**UNX511/DSP912 – Assignment 2: Embedded Debug Logging**

**Due: Monday, April 15th, 11:59PM**

**Business Case** (Marketing, Project Manager, Senior Executives)

A product goes through the phases of business case, requirements, design, implementation, testing and rework, and eventually deployment to customers. Testing and rework attempts to simulate conditions at a customer site, but does not always catch every defect in the product. There are cases where the product malfunctions or stops working at a customer site, with very little clues as to what went wrong.

Linux has some built in logging mechanisms such as **syslog**, **rsyslog** and **syslog-ng**. These logs can be found in the **/var/log/** directory and can be useful. More useful is if the code for the product itself had logging embedded into it. Embedding debug logs in the code assists in debugging such problems in the field which are difficult to recreate in the lab. This will greatly assist in working through problems with a customer without losing customer confidence and thereby harming the reputation of the company.

Our task then is to develop an embedded logging mechanism in our code with logs being sent to a central server which can keep track of how the code is performing.

**Requirements** (Marketing, Project Manager, Project Lead)

* Logging must be terse but contain useful information.
* Each embedded log entry must contain a log severity. It must also contain the filename, the function name, the line number that the log appears in. Finally, it must contain a brief message.
* Logging must be fast. It cannot slow down the performance of the product.
* Logging must the filtered by the filter level severity. This means any logs at the filter level severity or higher will be reported. Any logs of lower severity than the filter level severity will be ignored.
* The severity levels will be debug, warning, error, and critical.
* The code itself will initialize logging and set the filter severity level.
* Logs must be sent to a central server on another machine for future reference.
* The server will collect all log messages on one common log file called the server log file.
* The server gives the option to dump the server log file to the screen.
* The server gives the option to overwrite the filter level severity.

**Design** (Project Lead, Senior Engineers)

**Each Process:**

* Each process will initialize logging for its process, set the filter level severity, call log functions to report any unusual activity, and shutdown the logger before exiting.

**The Logger:**

* The logging code will be contained within one file which can be included in the **Makefile** for each process.
* The logging code will communicate with the server via an IP address and port on the server.
* The logging code will collect logs from the process.
  + If the logs are below the filter level severity, they are thrown away.
  + If the logs are at or higher than the filter level severity, they are written to a server via UDP for fast delivery.
* The logging code will have a separate thread for reading UDP messages from the server.

**The Server:**

* The server must bind a socket to its IP address and to an available port.
* The server must have the ability to write messages to the logger via UDP for fast delivery.
* The server will have a separate thread for reading UDP messages from the logger.
* The server must give the user the option to dump the server log file to the screen.
* The server must give the user the option to overwrite the filter level severity of the logs.

**Implementation** (Junior and Senior Engineers)

**Each Process:**

* The process code will include a **Logger.h** file.
* The process **Makefile** will include **Logger.cpp** to its list of files.
* The process will initialize logging with a call to **InitializeLog()** of the logger.
* The process will set the filter severity level with a call to **SetLogLevel()**.
* The process will send log messages to the logger by calling **Log()** of the logger with the following information:
  + Log Severity – DEBUG, WARNING, ERROR, CRITICAL.
  + File name - **\_\_FILE\_\_**. This is a Linux macro which contains the filename. It is of type **char \***.
  + Function name - **\_\_func\_\_**. This is a Linux macro which contains the function name. It is of type **char \***.
  + Line number - **\_\_LINE\_\_**. This is a Linux macro which contains the line number of the log. It is of type **int**.
  + Message – The log message. This can be any user defined message contained as a string.

Example:

**Log(DEBUG, \_\_FILE\_\_, \_\_func\_\_, \_\_LINE\_\_, "Created the objects");**

* When the process is ready to exit, it will call **ExitLog()** to shut down the logger.

**The Logger:**

* The logger will type-define an enumeration called **LOG\_LEVEL** in its header file (**Logger.h**) specifying the log levels: DEBUG, WARNING, ERROR, CRITICAL.
* The logger will expose the following functions to each process via its header file (**Logger.h**):
  + **int InitializeLog()**;
  + **void SetLogLevel(LOG\_LEVEL level)**;
  + **void Log(LOG\_LEVEL level, const char \*prog, const char \*func, int line, const char \*message)**;
  + **void ExitLog()**;
* The logger will implement the above functions in its CPP file (**Logger.cpp**):
  + **InitializeLog()** will
    - create a non-blocking socket for UDP communications (**AF\_INET, SOCK\_DGRAM**).
    - Set the address and port of the server.
    - Initialize a thread mutex.
    - Start the receive thread and pass the file descriptor to it.
  + **SetLogLevel()** will set the filter log level and store in a variable global within Logger.cpp.
  + **Log()** will
    - compare the severity of the log to the filter log severity. The log will be thrown away if its severity is lower than the filter log severity.
    - create a timestamp to be added to the log message. Code for creating the log message will look something like:

**memset(buf, 0, BUF\_LEN);**

**char levelStr[][16]={"DEBUG", "WARNING", "ERROR", "CRITICAL"};**

**len = sprintf(buf, "%s %s %s:%s:%d %s\n", dt, levelStr[level], file, func, line, message)+1;**

**buf[len-1]='\0';**

* + - The message will be sent to the server via UDP **sendto()**.
  + **ExitLog()** will stop the receive thread via an **is\_running** flag and close the file descriptor.
  + The receive thread is waiting for any commands from the server. So far there is only one command from the server: **“Set Log Level=<level>”**. The receive thread will
    - accept the file descriptor as an argument.
    - run in an endless loop via an **is\_running** flag.
    - mutex the **recvfrom()** function to ensure two receives do not run into each other.
    - ensure the **recvfrom()** function is non-blocking with a sleep of 1 second between receives.
    - act on the command **“Set Log Level=<level>”** from the server to overwrite the filter log severity.

