

```
1  /*
2  * Example driver for learning purposes
3  *
4  * This driver does nothing really useful except demonstrate how one
5  * writes certain drivers functions.
6  *
7  * Copyleft (c) 2002 Deepak Saxena <dsaxena@mvista.com>
8  *
9  */
10
11
12 #include <linux/config.h>
13 #include <linux/module.h>
14 #include <linux/version.h>
15 #include <linux/types.h>
16 #include <linux/errno.h>
17 #include <linux/kernel.h>
18 #include <linux/sched.h>
19 #include <linux/spinlock.h>
20 #include <linux/timer.h>
21 #include <linux/init.h>
22 #include <linux/poll.h>
23 #include <linux/slab.h>
24 #include <linux/string.h>
25 #include <linux/devfs_fs_kernel.h>
26 #include <linux/vmalloc.h>
27 #include <asm/uaccess.h>
28 #include <asm/semaphore.h>
29
30
31 #define DRIVER_VERSION "3.14195654"
32 #define MAX_DEVICES 12
33 #define BUFFER_SIZE 32
34
35 #define DEBUG
36 #ifdef DEBUG
37 #define dprintk(x...) printk(## x)
38 #else
39 #define dprintk(x...)
40 #endif
41
42 #define IOMYDEV_ALLOC_MEM 0x6b01
```

```

43 #define IOMYDEV_MEM_INFO      0x6b02
44
45
46 /*
47  * This structure represents a device data block
48  */
49 struct device_data {
50     char * ptr;
51     struct device_data* next;
52 };
53
54 /*
55  * This structure represents our device itself
56  */
57 struct my_device
58 {
59     char name[80];
60     unsigned int memsize;          /* Current memory size */
61     unsigned int needed_mem;      /* Memory being requested */
62     wait_queue_head_t waitq;      /* waitq for waiting on read */
63     struct device_data *dev_data; /* Device data */
64     struct device_data *end_data; /* Last data ptr */
65     struct timer_list timer;      /* Timer */
66     struct rw_semaphore sem;      /* Semaphore for concurrent r/w */
67     struct fasync_struct *async_queue; /* Async notification queue */
68     spinlock_t lock;             /* Spin lock for memsize protection */
69 };
70
71 /*
72  * Our device table
73  */
74 static struct my_device my_devices[MAX_DEVICES];
75
76 /*
77  * Find out current location in memory list
78  */
79 static inline struct device_data*
80 get_current_ptr(struct my_device *dev, ssize_t f_pos)
81 {
82     struct device_data *dev_data = dev->dev_data;
83     unsigned int pos = 0;
84

```

```

85     dprintk("pos = %d, mask = %d\n", pos, f_pos & BUFFER_SIZE);
86
87     while(pos < (f_pos & ~(BUFFER_SIZE-1)) && dev_data) {
88         dev_data = dev_data->next;
89         pos += BUFFER_SIZE;
90     }
91
92     return dev_data;
93 }
94
95 /*
96  * Increase memory size to accomodate atleast max bytes
97  */
98 static inline void
99 allocate_mem(struct my_device *dev, unsigned int max)
100 {
101     unsigned long flags;
102
103     max += 3 * BUFFER_SIZE;
104
105     max = (max & ~(BUFFER_SIZE-1));
106
107     dprintk("Allocating extra %#010x bytes(max:%d)\n", max - dev->memsize, max);
108
109     do {
110         dev->end_data->next = (struct device_data*)kmalloc(sizeof(struct device_data), GFP_KERNEL);
111         dev->end_data = dev->end_data->next;
112         dev->end_data->ptr = vmalloc(BUFFER_SIZE);
113         memzero(dev->end_data->ptr, BUFFER_SIZE);
114
115         spin_lock_irqsave(&dev->lock, flags);
116         dev->memsize += BUFFER_SIZE;
117         spin_unlock_irqrestore(&dev->lock, flags);
118     } while(dev->memsize != max);
119
120     dev->end_data->next = NULL;
121
122     dprintk("Done extending memory\n");
123 }
124
125 /*
126  * Read len bytes from position *f_pos

