SHARING MEMORY BETWEEN KERNEL AND USER SPACE IN LINUX

Tomasz Motylewski
Institute of Physical Chemistry, University of Basel, Klingelbergstr. 80, 4056 Basel, Switzerland
motyl@stan.chemie.unibas.ch

Abstract

Techniques allowing kernel tasks and user space processes to share large blocks of memory are presented. In particular, "mbuff" device driver is introduced. Simple examples of synchronizing producer and consumer threads using only shared memory are given.

1 Introduction

Shared memory is the most efficient interprocess communication (IPC) mechanism when big amounts of data are to be transfered or when no built-in synchronization is required. Sharing memory between processes allows them to work like threads on shared variables but still have their own protected address spaces. The other IPC mechanisms like pipes, FIFOs, sockets and files are less efficient because they involve (sometimes multiple) copying of data between processes' memory and kernel buffers. While memory bandwidth of modern PC computers is high[1] also amount of data generated by PCI A/D converters and other equipment is increasing[2].
Sharing memory between user processes in Linux is well known and documented elsewhere[1, 2, 3]. This paper will focus on sharing memory blocks between kernel modules (including real time tasks) and user space.

2 Sharing memory not controlled by Linux

The historically first method of sharing memory between kernel and user processes[4, 5] requires the user to reserve required amount of memory at system boot using `mem=`*nn*`m` parameter where *nn* should be smaller than amount of physical memory in MB. This instructs Linux kernel to use only the first *nn* MB of memory and makes possible to use the rest by mapping (`mmap(2)`) of `/dev/mem`. In the kernel space pointer returned by `ioremap(start,size)` has to by used. Accidentally, since kernel memory map consists of 4 MB pages[3] on i86 architecture, the free memory until the next 4 MB border may be used without ioremap. This has lead to the false belief that the method described here can only be used for blocks smaller than 4 MB. Use of `ioremap` (`vremap` in Linux 2.0) solves this problem. It allows also to access memory-mapped PCI cards in the kernel.

3 Using mbuff driver

The `mbuff.o`[6] module and `/dev/mbuff` is intended to be used as a shared memory device making memory allocated in the kernel using vmalloc possible to map in the user space. Such memory does not need to be reserved at the system startup and its size is not limited by memory fragmentation. The allocated memory is logically (but not physically) continuous. It can not be swapped out, so is well suited for real time applications, especially communication between real time tasks and user space or other high bandwidth kernel-user data exchange. The `mbuff` code is derived from `bttv` driver implementation in Linux 2.2 and 2.4.
The simplest example is the file demo.c distributed with the mbuff package. The `mbuff_alloc()` func-

[1] measured: 44 MB/s Pentium 200; 137 MB/s AMD Athlon 800 MHz
[2] National Instruments PCI-6110E card may generate over 40 MB/s
[3] Used to remap the whole memory at different address at boot. Most other memory blocks in Linux kernel are mapped as 4 KB pages.

tion allocates new area and maps it, or just maps already existing area. The function returns the pointer to the mapped area or `NULL` in the case of failure (no `/dev/mbuff`, bad permissions, `mbuff.o` not loaded, not enough memory, or size greater than the size of already allocated area). The first call does real allocation (swapping out some programs if necessary), the next calls should use the `size` argument equal or less than the one used at the first allocation.

`mbuff_alloc()` should be called by each process accessing the memory, as well as in kernel module.

Every process calling `mbuff_alloc` is responsible for freeing it before exit. It can be done with `mbuff_free()` function. It will unmap the memory and decrease usage counter, so when the last process unmaps the memory, it will be freed. `mbuf` should be the pointer returned initially by `mbuff_alloc()`.

For people who often forget to deallocate the memory, there is `mbuff_attach()` function - it works like `mbuff_alloc()`, except it does not increase usage counter - memory can be deallocated automatically on `munmap()` (e.g. process gets killed). To unmap it earlier `mbuff_detach()` function may be used. All it does is just `munmap()` call. It makes sense to use `mbuff_attach()` and `mbuff_detach()` only in user space.

`mbuff_allocate()` calls `vmalloc()` which may need to swap out some memory - it should not be called from real time nor interrupt nor timer context. Interrupts have to be enabled when `mbuff_allocate()` is called. It is safe to call it from RT-FIFO handler as well as in `init_module()`.

