User Level DB: a Debugging API for User-Level Thread Libraries

Kevin Pouget, Marc Pérache, Patrick Carribault, and Hervé Jourdren

kevin.pouget@gmail.com, {marc.perache,patrick.carribault,herve.jourdren}@cea.fr

CEA, DAM, DIF, F-91297 Arpajon, France
Outline

- Context
- User-Level Thread Libraries
- User-Level Thread Libraries and Debugging
- User Level DB
- Conclusion and Future Work
Introduction

• Context
  • High Performance Computing
  • Supercomputer architecture
    • Cluster
    • Non-Uniform Memory Access (NUMA) hardware
    • Multicore
  • User-Level Thread Libraries
  • MPC (MultiProcessor Communications):
    • APIs: MPI 1.3, OpenMP 2.5, PThread

• How to debug threads in this context?
Thread Libraries
Thread Libraries

a) Kernel

b) User

c) Hybrid
Thread Libraries

- Why using user-level thread libraries?
  - Thread scheduler in user-space:
    - Can be specialized.
    - Low-cost context switching.
  - Portable.

- MPC framework:
  - Thread-based MPI: each MPI task is converted a thread.
  - POSIX Thread API and OpenMP API: efficient mix of threads and MPI
  - Everything is a thread.
  - Optimized scheduler:
    - Integrated polling method.
    - Collective communications.
User-Level Thread Libraries and Debugging
Debugging Threads

- User command
Debugging Threads

- User command
- Request to the Thread_DB library thanks to Thread_DB API
Debugging Threads

- User command
- Request to the Thread_DB library thanks to Thread_DB API
- Use the thread-library specific implementation of Thread_DB
Debugging Threads

- User command
- Request to the Thread_DB library thanks to Thread_DB API
- Use the thread library specific implementation of Thread_DB
- Use the Proc_service API to read into the process to debug
Debugging Threads

- User command
- Request to the Thread_DB library thanks to Thread_DB API
- Use the thread library specific implementation of Thread_DB
- Use the Proc_service API to read into the process to debug
- Return command results to the debugger
Debugging Threads

- User command
- Request to the Thread_DB library thanks to Thread_DB API
- Use the thread library specific implementation of Thread_DB
- Use the Proc_service API to read into the process to debug
- Return command results to the debugger

Have to be implemented for each thread library
Debugging User-Level Threads with GDB

- Example:
  - 6 user threads
  - 2 kernel threads

- With GDB

  (gdb) info threads
  Thread 2 (LWP 2) in bar ()
  Thread 1 (LWP 1) in foo ()

- Expected:

  (gdb) info threads
  Thread 5 (LWP 2) in mpc_setjmp ()
  Thread 4 (LWP 2) in bar ()
  Thread 3 (LWP 1) in foo ()
  Thread 2 (LWP 1) in mpc_setjmp ()
  Thread 1 (LWP 1) in mpc_setjmp ()
Debugging User-Level Threads with GDB

- **Example:**
  - 6 user threads
  - 2 kernel threads

- **With GDB**
  
  ```
  (gdb) info threads
  Thread 2 (LWP 2) in bar ()
  Thread 1 (LWP 1) in foo ()
  ```

- **Expected:**
  
  ```
  (gdb) info threads
  Thread 5 (LWP 2) in mpc_setjmp ()
  Thread 4 (LWP 2) in bar ()
  Thread 3 (LWP 1) in foo ()
  Thread 2 (LWP 1) in mpc_setjmp ()
  Thread 1 (LWP 1) in mpc_setjmp ()
  ```
Debugging User-Level Threads with GDB

• General case:
  • The user-level thread library uses the underlying POSIX Thread library.
  • Debuggers use the libthread_db associated to the underlying POSIX Thread library.
  • Debuggers ignore sleeping user thread.
  • Debuggers only see running threads.

=> Need a specialized libthread_db for each user-level thread library and force the debugger to use it.
Debugging User-Level Threads with GDB on Linux

- User command

Diagram:
- Debugger
- Thread_DB API
- Proc_service API
- Thread library
- X Debug module
- Debugger space
- Process space

MTAAP 2010

CEA,DAM,DIF
Debugging User-Level Threads with GDB on Linux

- User command
- Use registers of the kernel threads
Debugging User-Level Threads with GDB on Linux

- User command
- Use registers of the kernel threads
- Return command results to the debugger
Debugging User-Level Threads with GDB on Linux

- GDB on Linux do not use libthread_db correctly:
  - Skip the libthread_db library.
  - Due to bugs in libthread_db implementation.
  - Assume 1x1 threading model (true for Linux).

