FAQ 4

_Q_: can timer interrupt happens inside other timer interrupt?
_A_: No.

_Q_: converting msec to jiffies
_A_: use code of static function in *itimer.c* tvtojiffies

_Q_: settimer(1,0) handling
_A_: if it is too late then signal should be sent during system call.

_Q_: how to handle jiffies overflow
_A_: Assume that there isn’t any.

_Q_: what if kmalloc fails?
_A_: Assume that kmalloc never fail in ordinary circumstances.

_Q_: in do_timer appears only jiffies_64 and no sign for jiffies.
_A_: it is ok , they point to same memory address.

_Q_: Is it ok to allocate space with first call to timer interrupt?
_A_: No . First call to timer interrupt happens in swapper, no dynamic memory allocation.

_Q_: when we calculate entry for hash table: ASAP or after memory allocation?
_A_: it doesn't matter

_Q_: do we need to use memory barriers?
_A_: Probably, yes. In the end of this FAQ we put the explanation (taken from “Understanding Linux Kernel”) about what are memory barriers and how to use them.

_Q_: what about atomic instructions?
_A_: You should assume tha any instruction of type *a=b takes 2 assembly instructions and between them a timer interrupt can happen.

_Q_: how to send a signal?
_A_: send_sig_info(SIG_ALRM,(void*) 1,t) where t is pointer to 'struct task_struct'.

_Q_: how to allocate memory?
_A_: kmalloc(size,GFP_KERNEL);

_Q_: how to free memory?
_A_: kfree(buffer);

_Q_: where can interrupts happen?
_A_: interrupts can happen only at instruction boundaries.
Memory barriers explanation
(taken from “Understanding Linux Kernel”)

“When using optimizing compilers, you should never take for granted that instructions will be performed in the exact order in which they appear in the source code. For example, a compiler might reorder the assembly language instructions in such a way to optimize how registers are used. Moreover, modern CPUs usually execute several instructions in parallel, and might reorder memory accesses. These kinds of reordering can greatly speed up the program.

When dealing with synchronization, however, instructions reordering must be avoided. As a matter of fact, all synchronization primitives act as memory barriers. Things would quickly become hairy if an instruction placed after a synchronization primitive is executed before the synchronization primitive itself. A memory barrier primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive. Thus, a memory barrier is like a firewall that cannot be passed by any assembly language instruction.”

You can use ‘mb()’ macro for your memory barriers.
‘mb()’ macro defined in <asm/system.h>