```

```

127  */
128  static ssize_t
129  my_device_read(struct file *fp, char *data, size_t len, loff_t *f_pos)
130  {
131      struct my_device *dev = (struct my_device *)fp->private_data;
132      struct device_data *dev_data = NULL;
133      unsigned int start_pos = *f_pos, read = 0, size = 0;
134      unsigned offset;
135      unsigned long flags;
136      char* pos = 0;
137
138      dprintk("READ: %#06x @ %#010x - f_pos %d\n", len, data, *f_pos);
139
140      dprintk("READ: Trying to acquire lock\n");
141      spin_lock_irqsave(&dev->lock, flags);
142      dprintk("READ: Lock acquired\n");
143      if(*f_pos + len >= dev->memsize) {
144          if(*f_pos + len > dev->needed_mem)
145              dev->needed_mem = *f_pos + len;
146          spin_unlock_irqrestore(&dev->lock, flags);
147          dprintk("READ: Sleeping...need data: %d bytes\n", dev->needed_mem);
148          interruptible_sleep_on(&dev->waitq);
149          dprintk("READ: Woken up!\n");
150          if(signal_pending(current))
151              return -ERESTARTSYS;
152      }
153      spin_unlock_irqrestore(&dev->lock, flags);
154
155      dprintk("READ: Trying to acquire read semaphore\n");
156      down_read(&dev->sem);
157      if(signal_pending(current))
158          return -ERESTARTSYS;
159      dprintk("READ: Read semaphore acquired\n");
160
161      dev_data = get_current_ptr(dev, *f_pos);
162
163      if(!dev_data) {
164          up_read(&dev->sem);
165          return 0;
166      }
167
168      dprintk("READ: Current ptr = %#010x\n", dev_data);

```

```
169
170 offset = fp->f_pos % BUFFER_SIZE;
171
172 if(offset || len < (BUFFER_SIZE - offset))
173 {
174     if(len > (BUFFER_SIZE - offset))
175         size = len - (BUFFER_SIZE - offset);
176     else
177         size = len;
178
179     dprintk("READ: Copying initial %d bytes\n", size);
180     pos = dev_data->ptr + offset;
181     if(copy_to_user(data, pos, size)) {
182         up_read(&dev->sem);
183         return -EFAULT;
184     }
185
186     *f_pos += size;
187     len -= size;
188     data += size;
189
190     dev_data = dev_data->next;
191 }
192
193 while((len >= BUFFER_SIZE) && dev_data) {
194     dprintk("READ: Copying full buffer\n");
195     if(copy_to_user(data, dev_data->ptr, BUFFER_SIZE)) {
196         up_read(&dev->sem);
197         return -EFAULT;
198     }
199
200     len -= BUFFER_SIZE;
201     *f_pos += BUFFER_SIZE;
202     data += BUFFER_SIZE;
203
204     dev_data = dev_data->next;
205 }
206
207 if(len && dev_data) {
208     printk("READ: Copying remaining %d bytes\n", len);
209     if(copy_to_user(data, dev_data->ptr, len)) {
210         up_read(&dev->sem);
```

```

211         return -EFAULT;
212     }
213
214     *f_pos += len;
215 }
216
217 up_read(&dev->sem);
218 return (*f_pos - start_pos);
219 }
220
221 /*
222  * Write len bytes starting at position *f_pos
223  */
224 static ssize_t
225 my_device_write(struct file *fp, const char *data, size_t len, loff_t *f_pos)
226 {
227     struct my_device *dev = (struct my_device*)fp->private_data;
228     struct device_data *dev_data = NULL;
229     unsigned int start_pos = *f_pos, read = 0, size = 0;
230     char* pos = 0;
231     char *buffer;
232     unsigned offset;
233     unsigned long flags;
234
235     dprintk("WRITE: %#06x @ %#010x - f_pos %d\n", len, data, *f_pos);
236
237     down_write(&dev->sem);
238     if(signal_pending(current))
239         return -ERESTARTSYS;
240
241     buffer = kmalloc(len, GFP_KERNEL);
242     if(!buffer) {
243         up_write(&dev->sem);
244         return -ENOMEM;
245     }
246
247     if(*f_pos + len > dev->memsize)
248         allocate_mem(dev, *f_pos + len);
249
250     /*
251     * Unlike a read, we can't have partial writes as
252     * this could lead to data "corruption". If we can't read

