Scull's Memory Usage

We need to examine how and why *scull* performs memory allocation before looking at the *read* and *write* methods.

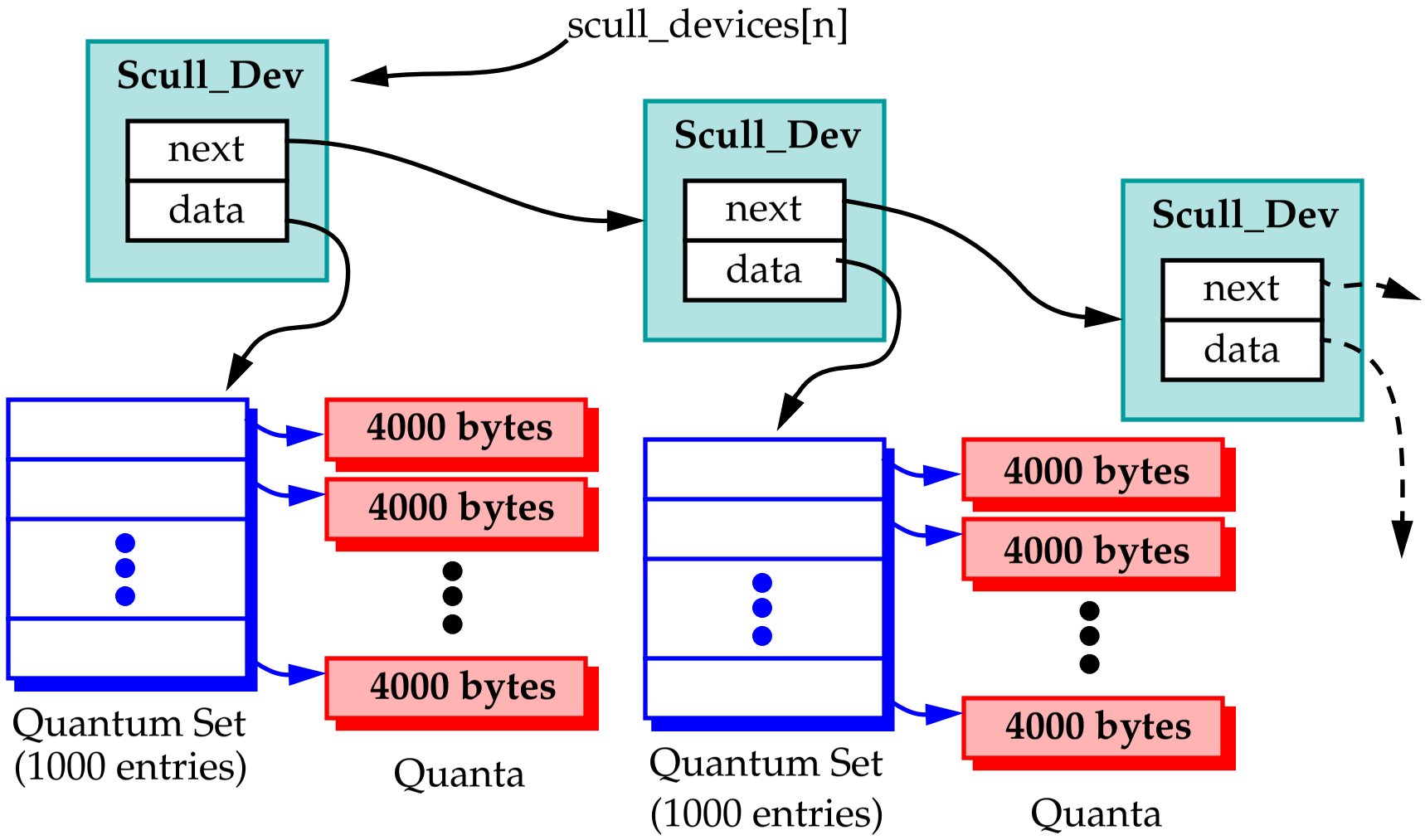
The *Scull_Dev* structure is defined first:

```
typedef struct Scull_Dev
   {
   void **data;
   struct Scull_Dev *next;   /* Pointer to next device struct. */
   int quantum;              /* Quantum size */
   int qset;                 /* Array size */
   unsigned long size;
   unsigned int access_key;  /* Use by sculluid and scullpriv. */
   unsigned int usage;       /* Lock on device during use. */
   } Scull_Dev;
```

A linked list of these structures is created for each device as its memory requirements grow.

