Endurance measuring

Author:	Eero Tamminen <eero.tamminen@nokia.com>
Scope:	Maemo v5 system and application analysis
Date:	2009-11-16
Status:	Draft

November 15, 2011

Abstract

The aim of endurance measurement is not (just) to detect the problems, but to get them fixed. Problems that cannot be replicated, cannot be fixed.

Re-producing endurance problems is not feasible schedule-wise and developer effort required even just to *test* whether problem is reproducible, may be too time consuming[1].

If the measurements don't indicate the correct component for the problem, there will be additional time wasted when the problem is moved between components[2].

*Only with sufficient measurements produced when endurance problem occurs, the problems can be fixed. Information that there is a problem, is **not** enough!*

[1] If triggering the bug took a week, developer can start assuming that the problem is not reproducible only after a week, and with confidence he can say that it's not reproducible only after several weeks of testing (or several developers testing for a week).

[2] Which involves the corresponding developers searching for information about suitable debugging tools, installing them, re-compiling dependent libraries, and finally noticing that the problem is not in their component before moving it to another component. Because this takes time that they don't have, they don't start this kind of things immediately. Result is huge waste of time.

Legal notice

Disclaimer

License

Changelog

TODO Add sp-rtrace & functracer. Document sp-memusage mallinfo thing. Consider la-trace. How to set clock in Fremantle from cmdline?

2009-11-24 Add license page

2009-11-16 List Maemo releases in Terminology. Listed issues where UBIFS is improved over JFFS-2. Mention file system usage issues on system updates. Discuss FAT and its corruption issues. Changed most mentions of Flash into (root) file system. Folded the separate UI reliability document to a new UI reliability issues section. Minor text fine-tuning

2009-05-22 Minor text improvements + additional TODOs

2009-04-30 Moved open/resolved issues to appendix, TODO and reference fixes

2008-09-16 Fixes to debugging sections and cleanup of previous changes

2008-09-05 Major updates as part of publication cleanup and IT2008 updates: removed useless risk matrix, items discussed in the performance measurement document and things implemented into sp-endurance, sp-memusage and sp-rich-core tools. libleaks or functracer?

2007-02-16 Major updates for IT2007: added sections on solved issues and sp-endurance usage, added notes about maemo-summoner and a SMAPS helper script, removed automatic backtraces on device section (it's not currently feasible on ARM) and added debugging issues appendix

2006-03-24 Updated the document for IT2006 in regards to open issues and how the problem detection and measurements are done. Updated instructions, references and URLs. Removed old/obsolete information

2005-12-21 Recommend first leak testing to be done on x68 with Valgrind. Everything else is waste of time because of the large amount of false positives reported by the other tools. Added note about X server resource usage and more URLs and document IDs

2005-12-20 Added notes about use-time, system V IPC resource measurements, script for endurance measurements and section on randomized tests

2005-12-15 Added Proposed measurements section and fixes according to comments. Script fixes

2005-12-13 Added out-of-scope, problem occurrence frequency sections and other updates to the overview and detection sections based on feedback. Moved battery removal issues to out-of-scope (for SW endurance). Added risk analysis section

2005-12-09 Wrote section on detecting and measuring the problems, major update to fixing section, improved subsection headers, added abstract, other minor updates

2005-12-01 Started the document

Contents

1 Introduction

This document describes the potential *software* endurance problems, how they can be measured and problems in measuring. The document is composed of the following sections:

- Overview of endurance measuring, see section 2
- Categorization of the endurance problems, see section 2.3
- What are the known endurance problems, how user can recover from them and the problem risk analysis, see section 3
- How to detect & debug and measure these problems, see section 4
- How to locate and fix these problems within the components where they are detected, see section 6

1.1 Target audience

Target audience of this document is system engineers and anybody who is interested about the possible software endurance problems and how to handle them.

1.2 Terminology and typography

Current Maemo release names are:

Harmattan Maemo v6, not yet released

Fremantle Maemo v5, released in 2009

Diablo Maemo v4, released in 2008

Earlier Maemo operating system releases used names like IT2005 which tell in which year they were released.

References to outside documents are marked like this: [Debugging Guide]. The references are listed in section 6.5.

Commands run in shell are marked with `monotype font`. In the commands *PID* should be replaced with the process ID of an appropriate process.

User refers to a typical device user.

2 Overview

Endurance means that software behaves consistently over a long period of (use-)time without functionality loss or performance degradation. The emphasize on long period of time has several implications:

1. Issues are hard to re-produce because in practice it's impossible to re-construct everything that user has done over a large period of time before encountering the issue. Users don't remember (potentially) crucial details after long period of ad-hoc testing
2. There are no clearly defined pre-conditions for the bugs, as the necessary pre-condition(s) can be determined properly only after the reason for the bug has been resolved from the code inspection
3. Endurance issues are often impossible to verify properly until the reason for the issue has been resolved

After the reason for the problem is found, the events triggering it can be simulated in much shorter time.

When the problem can be (quickly) re-produced, it's not anymore an endurance problem, just a normal bug.

2.1 Measuring objectives

Because of the endurance problems re-reproducibility issue, measurements done during the testing have to:

- Provide enough details to replicate the pre-conditions for triggering the issue, and/or
- Clearly identify the component responsible for the issue so that amount of code to inspect is reasonable

Therefore the point of the endurance work is to provide testers and developers the tools and processes which produce these measurements.

