Using flexible arrays in the kernel
Last updated for 2.6.32
Jonathan Corbet <corbet@lwn.net>

Large contiguous memory allocations can be unreliable in the Linux kernel. Kernel programmers will sometimes respond to this problem by allocating pages with vmalloc(). This solution not ideal, though. On 32-bit systems, memory from vmalloc() must be mapped into a relatively small address space; it's easy to run out. On SMP systems, the page table changes required by vmalloc() allocations can require expensive cross-processor interrupts on all CPUs. And, on all systems, use of space in the vmalloc() range increases pressure on the translation lookaside buffer (TLB), reducing the performance of the system.

In many cases, the need for memory from vmalloc() can be eliminated by piecing together an array from smaller parts; the flexible array library exists to make this task easier.

A flexible array holds an arbitrary (within limits) number of fixed-sized objects, accessed via an integer index. Sparse arrays are handled reasonably well. Only single-page allocations are made, so memory allocation failures should be relatively rare. The down sides are that the arrays cannot be indexed directly, individual object size cannot exceed the system page size, and putting data into a flexible array requires a copy operation. It's also worth noting that flexible arrays do no internal locking at all; if concurrent access to an array is possible, then the caller must arrange for appropriate mutual exclusion.

The creation of a flexible array is done with:

```
#include <linux/flex_array.h>

struct flex_array *flex_array_alloc(int element_size,
                                    unsigned int total,
                                    gfp_t flags);
```

The individual object size is provided by element_size, while total is the maximum number of objects which can be stored in the array. The flags argument is passed directly to the internal memory allocation calls. With the current code, using flags to ask for high memory is likely to lead to notably unpleasant side effects.

It is also possible to define flexible arrays at compile time with:

```
DEFINE_FLEX_ARRAY(name, element_size, total);
```

This macro will result in a definition of an array with the given name; the element size and total will be checked for validity at compile time.

Storing data into a flexible array is accomplished with a call to:

```
int flex_array_put(struct flex_array *array, unsigned int element_nr,
                   void *src, gfp_t flags);
```

This call will copy the data from src into the array, in the position indicated by element_nr (which must be less than the maximum specified when

the array was created). If any memory allocations must be performed, flags will be used. The return value is zero on success, a negative error code otherwise.

There might possibly be a need to store data into a flexible array while running in some sort of atomic context; in this situation, sleeping in the memory allocator would be a bad thing. That can be avoided by using GFP_ATOMIC for the flags value, but, often, there is a better way. The trick is to ensure that any needed memory allocations are done before entering atomic context, using:

```
int flex_array_prealloc(struct flex_array *array, unsigned int start,
                        unsigned int end, gfp_t flags);
```

This function will ensure that memory for the elements indexed in the range defined by start and end has been allocated. Thereafter, a flex_array_put() call on an element in that range is guaranteed not to block.

Getting data back out of the array is done with:

```
void *flex_array_get(struct flex_array *fa, unsigned int element_nr);
```

The return value is a pointer to the data element, or NULL if that particular element has never been allocated.

Note that it is possible to get back a valid pointer for an element which has never been stored in the array. Memory for array elements is allocated one page at a time; a single allocation could provide memory for several adjacent elements. Flexible array elements are normally initialized to the value FLEX_ARRAY_FREE (defined as 0x6c in <linux/poison.h>), so errors involving that number probably result from use of unstored array entries. Note that, if array elements are allocated with __GFP_ZERO, they will be initialized to zero and this poisoning will not happen.

Individual elements in the array can be cleared with:

```
int flex_array_clear(struct flex_array *array, unsigned int element_nr);
```

This function will set the given element to FLEX_ARRAY_FREE and return zero. If storage for the indicated element is not allocated for the array, flex_array_clear() will return -EINVAL instead. Note that clearing an element does not release the storage associated with it; to reduce the allocated size of an array, call:

```
int flex_array_shrink(struct flex_array *array);
```

The return value will be the number of pages of memory actually freed. This function works by scanning the array for pages containing nothing but FLEX_ARRAY_FREE bytes, so (1) it can be expensive, and (2) it will not work if the array's pages are allocated with __GFP_ZERO.

It is possible to remove all elements of an array with a call to:

```
void flex_array_free_parts(struct flex_array *array);
```

This call frees all elements, but leaves the array itself in place.
Freeing the entire array is done with:

```
void flex_array_free(struct flex_array *array);
```

As of this writing, there are no users of flexible arrays in the mainline
kernel. The functions described here are also not exported to modules;
that will probably be fixed when somebody comes up with a need for it.