Libmonitor: A Tool for First-Party Monitoring

Mark W. Krentel
Dept. of Computer Science
Rice University
6100 Main St., Houston, TX 77005
krentel@rice.edu

ABSTRACT

Libmonitor is a library that provides hooks into a program and provides callback functions for monitoring the begin and end of processes and threads and maintains control across fork, exec and in the presence of signals. It provides a layer on which to build first-party profiling tools for performance or correctness. Libmonitor is lightweight, fully scalable, easy to use and does not require access to an application's source code.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

libmonitor, process and thread control

1. INTRODUCTION

Libmonitor is a library that provides hooks into a program for first-party monitoring and control of process and threads. Libmonitor gains control and provides callback functions for the begin and end of processes and threads and maintains control across fork, exec and in the presence of signals. It provides a separable layer on which to build profiling tools for performance or correctness. For example, the Rice HPCToolkit [1, 8] uses libmonitor to attach to a process, turn profiling on and off, assist in unwinding the call stack and manage exceptional events such as signals.

Libmonitor was designed for large parallel systems such as IBM's BlueGene and Cray's XT. These systems run restricted microkernels on separate compute nodes where standard Unix tools such as `ptrace`, `fork` and `dlopen` are often unsuitable or unavailable. For efficiency, the compute nodes often do not support dynamically linked executables. On these systems, the profiling tools must adapt to first-party monitoring and the entire program, including application, profiling tools and libmonitor must run as a single statically linked binary.

Libmonitor works by overriding functions such as `main`, `exit` and `pthread_create`. This allows libmonitor to gain control of a process and provide a callback function to the client. There are callbacks for the begin and end of processes and threads, `fork`, `dlopen` and `MPI_Init`, as summarized in Figure 1. After the callback returns, libmonitor invokes the real function to allow the application to continue. There are also support functions for intercepting the application's signals, to assist in unwinding the call stack and to provide access to the real versions of some of the functions that libmonitor overrides. Figure 2 summarizes the support functions.

A key feature in how libmonitor attaches to a process is that it does so without access to the application's source code. For proper insight into how a program runs, it is vital to observe it in its native state at full optimization. There are technical differences in the statically and dynamically linked cases, but neither one requires modifying or even reading the application's source code. Instead, libmonitor intercepts the application's function calls at the object layer. This approach works with multi-lingual code, does not require any specific language features or compilers and works at all optimization levels. This allows for monitoring programs in their native state.

The libmonitor layer is also lightweight and fully scalable. Libmonitor does not require any inter-process communication and does not create any new processes or threads for itself. Thus, any overhead does not increase with the number of processors. As such, libmonitor is suitable for running on parallel systems with hundreds of thousands or even millions of cores.

The advantage of libmonitor as a separate tool is that it gives a clean separation between the application and the libmonitor client. Libmonitor distinguishes between the *client*, the *application* and libmonitor itself. The client knows about libmonitor and libmonitor provides callback and support functions to the client, but the application doesn't know that the client or libmonitor are part of its process. The client can write its callback functions without worrying about the details of how libmonitor attaches its hooks into the application.

This separation also makes libmonitor very easy to use, allowing for quick prototypes of profiling tools. A libmonitor client only needs to define the callback functions that it wants and compile them into an object file or library. For example, the `papiex` program [7] reports the totals of performance counter events in a program's execution. A skeletal version of `papiex` can be written by combining libmonitor and PAPI [9] in about half a page of code as shown in Figure 3. The program simply needs to start PAPI in the init process callback and then stop it and print the counter results in the fini process callback.

The original monitor program was written by Philip Mucci at the University of Tennessee [6]. Libmonitor as described in this paper is a rewrite from scratch of Mucci's monitor. Our original motivation was to extend the old monitor to handle the case of statically linked binaries. Since then, there have been several other improvements and better handling of corner cases. The new libmonitor has more extensive support for signal handlers, pre- and post-callbacks for the beginning of threads and fork, generic MPI support and support for finding a thread's stack bottom to assist unwinding the call stack.

Libmonitor is used by the Rice HPCTOOLKIT [1, 8] and Open SpeedShop [2] projects. Both projects use libmonitor to attach their profiling tools to a process.

Libmonitor is licensed under the three-clause BSD license and is available via anonymous subversion download from the SciDAC Outreach Center [3].

2. LIBMONITOR FEATURES

Libmonitor's main function is to give the client control over an application's processes and threads. And it must do so without the application noticing and without access to its source code. To meet these requirements, libmonitor overrides functions such as `main` and `exit`. When libmonitor gains control, it invokes a callback function that the client uses to insert its own code. When the callback returns, libmonitor calls the real function to return control to the application. For example, HPCTOOLKIT [1, 8] turns on profiling interrupts via `setitimer` or `PAPI_overflow` at the beginning of the process and then turns them off at the end.