The fields *next*, *data*, *quantum*, *qset* and *size* are used to track the memory allocation.

Scull's Memory Usage



An initial one byte allocation allocates **8K** of memory, a quantum set + one quanta.

Scull's Memory Usage

scull_trim is in charge of the deallocating this data structure.

```
int scull_trim(Scull_Dev *dev)
   {
   Scull_Dev *next, *dptr;
   int qset = dev->qset;
   int i;

   for (dptr = dev; dptr; dptr = next){
      if (dptr->data)
         {
         for (i = 0; i < qset; i++)
            if (dptr->data[i])
               kfree(dptr->data[i]);
         kfree(dptr->data);
         dptr->data = NULL;
         }
      next = dptr->next;  /* Goto next quantum set. */
      if (dptr != dev)
         kfree(dptr);
      }
```

Scull's Memory Usage

```
        dev->size = 0;
        dev->quantum = scull_quantum;
        dev->qset = scull_qset;
        dev->next = NULL;
        return 0;
        }
```

This routine is called in *scull_open* when the file is opened for writing.

Race Conditions:

A race condition exists when the "correctness" of a computation depends on the order of concurrent execution among a set of competing processes.

Semaphores are used to "synchronize" access to a data structure or device as a means of ensuring "correctness" across concurrent executions.

Race Conditions

On a uniprocessor system, the semaphore given above is not needed since kernel code is not preemptable.

However, on SMP systems, two (or more) processes on different processors may concurrently call *open*.

The call to *skull_trim* needs to be protected by a semaphore in this case.

Semaphores need to be initialized (this code is inserted into *scull_init*).

```
for ( i=0; i < scull_nr_devs; i++ )
   scull_devices[i].quantum = scull_quantum;
   scull_devices[i].qset = scull_qset;
   sema_init(&scull_devices[i].sem, 1);
   }
```

The call to *down_interruptible* test the value of the semaphore to see if it is greater than 0.

If it isn't, they are put to sleep and are later awakened by a process leaving a "critical section".

Read and Write Methods

Read and write methods transfer data between the user address space and the kernel address space.

*ssize_t read(struct file *filp, char *buff, size_t count, loff_t *offp);*
*ssize_t write(struct file *filp, const char *buff, size_t count, loff_t *offp);*

count is the size of the requested data transfer.
buff points to the user buffer holding the data to be written or the empty buffer where the newly read data should be placed.
offp indicates the file position the user is accessing.

Note that you cannot use the libc routines, e.g., *memcpy*, because the data buffer pointers operate in user (virtual) address space, and not in kernel space.

Also, memory in user-space can be swapped out.

Read and Write Methods

Cross copies of data between user and kernel space is performed by functions in *<asm/uaccess.h>*

The following kernel functions allow data to be copied between kernel and user address space:

```
unsigned long copy_from_user(void *to,
   const void *from, unsigned long count);
unsigned long copy_to_user(void *to,
   const void *from, unsigned long count);
```