2.2 Out of scope

Following are out of scope for this document:

- Problems with hardware endurance such as low battery, flash wearing, plugging things in&out etc.
- Measuring factors affecting the device use-time. Discussed more in [Performance measuring and analysis].
- Endurance regression testing. This document doesn't concern problem fix followup; only detecting, measuring and fixing them.
- Power users' use-cases. Power users can enable device R&D mode, do things from the command line or as root and therefore break the device in many interesting ways that normal users cannot or won't.

2.3 Problem categories

Endurance problems can be divided into a couple of categories (with decreasing severity):

1. System functionality or performance significantly decreases after use and cannot be recovered by the user (except by reflashing the device)
2. Specific functionality or its performance significantly decreases after use and cannot be recovered by the user
3. Loss of user data
4. Functionality or performance significantly decreases after use, but it can be recovered with reboot
5. Functionality or performance significantly decreases after use, but user can recover it easily without reboot
6. Decreased device use-time

Only way to create degradation in software functionality or performance that persists over reboot is to modify state that persists over reboot. RAM contents don't persist over reboot, disk contents do. Therefore categories 1-2 problems are related to files and file system and problems in categories 4-5 are typically related to resource leaks, memory fragmentation and memory usage in general.

User data loss that occurs as a result of system functionality/performance degradation over time[3], are category 3 problems. They can be results of both disk and memory usage problems (disk full, out of memory, no free file descriptors).

There are also bugs that are very hard to trigger, and therefore they usually appear only after a long use-time. For example low memory situation makes the device slower and can therefore be only way to trigger timing related bugs in normal use. If process termination in such conditions results in other processes terminating, and especially if that's a user visible issue, the device is be rebooted automatically[4] to return it to better working condition.

Some applications can get slow although device itself still works quite well. These problems are too application specific to be be discussed in this document.

2.4 Problem occurrence frequency

For risk analysis the device use time can be divided into few significant categories:

Hours Time it takes until full battery runs out when device is *continuously* used

Days Time it takes until full battery runs out when device is *idling* after the device has been used

[3] Other types of data loss are not endurance problems.

[4] The automated reaction naturally needs to be optional for debugging purposes.

Weeks Time the device can be used normally without rebooting just by recharging it occasionally

Months Prolonged time the device can be used without rebooting just by recharging it occasionally. Assumes battery nor device performance doesn't degrade significantly to force user to an early reboot and that user doesn't do operations requiring reboot (SW image update, backup restore, language change)

Years The device lifetime, includes numerous reboots in addition to recharging

Time when the device is powered off doesn't count because that doesn't exercise its software or file system, and has minimal or no effect on hardware.

3 Known endurance problems and recovering from them

This section describes potential endurance problems in the present and past Maemo releases and how system and user recover from them. Some of the larger old issues and their solutions are listed as reference for future systems designs.

3.1 System problems from which user cannot recover the device

Complex desktop like systems such as Maemo need to do writes to the file system at bootup, so there always needs to be some free space on the root filesystem for bootup to work properly. There's some flash space reserved by the file system which only the *root* user processes can fill, non-*root* user processes cannot fill the root file system completely.

If a *root* user process (such as package installation scripts called by `dpkg`) manages to fill the root file system completely, the device can end up in a reboot-loop. This is the worst category 1 problem, *user cannot correct it*. Package manager usually recovers from installation errors by removing the failed package, but this cannot always free (enough) space (see also 3.3).

If a process involved in the device startup and covered by the SW watchdog doesn't handle device root file system being full (to an amount non-*root* user process can fill it) and crashes, this causes also a reboot loop.

It's also possible that some cache etc. files grow without limits (see e.g. 3.1.2). If user doesn't see and cannot remove these files, *user cannot correct the issue* and eventually such files will fill the file system and device will stop working.

3.1.1 Problems specific to JFFS-2 (Diablo and earlier releases)

When the JFFS-2 partition starts to get full, the device performance will start to significantly degrade. There are several factors contributing to this:

- JFFS-2 keeps the whole file system structure in memory. This takes more RAM (and slows down mounting)
- Does too optimistic free space estimation for user-space (which causes more out-of-space issues) and reserves fairly little space for itself (which make next issue worse)
- Less space for garbage collection

UBIFS replaced JFFS-2 file system in Fremantle. It fixes or at least significantly improves these issues. For more details, see UBIFS documentation `http://www.linux-mtd.infradead.org/doc/ubifs.html`.

3.1.2 IT2005 specific problems

In IT2005 image thumbnails were never removed, but in later 770 device releases File manager takes care of that by removing the corresponding thumbnail when local file is deleted, and purging old thumbnails regularly.

As mentioned above, JFFS2 stores the whole file system structure in memory. Older JFFS-2 memory usage increased and performance decreased dramatically with very large (megabytes) files which were written in very small pieces[5]. The effect was worst if some process had the file open[6], but only *removing the file corrected the problem* completely. Many small files were less of a problem than one large one because their fragments are more likely to aligned and it's less likely that all of them are open at the same. Such files are usually generated to places where user cannot remove them, and if they are not removed automatically, they are category 1 problem.

In releases after IT2005 this is solved by JFFS2 rewriting too fragmented (4kB) parts of the files as they get filled.

3.2 Specific use-case problems from which user cannot (easily) recover the device

3.2.1 Corrupted FAT file system

User data is on a FAT file system for compatibility reasons. FAT itself isn't a robust file system (no journaling etc), but additionally it's also exported out of the device over USB as a mass-storage device, so that its contents can be changed from the PC, outside of the device control.

While the FAT file system contents are being changed from the device[7], or from the PC over USB, following things can corrupt its contents:

- Unplugging the USB cable or rebooting / powering off the device without using the "Safely remove" functionality on the PC first
- Removing the battery (or even just removing the battery cover)
- Device HW watchdog rebooting the device (except for USB mass-storage case, SW watchdog reboots are safe)

User should get a notification about the file system corruption when system first notices it, but if user happens to miss that, applications may not just work correctly due to the file system being read-only (to prevent further damage).

