Description of the bxilog library architecture

Pierre Vignéras

2014-04-24

1 Objectives

This document describes the architecture of the bxi logging api named bxilog.

2 Introduction

Any serious project needs a logging library. As an example, Python provides the `logging` module and Java provides the `java.logging` package. Unfortunately neither the C language nor the POSIX standard provides an efficient logging library[1]. A state of the art study has been conducted, but conclusions were disappointing. See document bxiclib/misc/sdk-choosen.txt for details.

Therefore, a new logging solution has been developped with the following features:

- Easy: straightforward to use;
- Complete: be sufficient in itself to produce all sort of messages from a given process;
- Performant: reduces the overhead as far as possible, in particular, the actual business code should not suffer too much from logging;
- Flexible: fits many different applications, in particular daemons and non-daemons processes;
- Lightweigth: does not require a daemon;
- Robust: supports forking, signals and multi-threading;
- Scalable: maintains good performance with a huge amount of threads and logs.

To our knowledge, no current logging solution in C provides these features.

3 Architecture

The bxilog solution is thread-safe, but lockless by design: it uses an asynchronous message passing paradigm, implemented by the `zeromq` library for internal and external communication. The architecture is illustrated on figure 1.

3.1 Business Code parts

The business code is traditionnally made of several modules, each having one or many `logger` data-structure (object like). A `logger` has a `name` and a `level`. The name is just a string (`char *`), a level is defined by an integer among the following:

- `CRITICAL:` used in critical cases, usually, the process is no more able to run after such a message;
- `ERROR:` used to indicate normal errors;
- `WARNING:` used to indicate warnings;
- `OUTPUT:` used for any normal messages that should be displayed, this should replace `printf()`;
- `INFO:` used for messages that can help end-user understand what is going on;

[1] Whereas `syslog` is defined by POSIX, it is not considered here, since it is quite heavy: it requires a daemon `syslogd` and lack several features described in this document.

- `DEBUG:` used for detailed messages that can help debugging the application;
- `TRACE:` used for very detailed informations, probably only helpful for application developpers.

The `OUTPUT` level is very important: replacing all `printf()` output by a log at this level provides various advantages:

- performance: the business code thread won't perform any costly output operation, the bxilog will do it on its behalf (in another thread or process), letting the business code thread moving forward;
- flexibility: since all messages are sent through bxilog including outputs they can be treated in various ways. As an example, all `OUTPUT` and `WARNING` messages can be sent to `stdout`, all `ERROR` and `CRITICAL` can be sent to `stderr`, while at the same time, all logs including `TRACE`, `DEBUG`, and `INFO`, can be flushed to storage in a file in a very detailed format. With such a mechanism, a developper of an application can ask one of its user facing a strange behavior to send the file: it will contain all the data seen by the end-user (`OUTPUT, WARNING, ERROR, CRITICAL`) and also the low level messages (`TRACE, DEBUG, INFO`).

The business code can have multiple threads traversing its own code, and therefore, using its own modules: there is no relationship between `loggers` and business code threads. Such a thread can traverse all business code modules and therefore produce logs with different `loggers`.

3.2 bxilog part

The main component of the bxilog is the internal thread that runs a very thin layer of code. When the API is initialized, an internal thread is created. This internal thread creates a special `zeromq PULL` socket that will receive all messages from business code threads through the `zeromq inproc` protocol. This protocol does not induce costly input/output operations. The main role of this internal thread is therefore to release each business code thread from costly input/output operations: as soon as a message has been given to the bxilog library, the thread can move on to its actual processing code.

The internal thread main code is reduced to the transformation of raw log data into a language neutral log format, and their publishing to subscribers using a `zeromq PUB` socket. The actual protocol used can be specified during the initialization. It can be any of `inproc, ipc, tcp,` or even `pgm` or `epgm`. A log includes many information such as:

- log level,
- precise timestamp,
- process id, kernel thread id, software thread id,
- program name,
- source file, line number, and function name,
- logger name,
- and the message.

A subscriber can be internal to the process that emits the log, or external.

Typically, a single internal subscriber will create a message from the log and will display it on the console and store it as well on storage according to some parameters given from the command line and/or the environment.

An external subscriber might filter according to system level criteria and forward only interesting messages to a dedicated system logger such as syslog.

Note that if no subscriber exist, then all logs will just be dropped.