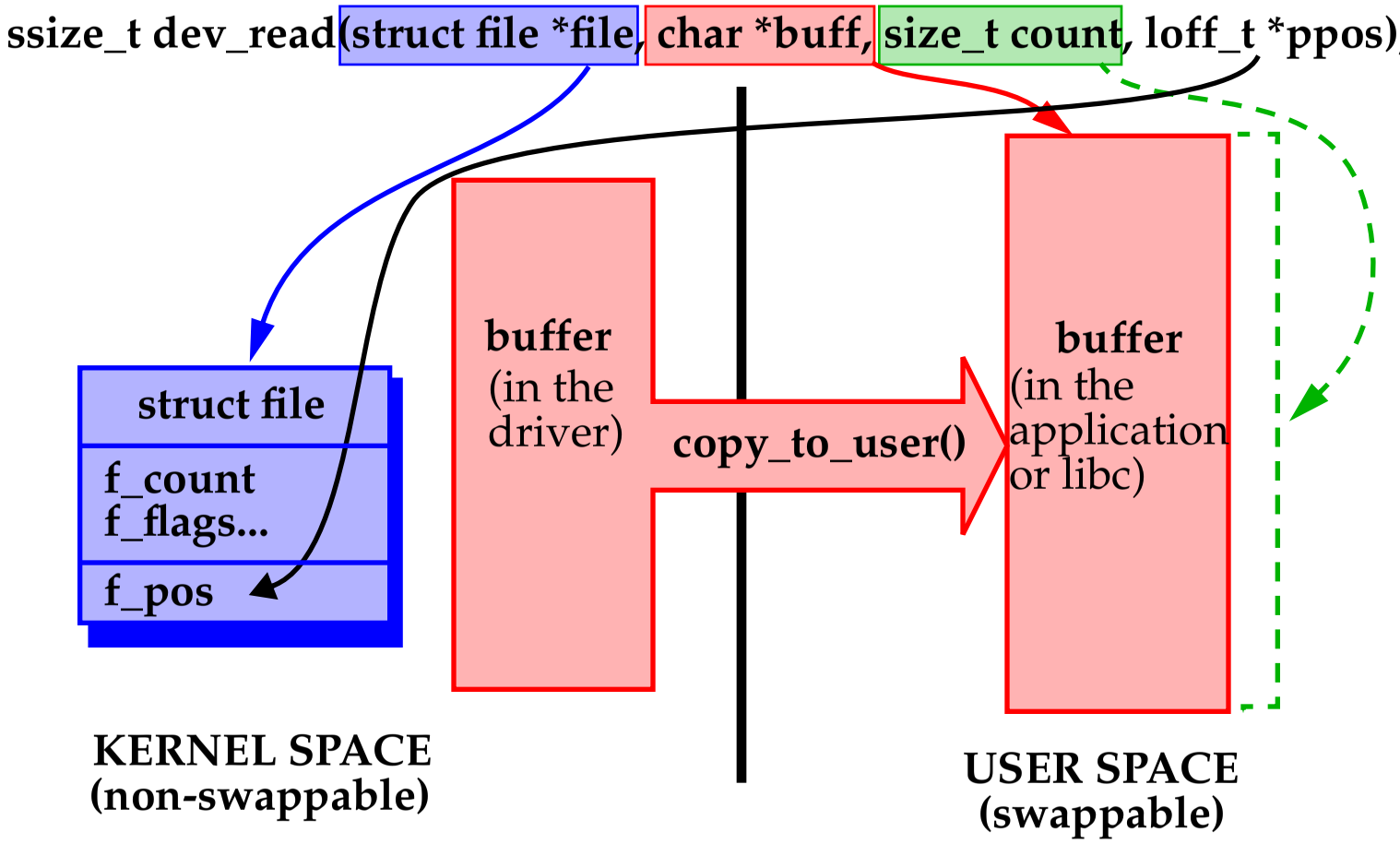
Once again, the code that uses these functions must be **reentrant** since a page fault will put the calling process to sleep.

Calls to the *read* and *write* methods request a transfer of a specific number of bytes.

However, the driver is free to transfer less data, as we will see.

Read and Write Methods

Both *read* and *write* return a negative value if an error occurs while a positive value tells the calling program how many bytes were actually transferred.



User programs always see -1 if an error occurs and need to check *errno*.

The Read Method

The return value from *read* is interpreted by the calling program as follows:

- When *return value* == *count*, the transfer succeeded.
- If *return value* > *0 && return value* < *count*, only a partial transfer occurred.
  The caller is free to retry (which occurs in the *fread* library function).
- If *return value* == *0*, end-of-file is reached.
- If *return value* < *0*, an error occurred.
  The caller can use this value to look up the error in <*linux/errno.h*>

The only condition not tested for is "**there is no data, but it may arrive later**".
In this case, the *read* system call should block.
This will be covered later.

The *read* method of scull returns a maximum of size **quantum**.
If more data is requested, the caller must iterate the call.

If the current read position is greater than the device size, read returns 0.
This occurs if process A reads and process B opens for writing.

