Embedded

uClinux for Linux Programmers

Adapt your software to run on processors without memory management—it's easier than you think.

by David McCullough

uClinux has seen a huge increase in popularity and is appearing in more commodity devices than ever before. Its use in routers (Figure 1), Web cameras and even DVD players is testimony to its versatility. The explosion of low-cost, 32-bit CPUs capable of running uClinux is providing even more options to manufacturers considering uClinux. Now with uClinux's debut as part of the 2.6 kernel, it is set to become even more popular.

Figure 1. The SnapGear LITE2 VPN/Router runs uClinux.

With more embedded developers facing the possibility of working with uClinux, a guide to its differences from Linux and its traps and pitfalls is an invaluable tool. Here we discuss the changes a developer might encounter when using uClinux and how the environment steers the development process.

No Memory Management

The defining and most prevalent difference between uClinux and other Linux systems is the lack of memory management. Under Linux, memory management is achieved through the use of virtual memory (VM). uClinux was created for systems that do not support VM. As VM usually is implemented using a processing unit called an MMU (memory management unit), you often hear the term NOMMU when traveling in uClinux circles.

With VM, all processes run at the same address, albeit a virtual one, and the VM system takes care of what physical memory is mapped to these locations. So even though the virtual memory the process sees is contiguous, the physical memory it occupies can be scattered around. Some of it even may be on a hard disk in swap. Because arbitrarily located memory can be mapped to anywhere in the process' address space, it is possible to add memory to an already running process.

Without VM, each process must be located at a place in memory where it can be run. In the simplest case, this area of memory must be contiguous. Generally, it cannot be expanded as there may be other processes above and below it. This means that a process in uClinux cannot increase the size of its available memory at runtime as a traditional Linux process would.
Although all programs need to be relocated at run time so that they can execute, it is a fairly transparent task for the developer. It is the direct effect of no VM that is the thorn in every uClinux developer's side. The net effect is that no memory protection of any kind is offered—it is possible for any application or the kernel to corrupt any part of the system. Some CPU architectures allow certain I/O areas, instructions and memory regions to be protected from user programs but that is not guaranteed. Even worse than the corruption that crashes a system is the corruption that goes unnoticed, and tracking down random interprocess corruption can be extremely difficult.

Without VM, swap is effectively impossible, although this limitation is rarely an issue on the kinds of systems that run uClinux. They often do not have hard drives or enough memory to make swap worthwhile.

**Kernel Differences**

To a kernel developer, uClinux offers little in the way of differences from Linux. The only real issue is that you cannot take advantage of the paging support provided by an MMU. In practice, this doesn't affect much of the kernel. tmpfs, for example, does not work on uClinux because it relies on the VM system.

Similarly, all of the standard executable formats are unsupported, because they make use of VM features that do not exist under uClinux. Instead, a new format is required, the flat format. Flat format is a condensed executable format that stores only executable code and data, along with the relocations needed to load the executable into any location in memory.

Device drivers often need some work when you move to uClinux, not because of differences in the kernels, but due to the kinds of devices the kernel needs to support. For example, the SMC network driver supports ISA SMC cards. They usually are 16-bit and are located at I/O addresses below 0x3ff. The same driver easily can be made to support the non-ISA embedded versions of the chip, but it may need to run in 8-, 16- or 32-bit mode, at an I/O address that is a full 32-bit address and at an interrupt number quite often higher than ISA's maximum of 16. So despite the fact that the bulk of the driver is the same, the hardware specifics can require a little porting effort. Quite often, older drivers store I/O addresses in short format, which does not work on an embedded uClinux platform with devices appearing at memory-mapped I/O addresses.

The implementation of mmap within the kernel is also quite different. Though often transparent to the developer, it needs to be understood so it is not used in ways that are particularly inefficient on uClinux systems. Unless the uClinux mmap can point directly to the file within the filesystem, thereby guaranteeing that it is sequential and contiguous, it must allocate memory and copy the data into the allocated memory. The ingredients for efficient mmap usage under uClinux are quite specific. First, the only filesystem that currently guarantees that files are stored contiguously is romfs. So one must use romfs to avoid the allocation. Second, only read-only mappings can be shared, which means a mapping must be read-only in order to avoid the allocation of memory. The developer under uClinux cannot take advantage of copy-on-write features for this reason. The kernel also must consider the filesystem to be “in ROM”, which means a nominally read-only area within the CPU's address space. This is possible if the filesystem is present somewhere in RAM or ROM, both of which are addressable directly by the CPU. One cannot have a zero allocation mmap if the filesystem is on a hard disk, even if it is a romfs filesystem, as the contents are not directly addressable by the CPU.

