Character Device Driver Tutorial (Part 3)

BY JOHN BALDWIN

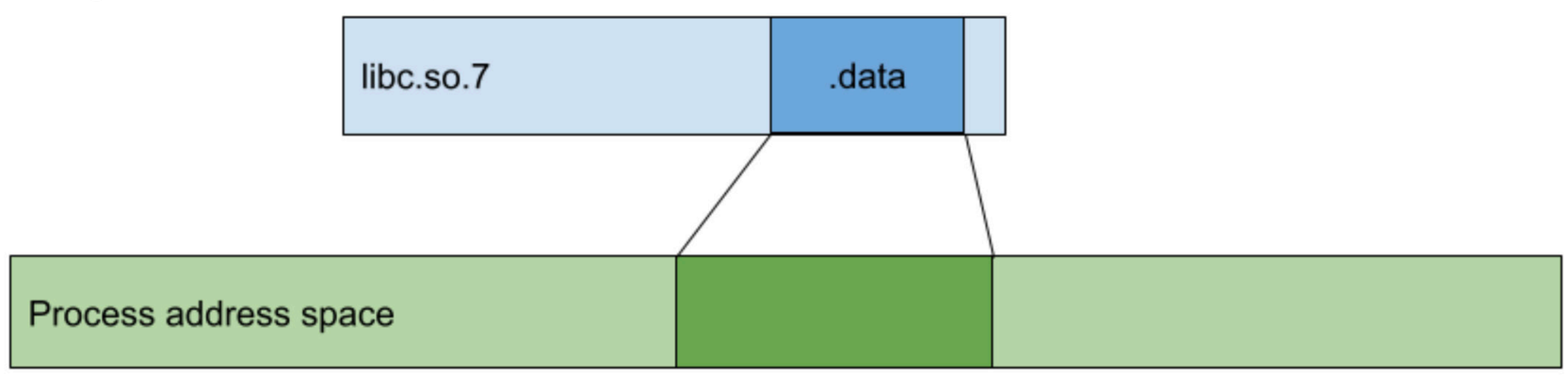
In Part 1 and Part 2, we implemented a simple character device driver that implemented support for basic I/O operations. In this final article in this series, we will explore how character devices can provide backing store for memory mappings in user processes. Unlike the previous article, we will not be extending the echo device driver but will instead implement new drivers to demonstrate memory mapping. These drivers can be found in the same repository as the echo driver at https://github.com/bsdjhb/cdev_tutorial.

Memory Mappings in FreeBSD

To understand how memory mapping works in character devices, one must first understand how the FreeBSD kernel manages memory mappings in general. FreeBSD's virtual memory subsystem is derived from the Mach virtual memory subsystem inherited from 4.4BSD. While FreeBSD's VM has seen substantial changes over the past thirty years, the core abstractions remain the same.

In FreeBSD, a virtual memory address space is represented by a virtual memory map (`struct vm_map`). A VM map contains a list of entries (`struct vm_map_entry`). Each entry defines the properties for a contiguous range of address space including the permissions and the backing store. Virtual memory objects (`struct vm_object`) are used to describe the backing store for mappings. A VM object has its own logical address space of pages. For example, each regular file on disk is associated with a VM object where the logical address of a page in the VM object corresponds to offsets within the file, and the contents of a logical page are the file contents at the given file offset. Each VM map entry identifies its backing store as a range of logically contiguous pages starting at a specific offset from a single VM object. Figure 1 shows how a single VM map entry can be used to map the .data section from the C runtime library into a process' address space.

Figure 1: Mapping of C Runtime Library .data Section



Each VM object is associated with a pager which provides a set of functions used to determine the contents of the pages associated with a VM object. The vnode pager is used for VM objects associated with regular files from both block-storage filesystems and net-

work filesystems. Its functions read data from the associated file to initialize pages and write modified pages back to the associated file. The swap pager is used for anonymous VM objects that are not associated with a regular file. Zero-filled pages are allocated on first use. If the system runs low on memory, the swap pager writes less frequently used dirty pages to swap until they are needed again.

Logical pages in VM objects are represented by a VM page (`struct vm_page`). During boot, the kernel allocates an array of VM pages such that each physical page of RAM is associated with a VM page object. VM pages are mapped into address spaces using architecture-specific page table entries (PTEs). Managed VM pages maintain a linked list of mappings using architecture-specific structures called PV entries. This list can be used to remove all the mappings for a VM page by invalidating the associated PTEs so that a VM page can be reused to represent a different logical page, either for a different VM object or a different logical page address within the same VM object.