The core libmonitor callback functions include four callbacks for the begin (init) and end (fini) of processes and threads. These functions are called from within the thread itself. Another callback is for the beginning of thread support. This function is called from within the main thread just before the application creates the first new thread. Libmonitor promises that the init process callback is called before the begin thread callback and that is called before any init thread callback. At the end of the process, all of the fini thread callbacks are called before the final fini process. Figure 1 summarizes the callback functions.

To insert its code into an application, the client defines the callback functions that it wants to receive and compiles them into an object file or shared library. These callbacks are then linked with libmonitor and the application at run time with `LD_PRELOAD` in the dynamic case or at link time with the linker option `--wrap` in the static case. The client does not need to define all of the callbacks. Libmonitor provides default definitions as weak symbols for all of the callbacks, so the client only needs to define the ones it wants.

Fork and exec. In Linux, new processes are created and run with `fork` and `exec`. Accordingly, libmonitor overrides these functions in order to maintain control so that an application cannot launch new processes without libmonitor noticing. When the application calls `fork`, libmonitor delivers three callbacks. First, it delivers a pre-fork callback in the parent process, then an init process callback in the child and finally a post-fork callback in the parent. (Technically, the init process and post-fork callbacks occur in separate processes, so they could occur in either order.) This sequence enables the client to track the parent-child relationship among processes that it is monitoring.

Even though `exec` technically does not change the process id number (pid), `exec` represents the end of one process and the beginning of another. Accordingly, libmonitor delivers a fini process callback in the old process and then, if the new process is also monitored, an init process callback in the new process.

Threads. Besides the init and fini thread callbacks, libmonitor also provides support for matching parent and child threads. Libmonitor provides pre and post thread create functions analogous to the pre and post fork callbacks. When the application calls `pthread_create`, libmonitor calls thread pre create in the parent thread, then init thread in the new thread and finally thread post create in the parent thread. (Again, the init thread and post create callbacks occur in separate threads, so they could occur in either order.) This sequence enables the client to track where in the parent thread the new threads are created.

Libmonitor also maintains a thread local data pointer for each thread. This is a `void *` pointer which is set by the return value of the init thread callback and maintained throughout the lifetime of the thread. The pointer returned by the thread pre create callback is passed to the init thread callback in the new thread, thus making it easy for the client to match the parent and child threads.

Signals. Because signals are an important part of process control, libmonitor gives the client first access to any signal. That is, the client can register a handler with libmonitor via the `monitor_sigaction` support function (see Figure 2). Libmonitor will deliver this signal first to the client, even if the application has registered a handler for the same signal. The client can then process the signal or decline it. If the client declines the signal, libmonitor handles it according to the disposition of the application, either invoking the default action or delivering the signal to the application.

For example, unwinding the call stack is a tricky process and can sometimes produce segfaults due to bad debugging information. So, HPCTOOLKIT registers a handler for `SIGSEGV` with libmonitor. When it receives this signal, if the segfault occurred during an unwind attempt, then HPC-

TOOLKIT uses `siglongjmp` to exit the region. Otherwise, it declines the signal and libmonitor delivers the signal to the application.

Dlopen. Processes can insert new code into their address space with `dlopen` and `dlclose`. Profiling tools need to know when this happens to analyze the new regions. Accordingly, libmonitor provides callbacks when the application calls `dlopen` or `dlclose`. The client receives callbacks immediately before and after the `dlopen` or `dlclose` along with the path or handle to the library. This sequence allows the client to track changes in the application's address space.

Runtime stack bottom. An important task in performance and correctness tools is unwinding the call stack. Accordingly, libmonitor provides support functions to help make the unwind process more reliable. First, libmonitor identifies a range for the return address of the last (innermost) function on the call stack for every thread and makes this available to the client. Because libmonitor intercepts the beginning of every process and thread, there is already a libmonitor function at the bottom of the call stack. Thus, it is a simple matter for libmonitor to put assembly labels immediately before and after the call to the first application function, either `main` or the pthread start routine. Every application stack unwind should end at an address within one of these frames.

Second, libmonitor identifies the bottom of every thread's runtime stack. Again, because libmonitor runs before the application, it can use `alloca` to allocate a small amount of space on the stack and make this address available to the client. This way, the client can find the bottom of the runtime stack and can tell if it has unwound too far.

3. TECHNICAL CHALLENGES

Libmonitor faces a number of interesting technical challenges to maintain process control, including overriding functions in the static and dynamic cases, control of signals and catching process exit.

