Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Linux-tiny And Directions For Small Systems

Matt Mackall
Digeo, Inc.
`mpm@digeo.com`

Abstract

Linux-tiny is a project to reduce the memory and storage footprint of the 2.6 Linux kernel for embedded, handheld, legacy, and other small systems. I describe strategies for kernel size reduction, some of the major areas already investigated and the results achieved, as well as some avenues for further exploration.

1 Introduction

Historically, Linux had a reputation for running on very modest systems. My first dedicated Linux box, running a 0.99 kernel circa 1994, provided mail, FTP, web, dial-in, and shell services on a 16MHz 386SX with a mere 4 megabytes of RAM. In the 10 years since then, Linux has grown to the point where it runs on machines with over a thousand processors and a terabyte of RAM. Not surprisingly, a modern Linux distribution can have difficulty getting to a shell prompt on machines with less than 8 megabytes of RAM, let alone doing useful work.

1.1 What happened?

In the time between the 0.99 and 2.6 kernels, we've seen Linux become a serious commercial endeavor, we've seen kernel hackers get jobs (and get big machines on their desks), and we've seen a massive boom in Internet use and personal computing. Linux developers have been targeting high end computing and rising demand for hardware has seen prices drop tremendously.

But there are still small machines! Hand-helds and embedded systems are perennially pressed for space to match their desktop counterparts and many people throughout the world still rely on legacy machines to get their work done. What can be done to recapture the 'small is beautiful' utility of those early systems?

1.2 Where is the growth?

The process by which any large software project grows can aptly be described as *death by a thousand cuts*. The accumulation of bloat occurs change by change and creeps in from several different directions.

Perhaps the most visible is the addition of new **features**, which generally requires the introduction of wholly-new code. Frequently features are considered so small or so essential that no thought is given to making them optional. As the median system size grows, this new code tends to be more verbose and less concerned with space issues.

The next, more subtle culprit is **performance**. Given the fundamental importance of kernel performance to overall system performance, trade-offs of size for speed are easy to justify. Unfortunately the accumulation of many such trade-offs can leave us with a system that no longer boots. Ironically, the evolution of processors has brought us to a point where cache footprint can be critical to performance so a lot of the choices that have been made in this area bear rethinking.

Next we have **compatibility** and **correctness**. Every time the system is extended to better support a slightly different piece of hardware or work around another corner case, more code is added. Occassionally cleanups and unifications make some of this code redundant, but this is the exception. A related phenomenon is the evolution of the kernel APIs and the accumulation of obsolete code for the sake of backward compatibility.

2 Linux-Tiny for the small system niche

There have been numerous efforts to address the above phenomena for various components of Linux systems, but most of the attention has been addressed at userspace (arguably the biggest offender). Experiments with pre-2.6.0 kernels however suggested it was time to pay some more attention to the kernel itself. So in December of 2003, I decided to create a new 2.6-based tree dedicated to small systems which I named Linux-Tiny [3] (someone had already borrowed my initials for their tree).

2.1 Methodology

With stated targets of embedded, hand-held, and legacy machines, the -tiny tree attempts to tailor the kernel to the needs of small systems. The tree is maintained as a series of small patches stacked on top of mainline kernel releases, managed with the quilt tool [1] (previously with Andrew Morton's patch scripts [4]).

Patches try to observe the following criteria:

- **configurable**: changes that are not clearly wins for all systems should be configurable so that users can make their own trade-offs
- **non-invasive**: patches should be small, self-contained, and largely independent so that integrators can cherrypick the patches they'd like to use
- **mergeable**: while not mandatory, patches should try to be acceptable to the mainline kernel in both style and approach; merging to mainline is a priority

In addition to patches focusing on reducing kernel footprint, I've also added a number of patches to do debugging and auditing including netconsole, kgdb, and kgdb-over-ethernet support.

2.2 Setting goals

Everyone has a different set of functionality requirements in mind for small systems. The features needed on a handheld are very different from those needed for a network appliance or a kiosk. Thus, choosing a subset of features to develop towards is tricky.

The approach I've taken is to choose a series of targets to optimize, and the first is a minimal x86 kernel with filesystem, console, and TCP/IP support. How small can we make this kernel? This puts a focus on the most of the common core functionality of Linux and provides a useful benchmark for progress.