4 Usage examples

Characteristic feature of shared memory is asynchronous ("random") access. It results in less danger of deadlocks, because one thread accessing shared area does not block the others from doing so. Also, small changes in huge data structures do not require writing the whole block again. But care has to be taken to ensure the consistency of data seen by one thread, while the other is changing it. It includes prohibiting the compiler from optimizing (caching in registers) accesses to shared memory (declare all variables kept there `volatile`) and understanding which thread may preempt which other and at which time. Critical regions of the program may be protected by disabling interrupts or using mutexes or semaphores, alternatively designed to keep data structures consistent at any time or to detect whether data has been changed while being read[4].

4.1 Ring buffer

Simplified ring buffer implementation, suitable for use with single producer and consumer threads is presented below. Since producer changes only `r->end` and consumer only `r->start` no locking is needed. The order of operations — "check status; update value; update index" is significant.

```
struct ring_buffer {
    volatile int start,end;
    volatile short buf[SIZE];
} *ring;
ring_init() {
    ring=(struct ring_buffer*)
        mbuff_alloc("mybuffer",
        sizeof(struct ring_buffer));
    ring->start=ring->end=0;
}
ring_destroy() {
mbuff_free("mybuffer",ring);
}
short ring_get(struct ring_buffer *r) {
    short val;
    if(r->start==r->end) {
        return EMPTY;
    }
    val=r->buf[r->start];
    r->start= ++r->start %SIZE;
    return val;
}
int ring_put(struct ring_buffer *r,
        short val) {
    int next= (r->end+1) % SIZE;
    if(next == r->start) {
        return FULL;
    }
    r->buf[r->end] = val;
    r->end = next;
    return OK;
}
```

4.2 Direct DMA to shared memory

Many PCI boards have scatter-gather DMA capability – they can be given the chain (or ring) of physical addresses of memory buffers and then save incoming data to them without further intervention (producer thread). To find physical addresses of `vmalloc`'ed memory pages `kvirt_to_bus()` (`kvmem.h`) should be used.

When the information about the number of bytes already transfered by DMA is not available, each empty page of the buffer may be marked at the beginning and at the end by a "magic" sequence of characters before the transfer is started. After start

the consumer process may check whether both magic sequences on the next page have been already overwritten, if so the data can be read and the page marked as "empty" again. This leads to a very efficient implementation of circular buffer. It does not protect against overruns if consumer process is not fast enough, but allows the producer and consumer threads to work without additional synchronization. For example implementation see `crdsd.c` in [7].

5 Conclusion

Shared memory is an efficient and easy to use way to pass data and communicate between all user space and kernel processes/threads. Applications requiring physically continuous memory, possibly mapped at fixed address should reserve it using `mem=` boot argument and `mmap /dev/mem` (user space) or use `ioremap` (kernel). Other applications may use more flexible mbuff driver to allocate needed memory blocks at any time. Example uses of shared memory are:

- Fast transfer of data between user and kernel space (no copy)
- Exporting internal data and state of kernel tasks to make debugging easier
- Controlling kernel tasks by changing parameters in shared memory

Excellent article of Frederick M. Proctor[4] is recommended for more examples and general techniques used with shared memory. Instead of using `__va()`, `ioremap()` call should be utilized.
Updated version of this article will be available online[8].

References

[1] shmget(2) manual page, *Linux Programmer's manual*, Debian 2.2, manpages_1.29-2_all.deb, 2000.

[2] ipc(5) manual page, *Linux Programmer's Manual*, Debian 2.2, manpages_1.29-2_all.deb, 2000.

[3] Sven Goldt, Sven van der Meer, Scott Burkett and Matt Welsh, IPC chapter in *The Linux Programmer's Guide* version 0.4, March 1995, `http://www.ibiblio.org/pub/linux/docs/linux-doc-project/programmers-guide/`

[4] Frederick M. Proctor *Using Shared Memory in Real-Time Linux* `http://www.isd.cme.nist.gov/projects/emc/shmem.html`

[5] Michael Barabanov, *Shared Memory HOWTO*, `http://www.rtlinux.org/rtlinux/documents/shmem.html`

[6] Tomasz Motylewski, *Kernel – user space shared memory driver mbuff*, `http://crds.chemie.unibas.ch/PCI-MIO-E/mbuff-0.7.2.tar.gz`, 2000

[7] Tomasz Motylewski *Programs for control of CRDS experiment* `http://crds.chemie.unibas.ch/PCI-MIO-E/pcontrol-0.4.4.tar.gz`

[8] Tomasz Motylewski *Sharing Memory Between Kernel and User Space in Linux*, `http://crds.chemie.unibas.ch/linux/shm-orlando/`, 2000