I have listed this as category 2 as it's not obvious or easy for a normal user to recover the device functionality and it's possible that many users will encounter this issue. Recovery is done *by repairing the file system from the PC or by formatting it on the device,* of which latter option loses user's data on that file system.

[5] For example syslog writes files line at the time.
[6] The memory effect is ~4x worse when the file is open.
[7] directly by user, or indirectly e.g. by media stream buffering/caching

3.2.2 Full file systems

There may be (3rd party) applications which don't handle properly full file system; they write data or configuration files at startup and either crash, don't work properly or fail to report an appropriate error message to the user. In worst case they lose user data.

I have listed this as category 2 as it's not obvious to the user that the functionality may be recovered *by removing some files to make more space.*

3.3 Problems from which user can recover with a reboot

3.3.1 Resource leakage in system processes

If there are even small memory leaks in always running processes, eventually they will make the device run out of the memory. In this case either the offending process is killed or if it's OOM-protected[8] system process, some other process(es) are killed. This may (eventually) cause also a device reboot. File descriptor and System V IPC (shared memory, semaphore etc) leaks are more rare, but if the device runs out of file descriptors or shared memory (or its handles), it doesn't work properly.

Except for shared memory segments which have file system semantics, these problems go away when the offending process is re-started. To restart an always running system process, *user needs to reboot the device*.

System updates also cause a problem which can be (completely) solved only by a reboot. If files (such as libraries or data files used by system processes) are updated, but there are processes still keeping them open, file system needs to keep the old file contents around until those processes either terminate, or otherwise (are forced to) release the old versions of the files. If such files are replaced/modified when packages are removed, it's possible that the package removal actually decreases disk space instead of freeing it, at least until next reboot.

3.3.2 UI deadlocks

There are several things UI design needs to take into account so that it doesn't ruin the device relability:

- Provide ways both to terminate and switch away from frozen (fullscreen) applications. Even a well-tested, pre-installed application like Browser[9] can freeze or deadlock, not just 3rd party programs. Frozen application may also ruin device use-time if it's using CPU, not just frozen to user.
- Make sure that applications cannot block above kind of system functionality, by grabbing input (without a way to break the grab) or by (accidentally) stacking their windows above system windows etc. This is a basic requirement for well behaving UI libraries, not a guarantee against intentionally malicious programs.

[8] Process owned by *root* or which `/proc/PID/oom_adj` has value of -17.

[9] Browser loads and executes "random" programs from the internet in several different scripting languages / virtual machines with "random" data while going through millions of lines of code.

- Way to cleanly switch the device off in case its UI against all odds freezes. See 3.2.1 for reasons.

Whether these are properly designed can be checked just by sending SIGSTOP to appropriate UI processes and trying to close the application or switch the device off.

3.4 Problems from which user can easily recover without a reboot

Severity of resource leaks in applications depend on how large they are and how often they can happen (e.g. in a repeating network error case) and in how common use-case they are.

In applications where memory usage is highly dynamic (such as browser), it's also possible that the memory used by the application cannot be returned back to system because of memory fragmentation once the the application returns to a state where it wouldn't need all that memory. The effect of this is very specific to how such a process does in-memory caching and what kind of allocator(s) it uses for that.

The problem goes away when *user re-starts the application*.

Some applications might not terminate when user closes them, to speed up their later "startup". *Such applications MUST be very carefully reviewed and tested not to leak any resources.* If such a process can freeze while it's not visible, it's a higher category problem.

3.5 Problems with device use-time

On desktop, games often update the screen constantly in a busy-loop instead of as a response to the user activity. This kind of constant activity should be stopped when the device screen is blanked and otherwise it shouldn't be done constantly but with a short timer. Otherwise device battery drains significantly faster because the CPU cannot sleep.

The problem is not an issue if *user doesn't use such an application*.

It's also possible that some change in the environment causes a process to enter into a state where it uses CPU constantly[10]. If it's a system process, user can only reboot the device to get it's use-time back to normal and even this doesn't succeed if some change in the device data triggers this at bootup[11].

[10] bugs.maemo.org has had metalayer-crawler and Modest bugs on this.

[11] Like having corrupted memory card inserted could trigger in ITOS2006 infinitely looping media metadata collection.

4 Detecting and measuring the problems

4.1 Measurement testing overview

Measurements without a use-case that produced them are worthless. All measurements should be accompanied with as accurate use-case as possible, either one written manually or a log that contains all relevant information to replicate the use-case.

It's not enough to detect that there is a problem, with endurance problems measurements have to point out also what causes the problem. Therefore below are listed first a way to detect a problem, and then a way to detect the cause for it.

4.1.1 Testing for resource leakages

There are two kinds of resource leaks:

- Resources that application loses, and
- Resources that application "collects" unnecessarily

Former type of memory leaks can be reliably detected with the Valgrind tool (works only on x86, see section 6.2.3).

Latter type of resource leakage and other than memory leakage (disk space, file descriptors...) can with current tools be detected only by indirect observation i.e:

- First doing the use-case once or twice to initialize everything that requires initialization. Note that the size of usage increase this incurs depends from everything that is done before the use-case[12]
- Then repeating the same use-case multiple times
- And observing whether the component resource usage increases on the use-case repeats

If the use-case repeat times are exponential, they reveal the leakage trends more clearly and discount better the effect of buffer/cache size increases etc. that happen only occasionally. In most cases 2, 4, 8 and 16 successive repeats are enough to clearly indicate the leak, but sometimes more (32, 64...) are needed.