The Read Method

```
ssize_t scull_read(struct file *filp, char *buf,
   size_t count, loff_t *f_pos){
   Scull_Dev *dev = filp->private_data;
   Scull_Dev *dptr;
   int quantum = dev->quantum;
   int qset = dev->qset;
   int itemsize = quantum * qset;
   int item, s_pos, q_pos, rest;
   ssize_t ret = 0;

if (down_interruptible(&dev->sem))
   return -ERESTARTSYS;

/* Write open truncated data. */
   if (*f_pos > dev->size) goto out;

/* If current position plus request # bytes is greater than the # bytes
   available, reset count. */
   if (*f_pos + count > dev->size)
      count = dev->size - *f_pos;
```

The Read Method

```
/* Compute the quantum to read the data from. */
   item = (long) *f_pos/itemsize;
   rest = (long) *f_pos%itemsize;
   s_pos = rest/quantum;
   q_pos = rest%quantum;

/* Simply follow the pointers in the list up to the right  position. */
   dptr = scull_follow(dev, item);

/* Check for end-of-file in the above call, NULL quantum_set. */
   if (!dptr->data)
      goto out;

/* Check for end-of-file, NULL quantum. */
   if (!dptr->data[s_pos])
      goto out;
```

The Read Method

```
/* If the amount of data requested is greater than what is available in the
   rest of the quantum, read only up to the end of the quantum. */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

/* The following call may sleep. */
    if ( copy_to_user(buf, dptr->data[s_pos]+q_pos,
        count)) {
        ret = -EFAULT;
        goto out;
        }

/* Update the pointer to the read position. */
    *f_pos += count;
    ret = count;

    out:
    up(&dev->sem);
    return ret;
    }
```

The Write Method

The following semantics are implemented for the *write* method:

- If *return value* == *count*, the transfer succeeded.
- If *return value* > *0 && return value* < *count*, only a partial transfer occurred.

  The caller is free to retry, which is what *cp* will do for you.
- If *return value* == *0*, nothing was written.

  This is **not** an error and the standard library should retry the write.

  This occurs for blocking write, covered later.
- If *return value* < *0*, an error occurred.

  The caller can use this value to look up the error in <*linux/errno.h*>

As with *scull_read*, *scull_write* deals only with a quantum at a time.

The Write Method

```
ssize_t scull_write (struct file *filp,
   const char *buf, size_t count, loff_t *f_pos)
   {
   Scull_Dev *dev = filp->private_data;
   Scull_Dev *dptr;
   int quantum = dev->quantum;
   int qset = dev->qset;
   int itemsize = quantum * qset;
   int item, s_pos, q_pos, rest;
   ssize_t ret = -ENOMEM;

   if (down_interruptible(&dev->sem))
      return -ERESTARTSYS;

/* Compute the quantum to write the data to. */
   item = (long) *f_pos/itemsize;
   rest = (long) *f_pos%itemsize;
   s_pos = rest/quantum;
   q_pos = rest%quantum;
```

The Write Method

```
/* Simply follow the pointers in the list  up to the right position. */
   dptr = scull_follow(dev, item);

/* If the quantum set is NOT found, then allocate a quantum set and
   initialize the space to 0. */
   if (!dptr->data)
      {
      dptr->data =
         kmalloc(qset * sizeof(char *), GFP_KERNEL);
      if (!dptr->data)
         goto out;
      memset(dptr->data, 0, qset * sizeof(char *));
      }

/* If the quantum is NOT present, then allocate a quantum. */
   if (!dptr->data[s_pos])
      {
      dptr->data[s_pos] =
         kmalloc(quantum, GFP_KERNEL);
```

The Write Method

```
            if (!dptr->data[s_pos])
               goto out;
            }

        /* Check that the write request is less than the quantum size. */
            if (count > quantum - q_pos)
               count = quantum - q_pos;

        /* Once again, the write may sleep. */
            if ( copy_from_user(dptr->data[s_pos]+q_pos, buf,
                  count)) {
               ret = -EFAULT;
               goto out;
               }

            *f_pos += count;
            ret = count;
```

The Write Method

```
        /* Update the memory size of the device. */
            if ( dev->size < *f_pos)
            dev->size = *f_pos;

            out:
            up(&dev->sem);
            return ret;
            }
```

The device acts like a data buffer whose length is limited only by the available RAM on the system.

Try using *cp*, *dd* and *input/output redirection* to test out the driver, e.g.,

ls -l > /dev/scull0

You can also add *printk* statements in the appropriate places in the driver code to watch variables.

Or you can use the *strace* utility to monitor system calls issued by a program.