**Memory Allocation (Kernel and Application)**

uClinux offers a choice of two kernel memory allocators. At first it may not seem obvious why an alternative kernel memory allocator is needed, but in small uClinux systems the difference is painfully apparent. The default kernel allocator under Linux uses a power-of-two allocation method. This helps it operate faster and
quickly find memory areas of the correct size to satisfy allocation requests. Unfortunately, under uClinux, applications must be loaded into memory that is set aside by this allocator. To understand the ramifications of this, especially for large allocations, consider that an application requiring a 33KB allocation in order to be loaded actually allocates to the next power of two, which is 64KB. The 31KB of extra space allocated cannot be utilized effectively. This order of memory wastage is unacceptable on most uClinux systems. To combat this problem, an alternative memory allocator has been created for the uClinux kernels. It commonly is known as either page_alloc2 or kmalloc2, depending on the kernel version.

page_alloc2 addresses the power-of-two allocation wastage by using a power-of-two allocator for allocations up to one page in size (a page is 4,096 bytes, or 4KB). It then allocates memory rounded up to the nearest page. For the previous example, an application of 33KB actually has 36KB allocated to it; a savings of 28KB for a 33KB application is possible.

page_alloc2 also takes steps to avoid fragmenting memory. It allocates all amounts of two pages (8KB) or less from the start of memory up and all larger amounts from the end of free memory down. This stops transient allocations for network buffers and so on, fragmenting memory and preventing large applications from running. For a more detailed example of memory fragmentation, see the example in the Applications and Processes section below. page_alloc2 is not perfect, but it works well in practice, as the embedded environments that run uClinux tend to have a relatively static group of long-lived applications.

Once the developer gets past the kernel memory allocation differences, the real changes appear in the application space. This is where the full impact of uClinux's lack of VM is realized. The first major difference most likely to cause an application to fail under uClinux is the lack of a dynamic stack. On VM Linux, whenever an application tries to write off the top of the stack, an exception is flagged and some more memory is mapped in at the top of the stack to allow the stack to grow. Under uClinux, no such luxury is available as the stack must be allocated at compile time. This means that the developer, who previously was oblivious to stack usage within the application, must now be aware of the stack requirements. The first thing a developer should consider when faced with strange crashes or behavior of a newly ported application is the allocated stack size. By default, the uClinux toolchains allocate 4KB for the stack, which is close to nothing for modern applications. The developer should try increasing the stack size with one of the following methods:

1. Add `FLTFLAGS = -s <stacksize>` and `export FLTFLAGS` to the Makefile for the application before building.

2. Run `flthdr -s <stacksize> executable` after the application has been built.

The second major difference that strikes a uClinux developer is the lack of a dynamic heap, the area used to satisfy memory allocations with malloc and related functions in C. On Linux with VM, an application can increase its process size, allowing it to have a dynamic heap. This traditionally is implemented at the low level using the sbrk/brk system calls, which increase/change the size of a process' address space. The heap's management by library functions such as malloc then is performed on the extra memory obtained by calling sbrk() on behalf of the application. If an application needs more memory at any point, it can get more simply by calling sbrk() again; it also can decrease memory using brk(). sbrk() works by adding more memory to the end of a process (increasing its size). brk() arbitrarily can set the end of the process to be closer to the start of the process (reduce the process size) or further away (increase the process size).

Because uClinux cannot implement the functionality of brk and sbrk, it instead implements a global memory pool that basically is the kernel's free memory pool. There are pitfalls with this method. For example, a runaway process can use all of the system's available memory. Allocating from the system pool is not compatible with sbrk and brk, as they require memory to be added to the end of a process' address space. Thus, a normal malloc implementation is no good, and a new implementation is needed.
A global pool approach has some advantages. First, only the amount of memory actually required is used, unlike the pre-allocated heap system that some embedded systems use. This is extremely important on uClinux systems, which generally are running with little memory. Another advantage is that memory can be returned to the global pool as soon as it is finished being used, and the implementation can take advantage of the existing in-kernel allocator for managing this memory, reducing the size of application code.

One of the common problems new users encounter is the missing memory problem. The system is showing a large amount of free memory, but an application cannot allocate a buffer of size X. The problem here is memory fragmentation, and all of the uClinux solutions available at this time suffer from it. Because of the lack of VM in the uClinux environment, it is nearly impossible to utilize memory fully due to fragmentation. This is best explained by example. Suppose a system has 500KB of free memory and one wishes to allocate 100KB to load an application. It is easy to think that this would be possible. However, it is important to remember that one must have a contiguous 100KB block of memory in order to satisfy the allocation. Suppose the memory map looks like this. Each character represents approximately 20KB, and X marks areas allocated or in use by other programs or by the kernel:

```
 0    100   200   300  400   500   600   700  800   900   1000
+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+--
|XXXXX|XXXXX|---XX|--X--|-X---|XX---|-X---|-XX--|-X---|XXXXX|
```

In this case, 500KB are free, but the largest contiguous block is only 80KB. There are many ways to arrive at such a situation. A program that allocates some memory and then frees most of it, leaving a small allocation in the middle of a larger free block, often is the cause. Transient programs under uClinux also can affect where and how memory is allocated. The uClinux page_alloc2 kernel allocator has a configuration option that can help identify this problem. It enables a new /proc entry, /proc/mem_map, that shows pages and their allocation grouping. Documenting this is beyond the scope of this article, but more information can be found in the kernel source for page_alloc2.c.