Because of the required repetitiveness, for regression testing running of these use-cases should be automated once a suitable set of use-cases is found.

4.1.2 Too large base/default resource usage

First time the use-case is run gives the base resource usage (initializations etc.) needed by the use-case. Because of the limited resources, base memory usage can also be a problem, but that needs careful analysis about proper compromises between performance and memory usage etc., large memory usage cannot outright classified as a bug. For more information, see section 6.2.1.

[12] E.g. whether the font used to draw the UI text is already loaded by something else.

4.1.3 Testing for hard-to-reproduce crashes

Repeated test-cases don't catch hard-to-reproduce crashes, tests that use different timings or in general stress the software extraordinarily are required for this. I.e. "randomized" testing.

Required random elements may be:

- Test order
- Timings
- Input events
- Input data
- Load on the system (CPU, RAM, network usage)

There are some Open Source tools available for fuzzying i.e. randomizing program input and data, but there should be also program specific randomized functionality testing (and manual ad-hoc testing).

What is done needs to be logged along with the measurements for tests to be repeatable later on!

4.1.4 Test automation

Unit tests can be used for non-UI programs and libraries. As there's plenty of documentation on that (and tools like `check` which is used e.g. by Gstreamer), this concentrates on UI testing.

On Maemo there are several ways to simulate user actions without modifying the applications:

- Introspecting the actions available in the application UI through the widget accessibility interfaces and then invoking the actions[13]. This doesn't go through the widget and event handling mechanism and cannot do things like drag and drop
- Synthesizing stylus and key events through XTest Xserver extension[14]. With this it's possible to do everything user does, but the test result correctness verification needs to be done using some other mechanism
- Invoking application services through their D-BUS APIs

Best would be to use XTest to simulate user actions and accessibility APIs for introspecting what actions can be done and to verify the results. At least the synthetized events and their time-stamps need to be logged. Use-case setup might be sometimes easier with D-BUS (e.g. opening browser for a specific URL).

Major point of endurance testing work is to make sure that appropriate measurements can and are being collected while the tests are done, that the number of test repeats can be controlled and that the test-cases are linked to the measurements.

[13] For example with Dogtail or LDTP tools.
[14] For example with Xnee or Xresponse from the Maemo tools repository.

4.2 Detecting FAT file system corruption

If kernel detects problems in file system metadata, it will re-mount the file system as read-only. Kernel can send a message to user-space when this happens. However, as the file system might have already earlier been corrupted and its contents being corrupted more by further writes on it, this leaves something to be desired...

Detecting the corruption earlier requires full file system metadata consistency cross-check. Unfortunately Linux `dosfsck` has multiple issues which make running it infeasible before the device FAT file system is mounted (on bootup or when USB cable is disconnected):

- For the cross-check, `dosfsck` needs the whole file system metadata in memory. For the (unlikely) worst case file system with tens of GBs of data, this requires hundreds of MBs of RAM. If device is already low on memory, the check will cause large amount of swapping or even fail and cause some application(s) to abort due to their memory allocations failing which isn't acceptable.
- If `dosfsck` goes to swap, the checking will be very slow. For the worst case, it can easily take tens of minutes. User cannot wait this long until being able to use (write to) the mounted file system. It will also prevent mounting the file system back to PC, if user forgot something and re-connects USB-cable[15].

So, there's no good solution for this. There are no good and well tested alternatives for FAT either which would be as compatible.

4.3 Measuring system disk space usage increase

Total disk usage can be measured with the following command:
`df`

To see which files get modified by a use-case, use a date few days forward:
```
datetime=$(date -I|awk -F- '{printf("%s-%s-%02d/00:00:00",$1,$1,$3+2)}')
```

And set clock accordingly. On Maemo4 and earlier releases this can be done with:
```
/mnt/initfs/usr/bin/retutime -T $datetime
```

Do your test-case and check which files were modified:
```
find / -mtime -1|grep -vE '/(proc|sys|tmp|dev)/'
```

(You need to set clock forward because Busybox `find` doesn't support better granularity for file modification changes than days, such as the `-mmin` (minutes) option supported by the GNU `find`...)

4.3.1 Detecting file system writes that make the device slower

Currently there's no tool that would give the information about how fragmented a JFFS-2 or UBIFS file systems are.

[15] File indexer can prevent this too if there's lots of files.

However, once there's a list of files that are modified during the use-case, it can be checked whether any of them are written inefficiently / in an way that would slow down the device (i.e. in small pieces) by tracing the process writing the file.

First you need to check which file descriptors the process has open, so that you can discount socket and pipe writes:
`lsof -p PID|grep '[0-9]\+[wu]'`

Then you can check what sized writes the process does during the use-case. Writes which are (considerably) smaller than 4KB are problems if file contains a lot of them (First write() argument is file descriptor and return value is how much was written):
`strace -p PID 2>&1|grep 'write('`

You can use also strace-account `http://www.gnome.org/~mortenw/files/strace-account` to get an overview about issues like this.

With old JFFS-2 these small writes could be even a reliability issue (see 3.1.2), in newer releases they're "only" a performance issue.

4.4 Measuring file descriptor leaks

Count of used file descriptors in the whole system is in the `/proc/fs/file-nr` file. Note that you might need to wait a short while after the last process has exited as kernel might not free the file descriptors immediately. Kernel also allocates the file descriptors in larger amounts so the reported value can be larger than expected.