3 Finding bloat

As mentioned above, there are many sources of bloat. There are also several forms it can take: as superfluous code, statically or dynamically allocated data, inline functions or macros, compiler mis-optimizations, or cut-n-paste coding.

Given that the kernel is on the order of several hundred kilobytes, tackling bloat is going to be a matter of trimming several kilobytes here and a couple kilobytes there. While one could simply pick any source file and read through it searching for cleanup opportunities, there are some more straightforward ways of finding the "low-hanging fruit".

3.1 Using nm(1) and size(1)

The easiest place to begin is by using the `nm` tool to find large functions and data structures. Comparing the (hexadecimal) numbers from `nm(1)` with `size(1)` gives us a good start at understanding the relative sizes of some of the major subsystems and their components compared to the kernel as a whole. For instance, we can see by comparing Table 1 and Table 2 that the static `ide_hwifs` data structure alone takes 15360 bytes, over 2% of the data portion of the default kernel.

3.2 Measuring function inlining

Function inlining and macro expansion present a special problem for our bloat detection efforts. In the early 1990s, inlining was a very popular performance technique to avoid function call branches. A great number of key functions are marked for inlining in the kernel and their usage and size impact is obscured because they become a seamless part of the functions that use them. Auditing their usage becomes a matter of convincing the compiler to tell us when inlines are being instantiated in a build and then estimating how large these functions are when expanded inline.

Rather than modifying the compiler itself, the first part of this puzzle was hacked around by redefining `inline` to include the GCC extension `__attribute__((deprecated))`. This causes a very useful warning like the following to be generated:

```
arch/i386/kernel/semaphore.c:58:
warning:  'get_current' is
deprecated (declared at
include/asm/current.h:16)
```

By post-processing these voluminous warning messages, we can determine which inline functions are instantiated directly in C files as well as which are called as parts of other inlines and finally calculate the total number of direct or indirect instantiations of each (see Table 3).

The second part of this puzzle was more challenging. While we know in which modules and how often inlines are instantiated, we cannot yet calculate their sizes. I made several attempts to generate approximate size data by looking at GCC's symbolic debugging output, but this tended to be easily confused by inlining and was too inaccurate for use.

Recently Denis Vlasenko took another stab at this and wrote a set of scripts called inline_hunter [5] to generate a set of dummy functions wrapping single calls to inlines. While these sizes won't directly reflect the size of inline instantiations due to function call overhead and lost optimization opportunities, for larger inline functions, it has proven fairly representative. Some of the larger inlines found with this approach are shown in Table 4.

3.3 Tracking dynamic allocations

Of course much of the kernel's memory footprint is from dynamic allocations. Memory used for page tables, tracking running processes, indexing hashes and so forth is allocated at runtime and can vary with the size of the load. A number of these are hash tables to increase look-up performance, which for small systems can be less important than simply fitting in memory.

There are several important allocators in the kernel. First, the bootmem allocator which

```
2.6.5\$ nm --size -r vmlinux | head -20
00008000 b __log_buf
00007000 D irq_desc
00004e78 d pci_vendor_list
00004000 b bh_wait_queue_heads
00003c00 B ide_hwifs
0000213a T vt_ioctl
00002000 D init_thread_union
00001880 D contig_page_data
0000163b T journal_commit_transaction
00001500 b irq_2_pin
000012f5 T tcp_sendmsg
00001162 t n_tty_receive_buf
00001080 d per_cpu__tvec_bases
00001000 t translation_table
00001000 b sd_index_bits
00001000 D init_tss
00001000 b doublefault_stack
00001000 B con_buf
00001000 b cache_defer_hash
00000fe0 T cdrom_ioctl
```

Table 1: nm output for 2.6.5 default config

handles a number of critical allocations at startup. As there are not terribly many of these, they can be audited very simply with `printk()` techniques.

Second, the SLAB allocator is used to quickly allocate sets of objects of the same size and type. The kernel provides a way to track these allocations with `/proc/slabinfo`.