```

```

253     * all of the data from the user, we just fault.
254     */
255     if(copy_from_user(buffer, data, len)) {
256         up_write(&dev->sem);
257         return -EFAULT;
258     }
259
260     dev_data = get_current_ptr(dev, *f_pos);
261
262     dprintk("current_ptr = %#010x\n", dev_data);
263
264     /*
265     * Get a ptr into the memory descriptor
266     */
267     offset = fp->f_pos % BUFFER_SIZE;
268
269     if(offset || len < (BUFFER_SIZE - offset))
270     {
271         if(len > (BUFFER_SIZE - offset))
272             size = len - (BUFFER_SIZE - offset);
273         else
274             size = len;
275
276         pos = dev_data->ptr + offset;
277
278         dprintk("Inital copy of %#010x bytes offset by %#010x\n", size, offset);
279         memcpy(pos, buffer, size);
280         dprintk("Copy OK\n");
281
282         *f_pos += len;
283
284         len -= size;
285         buffer += size;
286         dev_data = dev_data->next;
287     }
288
289     while(len >= BUFFER_SIZE) {
290         dprintk("Copying full buffer size\n");
291         memcpy(dev_data->ptr, buffer, BUFFER_SIZE);
292
293         len -= BUFFER_SIZE;
294         buffer += BUFFER_SIZE;

```

```

295
296     *f_pos += BUFFER_SIZE;
297
298     dev_data = dev_data->next;
299 }
300
301 if(len > 0) {
302     dprintk("WRITE: Copying extra %d bytes\n", len);
303     memcpy(dev_data->ptr, buffer, len);
304     *f_pos += len;
305 }
306
307 spin_lock_irqsave(&dev->lock, flags);
308 if(dev->memsize >= dev->needed_mem) {
309     dprintk("WRITE: Waking up waiting readers: %d bytes\n", dev->needed_mem);
310     dev->needed_mem = dev->memsize;
311     wake_up_interruptible(&dev->waitq);
312 }
313 spin_unlock_irqrestore(&dev->lock, flags);
314
315
316 if(dev->async_queue) {
317     dprintk("WRITE: Sending async notification\n");
318     kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
319 }
320
321 dprintk("WRITE: Wrote %d bytes\n", *f_pos - (loff_t)start_pos);
322 dprintk("WRITE: Releaseing semaphore\n");
323 up_write(&dev->sem);
324
325 return (*f_pos - start_pos);
326 }
327
328
329 /*
330  * Configure ourselves for async notification
331  */
332 static int
333 my_device_fasync(int fd, struct file *fp, int on)
334 {
335     struct my_device *dev = (struct my_device *)fp->private_data;
336

```

```

337     return fasync_helper (fd, fp, on, &dev->async_queue);
338 }
339
340 /*
341  * Provide an ioctl interface to user
342  */
343 static int
344 my_device_ioctl(struct inode *ip, struct file *fp, unsigned int cmd,
345                unsigned long arg)
346 {
347     struct my_device *dev = (struct my_device *)fp->private_data;
348     unsigned long flags;
349
350     if(cmd == IOMYDEV_ALLOC_MEM) {
351         spin_lock_irqsave(&dev->lock, flags);
352         if(dev->memsize >= arg) {
353             spin_unlock_irqrestore(&dev->lock, flags);
354             return 0;
355         }
356         spin_unlock_irqrestore(&dev->lock, flags);
357         allocate_mem(dev, arg);
358     } else if (cmd == IOMYDEV_MEM_INFO) {
359         unsigned long *addr = (unsigned long *)arg;
360         if(!put_user(dev->memsize, addr))
361             return -EFAULT;
362     } else return -EINVAL;
363
364     return 0;
365 }
366
367 /*
368  * Open device
369  */
370 /*
371 static int
372 my_device_open(struct inode *ip, struct file *fp)
373 {
374     struct my_device *dev = NULL;
375     int minor = MINOR(ip->i_dev);
376
377     /*
378     * Does this device exist?