List of files open by a single process are listed under the `/proc/`*`PID`*`/fd/` directory:
`ls -l /proc/`*`PID`*`/fd/`

`lsof` utility can also be used to list the open files in system:
`lsof -d0-255|sort`

4.5 Measuring system memory leaks

Free memory in the system can be measured from the `/proc/meminfo` file by adding together the *MemFree*, *Buffers* and *Cached* values (and corresponding ones for swap if swap is in use). [Memory usage] package scripts can be used to monitor this.

System System V IPC resource usage can be measured from the files in the `/proc/sysvipc/` directory (see `man 5 ipc`). These report also which files are using the resources. Maximum values for these resources on the device are reported by the `/proc/sys/kernel/msg*` and `/proc/sys/kernel/shm*` files (see `man proc`).

Scripts in the [Endurance measuring] package can be used to store these (and many other) system values while the software test-cases are run.

4.5.1 Kernel memory usage

Although an approximation for the kernel memory usage at boot-up can be deduced by subtracting the `/proc/meminfo` *MemTotal* value from the amount of physical RAM

in the device, there's no really reliable way to tell how kernel internal memory usage changes when the device is actually used[16].

What one can do is to compare the changes in `/proc/meminfo` information against the changes in the processes memory usage information reported by SMAPS (see 6.2.1) and to see whether the missing amount of memory would be found from the changes in the `/proc/slabinfo` statistics about the kernel internal memory pool usage. GNU `procps` package provides `slabtop` tool to monitor `/proc/slabinfo` content changes. `/proc/slabinfo` doesn't necessarily tell where the used/leaked memory goes inside the kernel though, just from what memory pool it was taken.

4.5.2 System user-space memory usage

Easiest way to see into which processes system memory is going, is using `top` (or preferably, the separately installable `htop`). They get their information from the `statm`/`status` files in the corresponding process `/proc/PID/` directory.

However, these measurements are fairly inaccurate because part of the memory reported to be used is shared with the other processes. To measure more reliably where the system memory is really going i.e. what memory is private to given process, you need to read the data from `/proc/PID/smaps` files (see 6.2.1). For checking state of a single process, you can use tools and scripts provided by the [Memory usage] package. On Linux Desktop, the desktop environment system monitor tools can give similar information too.

To get private memory usage information on *all* processes, see `[?]`.

4.6 Measuring process specific memory leaks and fragmentation

For process specific memory leaks more detail is required because SMAPS reports only system view to the process memory usage, not the actual memory usage inside the process heap. Heap can contain data that application has already freed, but hasn't been able to return back to the system due to the memory fragmentation. The data may also be cached or allocated in larger blocks by the C-library or some higher level library to speed up further allocations. Heap size changes are multiples of 4KB (the memory page size).

Therefore the smaller leaks inside the process may show up in system memory measurements only after much larger number of test-case repeats. To find these faster, separate tools and measurements are needed. Easiest way to detect application memory usage leaks is to request memory information from the Glibc allocator `mallinfo()` function before and after each use-case repeat cycle and compare the results. `Mallinfo()` can also provide information necessary for analyzing device memory fragmentation i.e. how much of the memory freed by the application is not returned back to the system.

The reason why fixing even small memory leaks may be important is that there could for example be leaking error handling code that isn't triggered normally but could in specific conditions be triggered very frequently. Worst case is leaking error handling code that loops until the error goes away, as in some cases they don't, at least before the device runs out of memory...

[16] On normal device kernel, there are kernel patches / config options for allocation tracking though.

Error handling may be even 1/3 of mature code base, so testing it properly requires a lot of effort. Static code analysis tools are recommended to finding leaks in those though as leaks in error handling are often fairly trivial i.e. easy to find statically. Best static code analysis tools are commercial and out of scope for this document.

Because repeating the test-cases is time consuming and produced measurements can require extensive analysis for weaning out false positives, it's preferable that first testing is done on x86 to find and fix definitive leaks that are reliably detected with Valgrind, see section 6.2.3.

To see more in detail where the rest of the memory goes, Massif or e.g. Functracer can be used, see section 6.2.1.

4.7 Detecting process crashes and device reboots

Applications and other processes in the device are started through the following services:

DSME Lifeguard Starts system processes and either restarts the process or the device if the service exits, see `dsmetool -h`

D-BUS Starts the receiver for a message if message has auto-activation flag. This is used for starting all applications in the device

Maemo-launcher All built-in applications launched by D-BUS are started actually through this for performance reasons (application name is linked to `maemo-invoker` which asks `maemo-launcher` to fork, dlopen() the application .launch ELF binary, and call its main())

Hildon Desktop UI process which lists the applications and allows user to invoke them. Can also exec non-integrated 3rd party applications directly (instead of through D-BUS and/or Maemo-launcher)

DSME and Maemo-launcher log to the syslog the exit state of the applications run through them so that crashes of most of the applications can be catched. D-BUS doesn't log the exit status of processes started by it, but as all default applications are started through Maemo-launcher, that's not really a problem except for 3rd party applications. Product releases don't have syslog, but it (or `sp-error-visualizer`) can be installed from the Maemo tools repository.

DSME SW watchdog logs also the reason for SW reset (reboot) or shutdown; it could be either user action, battery low, or critical system process crashing. HW watchdog will reboot the device if it's unresponsive long enough. Information about these is available on next boot-up from `/proc/bootreason` and stored into files under `/var/lib/dsme/stats/`.

To find out what within the process is responsible for the crashing, backtraces are needed. There are two ways to get these, getting the backtraces directly from the processes, or enabling core files, see section 6.4.2.

4.8 Detecting polling and busy-looping

See [Performance measuring and analysis] document for more details on this.