Each invocation of the mmap(2) system call creates a new VM map entry in the calling process. The arguments to the system call provide various properties of the new entry including the permissions, length, and offset into the backing VM object. The file descriptor argument is used to identify the VM object to map into the calling process' address space. To map memory from a character device, a process passes an open file descriptor for the character device as the file descriptor argument to the `mmap()` system call. The role of the character device driver is to decide which VM object is used to satisfy a memory mapping request as well as the contents of the pages backing the VM object.

4.4BSD included a device VM pager to support character device memory mappings.

Default Character Device Pager

4.4BSD included a device VM pager to support character device memory mappings. This device pager is designed to map regions of physical memory that do not change while the OS is running. For example, it can expose MMIO regions like a frame buffer directly to userspace.

The device pager assumes that each page in a device VM object is mapped to a page of physical address space. This page can be a page of RAM or associated with an MMIO region. Importantly, once a logical address in a device VM object is associated with a physical page, that mapping cannot be changed. This assumption works both ways in that the device pager also assumes that once a page of physical address space is associated with a device VM object, that physical page can never be reused for any other purpose. As a result, the VM pages used by the device pager are unmanaged (no PV entries). However, this also means that the VM system is not easily able to find existing mappings of these VM pages to revoke existing mappings. In particular, destroying a character device via `destroy_dev(9)` does not revoke existing mappings.

The default character device pager uses the character device mmap method both to validate mapping requests and to determine the physical address associated with each logical page address. The mmap method should validate the offset and protection arguments. If the offset is not a valid logical page address or the requested protection is not supported,

this method should fail by returning an error code. Otherwise, the method should store the physical address for the requested offset in the physical address argument and return zero. If the page should be mapped with a memory attribute other than `VM_MEMATTR_DEFAULT`, the memory attribute should be returned on success as well. When a mapping is created, the device pager invokes this method on each logical page address of the requested mapping to validate the request. For the first page fault of a logical page address, the device pager invokes the mmap method to obtain the physical address and memory attribute of the backing page.

Listing 1 shows the mmap method for a simple character device driver that uses the default device pager. This device allocates a single page of RAM when loaded and saves a pointer to this page in the `si_drv1` field. Due to the limitations of the character device pager, this driver cannot be unloaded. Example 1 demonstrates a few interactions with the device once it is loaded using a `maprw` test program to read and write from a mapping of the device.

Listing 1: Using the Default Device Pager

```
static int
mappage_mmap(struct cdev *dev, vm_ooffset_t offset, vm_paddr_t *paddr,
    int nprot, vm_memattr_t *memattr)
{
      if (offset != 0)
            return (EINVAL);
      *paddr = pmap_kextract((uintptr_t)dev->si_drv1);
      return (0);
}
```

Example 1: Using the /dev/mappage Device

```
# maprw read /dev/mappage 16 | hexdump
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010
# jot -c -s "" 16 ‘A’ | maprw write /dev/mappage 16
# maprw read /dev/mappage 16
ABCDEFGHIJKLMNOP
```

Mapping Arbitrary VM Objects

Due to the limitations of the default character device pager, FreeBSD has extended the support for character device memory mappings. FreeBSD 8.0 introduced a new mmap_single character device method. This method is called on every `mmap()` invocation that maps a character device. The mmap_single method must validate the entire `mmap()` request including the offset, size, and requested protection. If the request is valid, the method should return a reference to a VM object to use for the mapping. The method can either create a new VM object or return an additional reference to an existing VM object. If the mmap_single method returns the `ENODEV` error (the default behavior), `mmap()` will use the default character device pager.

The mmap_single method can also alter the offset (but not size) used for the mapping when returning a VM object. This permits a character device to use the initial offset of a mapping as a key to identify a specific VM object to map. For example, a driver might have two internal VM objects and use offset 0 to map the first VM object, and an offset of `PAGE_SIZE` to map the second VM object. For the second case, the mmap_single method would reset the effective offset to 0 so that the resulting mapping starts at the beginning of the second VM object.

However, a character device doesn't have to use multiple VM objects to benefit from the mmap_single method. The ability to use VM objects with other pagers can be useful. For example, the physical pager creates VM objects backed by wired pages of physical RAM. Unlike the default device pager, these pages are managed and can be safely freed when the VM object is destroyed. Listing 2 updates the mappage device driver from earlier to use a physical pager VM object instead of the default character device pager. This version of the device driver can be safely unloaded since the VM object will persist after the driver is unloaded until all mappings have been destroyed.