The more general `kmalloc()` allocator has been rebuilt on top of the aforementioned SLAB allocator, translating kmalloc requests into requests from a set of ascending generic SLAB sizes. Thus all `kmalloc()` allocations are lumped together by size in the `/proc/slabinfo` output. That can be helpful if you know what you're looking for, but doesn't give many hints as to which parts of the kernel are using that memory.

To address this deficiency, I've created a small footprint tool for tracking allocations via `/proc/kmalloc` (see Table 5). This works by tracking the address of each allocation along with the address of the allocating function in a simple hash table. Also tracked are net and gross allocation sizes and counts per caller. When a `kfree()` call is made, it is matched up to its caller for accounting purposes and removed from the hash. Thus it is possible not only to determine how much dynamic memory is used by each function but also to easily identify memory leaks.

4 Some notable opportunities for code trimming

The above methods have revealed numerous opportunities for cutting back the kernel's

```
2.6.5\$ size vmlinux */built-in.o
   text    data     bss     dec     hex filename
3366220  673296  166824 4206340  402f04 vmlinux
1181276  250808   48000 1480084  169594 drivers/built-in.o
 735152   32593   30628  798373   c2ea5 fs/built-in.o
  18151    1120    1316   20587    506b init/built-in.o
  21841     172     204   22217    56c9 ipc/built-in.o
 159632   16115   42402  218149   35425 kernel/built-in.o
   2870       0       0    2870     b36 lib/built-in.o
 129669    9068    2884  141621   22935 mm/built-in.o
 580407   33816   18856  633079   9a8f7 net/built-in.o
   1869       0       0    1869     74d security/built-in.o
 325923   11114    3016  340053   53055 sound/built-in.o
    134       0       0     134      86 usr/built-in.o
```

Table 2: size output for 2.6.5 default config

memory footprint, many of which remain to be examined. What follows are some of the more notable areas that have been explored.

4.1 Debugging data

The kernel has numerous facilities for trapping and reporting problem conditions and other status information, including `printk()`, `bug()`, `warn()`, `panic()`, and friends. In ideal circumstances, these facilities go unexercised. And in the extreme, embedded boxes may have no means of reporting this data, due to lack of a display, writable storage, or the like. Unfortunately, not only do these facilities use a substantial amount of code, their users need extra space for error message strings, filenames, and line numbers.

Linux-tiny has a set of configuration options to compile out most of this code and remove the debugging strings and data from the kernel. Disabling support for `printk()` saves well over 100K. Independent options control the inclusion of the `bug()` infrastructure and support for trapping panics and doublefaults.

4.2 Optional interfaces

For systems with well-defined application requirements, many of the kernel's APIs are unnecessary. Cutting-edge, obsolete, or obscure features are obvious candidates for configurable removal.

- **sysfs**: The new sysfs filesystem makes substantial memory demands (which can be more than half a megabyte even on the smallest systems) but its features may well not be essential to current systems. The -tiny tree was a testbed for options to entirely remove sysfs or to use a lighter "backing store" version.
- **ptrace, aio, posix-timers**: These features are among those that are only used by a small set of applications. These and other Linux-tiny options are enabled under the CONFIG_EMBEDDED menu, which marks them as making the kernel non-standard.
- **uid16, vm86**: Some of the many legacy interfaces in the kernel. Modern applications and libraries use 32-bit user and

group IDs and vm86 support is used to run 16-bit code for emulators like DOSEMU and Wine and for some video drivers used by X.

- **ethtool, tcpdiag, igmp, rtnetlink**: One of the most complicated parts of the kernel is the networking layer, which has grown a variety of APIs to gain access to its many features. But for most users, the interfaces used by the classic `ifconfig(8)` and `route(8)` tools are sufficient.

4.3 4K stacks

During the 2.1 kernel series (circa 1998), the x86 kernel increased the size of the per-task kernel stacks from 4K to 8K to work around issues with stack depth. In addition to the obvious increase in overhead for every userspace process, several new kernel daemons have been added, all with their own stacks. Another issue is that finding pairs of contiguous pages to build an 8K stack can be very difficult on a machine with memory pressure and especially so on machines with a small number of total pages.

Many of the problems that made 4K stacks problematic have since been addressed and 4K stacks are now practical for most applications. Linux-tiny has served as an early testbed for reintroducing 4K stack support to the mainline 2.6 kernel and includes a developer tool called `checkstack` that will automatically disassemble a kernel to find the most extreme stack space users.