5 Endurance metrics

5.1 What information is collected

Here are listed the main endurance metrics measured by the [Endurance measuring] package scripts:

- System uptime, load average, CPU usage, context switches and interrupts:
  `/proc/uptime`
  `/proc/loadavg`
  `/proc/stat`
  `/proc/interrupts`
- System memory usage:
  `/proc/meminfo`
  `/proc/slabinfo`
  `/proc/sysvipc/*`
- Number of used system file descriptors:
  `/proc/sys/fs/file-nr`
- How many file descriptors each process has open:
  `/proc/[1-9]*/fd/*`
- Misc process information (memory usage, signal masks etc):
  `/proc/[1-9]*/status`
- Memory private to each running process as reported by SMAPS:
  `/proc/[1-9]*/smaps`
- Processes CPU usage:
  `/proc/[1-9]*/stat`
- X server resources used by its clients as reported by the XResource extension
- System disk usage:
  `df`
- System network usage:
  ifconfig
- DSME, Maemo-launcher and Glib error messages from the syslog
- DSME restart/reboot statistics

For more information, see the sources of that package.

5.2 When to collect the measurements

It's proposed that the endurance metrics scripts are run:

- After booting

- After the pre-conditions for the testing round are executed (starting an application etc.)
- After one full testing round
- After N additional test rounds, N grows exponentially: 2, 4, 8, 16...
- (After the device has idled overnight with screen blanked after the tests)

6 Fixing/avoiding the problems

6.1 Fixing file leaks

File system problems can be fixed by improving the code of offending processes and libraries:

- Limit their file growth
- Make them robust against file system errors

In general:

- Root processes file write code should be audited very carefully so that they don't try to write files if there's too little space and that they will clean up all their files in all error cases
- All processes should be able to handle file open and write failures gracefully and notify user about it if it's relevant to her
- Total size of all files a process writes MUST be limited when they are where user cannot acces them directly. It would be preferable if there's some way for the user to remove them (e.g. "Clear cache" option) and the space they take is listed separately in the memory applet

[Maemo MXR]tool can be used to find code accessing a particular file.

6.2 Fixing memory and file descriptor usage and leaks

There are several ways to improve the memory situation:

- Reduce system memory overhead
- Reduce generic application process memory overhead
- Reduce memory usage in specific use-case, and
- Fix all resource leaks

A rough approximation on how the application memory usage changes at real-time can be seen with:
`top`

6.2.1 Memory usage reduction

Best tools currently available for tracking where exactly the process memory goes are:

SMAPS Smaps is 2.6.14+ kernel feature which reports reliably memory that is private to the process or swapped out from it[17]. Whole system SMAPS data can be collected with `sp-smaps-measure` package (used by [Endurance measuring]) and post-processed by the [SMAPS scripts] package. [Memory usage] package scripts give an overview of system and process memory usage using SMAPS.

Sp-endurance Sp-endurance is a package for the target device which collects endurance information, like SMAPS data. Its post-processing scripts visualize the resources usage changes and list logged errors. It can be used to analyze reliability issues as it *detects* resource usage changes. However, it doesn't help in *analyzing* the cause for these resource leaks. See 6.2.2

Massif This is a Valgrind plugin i.e. x86 only. It gives an ASCII graph of the application data+heap usage over time and function callgraphs to the largest allocations at suitable intervals (including the peak allocation). This is easiest to use of the tools and part of the SDK. It's best to use Massif through the `run-with-massif` helper script from [**?**]. This flags some functions from the Maemo libraries as allocation functions which makes the callgraphs more readable and knows about maemo-launched applications

Kmemleaks A patch to kernel which will report (some of the potential) kernel memory leaks

Kmemtrace A 2.6.30+ kernel feature to trace kernel memory allocations. This gives information on how and how much memory kernel itself uses. However, Maemo kernel is older than this

Except for `/proc/slabinfo`, there's currently no tool in Maemo to track where kernel memory goes inside the kernel, how much each kernel driver accounts for etc. Device driver memory usage could be tested by disabling each of the kernel devices and configuration options one at the time, boot the device with new kernel and check the device memory usage.

6.2.2 Finding resource leakage and errors with sp-endurance

Note: Before using [Endurance measuring], it would be best if you automate your test-cases so that they are easy to repeat (see section 4).

Install *sp-endurance* package to the target and *sp-endurance-postproc* package to the PC (both are available from the repositories).

To collect endurance data, do following after each test-case repeat on the target:
`save-incremental-endurance-stats <test-case name> [step description]`

Then transfer the data stored under the *test-case-name* directory to the PC and run following:
`parse-endurance-measurements <test-case-name>/[0-9]*`

This will produce `endurance-report.html` report which contains bar-graphs of the device and processes memory & CPU usage changes, tables about other resource usage and process changes and lists of the logged errors etc.

[17] But not whether the swapped out memory was private to process or shared with other processes.

6.2.3 Finding memory and file descriptor leaks with Valgrind

Locating memory and file descriptor leaks within a process is easiest with Valgrind.

Just run the leaking process with:
`valgrind --tool=memcheck --num-callers=50 --leak-check=yes --track-fds=yes /usr/bin/app`

For Maemo UI applications it's best to do this with the `run-with-memcheck` script from [**?**] package as this automates some things that are needed when using Maemo applications with Valgrind (running maemo-launched binaries with `maemo-summoner`, exporting `G_SLICE=always-malloc` for glib).

This will give you a text report of all the allocations and file descriptors which process has lost when the process exits (or is killed). See also [Debugging Guide].

It's better to kill the application (with a signal like `SIGUSR2` which processes don't normally catch but Valgrind *can*) instead of closing it normally as usually you're not interested about leakage that application may have at exit[18].

