimize the disk access
  - Ergo, file buffers are large (the larger the amount of free RAM, the larger are the buffers), and are not flushed very often
- Conclusion: it is very important to flush after each critical message, such as messages that signal an error condition

Advantages and limitations of redirection

- Flexibility as an advantage: When a programmer writes the software, it does not need to know where the log messages go
  - The programmer uses instead the standard descriptors for the output and error streams
  - That they are actually redirected is immaterial as far as the code is concerned (save for the issue with the buffer size, but then it's a good idea to flush the output anyway)
  - The server can be compiled on different machines without any change
    - even if one machine uses a system console to print out the messages while another machine uses log files
- Flexibility as a limitation:
  - A system administrator may want to forward all the log messages to some other machine
  - Another sysadmin may want to forward the log messages to another program
  - Using redirection all of these scenarios are awkward to implement
    - One can use some system of remote file systems, but this is overkill
  - In order to overcome these limitations, we can use (drum roll)... a client-server approach!

The log server

- Each participating computer operates a log server
  - This server accepts and handles messages intended for the system log
  - Once messages are received, the server can be programmed to print them to the console, write them to a file, or even send them to another machine
  - The server accepts local as well as remote connections
- When a program emits an error message, it becomes a client of the log server
  - The program send then the message and continues execution
  - The code of the program contains only information about how to send messages to the log server
    - It does not know and does not care how those messages are actually handled by the server
    - Much like redirection, but even more flexible

Linux log server: syslog

- Syslog consists of
  - A server (`syslogd`)
  - A series of library routines (i.e., C functions and shell commands) that can be used by a program to contact the log server
- Features:
  - Syslog groups messages into classes (according to their source and importance)
  - Syslog uses a configuration file, to allow the sysadmin to specify how different message classes are handled
    - For instance, if a serious error occurs, the messages might be sent to the console, whereas low priority (or informational) messages might be redirected to a file

Syslog message classes

- First, syslog partition programs into facilities, i.e., groups of programs with a common characteristic
  - Each log message must originate from one of these facilities
  - Each facility can be handled in a different way

Facility name	Subsystem using the facility
`LOG_KERN`	The kernel, i.e., the core of the OS
`LOG_USER`	Any user process (i.e., normal application)
`LOG_MAIL`	The email system
`LOG_FTP`	The ftp system
`LOG_DAEMONS`	System daemons
`LOG_AUTHPRIV`	Security/authorization messages
`LOG_LPR`	The printing system
`LOG_CRON`	Clock daemons
`LOG_SYSLOG`	Messages generated by `syslogd` itself
`LOG_LOCAL`i ($0 \leq i \leq 7$)	Reserved for local use

Syslog message classes (cont'd)

- Log messages are also classified according to priority levels

Priority	Meaning
`LOG_EMERG`	Extreme emergency, message should be communicated to all the users; system is unusable
`LOG_ALERT`	A condition that requires immediate action e.g., a corrupted passwd file
`LOG_CRIT`	A critical error, such as hardware failure
`LOG_ERR`	An error that requires attention, but is not critical
`LOG_WARNING`	A warning (there might be an error)
`LOG_NOTICE`	Normal, but significant, condition
`LOG_INFO`	Informational message e.g., messages printed by a daemon during normal startup
`LOG_DEBUG`	Messages used by programmers for debugging

  - Each priority level can be handled in a different way

Using syslog

- You should include, as usual, a header with declarations for the library functions

```
#include <syslog.h>
```

- Then you have to open access for your program to the syslog

```
openlog(identity, options, facility);
openlog("shfd", LOG_CONS|LOG_PID, LOG_USER);
```

  - No status is returned (why?)
- Once the syslog is opened, you can write to it using `syslog`

```
syslog(LOG_WARNING, "Message %d corrupted", message);
```

  - First argument is the priority, the rest are the same as for `printf`
- Once you are done with the syslog you do

```
closelog();
```

  - Even if you don't see it, opening the syslog allocated an entry in the descriptor table, so it is a good idea to close it when no longer needed

A syslog configuration file

- A configuration file (`/etc/syslog.conf`) specifies which messages goes where
  - Generic form: facility.priority where
  - "Globbing" (i.e., pattern matching) is allowed; priority may be `none` (meaning explicit exclusion of messages from that facility)

```
# Log all kernel messages to the console.
kern.*                                       /dev/console
# Log anything (except mail) of level info or higher.
# Don't log private authentication messages!
*.info;mail.none;authpriv.none;cron.none     /var/log/messages
# The authpriv file has restricted access.
authpriv.*                                   /var/log/secure
# Log all the mail messages in one place.
mail.*                                       /var/log/maillog
# Log cron stuff
cron.*                                       /var/log/cron
# Everybody gets emergency messages
*.emerg                                      *
# Save boot messages also to boot.log
local7.*                                     /var/log/boot.log
# Some logs also go to a world readable file
local6.*                                     /var/log/students.log
local6.*                                     /dev/tty12
```


Syslog issues

- A message is typically prefixed by syslog with the current date and the machine on which the event happened, e.g.

```
Mar 19 20:02:56 turing motion: [0:motion] [NTC] [ALL] motion_startup: Motion
4.0.1 Started
Mar 19 20:02:57 turing motion: [0:motion] [NTC] [ALL] motion_startup: Logging
to syslog
Mar 19 20:02:57 turing motion: [0:motion] [NTC] [ALL] motion_startup: Using
log type (ALL) log level (WRN)
Mar 20 06:35:47 hoare kernel: [4564877.497919] ata2: hard resetting link
Mar 20 06:35:47 hoare kernel: [4564877.806289] ata2: SATA link up 1.5 Gbps
(SStatus 113 SControl 310)
Mar 20 06:35:47 hoare kernel: [4564877.808856] ata2.00: configured for UDMA/33
Mar 20 06:35:47 hoare kernel: [4564877.808871] ata2: EH complete
Mar 20 20:55:00 clarke kernel: Kernel logging (proc) stopped.
Mar 20 20:55:00 clarke kernel: Kernel log daemon terminating.
Mar 20 20:55:01 chomsky kernel: Kernel logging (proc) stopped.
Mar 20 20:55:01 chomsky kernel: Kernel log daemon terminating.
Mar 20 20:55:01 clarke exiting on signal 15
Mar 20 20:55:02 chomsky exiting on signal 15
```

- Log files will grow with time; what do we do with them?
  - We rotate them periodically
  - Meaning that from time to time (typically: weekly) we create them afresh
  - A number of archival copies are usually kept