Embedded-Appropriate Crash Handling

Tim Bird
Sony Network Entertainment

Agenda

- Characterize the problem

- Anatomy of a crash

  - Linux features to leverage

- Existing approaches

- Describe my solution

- Point to some resources

The problem

- ## Stuff crashes

  - By Stuff, I'm talking about user-space programs

  - Kernel crash handling is a separate topic

- Developers and support personnel want as much information at possible to fix problems

- Various attributes of embedded products make this interesting

- Tradeoffs required

Information to report

- Need to provide enough information to figure out what went wrong

- It is easy to report instruction that crashed

- Harder to report what happened leading up to that instruction

- Default method is to dump the whole "core" of the file, and let the engineer use a debugger

  - Even this doesn't capture the entire machine state - only the state of the one process that crashed

- This doesn't include history

2/1/12

Embedded issues

- Constrained memory, storage and CPU

- Fixed workload

- Limited or intermittent connectivity

- Limited user interface

- Need illusion of continued operation

- Off-device processing

Constrained space

- Memory, File system, logs -- pretty much everything

  - Affects space for storing crash information

    - Have worked on device with 4M RAM

    - On TV devices, had a 64K exception reporting budget

- No symbols on the target!

- No tools on the target!

- Limited logging

  - Some embedded systems even turn off printk!

- Reports need to be concise, but still provide enough information

Constrained CPU

- Embedded processors are often under-spec'ed

  - Especially at hardware-driven companies

- No extra cycles to dedicate to runtime analysis and logging

  - Tracers are almost never an option

2/1/12

Fixed workload

- Some embedded products have fixed software

    - Have a fixed set of processes

    - May have deterministic pids

- May not need to save some information

Sparse connecrtivirty

- Some devices don't have regular connectivity

  - Report may be saved a long time

- Crash may be un-noticed to user

  - E.g. process crash on TV - looks like video hiccup

- When the crash report is transmitted logs have cycled and are useless

  - Need to preserve some log info at time of crash

- How much to preserve at crash time?

- How persistent to make the logs and reports

  - ramfs vs. long-term storage

Limited user interfaces

- Devices with limited interfaces may be difficult to get permission

  - E.g. Walkman audio player

- Security/privacy issues with logs and crash_reports

- Need user permission to send crash reports

Off-device analysis issues

- ## What can be deferred to off-device, post-mortem analysis?

  - Biggest issue is mapping addresses to symbols

  - If you save whole stack, you can do unwinding on host

- ## Must ensure to keep symbols around on the host

  - My experience is that you frequently can't find or rebuild exactly matching binaries

    - I know this is terrible

Fixed-up mapfile generator

- Wrote special symbol converter with fuzzy matching

  - Em-log-process

- Can produce fixed-up maps from stripped binaries (from target) and unstripped binaries generated on host

  - Functions shrink or stretch only a little bit between versions (size locality)

Anatomy of a crash

- When a program crashes, kernel sends signal to dying process

  - Signals which cause a core dump are specified in 'man 7 signal'

  - Can perform some functionality, before process death

- After signal handler returns

  - Kernel may create a 'core' file

    - Depending on settings – 'man 5 core'

Getting control when a crash occurs

- 1) While process is still active

  - Signal handler

    - Runs in process context

      - Can run on alternate stack

  - Memory state may be corrupted

    - It may be unsafe to proceed internally

  - Can contact an external agent

- 2) From kernel, right before exit

Where to get information to save

- /proc

- Ptrace

- Logs

- Traces

/proc

- ## Is available during signals, as well as during core generation

  - ### /proc is present as long as process has not been killed

- ## Stuff from /proc

  - ### maps, command line, stat, open files

ptrace

- ## Registers

  - Including PC, SP, link register (return address)

- ## Stack, text and data

  - Including memory maps for shared libraries

- ## Also ELF sections

  - Used for unwind tables, possibly symbols

Logs and traces

- Logs
  - /dev/kmsg
  - Syslog
  - Android log?