**The Server:**

* The server will shutdown gracefully via a ctrl-C via a **shutdownHandler**.
* The server’s **main()** function will create a non-blocking socket for UDP communications **(AF\_INET, SOCK\_DGRAM)**.
* The server’s **main()** function will bind the socket to its IP address and to an available port.
* The server’s **main()** function will initialize a thread mutex.
* The server’s **main()** function will start a receive thread and pass the file descriptor to it.
* The server’s **main()** function will present the user with three options via a user menu:
  + 1. Set the log level
    - The user will be prompted to enter the filter log severity.
    - The information will be sent to the logger. Sample code will look something like:

**memset(buf, 0, BUF\_LEN);**

**len=sprintf(buf, "Set Log Level=%d", level)+1;**

**sendto(fd, buf, len, 0, (struct sockaddr \*)&remaddr, addrlen);**

* + 2. Dump the log file here
    - The server will open its server log file for read only.
    - It will read the server’s log file contents and display them on the screen.
    - On completion, it will prompt the user with:

**“Press any key to continue: “**

* + 0. Shut down
    - The receive thread will be shutdown via an **is\_running** flag.
    - The server will exit its user menu.
    - The server will join the receive thread to itself so it doesn’t shut down before the receive thread does.
* The server’s receive thread will:
  + open the server log file for write only with permissions **rw-rw-rw-**
  + run in an endless while loop via an **is\_running** flag.
  + mutex the **recvfrom()** function to ensure two receives do not run into each other.
  + ensure the **recvfrom()** function is non-blocking with a sleep of 1 second between receives.
  + take any content from **recvfrom()** and write to the server log file.

On the process side you have been provided with a [**Makefile**](https://scs.senecac.on.ca/~miguel.watler/courses/unx511/Assignment2/logger/Makefile), [**TravelSimulator.cpp**](https://scs.senecac.on.ca/~miguel.watler/courses/unx511/Assignment2/logger/TravelSimulator.cpp), [**Automobile.cpp**](https://scs.senecac.on.ca/~miguel.watler/courses/unx511/Assignment2/logger/Automobile.cpp), and [**Automobile.h**](https://scs.senecac.on.ca/~miguel.watler/courses/unx511/Assignment2/logger/Automobile.h). You have to implement **Logger.cpp** and **Logger.h**.

On the server side you have to implement **LogServer.cpp** and you also have to create a **Makefile**.

**Testing and Rework** (Junior and Senior Engineers, Product Support)

* Start the server on one machine. Make sure you have the correct IP address and port.
* Start the test application on another machine, which in this case is called **TravelSimulator**.
* Make sure your logger has the correct IP address and port of the server.
* Logging has been added to the **TravelSimulator**, along with calls to **InitializeLog()**, **SetLogLevel()** and **ExitLog()**.
* The **TravelSimulator** runs indefinitely until the user administers a ctrl-C.
* Observe the server log file as it grows either by dumping to the screen or looking at the file itself.
* Change the filter log severity on the server.
* Observe the server log file again.
  + Is it getting all of the log messages?
  + Are the log messages in sequence?
  + For each log message verify the timestamp, log level, file name, function, line number and the message.

**Questions**

1. Generally, what are syslog and rsyslog? Specifically, name three features of syslog/rsyslog and compare them to your embedded debug logging. Will there be any overlap of information?
2. Name five features of syslog-ng.
3. Name five ways syslog-ng is an improvement over syslog/rsyslog.
4. Consider a Log Server that has to manage embedded logs for a massive amount of processes on a massive amount of machines. Name three ways the server could manage the connections to each process.
5. Consider a Log Server that has to manage embedded logs for a massive amount of processes on a massive amount of machines. With such a large amount of data in the logs, name three ways a user could extract useful information from them (be general).
6. Explain how gdb could be used on a Linux machine to attach to a process and get thread information. Is this also useful in debugging?

**Timeline**

**Week 1:** Create the Logger.

**Week 2:** Create the Server.

**Week 3:** Test and Rework. Answer questions. Submit assignment.

**Marking Rubric**

Assignment 2 is worth 15% of your final grade and as such is marked out of 15 as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Does not meet expectations** | **Satisfactory** | **Good** | **Exceeds Expectations** |
| **The Logger**  **(3 marks)** | Does not meet requirements | Meets the most important requirements | Meets all requirements with minor errors | Meets all requirements with no errors |
| **The Server**  **(3 marks)** | Does not meet requirements | Meets the most important requirements | Meets all requirements with minor errors | Meets all requirements with no errors |
| **Documentation**  **(3 marks)** | Does not contain documentation. | Contains header documentation for either all files or for all functions within each file | Contains header documentation for all files and for most functions within each file. | Contains header documentation for all files and for all functions within each file.  Documents unclear code. |
| **Questions**  **(6 marks)** | Answers no question correctly  (0 marks) | Answers some questions correctly  (1-2 marks) | Answers most questions correctly  (3-4 marks) | Answers all Questions correctly  (5-6 marks) |

**Submission**

Please email all source code and answers to questions to:

**miguel.watler@senecacollege.ca**

Your questions can be submitted in a separate document or embedded within your source code.

**Late Policy**

You will be docked 10% if your assignment is submitted 1-4 days late.

You will be docked 20% if your assignment is submitted 5-6 days late.

You will be docked 30% if your assignment is submitted 7-8 days late.

You will be docked 40% if your assignment is submitted 9-10 days late.

You will be docked 50% if your assignment is submitted over 10 days late.