4.4 The SLOB allocator

Most memory in the kernel is managed either directly or indirectly through the SLAB allocator. SLAB maintains separate caches for objects of given sizes and types and can very quickly manage allocations for them. In some cases, it can even arrange for objects to be pre-initialized without any additional overhead. SLAB also has some resistance to troublesome memory fragmentation issues. While simple in principle, the SLAB code ends up being quite complex from its efforts to squeeze the maximum possible performance out of the allocator.

The primary downside to SLAB is that because it maintains a collection of independent caches which are all one or more pages, it ends up leaving quite a bit of unused space in each SLAB cache. In addition, as `kmalloc` is implemented on top of SLAB using a set of preset object size SLABs, there is quite a bit of extra space allocated for the average `kmalloc` call. Measurements with the previously described `/proc/kmalloc` tool report that extra overhead can amount to 25-30% of the total memory allocated by `kmalloc`.

Linux-tiny provides an optional replacement for SLAB that I've dubbed *SLOB* (simple list of blocks). SLOB trades performance for space efficiency by implementing a more traditional list-based allocator that also understands requests for objects with particular alignments. The APIs used by SLAB and `kmalloc()` are provided by a small emulation layer.

SLOB manages all objects at a granularity of 8 bytes so overhead for odd object sizes is minimized. It also does away with the numerous partly-used caches of the SLOB approach. Finally, the SLOB code is much simpler and takes up less than one tenth of the space of the standard SLAB allocator.

4.5 TinyVT

As you can see from Table 1, the largest single function in the default kernel is `vt_ioctl()`, which manages many of the special features of the Linux console. As most early Linux users didn't have the memory for running a full-fledged X desktop, the native Linux text console is very powerful, with support for scrollback, selection, virtual console switching, Unicode translation and character sets, screen blanking, and so on.

These features can be very handy for some users, but on a palmtop or kiosk running a GUI, or for a minimal rescue disk, they're dead weight. Linux-tiny includes a heavily trimmed down replacement for the standard console code which drops many of these features and can trim a couple percent off the size of the kernel image.

5 Results

Recent releases of Linux-tiny contain the above options and numerous others. My test configuration, with support for a text console, IDE disks, the Ext2 filesystem, TCP/IP, and a PCI-based network card results in a 363K compressed kernel image. Other users of Linux-tiny have reported kernel configurations resulting in images as small as 191K.

Booting the test configuration with `mem=2M`, which gives a total of of 1664K after accounting for BIOS memory holes, still leaves ample room for a lightweight userspace (see Table 6). A similarly configured mainline kernel without the -tiny patches compiles to a kernel image of over 500K and has difficulty booting with `mem=4M`.

For comparison, the earliest Linux distribution kernel I've been able to locate, a 0.99pl15 kernel from Slackware 1.1.2 circa 1994, is a mere 301K. Modern `highly-modularized` 2.6 kernels from Fedora Core 2 and SuSE 9.1 weigh in at 1.2M and 1.5M respectively while the default 2.6.5 kernel config builds a 1.9M compressed kernel.

6 Further directions

There are many further avenues to pursue and subsystems to trim. Some of the more aggressive ideas on the to-do list include:

- A lightweight replacement network stack: Minimal TCP stacks like uIP [2] have sufficient functionality for simple network applications and have extremely small footprints.

- Replacements for fixed-sized hash tables: Existing kernel hash tables have difficulty scaling with workloads and memory sizes. Other approaches like radix trees might be better in some areas and avoid wasted memory when the indexes are empty.

- Support for bunzip2: Linux-tiny now has a simplified interface to the boot-time decompressor and allows for replacements to be easily dropped in. While bzip2 compression won't save any memory at runtime, it will save valuable storage space on embedded systems.

- Pageable kernel memory: Following an approach similar to the `__init` approach in current kernels, it should be possible to mark specific functions and data in the kernel core as pageable, provided they meet some specific requirements.

- Tracking kernel growth: Using automated tools to track the size of kernel functions and subsystems from release to release will help catch new bloat when it appears.