4 Complexities

4.1 Initialization and termination

When the internal thread is started, the thread that initialize the bxilog API should wait until the internal thread is ready to process logging message.

4.2 Logger configuration

When a log is emmitted, a check is made to know wether the given `logger` has been configured or not. If not, the logger is configured according to rules specified during the bxilog initialization.

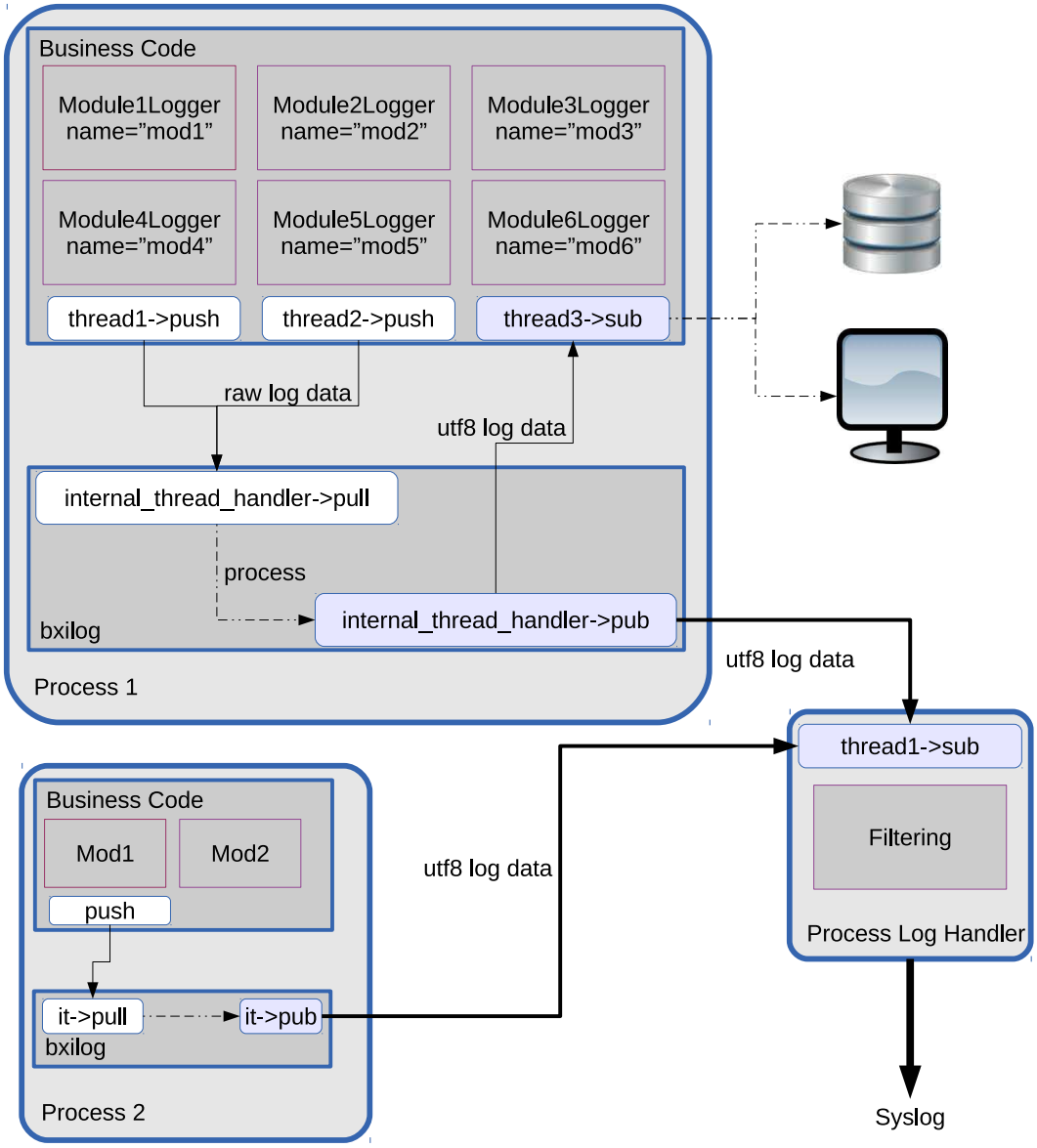

Figure 1: The architecture of bxilog.

