Summary

1. Introduction

2. Shared Resource Tracer/Runtime

3. Shared Resource Tracer/External

4. Other Challenges

5. Conclusion
Summary

1. Introduction

2. Shared Resource Tracer/Runtime

3. Shared Resource Tracer/External

4. Other Challenges

5. Conclusion
Introduction

- More than a decade of experience and problem solving
- Lots of feedback from users
- We wish to share this
Introduction

• More than a decade of experience and problem solving
• Lots of feedback from users
• We wish to share this
• Challenges of integrating a user-space tracer in Linux ecosystem
• Apply to other tools and applications
User-space Tracer Properties Trifecta

1. **Integrity** [I] of application
   - Don’t crash the application
   - Don’t corrupt application data
   - Predictable timing impacts on Real-Time applications
User-space Tracer Properties Trifecta

1. **Integrity** [I] of application
   - Don’t crash the application
   - Don’t corrupt application data
   - Predictable timing impacts on Real-Time applications

2. **Reliability** [R] of results
   - Report discarded events
   - Report tracing setup complete or partial failures
User-space Tracer Properties Trifecta

1. **Integrity** [I] of application
   - Don’t crash the application
   - Don’t corrupt application data
   - Predictable timing impacts on Real-Time applications

2. **Reliability** [R] of results
   - Report discarded events
   - Report tracing setup complete or partial failures

3. **Adaptability** [A] of tracer
   - Automatically adapt to the software and hardware environments
   - Minimize the amount of user intervention and configuration required for tracing
User-space Tracer Properties Trifecta (continuation)

- \( R + A = \) user **distrusts** the tracer; won't deploy it
- \( I + A = \) results are **doubted** by the user
- \( I + R = \) increased of **burden** put on the user
Summary

1. Introduction
2. Shared Resource Tracer/Runtime
3. Shared Resource Tracer/External
4. Other Challenges
5. Conclusion
Memory Usage

• Problems
  • Impact on application memory layout
    • Observable effect only when tracing (e.g., observable double-free by application)
    • Reproducibility of memory access patterns
Memory Usage

- Problems
  - Impact on application memory layout
    - Observable effect only when tracing (e.g., observable double-free by application)
    - Reproducibility of memory access patterns
  - Quiescent state cannot be guaranteed with pthread_atfork(3)
    - Endless loop in malloc(3) and free(3)
    - Prevent rendez-vous point with LTTng-UST [1] listener threads
Memory Usage

- Problems
  - Impact on application memory layout
    - Observable effect only when tracing (e.g., observable double-free by application)
    - Reproducibility of memory access patterns
  - Quiescent state cannot be guaranteed with pthread_atfork(3)
    - Endless loop in malloc(3) and free(3)
    - Prevent rendez-vous point with LTTng-UST [1] listener threads

- Current solution [1]
  - LD_PRELOAD lttng-ust-fork.so [2]
Memory Usage

- Problems
  - Impact on application memory layout
    - Observable effect only when tracing (e.g., observable double-free by application)
    - Reproducibility of memory access patterns
  - Quiescent state cannot be guaranteed with pthread_atfork(3)
    - Endless loop in malloc(3) and free(3)
    - Prevent rendez-vous point with LTTng-UST [1] listener threads

- Current solution [I]
  - LD_PRELOAD lttng-ust-fork.so [2]

- Future solution [A]
  - Implement own memory allocator within LTTng-UST
File Descriptor Table

- Problems
  - Single-threaded applications can close all file descriptors
    - Recurrent pattern in daemon
File Descriptor Table

- Problems
  - Single-threaded applications can close all file descriptors
    - Recurrent pattern in daemon
  - Tracer needs to communicate with external processes via Unix sockets
    1. Tracer fails to read/write to its file descriptors (EBADF)
    2. Tracer reads/writes to application file descriptors (recycle of fd)
File Descriptor Table

• Problems
  • Single-threaded applications can close all file descriptors
    • Recurrent pattern in daemon
  • Tracer needs to communicate with external processes via Unix sockets
    1. Tracer fails to read/write to its file descriptors (EBADF)
    2. Tracer reads/writes to application file descriptors (recycle of fd)
  • Similar problem that prevents glibc from using io_uring(7)
File Descriptor Table