The question is often asked, why can't this memory be defragmented so it is possible to load a 100KB application? The problem is that we don't have VM and we cannot move memory being used by programs. Programs usually have references to addresses within the allocated memory regions, and without VM to make the memory always appear to be at the correct address, the program will crash if we move its memory. There is no solution to this problem under uClinux. The developer needs to be aware of the problem and, where possible, try to utilize smaller allocation blocks.

**Applications and Processes**

Another difference between VM Linux and uClinux is the lack of the fork() system call. This can require quite a lot of work on the developer's part when porting applications that use fork(). The only option under uClinux is to use vfork(). Although vfork() shares many properties with fork(), the differences are what matter the most.

fork() and vfork(), for those unfamiliar with these system calls, allow a process to split into two processes, a parent and a child. A process can split many times to create multiple children. When a process calls fork(), the child is a duplicate of the parent in all ways, but it shares nothing with the parent and can operate independently, as can the parent. With vfork() this is not the case. First, the parent is suspended and cannot continue executing until the child exits or calls exec(), the system call used to start a new application. The child, directly after returning from vfork(), is running on the parent's stack and is using the parent's memory and data. This means the child can corrupt the data structures or the stack in the parent, resulting in failure. This is avoided by ensuring that the child never returns from the current stack frame once vfork() has been called and that it calls _exit when finishing—exit cannot be called as it changes data structures in the parent. The child also must avoid changing any information in global data structures or variables, as such changes
may break the execution of the parent.

Making an application use vfork instead of fork usually falls into the absolutely simple or incredibly difficult category. Generally, if the application does not fork and then exec() almost immediately, it needs to be checked carefully before fork() can be replaced with vfork().

The uClinux flat executable format, though it doesn't directly affect applications and their operations, does allow quite a few options that the usual ELF executables under Linux do not. Flat format executables come in two basic flavors, fully relocated and a variation of position-independent code (PIC). The fully relocated version has relocations for its code and data, while the PIC version generally needs only a few relocations for its data.

One of the most advantageous features to the embedded developer is execute-in-place (XIP). This is where the application executes directly from Flash or ROM, requiring the absolute minimum of memory, because only the memory for the data of the application is needed. This allows the text or code portion to be shared between multiple instances of the application. Not all uClinux platforms are capable of XIP, as it requires compiler support and the PIC form of the flat executable. So unless the toolchain for a given platform can do PIC, it cannot do XIP. Currently, only the m68k and ARM toolchains provide the required level of support for flat format XIP. romfs is the only filesystem to support XIP under uClinux, because the application must be stored contiguously within the filesystem for XIP to be possible.

The flat format also defines the stack size for an application as a field in the flat header. To increase the stack allocated to an application, a simple change of this field is all that is required. This can be done with the flthdr command, like this:

```
flthdr -s flat-executable
```

The flat format also allows two compression options. The entire executable can be compressed, providing maximum ROM savings. It also offers the often useful side effect that the application is loaded entirely into a contiguous RAM block. You also may choose data-segment-only compression. This is important if you want to save ROM space but still want the option to utilize XIP. The following:

```
flthdr -z flat-executable
```

creates a fully compressed executable, and

```
flthdr -d flat-executable
```

compresses only the data segment.

Shared Libraries

Although a complete discussion of shared libraries is beyond the scope of this article, they are quite different under uClinux. The currently available solutions require compiler changes and care on the part of the developer. The best way to create shared libraries is to start with an example. The current uClinux distributions provide shared libraries for both the uC-libc and uClibc libraries. The method for creating a shared library isn't difficult, and both of these libraries provide a good, clean example of how it is done. To set expectations appropriately, the GCC -shared option is not part of the shared library creation process, so do not expect it to be familiar. Shared libraries under uClinux are flat format executables, just like applications, and to be truly shared must be compiled for XIP. Without XIP, shared libraries result in a full copy of the library for each application using it, which is worse than statically linking your applications.
Summary

The step into uClinux from Linux often is more than the differences between uClinux and Linux. uClinux systems tend to be more deeply embedded systems, with smaller memories and ROM footprints and an unusual array of devices. The loss of a hard drive and the tight resource limits, coupled with no memory protection and a number of other subtle differences can make a developer's first adventure into uClinux more difficult than imagined. The best way to get started is to look at the uClinux Emulators (Figure 2) and cheap hardware (Figure 3) options available.

Figure 2. uClinux Running under Xcopilot (Palm Emulator)
Figure 3. uClinux Running on a Real Palm IIIx (with Microwindows)

Hopefully, highlighting these issues will help the wary developer be prepared beforehand and avoid some of the common pitfalls and misconceptions of working with uClinux.

Resources for this article: [www.linuxjournal.com/article/7546](http://www.linuxjournal.com/article/7546).

David McCullough is a senior software engineer and a veteran embedded software developer. Prior to working at SnapGear and Lineo, he held software development and engineering management positions at Stallion Technologies and was involved in the development of products based on SCO and BSD UNIX. David ported and maintained XFree86 on SCO UNIX for several years and recently was instrumental in the development of the uClinux port of Linux 2.6.