6.2.4 Finding memory leaks with Functracer

There can also be leaks within the use-case into which process still retains a pointer. I.e. they are not lost, just not freed. Memcheck is not very good at pointing those out and Massif is useful for that only if the leaks are large. Functracer can be used to find also small leaks.

Using Functracer [Functracer] can either start a process directly or attach to an already existing process. It then traces all allocations done by the process and collects their backtraces. This tracing can be started when `functracer` is started:
`functracer --file --start --depth=8 xrestop`

and/or toggled by signaling the traced process with `SIGUSR1`.

`functracer` doesn't by default resolve the function names in the backtraces for performance reasons and because `functracer-postproc` can resolve them better:
`functracer-postproc --leak allocs-4254.0.trace`

To get useful backtraces (more than 1-2 frames on ARM) and for resolving function names, matching debug symbol packages need to be installed for the binary and all relevant libraries though.

As this tool just reports allocations and post-processing only removes non-freed allocations from the list, it's not possible to know whether they are actually leaks. For this the use-case needs to be repeated multiple times to see whether the amount of allocations increases correspondingly:
`functracer -f -s app &`
<check process PID with 'ps'>
`kill -SIGUSR1 PID`
<do the use-case first time>

[18] Once the process disappears, kernel automatically cleans all memory used by it so freeing memory just before exit() just slows down program exit.

```
kill -SIGUSR1 PID
kill -SIGUSR1 PID
```
<do the use-case multiple times>
```
kill -SIGUSR1 PID
```
<repeat above three steps with exponentially increasing number of use-case repeats>

Then you can post-process the resulting allocation logs and compare them with each other to see whether there's leakage.

6.2.5 Finding application resource leaks on the X server side

Applications can keep a lot of their data on the X server side[19], and application X resource leaks are seen as X server memory increases. These can be found with the *xrestop*[20] tool which works similarly to `top`, it just reports X client resource usage[21] instead of process memory usage.

6.3 Dealing with unavoidable memory leaks and fragmentation

6.3.1 Detecting memory fragmentation

Memory fragmentation can be detected by calling Glibc mallinfo() function and comparing amounts of allocated and freed memory[22].

6.3.2 Using different allocator

Applications which have very dynamic and complex memory allocation patterns can use a different memory allocator which handles the application specific allocation needs better than Glibc ptmalloc (generic allocator in C-library) or Glib Gslice allocator (improves allocation performance with threads). This can help with the memory fragmentation but it doesn't completely fix it.

For example in IT2005 the BSD allocator was used in the Browser. The base memory usage with BSD allocator is significantly larger than with the default Glibc allocator and it's somewhat slower, especially in freeing the memory, but it's more resistant (uses `mmap`() and `madvise`() instead of heap, Glibc uses `mmap`() only for larger allocations) to memory fragmentation which with Glibc is much worse problem, *especially* when the process leaks memory.

6.3.3 Workarounds (mainly in ITOS2008)

Sometimes it's not possible to fix memory usage growth induced either by memory fragmentation or leaks.

[19] For example ITOS2008 World Clock application uses several times more memory from the X server for it's world map than what is private in its own process.

[20] See: Maemo tools `http://maemo.org/development/tools/`

[21] It has some issues in reporting Composite extension related images.

[22] So far only problems in this are have been Browser/Flashplayer (do complex allocations and caching) and D-BUS (has complex, badly performing allocator)

This can be worked around by making the components re-startable. This works already for most applications through state saving and background killing mechanisms explained in the Maemo Architecture document ([Maemo architecture]), but all processes using a lot of memory don't implement it:

Browser Web-service pages and AJAX applications can have state also on the server side which user would lose when the Browser connection breaks at process exit.

Home In ITOS2008 and earlier releases Home was part of the Desktop process along with the Task Navigator and Status Bar (to save RAM). Re-starting it takes a long time due to all the graphics and applets and would be too user visible.

There are also several system UI processes which are restartable, but do not participate in background killing because they need to be "instantly" available. These are **System UI**, **Connectivity dialogs** and **Input method**.

There hasn't been any issue in their memory usage, but if there were services that do have issues, they could be modified to exit() themselves when they:

- receive memory low notification,
- notice that their memory has fragmented too much[23], and
- they're not visible.

Lifeguard (software watchdog) would then restart them.

It's also possible that kernel memory fragments. Upstream kernel is moving to 4KB stack (from 8KB) which could help in this and there are kernel facilities for monitoring its stack usage.

6.4 Fixing process crashes

See [Debugging Guide] for the instructions and issues that there are on debugging in the device and Maemo Scratchbox environment.

6.4.1 Getting crash backtraces from within the processes

On x86 the crashes could be catched and a backtrace printed with the Glibc `backtrace()` function. It works in in most cases on Fremantle too, but not on earlier releases.

However, catching SIGABRT etc. signals within the process and doing other operations in the signal handler context can ruin your core-dumps completely (e.g. if you catch abort Glibc does when it detects memory corruption) so this is *strongly* discouraged. Also, Glibc `backtrace()` cannot give completely backtraces, for that you need to resolve them with debug symbols (sometimes code needs even to be re-compiled & re-run to get full function frames needed for this).

Therefore it's best to use Gdb and provide full debugging symbols anything for which one needs backtraces (to debug crashes, performance, memory usage...).

[23] I.e. amount of freed memory reported by mallinfo() that's not returned to the system has exceeded certain limit.

6.4.2 Getting core dumps from crashes

In the Maemo releases core dumping can be enabled just by adding `core-dumps` directory to a memory cards root directory (that is is set to the core dump file name pattern in `/proc/sys/kernel/core_pattern`) or in Fremantle release under (`/home/user/`)`MyDocs` directory.