Listing 2: Using the Physical Pager

```
static int
mappage_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
      vm_object_t obj;

      obj = cdev->si_drv1;
      if (OFF_TO_IDX(round_page(*offset + size)) > obj->size)
            return (EINVAL);

      vm_object_reference(obj);
      *object = obj;
      return (0);
}

static int
mappage_create(struct cdev **cdevp)
{
      struct make_dev_args args;
vm_object_t obj;
int error;

obj = vm_pager_allocate(OBJT_PHYS, NULL, PAGE_SIZE,
    VM_PROT_DEFAULT, 0, NULL);
if (obj == NULL)
      return (ENOMEM);
make_dev_args_init(&args);
args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
args.mda_devsw = &mappage_cdevsw;
```

```
args.mda_uid = UID_ROOT;
args.mda_gid = GID_WHEEL;
args.mda_mode = 0600;
args.mda_si_drv1 = obj;
error = make_dev_s(&args, cdevp, "mappage");
if (error != 0) {
      vm_object_deallocate(obj);
      return (error);
}
      return (0);
}

static void
mappage_destroy(struct cdev *cdev)
{
      if (cdev == NULL)
            return;

      vm_object_deallocate(cdev->si_drv1);
      destroy_dev(cdev);
}
```

Per-Open State

In the first article in this series, we demonstrated support for per-instance data using the **`si_drv1`** field. Some character device drivers need to maintain a unique state for each open file descriptor. That is, if a character device is opened multiple times, the driver wishes to provide different behavior to each open reference.

FreeBSD provides this feature via a family of functions. Typically, a character device driver creates a new instance of per-open state in the open method and associates that instance with the new file descriptor by calling devfs_set_cdevpriv(9). This function accepts a void pointer argument and a destructor callback function. The destructor is invoked to clean the per-open state when the last reference to the file descriptor is closed. Other character device switch methods call devfs_get_cdevpriv(9) to retrieve the void pointer associated with the current file descriptor. Note that this family of functions always operates on the current file descriptor as determined implicitly by the caller context. The driver does not pass an explicit reference to a file descriptor to these functions.

In the first article in this series, we demonstrated support for per-instance data using the si_drv1 field.

Listing 3 shows the open and mmap_single methods as well as the **`cdevpriv`** destructor for a new **`memfd`** character device driver. This simple driver provides similar functionality to the **`SHM_ANON`** extension in FreeBSD's shm_open(2) implementation. Each open file descriptor of this device is associated with an anonymous VM object. The VM object's size grows, if necessary, when it is mapped. The VM object can be shared with other processes by sharing the file descrip-

tor, for example by passing the file descriptor over a UNIX domain socket. To implement this, the driver allocates a new VM object in the open method and associates that VM object with the new file descriptor. The mmap_single object retrieves the VM object for the current file descriptor, grows it if necessary, and returns a reference to it. Finally, the destructor function drops the file descriptor's reference on the VM object.

Listing 3: Per-Open Anonymous Memory

```
static int
memfd_open(struct cdev *cdev, int fflag, int devtype, struct thread *td)
{
	vm_object_t obj;
	int error;

	/* Read-only and write-only opens make no sense. */
	if ((fflag & (FREAD | FWRITE)) != (FREAD | FWRITE))
		return (EINVAL);

	/*
	 * Create an anonymous VM object with an initial size of 0 for
	 * each open file descriptor.
	 */
	obj = vm_object_allocate_anon(0, NULL, td->td_ucred, 0);
	if (obj == NULL)
		return (ENOMEM);
	error = devfs_set_cdevpriv(obj, memfd_dtor);
	if (error != 0)
		vm_object_deallocate(obj);
	return (error);

}

static void
memfd_dtor(void *arg)
{
	vm_object_t obj = arg;

	vm_object_deallocate(obj);
}

static int
memfd_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
	vm_object_t obj;
	vm_pindex_t objsize;
	vm_ooffset_t delta;
```

```
void *priv;
int error;

error = devfs_get_cdevpriv(&priv);
if (error != 0)
      return (error);
obj = priv;

/* Grow object if necessary. */
objsize = OFF_TO_IDX(round_page(*offset + size));
VM_OBJECT_WLOCK(obj);
if (objsize > obj->size) {
      delta = IDX_TO_OFF(objsize - obj->size);
      if (!swap_reserve_by_cred(delta, obj->cred)) {
            VM_OBJECT_WUNLOCK(obj);
            return (ENOMEM);
      }
      obj->size = objsize;
      obj->charge += delta;
}

vm_object_reference_locked(obj);
VM_OBJECT_WUNLOCK(obj);
      *object = obj;
      return (0);
}
```

Extended Character Device Pagers

The mmap_single method mitigates some of the limitations of the default character device pager by permitting a character device to use VM objects backed by any pager as well as permitting a character device to associate different VM objects with different offsets. However, some limitations remain. The device pager is unique among other pagers in that it can map physical addresses that are not associated with physical RAM such as MMIO regions. Due to its use of unmanaged pages, there is no way to revoke mappings of the device pager nor a way for a driver to know if all mappings have been removed. FreeBSD 9.1 introduced a new interface to the device pager that provides solutions to both problems.