- Traces
  - Strace
  - Ftrace
  - Perf
  - Lttng buffer

- Need to filter to conserve space

Approaches to Crash handling

- Core files

- Desktop crash handling

- Android debuggerd

Core files

- Linux creates a file with a snapshot of the memory image of the dying process

    - Intent is to allow use of a debugger on that image

- If your process has weird permissions (is suid) a core is not generated automatically

    - Configure /proc/sys/fs/suid_dumpable to fix

- /proc/sys/kernel/core_pattern

    - Originally used to control name of core file

    - Since 2.6.19, can specify a 'pipe' to send the core file to

        - Pattern string starts with '|'

        - Kernel can pass process information to core file handler with command line args

- Core file handler can access /proc and use ptrace on dying process

Desktop crash handling

- ABRT – Automatic Bug Reporting System

  - Modular, with plugins for different languages (C, Python), different reporters, databases for crash reports

  - Crash reports have lots of information, about kernel, package, machine environment

  - abrt-gui – Can view crash database, manipulate reports, configure plugins

- Needs separate '–debuginfo' packages installed

- Uses sqlite backend for crash data

- Too heavy-weight for embedded

Android debuggerd

- Very cool embedded-specific crash handler

- Used on all android devices

- Crash report data placed in log and in tombstone file

- Debuggerd also facilitates connecting debugger to dying process

    - Can halt and wait for gdb to attach to the process

- Is Apache-licensed

Debuggerd (cont.)

- ## How does it work:

  - Debuggerd is a crash-handling daemon that is always running

  - Adds default signal handler to each process, which handles any signals that generate core

    - Included in bionic – every application gets it

  - Signal handler captures deadly signal and contacts debuggerd

  - Debuggerd records information using ptrace (registers, stack, memory areas), and /proc

- ## Is aware of emulator

  - Does some extra stuff if running under emulator

Miscelaneous features

- Has builtin ARM stack unwinder for generating a backtrace

- Automatically rotates a fixed number of crash reports

Some notes on unwinding

- Unwinding = processing stack and memory image to create a backtrace

- Backtrace is very compact - summarizes stack information nicely

- Local variables usually not available

- Different methods available, depending on compilation flags

Unwinding methods

- Frame pointers

- Unwind tables

- Virtual Machine

- Best-guess

Unwind methods details

- ## Frame pointers

  - Stack frame information maintained by runtime code, on stack

  - Requires runtime overhead in time and space

  - Not often used (-fomit-frame-pointer compiler flag)

- ## Unwind tables

  - Compact information about unwinding stored out-of-band

  - Requires space overhead, but no runtime overhead

  - Use –funwind-tables

  - See https://wiki.linaro.org/KenWerner/Sandbox/libunwind

Unwind methods details (cont.)

- Virtual machine

  - Emulate processor to determine control flow

  - Public domain McTernan ARM unwinder

  - No per-process overhead

  - May not be accurate

- Best-guess

  - Match stack values with possible call sites

  - No overhead

  - Least reliabl

Crash Handler

- New crash handler written by Sony (Tim)

- Based on debuggerd from Android

- Implemented as a core file handler

- Writes crash report to a "tombstone_0x" file in /tmp/tombstones

  - Writes information from /proc, ptrace, and kernel log buffer

- Also writes some information to the kernel log

Crash Handler (cont.)

- Implements a compact flash journal

  - <4k text summary of crashes since inception

- Can 'install' self (set core_pattern)

- Currently implements 2 unwinder methods

Miscellaneous hard-won details

- Must read /proc before PTRACE_ATTACH

- Had to patch kernel to allow ptrace for process in STOPPED state

  - Still working on mainlinable code for this

- Kernel releases process memory before core-handler exits

  - Causes race between ptrace accesses and kernel

  - Set core_pipe_limit to 1 to cause kernel to wait

  - Must access memory with ptrace before reading memory image on stdin

Miscellaneous compiler flag notes

- -mno-sched-prolog

    - Avoid merging function prolog with the rest of the function body

    - Should make prologs more uniform and detectable by crash_handler

- -fomit-frame-pointer and –mapcs-fram

    - Adjust stack frame layout and register usage

- -mpoke-function-name

    - Add function name before each function in text segment

Ideas for the future

- Crash-intelligent logging

Crash-intelligent logging

- Have program that has crashed log more information, in case of subsequent crash

  - Adjust log level or verbosity

  - Turn on tracer

- Turn off automatically after some period, for end-user devices, to reduce overhead

Some random issues

- Extra logging may make the bug go away

- Need a timeout, to avoid putting machine into persistent high-overhead state

  - Under normal circumstances, don't want every process to be verbose

    - That would use up logging space and crowd out other messages

- Application would need to voluntarily adjust logging level

- Need user permission for extra overhead?

To do:

- Find a spiffy name:

  - tow_truck?

  - grim_reaper?

- Resolve issue with required kernel patches

- Make a library routine for querying journal

- Polish up the code

2/1/12

Download

- Source is available on elinux wiki at:

  - http://elinux.org/Crash_handler

Questions?

<tim.bird (at) am.sony.com>