Dynamically linked case. For dynamically linked applications on Linux systems, libmonitor uses the dynamic linker-loader's `LD_PRELOAD` feature [5]. Libmonitor defines overrides for functions such as `__libc_start_main`, `_exit`, `fork`, `exec`, etc. These overrides plus the rest of the libmonitor code are compiled into a shared library which is inserted into a process's address space by setting the `LD_PRELOAD` environment variable. (It is common to use a launch script to set `LD_PRELOAD` and exec the application.) When the process starts, it runs `__libc_start_main` from libmonitor instead of libc. Libmonitor replaces the argument for the application's `main` function with the address of its own main and calls the real `__libc_start_main`. Then, from the libmonitor main, it delivers the init process callback to the client and finally calls the application's `main` function.

The advantage of the `LD_PRELOAD` solution is that it works with prebuilt executables and requires no modifications to any source files or the build system. It also works when source files are not available. The disadvantage is that this approach only works with functions from shared libraries. Although it might seem more natural to override `main` instead of `__libc_start_main`, that idea won't work. Functions such as `main` are statically linked and cannot be overridden with `LD_PRELOAD`. On Linux systems, `_start` calls `__libc_start_main` in libc which then calls `main` and this approach works. On other systems such as FreeBSD, `_start` calls `main` directly and so this approach will not work there.

Statically linked case. For statically linked applications, libmonitor uses the `--wrap` option from GNU's `ld` [4]. This is a feature for overriding functions at program link time. For example, to intercept the function `main`, libmonitor defines the function `__wrap_main` and adds the option "`--wrap main`" to the link line. This causes the linker to replace calls to `main` with libmonitor's `__wrap_main` and to replace calls to `__real_main` with the application's `main`. Libmonitor gains control of the process in its `__wrap_main` function. There, it delivers the init process callback and then calls `__real_main` to return control to the application.

The advantage of the `--wrap` solution is that it works with any function, not just functions from libc. This method also works on systems like FreeBSD that don't use `__libc_start_main`. The disadvantage is that it requires relinking the application to add the libmonitor code. Relinking does not require modifying any source files and technically, it does not require recompiling any files (that is, compiling from source to object). However, it does require modifying and rerunning the final link step.

Control of signals. Libmonitor maintains control of the application's signals for two reasons. First, signals can cause the process to terminate. Libmonitor needs to catch all forms of exit, so it cannot allow the process to terminate by a signal without noticing. Second, libmonitor gives the client first access to any signal. That is, the client can register a handler with libmonitor, receive a signal and decide if it wants to handle the signal. If the client declines the signal, libmonitor handles the signal according to the disposition of the application, either invoking the default action or delivering the signal to the application.

To maintain control of signals, libmonitor installs a signal handler for every catchable signal, that is, every signal except `SIGSTOP` and `SIGKILL`. Libmonitor must also override `signal` and `sigaction` and keep track of the application's disposition for every signal, whether the signal is caught, ignored or has the default action. Then, on every signal delivery, libmonitor must catch the signal, possibly offer it to the client and then take the appropriate action for that signal.

Process exit. An essential feature of libmonitor is that it catches and delivers a callback for all forms of process exit. A process can terminate by calling `exit` or `exec`, returning from `main` or by exiting via a signal. To catch these, libmon-

itor overrides all forms of exit: `exit`, `_exit` and `_Exit`; and all forms of exec: `execl`, `execlp`, `execle`, `execv`, `execvp` and `execve`. Libmonitor can then deliver the callback function before calling the real function.

To catch termination by a signal, libmonitor installs a signal handler for every catchable signal and maintains a table of every signal's disposition. When it receives a signal, libmonitor must determine if this signal will lead to process termination. If so, it must deliver the fini process callback before allowing the signal to take effect.

Thread shootdown. Threaded processes are not always well behaved at process exit time. It would be nice if all threads called `pthread_exit` before the main process called `exit`, but this is not always the case. Further, `exit` is not always called from the main thread. Libmonitor promises that before the process exits, every thread (except the main thread) receives a fini thread callback from within that thread. But merely overriding `pthread_exit` is not sufficient because another thread may exit the process without calling `pthread_exit`.

Libmonitor addresses these issues by using signals to run code in another thread. When one thread initiates process exit, libmonitor sends `pthread_kill` to the other threads to tell them to run their fini thread callback. This use of signals is how libmonitor runs code in another thread so that the fini thread callback will be delivered from within the thread itself. Although libmonitor calls this technique "thread shootdown," it is important to note that libmonitor does not actually terminate the other threads. After the fini thread callback, libmonitor returns control to the application to allow it to terminate naturally.

