

Guidelines for Creating Robust Embedded Systems

Part 5 – Handling Out of Memory Conditions

By Bob Japenga

Scope

This is Part 5 of a white paper intended to provide a brief overview of lessons learned in creating robust systems over the past 35 years of embedded systems development (starting with the Intel 4040). This part will address how to gracefully handle out of memory conditions.

Gracefully Handling Out of Memory Conditions

The first time I tried to handle an out of memory condition I found that I had two problems. First, I was not detecting every out of memory condition. It is so easy to allocate memory in C and C++ that often we don't check the return value from our malloc's or our new's. So guideline number one in gracefully handling out of memory conditions is:

1. Check that every dynamic memory allocation succeeds. In many systems we simply restart either the thread or the entire system when the memory allocation fails.

The second problem was that the mechanisms I provided to notify the user or log the error required dynamic memory (putting things up on the screen or logging to an error log). Even rebooting can require dynamic memory. For example in Linux, using

```
system (reboot);
```

may not work because you may not have the dynamic memory to spawn the shell (like BusyBox). It is necessary to use a call rather than using the system command.

So guideline number two in gracefully handling out of memory conditions is:

2. Make sure that everything you do after the "out of memory" condition does not use dynamic memory or pulls it from a stash that you squirreled away.

What I did in that first case was to determine how much memory I needed to flag and log the error and then allocate twice that much as startup. When the out of memory condition manifested itself, I deleted my stash and flagged the error.

Finally, the most important two guidelines are:

3. You must rigorously design your system so that you can never run out of memory
4. You must test, re-test and test some more

However all of what I learned has been thrown out the window with Linux. There is a major problem that we have had with using Linux in creating robust systems as it relates to dealing with Out of Memory conditions. Linux has this wonderful “feature” of over committing dynamic memory allocations. Simply stated it means that no matter how much memory is actually left, Linux almost always returns true to a memory allocation (assuming that you are configured in the default mode).

This is a kernel option that you can build out of your kernel (in later versions of the kernel it can be controlled with a `sysctl`¹)– but then virtually none of the great applications that we get with Linux will run because they allocate gigantic amounts of memory – more than your normal embedded system would have – and then just don’t use it. Linux doesn’t actually allocate the memory until you actually use it.

However, if you plan to use only the kernel and write all of your applications (no ftp, telnet, ntp, busybox, etc) yourself, I would strongly recommend creating a kernel without the overbooking feature (mode 2).

One Linux detractor says this about the feature:

Linux on the other hand is seriously broken. It will by default answer "yes" to most requests for memory, in the hope that programs ask for more than they actually need. If the hope is fulfilled Linux can run more programs in the same memory, or can run a program that requires more virtual memory than is available. And if not then very bad things happen.

What happens is that the OOM killer (OOM = out-of-memory) is invoked, and it will select some process and kill it. One holds long discussions about the choice of the victim. Maybe not a root process, maybe not a process doing raw I/O, maybe not a process that has already spent weeks doing some computation. And thus it can happen that one's emacs is killed when someone else starts more stuff than the kernel can handle. Ach. Very, very primitive.²

Yes – that is actually how Linux works and it makes designing robust Linux systems very difficult when you have limited memory. As of this writing, there is no easy way to control the OOM Killer without writing your own or modifying the existing one. One author has described a method of self management that looks promising³ but requires a significant amount of work that must be re-done every time changes are made to the system. And of course we never change our embedded Linux systems! (sarcasm)

The approach is to profile the memory usage of every application to determine its memory needs. Then create a limit of the amount of physical memory that you will allow for each application to use. For every malloc, you check to see if the amount allocated is less than the limit minus the total amount of rss memory available and the amount is less than the total system free memory. If that check fails, the malloc should fail.

Is there any hope? We have built robust Linux systems with very limited memory. Here are our guidelines.

1. You must minimize your usage of dynamic memory.
2. You must know your total dynamic memory usage (not an easy feat with a large system) including your other open source applications.
3. You must make sure that you never run out.

We have seen totally bizarre things happen when you start running out of memory. You never know what the OOM killer will do. With Linux, you must avoid this at all costs because there is virtually no way to predict what will happen when you run out of memory when using the default “over commit” One comedian has described this feature of Linux as follows:

An aircraft company discovered that it was cheaper to fly its planes with less fuel on board. The planes would be lighter and use less fuel and money was saved. On rare occasions however the amount of fuel was insufficient, and the plane would crash. This problem was solved by the engineers of the company by the development of a special OOF (out-of-fuel) mechanism. In emergency cases a passenger was selected and thrown out of the plane. (When necessary, the procedure was repeated.) A large body of theory was developed and many publications were devoted to the problem of properly selecting the victim to be ejected. Should the victim be chosen at random? Or should one choose the heaviest person? Or the oldest? Should passengers pay in order not to be ejected, so that the victim would be the poorest on board? And if for example the heaviest person was chosen, should there be a special exception in case that was the pilot? Should first class passengers be exempted? Now that the OOF mechanism existed, it would be activated every now and then, and eject passengers even when there was no fuel shortage. The engineers are still studying precisely how this malfunction is caused.

Although humorous, this is not funny and a serious problem that we hope the Linux community quickly addresses. We cannot all build our small systems with 1 gig of memory. Memory demands of embedded systems will continue to grow and outstrip the memory that is available on a lot of systems.

We would love to hear from you about your experience with this if you have suggestions. Please email me at rjapenga@mtiemail.com

In [Part 6](#) we will discuss handling out of disk conditions.

¹ Taken from the 2.6.26 documentation
The Linux kernel supports the following overcommit handling modes

-
- 0 Heuristic overcommit handling. Obvious overcommits of address space are refused. Used for a typical system. It ensures a seriously wild allocation fails while allowing overcommit to reduce swap usage. root is allowed to allocate slightly more memory in this mode. This is the default.
 - 1 Always overcommit. Appropriate for some scientific applications.
 - 2 Don't overcommit. The total address space commit for the system is not permitted to exceed swap + a configurable percentage (default is 50) of physical RAM. Depending on the percentage you use, in most situations this means a process will not be killed while accessing pages but will receive errors on memory allocation as appropriate.

The overcommit policy is set via the `sysctl 'vm.overcommit_memory'`.

² <http://www.win.tue.nl/~aeb/linux/lk/lk-9.html>

³ http://www.celinux.org/elc08_presentations/CELF_AvoidOOM.pdf April 15, 2008 by YoungJun Jang
yj03.jang@samsung.com