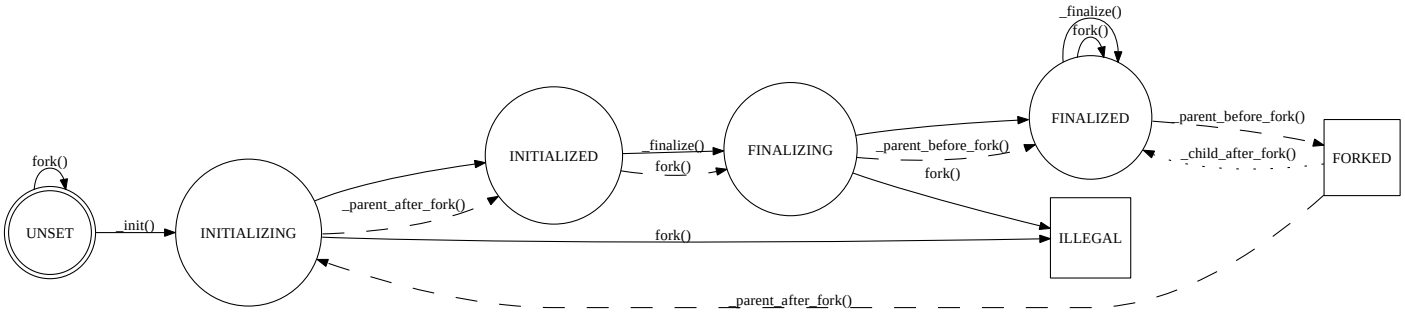

Figure 2: Finite State Machine of bxilog.

4.3 Synchronization of the internal thread with subscribers

Due to the nature of ZeroMQ PUB/SUB sockets, if a synchronization mechanism is not implemented between subscribers and publishers, a significant proportion of first logs will be missed by subscribers. Therefore, the internal threads will immediatly emit special `SYNC` messages on its PUB ZeroMQ socket. The internal thread is initially given the number of subscribers to synchronize with. It then regularly polls a specific socket (not shown on figure) for a `SYNC_ACK` message from a subscriber, stating that a new subscriber has received the `SYNC` message, and is therefore ready to receive real logs. Once the expected number of subscribers have been synchronized, the internal thread will notify the business code thread that it is ready to proceed.

4.4 Supporting fork() system call

Forking a multithreaded process is undefined in POSIX. And the bxilog API is not only thread-safe, it is itself multithreaded since at least the internal thread and the main thread cohabits in the same process. Therefore, forking such a process is challenging. The main solution is to gently kill the bxilog specific threads (internal thread and any other subscriber threads), helping the whole process to turn into a single threaded program just before the fork, then to fork, and then in the parent to come back to the state recorded before the fork, and in the child to stay in the uninitialized state as if a new process has been launched. If the child wants to perform some logs using bxilog, it should initialize the bxilog library as any process does. Therefore the bxilog library maintains a finite state machine given by figure 2.

4.5 Dealing with signals

Handling UNIX signals is tricky. It becomes even trickier in a multi-threaded applications. Two cases should be distinguished: signals received by the internal thread and signals received by the business code.

4.5.1 Internal Thread Signals Handling

Inside the internal thread: asynchronous signals such as SIGINT, SIGTERM, SIGQUIT and so on should not be received. It is not the role of the internal thread to deal with them. Those signals should be dealt with by the business code itself. Other signals (SIGSEGV, SIGBUS, ...) might happenend in the internal thread due to bugs, memory corruption, etc. It should therefore deal with them.

The solution is to include those signals in the internal thread polling loop. If such a signal is received, a log is produced internally, flushed

and the signal is raised again with the default signal handler in order to kill the application (and produce a core dump if required).

4.5.2 Business Code Signals Handling

When the business code receive a signal, the good thing to do is to emit a log with a backtrace. Therefore, bxilog provides a function that sets up signal handlers for doing that. The business code is free to use it or to provide its own signal handlers.

Since after a SIGSEGV the best thing to do is to quit, and since exiting without calling `bxilog_finalize()` will lost messages, the end result will be that the log produced by the SIGSEGV handling won't be seen at all in the log. The issue is to make sure the internal thread has flushed all logs before exiting. Therefore, the sighandler, ask for the termination of the internal thread which flushes the logs as far as it can , waits some time and then exit (raising the original signal with the default signal handler).