It's recommented to install also [Rich core] as that adds some additional system information to the core dumps and compresses them to save space and speed up large core dump saving. The data from the "rich cores" can be extracted with `rich-core-extract`.

6.4.3 Finding the crash reason without a useful core or backtrace

If there's no core file but crash is re-producible, then finding it's cause should be easy. You could either:

- Run the code under *Gdb* on x86 if the problem is reproducible also on x86. This is the preferably method as you have enough memory to run a debugger with all the debug symbols and unlike on target the backtraces work without re-compiling everything with `-g`.
- Run the code under gdbserver on the device and connect to it from the PC with Gdb, for more information see Scratchbox.org `http://www.scratchbox.org/` documentation. If you find gdbserver awkward to use, you might also try swapping to an MMC card or USB hard disk so that device will have enough memory to run Gdb.

Note: The above memory issues should be mostly resolved with Fremantle release.

If what Gdb shows when the crash happens doesn't make sense, the problem could have happened earlier or the process stack is smashed. In this case it might help to use *Valgrind* (described above) on x86 to see whether there are any memory access errors before the crash and fixing those first. Valgrind can launch Gdb when it detects a problem so that you can examine the process state.

On x86 GCC also supports the `--fstack-protector` option which can be used to find (some) stack overwrite issues. For more information, see the GCC manual page.

For issues on debugging in this environment, see [Debugging Guide].

Sometimes *Duma*[24] malloc debugger library produces more useful results than Valgrind. Because of memory requirements it may also be easier to run on x86.

Any Glib errors reported from the application should be fixed before wasting time with debugging. Critical warnings indicate the application internal state isn't anymore to be trusted. Glib errors can be catched e.g. by setting a breakpoint to the `g_logv()` function (which outputs these Glib assert error messages) before doing the action triggering the error.

[24] Duma is a further enhanced version of *Electric Fence*, see Duma `http://duma.sourceforge.net/`

6.5 Fixing device use-time issues

Besides avoiding polling in normal operations, nothing should be recursed infinitely and there should be strict retry count limits for failed operations as data coming from the outside may be broken (such as infinite MMC directory lenght or depth) or never available. Failure should be graceful.

References

[Performance measuring and analysis] Performance measuring and analysis, **TODO**

[Debugging Guide] Maemo developer documentation, Debugging Guide, Debugging Guide `maemo.org/maemo_release_documentation/maemo4.1.x/Debugging.pdf`

[Maemo architecture] Maemo architecture, Maemo architecture `http://maemo.org/maemo_release_documentation/maemo4.1.x/Architecture.pdf`

[Endurance measuring] `sp-endurance` and `sp-endurance-postproc` tools for measuring resource usage and logged software faults, Maemo tools `http://maemo.org/development/tools/`

[Memory usage] `sp-memusage` tools for measuring system and process memory usage reliably, Maemo tools `http://maemo.org/development/tools/`

[SMAPS scripts] SMAPS data visualization scripts, Maemo tools `http://maemo.org/development/tools/`

[Functracer] Allocation tracking utility, Maemo tools `http://maemo.org/development/tools/`

[Rich core] System information collection for richer core dumps, Maemo tools `http://maemo.org/development/tools/`

[Debug scripts] Maemo-debug-scripts package contains helper scripts for debugging and wrappers for different Valgrind plugins, Maemo tools `http://maemo.org/development/tools/`

[Valgrind] Valgrind `http://valgrind.org/`

[Maemo MXR] Maemo source code cross-reference, MXR source-code cross-reference `http://mxr.maemo.org/`

A Open issues

- Could some common library have a signal handler that reports the `mallinfo()` values and which could be enabled with an environment variable? This would help in detecting small memory usage increase and fragmentation and (the code for that would be trivial/small and wouldn't affect performance so it could be enabled all the time).
- Is it possible to have a *HW* watchdog handler that saves information about what caused the reboot (where kernel or watchdog kicker got stuck so that watchdog couldn't be updated)?
- Could the SW watchdog show on screen what service exit/crash will causes the device reboot e.g. when R&D mode is enable (it would be much more convenient for testers and developers than digging this information from /var/lib/dsme/)?

A.1 Resolved issues

- Can there be beta releases of all opened Maemo SW packages in the device so that power users and 3rd party developers can also test them and report bugs before they are put into sales release?
  - Yes, starting from Fremantle SDK. The software is also included into the Maemo community *Mer* distribution working on top of Ubuntu
- How to easily enable core dumps? See section 6.4.2
  - To generate core-dumps in Diablo, it's enough to insert a memory card having a "core-dumps" directory
  - In Fremantle one needs to install `sp-rich-core` and create "core-dumps" directory either to MyDocs directory or to memory card.
- How to measure which files in the JFFS-2 are fragmented (through a new /proc/ API)?
  - IT2006 JFFS-2 rewrites completed blocks that are too fragmented, so this isn't anymore such an issue. See section 4.3.1
- Device doesn't handle full rootfs gracefully
  - In IT2006 some space is reserved only for root so that system can be booted. Device functionality (naturally) will be limited until space is available also for other processes (configuration cannot be changed permanently, application state save may be limited, saving doesn't work, but all applications can be started without issues)
- Will Maemo-launcher log application crashes?
  - Yes, to syslog, in IT2006
- Will there be crash backtrace logging either in libosso or Maemo-launcher?

- In IT2006 Desktop can show to user abnormal application exits reported by maemo-launcher. Backtraces are not possible on target without debug symbols and is otherwise inadvisable, see 6.4.1