Of course, as most of the bloat in the kernel has been introduced in small increments, most of the improvements will be of the same variety. Contributions are encouraged!

References

[1] patchwork quilt patch management tools. `http://savannah.nongnu.org/projects/quilt`.

[2] Adam Dunkels. The uip tcp/ip stack for embedded microcontrollers. `http://www.sics.se/~adam/uip/index.html`.

[3] Matt Mackall. The linux-tiny homepage. `http://www.selenic.com/tiny-about/`.

[4] Andrew Morton. Patch-scripts. `http://www.zip.com.au/~akpm/linux/patches/`.

[5] Denis Vlasenko. inline_hunter 2.0 and its results, 2004. `http://lkml.org/lkml/2004/4/16/191`.

```
1560  get_current (1294 in *.c)
calls:
callers: <other>(336) capable(122) unlock_kernel(44) lock_kernel(33)
flush_tlb_page(11) flush_tlb_mm(10) find_process_by_pid(6)
flush_tlb_range(4) current_is_kswapd(4) current_is_pdflush(3)
rwsem_down_failed_common(2) on_sig_stack(2) do_mmap2(2) __exit_mm(2)
walk_init_root(1) scm_check_creds(1) save_i387_fsave(1)
sas_ss_flags(1) restore_i387_fsave(1) read_zero_pagealigned(1)
handle_group_stop(1) get_close_on_exec(1) fork_traceflag(1)
ext2_init_acl(1) exec_permission_lite(1) dup_mmap(1) do_tty_write(1)
de_thread(1) copy_signal(1) copy_sighand(1) copy_fs(1) check_sticky(1)
cap_set_all(1) cap_emulate_setxuid(1) arch_get_unmapped_area(1)

546  current_thread_info (286 in *.c)
calls:
callers: <other>(207) copy_to_user(95) copy_from_user(86)
tcp_set_state(22) test_thread_flag(20) verify_area(13)
tcp_enter_memory_pressure(6) sock_orphan(3) icmp_xmit_lock(2)
csum_and_copy_to_user(2) tcp_v4_lookup(1) sock_graft(1)
set_thread_flag(1) neigh_update_hhs(1) ip_finish_output2(1) gfp_any(1)
fn_flush_list(1) do_getname(1) clear_thread_flag(1) alloc_buf(1)
activate_task(1)

413  atomic_dec_and_test (55 in *.c)
calls:
callers: put_page(103) kfree_skb(101) <other>(47) mntput(34)
in_dev_put(23) neigh_release(19) tcp_tw_put(18) fib_info_put(17)
sock_put(15) put_namespace(6) mmdrop(6) __put_fs_struct(4)
tcp_listen_unlock(3) ipq_put(3) finish_task_switch(2) __detach_pid(2)
task_state(1) de_thread(1)

255  tcp_sk (134 in *.c)
calls:
callers: <other>(117) tcp_reset_xmit_timer(30) tcp_set_state(22)
tcp_current_mss(13) tcp_initialize_rcv_mss(6) tcp_free_skb(6)
tcp_check_space(6) tcp_data_snd_check(5) tcp_clear_xmit_timer(5)
tcp_synq_removed(3) tcp_select_window(3) westwood_update_rttmin(2)
westwood_acked(2) tcp_synq_len(2) tcp_synq_drop(2)
tcp_ack_snd_check(2) __tcp_inherit_port(2) tcp_use_frto(1)
tcp_synq_young(1) tcp_synq_is_full(1) tcp_synq_added(1)
tcp_prequeue(1) tcp_listen_poll(1) tcp_event_ack_sent(1)
tcp_connect_init(1) tcp_acceptq_queue(1) do_pmtu_discovery(1)
```