```

```

379     */
380     if(minor > MAX_DEVICES)
381         return -ENODEV;
382
383     fp->private_data = (void*)&my_devices[minor];
384
385     MOD_INC_USE_COUNT;
386
387     return 0;
388 }
389
390 /*
391  * Close device
392  */
393 static int
394 my_device_release(struct inode *ip, struct file *fp)
395 {
396     struct my_device *dev = (struct my_device*)fp->private_data;
397
398     my_device_fasync(-1, fp, 0);
399
400     MOD_DEC_USE_COUNT;
401
402     return 0;
403 }
404
405 /*
406  * Our file operations table
407  */
408 static struct file_operations my_device_fops =
409 {
410     .owner          = THIS_MODULE,
411     .read           = my_device_read,
412     .write          = my_device_write,
413     .ioctl          = my_device_ioctl,
414     .fasync        = my_device_fasync,
415     .open           = my_device_open,
416     .release       = my_device_release,
417 };
418
419
420 /*

```

```

421  * Module initialization
422  */
423  static int
424  my_device_init(void)
425  {
426      int ret, i;
427
428      printk(KERN_INFO "MyDriver Version %s (built %s:%s)\n",
429             DRIVER_VERSION, __DATE__, __TIME__);
430
431      if((ret = devfs_register_chrdev(240, "mydev", &my_device_fops)) {
432          printk(KERN_ERR "Could not register device with kernel\n");
433          return ret;
434      }
435
436
437      for(i = 0; i < MAX_DEVICES; i++) {
438          /*
439           * Allocate device, allocate device data
440           */
441          my_devices[i].dev_data =
442              (struct device_data*)kmalloc(sizeof(struct device_data), GFP_KERNEL);
443
444          if(!my_devices[i].dev_data) {
445              printk (KERN_WARNING "Could not allocate memory\n");
446              return -ENOMEM;
447          }
448          my_devices[i].dev_data->next = NULL;
449          my_devices[i].end_data = my_devices[i].dev_data;
450
451          my_devices[i].dev_data->ptr = vmalloc(BUFFER_SIZE);
452          if(!my_devices[i].dev_data->ptr) {
453              printk (KERN_WARNING "Could not allocate memory\n");
454              return -ENOMEM;
455          }
456
457          memzero(my_devices[i].dev_data->ptr, BUFFER_SIZE);
458
459          my_devices[i].memsize = BUFFER_SIZE;
460          my_devices[i].needed_mem = BUFFER_SIZE;
461
462          init_timer(&my_devices[i].timer);

```

```
463         my_devices[i].timer.data = (unsigned long)&my_devices[i];
464         my_devices[i].timer.function = my_device_timer;
465
466         init_waitqueue_head(&my_devices[i].waitq);
467
468         init_rwsem(&my_devices[i].sem);
469     }
470
471     return 0;
472 }
473
474 /*
475  * Module exit
476  *
477  * ...we should delete all memory here :)
478  */
479 static void
480 my_device_exit(void)
481 {
482     devfs_unregister_chrdev(240, "mydev");
483
484     return;
485 }
486
487 MODULE_LICENSE("GPL");
488 MODULE_AUTHOR("Deepak Saxena <dsaxena@mvista.com>");
489 MODULE_DESCRIPTION("A test driver");
490
491 module_init(my_device_init);
492 module_exit(my_device_exit);
493
494
```