=> Need to patch GDB in order to use libthread_db correctly.
Our contribution:
User Level DB
User Level DB Library Goals

- Provide debugger support to user-level thread libraries.
- Low development cost to support a new thread library.
- Portable:
  - Thread libraries.
  - Processors.
  - Operating systems.
  - Debuggers.
User Level DB Library Architecture

[Diagram showing the architecture of the user level database library, including components such as Thread_DB API, ULDB module, Thread module, Proc_service API, Debug module, and ULDB API.]
User Level DB Library

Provided by ULDB
User Level DB Library

Have to be implemented by each thread library
User Level DB API

- /* Enable or disable the ULDB library */
  - tdb_err_e uldb_enable_lib_thread_db (void)
  - tdb_err_e uldb_disable_lib_thread_db (void)
- /* Provide the offsets of x86 registers */
  - tdb_err_e uldb_set_eip_offset (size_t offset)
  - tdb_err_e uldb_set_esp_offset (size_t offset)
  - tdb_err_e uldb_set_ebp_offset (size_t offset)
  - tdb_err_e uldb_set_ebx_offset (size_t offset)
  - tdb_err_e uldb_set_edi_offset (size_t offset)
- /* Provide a lock object and its accessors */
  - tdb_err_e uldb_set_lock (void *lock, int (*acquire) (void *), int (*release) (void *), int (*lock_free) (void *))
User Level DB API

- /* Add or remove a thread to the internal list */
  - uldb_add_thread (const void *tid, tdb_thread_debug_t **thread)
  - uldb_remove_thread (tdb_thread_debug_t *thread)

- /* Report the events monitored by the debugger*/
  - uldb_report_creation_event (tdb_thread_debug_t *thread)
  - uldb_report_death_event (tdb_thread_debug_t *thread)

- /* Provide the address of the context of a sleeping thread */
  - uldb_set_thread_context (tdb_thread_debug_t *thread, void *context)
User Level DB API

- /* Provide further details about a thread (optional) */
  uldb_set_thread_startfunc (tdb_thread_debug_t *thread, char *tls)
- uldb_set_thread_stkbase (tdb_thread_debug_t *thread, void *stkbase)
- uldb_set_thread_stksize (tdb_thread_debug_t *thread, int stksize)
- uldb_set_thread_type (tdb_thread_debug_t *thread, td_thr_type_e type)
- /* Provide the current LWP id / state (active or not) of a thread */
  uldb_update_thread_lid (tdb_thread_debug_t *thread, lwpid_t lid)
- uldb_update_thread_state (tdb_thread_debug_t *thread, td_thr_state_e state)
Experimentations

- Debuggers:
  - DBX: just set the environment variable `_DBX_LIBTHREAD_DB_OVERRIDE` to the path of the ULDB library.
  - GDB:
    - Create an environment variable `GDB_LIBTHREAD_DB` to select another libthread_db.
    - GDB does not use correctly libthread_db.
    - Rewrite the thread module part of GDB from the GDB Solaris support.
  - All GDB-based frontend: DDD, Insight, Eclipse-Debug, DDT, ..
Experimentations

- Thread libraries:
  - MPC
    - MxN thread library.
    - 600 lines of patch.
  - GnuPTH
    - Mx1 thread library.
    - 100 lines of patch, less than a day.
  - Marcel
    - MxN thread library.
    - 500 lines of patch, few weeks.
Experimentations

- Architectures:
  - Operating system:
    - Solaris.
    - Linux.
  - Processor:
    - Sparc.
    - X86.
    - X86_64.
Conclusion
& Future Work
Conclusion

- ULDB allows to debug user-level thread.
- ULDB is portable:
  - 2 command line debuggers.
  - 4 graphical frontends.
  - 3 thread libraries.
  - 3 processor types.
  - 2 operating systems.
- ULDB provides user-level threads with a low cost:
  - Very low level of expertise required.
  - Small patches (< 1000 lines) for each thread library.
Future Work

- Thread-synchronization debugging support:
  - ULDB API extension.
  - GDB extension (part of the libthread_db not supported yet).

- Debugging support of tasks in OpenMP 3.0:
  - An OpenMP task is almost a user thread.
  - Currently not supported by GDB and the libthread_db.

Contact:

marc.perache@cea.fr, patrick.carribault@cea.fr
http://mpc.sourceforge.net/