Table 3: Some large inline counts and users for 2.6.5-tiny1

```
Size  Uses Wasted Name and definition
===== ==== ====== ================================================
   56  461  16560 copy_from_user         include/asm/uaccess.h
  122  119  12036 skb_dequeue            include/linux/skbuff.h
  164   78  11088 skb_queue_purge        include/linux/skbuff.h
   97  141  10780 netif_wake_queue       include/linux/netdevice.h
   43  468  10741 copy_to_user           include/asm/uaccess.h
   43  461  10580 copy_from_user         include/asm/uaccess.h
  145   77   9500 put_page               include/linux/mm.h
   49  313   9048 skb_put                include/linux/skbuff.h
  109  101   8900 skb_queue_tail         include/linux/skbuff.h
  381   21   7220 sock_queue_rcv_skb     include/net/sock.h
   55  191   6650 init_MUTEX             include/asm/semaphore.h
   61  163   6642 unlock_kernel          include/linux/smp_lock.h
   59  165   6396 lock_kernel            include/linux/smp_lock.h
  127   59   6206 dev_kfree_skb_any      include/linux/netdevice.h
   41  289   6048 list_del               include/linux/list.h
   73   83   4346 dev_kfree_skb_irq      include/linux/netdevice.h
  131   39   4218 netif_device_attach    include/linux/netdevice.h
  110   44   3870 skb_queue_head         include/linux/skbuff.h
   84   59   3712 seq_puts               include/linux/seq_file.h
   57   75   2738 skb_trim               include/linux/skbuff.h
   45   96   2375 skb_queue_head_init    include/linux/skbuff.h
   41  111   2310 list_del_init          include/linux/list.h
  102   23   1804 __nlmsg_put            include/linux/netlink.h
```

Table 4: Size estimates found by inline_hunter

```
# cat /proc/kmalloc
total bytes allocated:   266848
slack bytes allocated:    37774
net bytes allocated:     145568
number of allocs:           732
number of frees:            282
number of callers:           71
lost callers:                 0
lost allocs:                  0
unknown frees:                0

   total    slack      net alloc/free  caller
     256      203      256      8/0     alloc_vfsmnt+0x73
    8192     3648     4096      2/1     atkbd_connect+0x1b
     192       48       64      3/2     seq_open+0x10
   12288        0     4096      3/2     seq_read+0x53
    8192        0        0      2/2     alloc_skb+0x3b
     960        0        0     10/10    load_elf_interp+0xa1
    1920      288        0     10/10    load_elf_binary+0x100
     320      130        0     10/10    load_elf_binary+0x1d8
     192       48       96      6/3     request_irq+0x22
    7200     1254     7200     75/0     proc_create+0x74
      64       43       64      2/0     proc_symlink+0x40
    4096      984        0      1/1     check_partition+0x1b
   69632        0    45056     17/6     dup_task_struct+0x38
     128       48      128      2/0     netlink_create+0x84
     128       20      128      1/0     ext2_fill_super+0x2f
      32       28       32      1/0     ext2_fill_super+0x385
      32       31       32      1/0     ext2_fill_super+0x3b6
     608       76      384     19/7     __request_region+0x18
      64       32       64      2/0     rand_initialize_disk+0xd
    8192     2016     8192      2/0     alloc_tty_struct+0x10
     128       56      128      2/0     init_dev+0xba
     128       56      128      2/0     init_dev+0xf3
     128        0      128      2/0     create_workqueue+0x28
    8960     1680     8960     70/0     tty_add_class_device+0x20
    2048      960     2048      4/0     alloc_tty_driver+0x10
    9280     2332     9280      4/0     tty_register_driver+0x2d
     288        0      288      9/0     mempool_create+0x16
    1280      196     1280      9/0     mempool_create+0x41
    1536      384     1536      8/0     mempool_create+0x8f
      64       28       64      1/0     kbd_connect+0x3e
     928      348        0     29/29    kmem_cache_create+0x235
   28288     1448    28288     81/0     do_tune_cpucache+0x2c
...
```

Table 5: Tracking usage of kmalloc/kfree in -tiny

```
Uncompressing Linux... Ok, booting the kernel.
# mount /proc
# cat /proc/meminfo
MemTotal:          980 kB
MemFree:           312 kB
Buffers:            32 kB
Cached:            296 kB
SwapCached:          0 kB
Active:            400 kB
Inactive:           48 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:          980 kB
LowFree:           312 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:            380 kB
Slab:                0 kB
Committed_AS:      132 kB
PageTables:         24 kB
VmallocTotal:  1032172 kB
VmallocUsed:         0 kB
VmallocChunk:  1032172 kB
#
```

Table 6: Boot log for a 2.6.5-tiny1 test configuration with mem=2m