Multiple MPI implementations. For MPI programs, libmonitor provides the client with the MPI size and rank relative to `MPI_COMM_WORLD`. But this presents a technical problem. Getting the process's rank by calling `MPI_Comm_rank` requires knowing the value for `MPI_COMM_WORLD`. But the value and even the type of `MPI_COMM_WORLD` varies by implementation. On the surface, this means that libmonitor must be tied to a single MPI implementation.

Libmonitor's solution is to wait for the application to call `MPI_Comm_rank` itself, capture the first argument and use the same value as the communicator. In this way, it is possible to build a single libmonitor installation that works with essentially any MPI implementation. This method does require that the application calls `MPI_Comm_rank` early in the program and uses `MPI_COMM_WORLD` as its first communicator, but virtually any MPI program will already do this.

4. SUMMARY

In summary, libmonitor is a useful library for process and thread control. Libmonitor is mainly designed for first-party monitoring on large parallel systems such as the IBM BlueGene and the Cray XT. Libmonitor is easy to use, fully scalable and works with both statically and dynamically linked executables.

5. ACKNOWLEDGMENTS

Development of libmonitor is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762.

6. REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Krell Institute. Open SpeedShop for Linux. `http://www.openspeedshop.org/wp/`.

[3] M. Krentel. Libmonitor. `https://outreach.scidac.gov/projects/libmonitor/`.

[4] The `ld(1)` man page.

[5] The `ld.so(8)` man page.

[6] P. Mucci. Monitor. `http://icl.cs.utk.edu/~mucci/monitor/`.

[7] P. Mucci. Papiex. `http://icl.cs.utk.edu/~mucci/papiex/`.

[8] Rice University. HPCToolkit performance tools. `http://hpctoolkit.org/`.

[9] University of Tennessee, Knoxville. The performance API. `http://icl.cs.utk.edu/papi/`.

```
void *monitor_init_process(int *argc, char **argv, void *data);
void  monitor_fini_process(int how, void *data);
void *monitor_init_thread(int tid, void *data);
void  monitor_fini_thread(void *data);
void  monitor_init_thread_support(void);
void  monitor_pre_dlopen(const char *path, int flags);
void  monitor_dlopen(const char *path, int flags, void *handle);
void  monitor_dlclose(void *handle);
void  monitor_post_dlclose(void *handle, int ret);
void *monitor_pre_fork(void);
void  monitor_post_fork(pid_t child, void *data);
void *monitor_thread_pre_create(void);
void  monitor_thread_post_create(void *);
void  monitor_init_mpi(int *argc, char ***argv);
void  monitor_fini_mpi(void);
```

Figure 1: Selected libmonitor callback functions.

```
void *monitor_real_dlopen(const char *path, int flags);
int   monitor_real_dlclose(void *handle);
void  monitor_real_exit(int status);
int   monitor_real_system(const char *command);
int   monitor_real_sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int   monitor_real_pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
int   monitor_sigaction(int sig, monitor_sighandler_t *handler, int flags,
                        struct sigaction *act);
int   monitor_is_threaded(void);
void *monitor_get_user_data(void);
int   monitor_get_thread_num(void);
void *monitor_stack_bottom(void);
int   monitor_in_start_func_wide(void *addr);
int   monitor_in_start_func_narrow(void *addr);
int   monitor_mpi_comm_size(void);
int   monitor_mpi_comm_rank(void);
```

Figure 2: Selected libmonitor support functions.

```
/*
 *  Combine libmonitor and PAPI to count PAPI_TOT_CYC in a process.
 */
#include <err.h>
#include <errno.h>
#include <stdio.h>
#include "monitor.h"
#include "papi.h"

static int EventSet = PAPI_NULL;

void *
monitor_init_process(int *argc, char **argv, void *data)
{
    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
        errx(1, "PAPI_library_init failed");

    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        errx(1, "PAPI_create_eventset failed");

    if (PAPI_add_event(EventSet, PAPI_TOT_CYC) != PAPI_OK)
        errx(1, "PAPI_add_event failed");

    if (PAPI_start(EventSet) != PAPI_OK)
        errx(1, "PAPI_start failed");

    return NULL;
}

void
monitor_fini_process(int how, void *data)
{
    long long values[1];

    if (PAPI_stop(EventSet, values) != PAPI_OK)
        errx(1, "PAPI_stop failed");

    fprintf(stderr, "This process used %lld PAPI_TOT_CYC.\n", values[0]);
}
```

Figure 3: Skeleton papiex callbacks to count `PAPI_TOT_CYC` in a process.