The new interface requires character device drivers to explicitly create device VM objects. These VM objects are then used by the mmap_single method to provide a backing store for mappings. In the new interface, the mmap character device method is replaced by a new method structure (**`struct cdev_pager_ops`**). This structure includes methods invoked when a VM object is created (**`cdev_pg_ctor`**), a page fault requests a page from a VM object (**`cdev_pg_fault`**), and a VM object is destroyed (**`cdev_pg_dtor`**). VM objects using the extended device pager are created by calling **`cdev_pager_allocate()`**. The first argument to this function is an opaque pointer stored in the handle member of the new VM object. This pointer is also passed as the first argument to the constructor and destructor pager methods. The second argument to **`cdev_pager_allocate()`** is the object type, either

`OBJT_DEVICE` or `OBJT_MGTDEVICE`. The third argument is a pointer to a `struct cdev_pager_ops` instance.

The `cdev_pager_allocate()` function only creates a single VM object for each opaque pointer. If the same opaque pointer is passed to a subsequent call to `cdev_pager_allocate()`, the function will return a pointer to the existing VM object instead of creating a new one. In this case, the VM object's reference count is increased, so `cdev_pager_allocate()` always returns a new reference to the returned VM object.

Let's make use of this interface to extend the original version of the mappage driver from Listing 1 so that it can be safely unloaded while there are no active mappings. In this case, we will use an `OBJT_DEVICE` VM object. This still uses unmanaged mappings of a single wired page allocated when the driver is loaded. However, there is now additional state needed to determine if that allocated page is in use, so this version of the driver defines a softc structure containing the pointer to the page, a boolean variable to track if the page is actively mapped, a boolean to track if the driver is being unloaded (in which case new mappings are disallowed), and a mutex to guard access to the boolean variables. A pointer to the softc structure is stored in the `si_drv1` field of the character device and is also used as the opaque handle for the VM object. The mmap_single character device method validates each mapping request (including failing requests while an unload is pending) and calls `cdev_pager_allocate()` to obtain a reference to the VM object mapping the wired page. Note that the mmap_single method doesn't have to handle the cases of creating a new VM object or reusing an existing VM object separately. The constructor pager method sets the boolean `mapped` softc member to true. Once the last mapping of the VM object is removed and the VM object is destroyed, the destructor pager method is called which sets the `mapped` softc member to false. The `mappage_destroy()` function fails to unload with the `EBUSY` error if the `mapped` member is true when an unload is requested.

The cdev_pager_allocate() function only creates a single VM object for each opaque pointer.

The page fault pager method is more complex than the mmap character device method it replaces. The page fault method works more directly with the VM system and how a fault is normally handled by VM pagers. When a page fault occurs, the VM system allocates a free page of RAM and invokes a pager method to fill that page with the appropriate contents. The swap and physical pagers zero new pages in this method, while the vnode pager reads the appropriate contents from the associated file. The default device pager takes a different route. Since it is generally designed to map non-RAM addresses such as MMIO regions, the default device pager allocates a "fake" VM page tied to the physical address returned by the mmap method and replaces the new VM page allocated by the VM system with the "fake" VM page (the new VM page is released back to the system as a free page). The page fault pager method allows a driver to implement either approach by passing in a pointer to the new VM page allocated by the VM system. The page fault pager method is responsible for either filling that page with suitable content or replacing it with a "fake" VM page. For our driver, we compute the physical address of our wired page the same as before but use that physical address to construct a "fake" VM page.

Listing 4 shows the mmap_single character device method, the three device pager meth-

ods, and the `mappage_destroy()` function called during module unload. In example 2, we suspend the `maprw` test program while it has the page from the mappage device mapped and attempt to unload the driver which fails. After resuming the test program and letting it unmap the device by exiting, the driver is unloaded successfully.