- Problems
  - Single-threaded applications can close all file descriptors
    - Recurrent pattern in daemon
  - Tracer needs to communicate with external processes via Unix sockets
    1. Tracer fails to read/write to its file descriptors (EBADF)
    2. Tracer reads/writes to application file descriptors (recycle of fd)
  - Similar problem that prevents glibc from using io_uring(7)
- Current solution [I]
  - LD_PRELOAD liblttng-ust-fd.so [3]
    - Wrappers for close(2), fclose(3), closefrom(2), ...
    - Prevent "close all" behavior on tracer file descriptors
File Descriptor Table

- **Problems**
  - Single-threaded applications can close all file descriptors
    - Recurrent pattern in daemon
  - Tracer needs to communicate with external processes via Unix sockets
    1. Tracer fails to read/write to its file descriptors (EBADF)
    2. Tracer reads/writes to application file descriptors (recycle of fd)
  - Similar problem that prevents glibc from using io_uring(7)
- **Current solution [I]**
  - LD_PRELOAD liblttng-ust-fd.so [3]
    - Wrappers for close(2), fclose(3), closefrom(2), ...
    - Prevent ”close all” behavior on tracer file descriptors
- **Future solution [A]**
  - LTTng-UST listener threads with different file descriptor table
Signal Handling

- Problems
  - Signal number could be used by application
  - Starvation of signalfd [4]
Signal Handling

• Problems
  • Signal number could be used by application
  • Starvation of signalfd [4]

• Solution [I]
  • LTTng–UST does not rely on signals for IPC (Inter Process Communication)
  • LTTng–UST listener threads block all signals
Locks

- Problems
  - Deadlocks caused by lock-dependencies chain (fixed in glibc 2.24) [5]
    - Between tracer and dynamic loader [6]
Locks

- Problems
  - Deadlocks caused by lock-dependencies chain (fixed in glibc 2.24) [5]
    - Between tracer and dynamic loader [6]
  - Current solution [I]
    - Ensure consistent locking order
Locks

- **Problems**
  - Deadlocks caused by lock-dependencies chain (fixed in glibc 2.24) [5]
    - Between tracer and dynamic loader [6]
- **Current solution** [I]
  - Ensure consistent locking order
- **Possible solution**
  - Protect dynamic loader structures with RCU (Read Copy Update) or reference counters
Resources Management After Fork

- Problems
  - Resources can be leaked in child process (if no `execve(2)`)
    - Allocated memory
    - Opened file descriptors
Resources Management After Fork

- Problems
  - Resources can be leaked in child process (if no `execve(2)`)
    - Allocated memory
    - Opened file descriptors
- Current solution [1]
  - `LD_PRELOAD liblttng-ust-fork.so` [2]
    - Wrappers for `fork(2)`, `clone(2)`, `daemon(3)` ...
    - Put LTTng-UST listener threads in quiescent state
    - Release resources within child
Resources Management After Fork

- **Problems**
  - Resources can be leaked in child process (if no `execve(2)`)  
    - Allocated memory  
    - Opened file descriptors
- **Current solution [I]**
  - **LD_PRELOAD** `liblttng-ust-fork.so` [2]
    - Wrappers for `fork(2)`, `clone(2)`, `daemon(3)`...
    - Put LTTng-UST listener threads in quiescent state
    - Release resources within child
- **Future solution [A]**
  - Use `pthread_atfork(3)`
    - Require own memory allocator
Transparent Multi-Threading

• Problems
  • Single-threaded application are not expecting other threads
  • Global states (e.g., umask(2))
Transparent Multi-Threading

• Problems
  • Single-threaded application are not expecting other threads
  • Global states (e.g., umask(2))

• Solution [I]
  • LTTng-UST forks a worker process
Summary

1. Introduction
2. Shared Resource Tracer/Runtime
3. Shared Resource Tracer/External
4. Other Challenges
5. Conclusion
Asynchronous Process Termination

• Problems
  • IPC over shared memory **per-uid**
    • Must be resilient with respect to application terminations
Asynchronous Process Termination

- Problems
  - IPC over shared memory **per-uid**
    - Must be resilient with respect to application terminations
  - 3 steps protocol: reserve, write, commit
    - Issues when an application is terminated between reserve and commit
    - What if the application is simply stopped?

Why not TLS-based ring buffers?