Listing 4: Using the Extended Device Pager

```
static struct cdev_pager_ops mappage_cdev_pager_ops = {
	.cdev_pg_ctor = mappage_pager_ctor,
	.cdev_pg_dtor = mappage_pager_dtor,
	.cdev_pg_fault = mappage_pager_fault,
};

static int
mappage_mmap_single(struct cdev *cdev, vm_ooffset_t *offset, vm_size_t size,
    struct vm_object **object, int nprot)
{
	struct mappage_softc *sc = cdev->si_drv1;
	vm_object_t obj;

	if (round_page(*offset + size) > PAGE_SIZE)
		return (EINVAL);

	mtx_lock(&sc->lock);
	if (sc->dying) {
		mtx_unlock(&sc->lock);
		return (ENXIO);
	}
	mtx_unlock(&sc->lock);

	obj = cdev_pager_allocate(sc, OBJT_DEVICE, &mappage_cdev_pager_ops,
	    OFF_TO_IDX(PAGE_SIZE), nprot, *offset, curthread->td_ucred);
	if (obj == NULL)
		return (ENXIO);

	/*
	 * If an unload started while we were allocating the VM
	 * object, dying will now be set and the unloading thread will
	 * be waiting in destroy_dev().  Just release the VM object
	 * and fail the mapping request.
	 */
	mtx_lock(&sc->lock);
	if (sc->dying) {
		mtx_unlock(&sc->lock);
		vm_object_deallocate(obj);
		return (ENXIO);
	}
```

```
	mtx_unlock(&sc->lock);

	*object = obj;
	return (0);
}

static int
mappage_pager_ctor(void *handle, vm_ooffset_t size, vm_prot_t prot,
    vm_ooffset_t foff, struct ucred *cred, u_short *color)
{
	struct mappage_softc *sc = handle;

	mtx_lock(&sc->lock);
	sc->mapped = true;
	mtx_unlock(&sc->lock);

	*color = 0;
	return (0);
}

static void
mappage_pager_dtor(void *handle)
{
	struct mappage_softc *sc = handle;

	mtx_lock(&sc->lock);
	sc->mapped = false;
	mtx_unlock(&sc->lock);
}

static int
mappage_pager_fault(vm_object_t object, vm_ooffset_t offset, int prot,
    vm_page_t *mres)
{
	struct mappage_softc *sc = object->handle;
	vm_page_t page;
	vm_paddr_t paddr;

	paddr = pmap_kextract((uintptr_t)sc->page + offset);

	/* See the end of old_dev_pager_fault in device_pager.c. */
	if (((*mres)->flags & PG_FICTITIOUS) != 0) {
		page = *mres;
		vm_page_updatefake(page, paddr, VM_MEMATTR_DEFAULT);
	} else {
		VM_OBJECT_WUNLOCK(object);
```

```
                page = vm_page_getfake(paddr, VM_MEMATTR_DEFAULT);
                VM_OBJECT_WLOCK(object);
                vm_page_replace(page, object, (*mres)->pindex, *mres);
                *mres = page;
        }
        vm_page_valid(page);
        return (VM_PAGER_OK);
}
...

static int
mappage_destroy(struct mappage_softc *sc)
{
        mtx_lock(&sc->lock);
        if (sc->mapped) {
                mtx_unlock(&sc->lock);
                return (EBUSY);
        }
        sc->dying = true;
        mtx_unlock(&sc->lock);

        destroy_dev(sc->dev);
        free(sc->page, M_MAPPAGE);
        mtx_destroy(&sc->lock);
        free(sc, M_MAPPAGE);
        return (0);
}
```

Example 2: Safely Unloading via the Extended Device Pager

```
# maprw write /dev/mappage 16
^Z
Suspended
# kldunload mappage
kldunload: can't unload file: Device busy
# fg
maprw write /dev/mappage 16
maprw: empty read
# kldunload mappage
```

The extended device pager interface also adds a new type of device pager. The `OBJT_MGTDEVICE` pager differs from `OBJT_DEVICE` in that it always uses managed pages for mappings instead of unmanaged pages. This means that mappings for a page can be forcefully revoked even while the page is mapped. For fictitious pages mapping non-RAM pages, "fake" VM pages must be explicitly created before using them in the pager via the v`m_phys_fictitious_reg_range()` function.

Conclusion

In this article, we dove into some more unusual use cases for character devices including memory mappings and per-open state. Thanks for reading this series of articles. Hopefully, it was a useful introduction to character device drivers in FreeBSD.

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (in-cluding x86 platform support, SMP, various device drivers, and the virtual memory subsys-tem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.