- Do not scale with frequent and short lifetime threads (customer requirement)
- Allocation and publication overheads

Current solution [R]

- Recommend to use per-pid ring buffers

Future solution [R, A]

- Introduce the notion of sub-buffer producer ownership
- Only a single owner by sub-buffer (between step 1 and 3) by tagging it
- Can detect stalled vs terminated owner
Asynchronous Process Termination

- Problems
  - IPC over shared memory **per-uid**
    - Must be resilient with respect to application terminations
  - 3 steps protocol: reserve, write, commit
    - Issues when an application is terminated between reserve and commit
    - What if the application is simply stopped?
  - Why not TLS-based ring buffers?
    - Do not scale with frequent and short lifetime threads (customer requirement)
    - Allocation and publication overheads
Asynchronous Process Termination

- Problems
  - IPC over shared memory **per-uid**
    - Must be resilient with respect to application terminations
  - 3 steps protocol: reserve, write, commit
    - Issues when an application is terminated between reserve and commit
    - What if the application is simply stopped?
  - Why not TLS-based ring buffers?
    - Do not scale with frequent and short lifetime threads (customer requirement)
    - Allocation and publication overheads
- Current solution [R]
  - Recommend to use **per-pid** ring buffers
Asynchronous Process Termination

- Problems
  - IPC over shared memory **per-uid**
    - Must be resilient with respect to application terminations
  - 3 steps protocol: reserve, write, commit
    - Issues when an application is terminated between reserve and commit
    - What if the application is simply stopped?
  - Why not TLS-based ring buffers?
    - Do not scale with frequent and short lifetime threads (customer requirement)
    - Allocation and publication overheads
- Current solution [R]
  - Recommend to use **per-pid** ring buffers
- Future solution [R, A]
  - Introduce the notion of sub-buffer producer ownership
    - Only a single owner by sub-buffer (between step 1 and 3) by tagging it
    - Can detect stalled vs terminated owner
Summary

1. Introduction
2. Shared Resource Tracer/Runtime
3. Shared Resource Tracer/External
4. Other Challenges
5. Conclusion
CPU Topology in Containers

- Problems
  - Per-cpu ring buffers over allocating memory
  - For example, only a subset of CPU used in container
CPU Topology in Containers

- Problems
  - Per-cpu ring buffers over allocating memory
  - For example, only a subset of CPU used in container

- Current state
  - Adaptative per-cpu allocation (single process)
    - Based on RSEQ (Restartable SEQuence) concurrency level (mm_cid) [7]
    - Not NUMA (Non-Uniform Memory Access) aware
CPU Topology in Containers

- Problems
  - Per-cpu ring buffers over allocating memory
  - For example, only a subset of CPU used in container

- Current state
  - Adaptative per-cpu allocation (single process)
    - Based on RSEQ (Restartable SEQuence) concurrency level (mm_cid) [7]
    - Not NUMA (Non-Uniform Memory Access) aware

- Future solution [A]
  - Adaptative per-cpu allocation (shared memory)
    - NUMA aware (RSEQ numa_mm_cid)
    - RSEQ concurrency IDs for IPC namespace
Limited I/O, CPU Time and Persistent Storage

- Problems
  - Tracing when system resources are scarce
Limited I/O, CPU Time and Persistent Storage

• Problems
  • Tracing when system resources are scarce

• Current solution
  • Dynamic filtering
  • Snapshots (flight recorder tracing)
  • Triggers
Limited I/O, CPU Time and Persistent Storage

- Problems
  - Tracing when system resources are scarce

- Current solution
  - Dynamic filtering
  - Snapshots (flight recorder tracing)
  - Triggers

- Future solution
  - Trace hit counters [8]
Structured Instrumentation in Runtimes Other than C

- Problems
  - Structural tracing in runtimes other than C/C++
    - Python
    - Golang
    - Java
    - Javascript
Structured Instrumentation in Runtimes Other than C

- **Problems**
  - Structural tracing in runtimes other than C/C++
    - Python
    - Golang
    - Java
    - Javascript
  
- **Future solution [A]**
  - Use ABI proposed by libside [9]
Summary

1. Introduction
2. Shared Resource Tracer/Runtime
3. Shared Resource Tracer/External
4. Other Challenges
5. Conclusion
References 1


References II


Questions