


Symmetrical Multiprocessing under Linux



MORE DILITHIUM CRYSTALS FOR THE WARP DRIVE

BERNHARD KUHN

Despite the fact that amazingly fast processors are now available, some situations require more power than even 1GHz or faster CPUs can offer. The solution, of course, is to link more than one CPU together. More complex hardware arrangements mean higher the demands on the operating system and its programmers, however, as you'll find out here.

Multiprocessor systems can be used to great effect in many ways (see box). In the simplest case, several users are logged onto the same multi-processor computer (via X-terminals, for example) and running programs that are not themselves optimised for multiprocessor machines. In such a case the kernel scheduler must distribute user processes optimally over its available CPUs. But if the machine is only intended to deal with a single task, its implementation must be done in parallel in order to be able to make use of all the processors.

The Linux kernel offers two system calls for this; *fork()* and *__clone()*

fork() creates one copy each of code, data and stack elements of the process to be retrieved. Communication between the processes can only occur with the aid of shared memory and IPC calls – both of which are complex and greedy for computer time. For this reason, *__clone()* allows child processes to run in the same address space as the parent process, but each child needs its own stack. Listing 1 shows a simple example of this. The processes here follow notional parallel-running “threads”.

POSIX Threads

The distribution of tasks is only the beginning: Threads sometimes have to be able to synchronise themselves, maybe to protect against accesses to critical resources by other threads. This is done with the aid of what's known as semaphores and mutexes. To make the programmer's life easier, Xavier Leroy has developed a Linux Thread Library, the basic functions of which conform to POSIX 1003.1c. This implementation does have its drawbacks compared to commercial Unix systems. For a start, threads are supposed to display the same process number (PID) in the process table as the parent process (using the flag *CLONE_PID*), but the Linux kernel doesn't withstand this, as large parts of

the system assume that each process number has been issued only once – a minor, but significant flaw. A much more serious problem is the fact that threads are controlled as regular processes and therefore have to be taken into account in every kernel scheduling cycle. Other Unix systems allow the POSIX Thread Library to administer the partial tasks in the user space – with correspondingly lower context interaction times (there is only one regular process per processor and application). The Linux kernel developers, however, have foregone this performance-enhancing feature in favour of stability: simpler implementation makes for more robust code.

Kernel Threads for the user space

Apart from user space processes and threads, tasks relating to the system can also be performed in the address space of the kernel, where the context interaction times are considerably shorter. In this case there is no need to reprogram the memory management unit. The “kernel applications” (for example *kflushd* and *kswapd*) are in fact controlled in the regular process table, but are given preferential treatment – if there is work queuing for them.

Although spectacular performance is possible using kernel threads, the kernel web server “tux 1.0” being a good example, “true” applications should always be laid out as user space program for the sake of system stability. In addition, as we've mentioned elsewhere, the kernel thread API does not conform to POSIX and thus makes it harder to port applications.

Behind the scenes

Every processor has its own kernel scheduler, which differs only slightly from that of a single processor system. If one processor is always more heavily loaded than another one, processes are automatically redistributed. There are patches to bind processes permanently to processors, though. The rumor on the mailing lists at the moment is that this “processor-set

Listing 1: Clone Demo

```
/* clone.c, compile with */
/* gcc -o clone clone.c -D_REENTRANT */

#include sched.h

int loop(void *arg) {
    /* determine number of clones */
    int x=(int)arg;

    /* Closed loop */
    for(;;) printf("%i\n",x);
};

main() {
    /* Stacks for the clones */
    void* stack1[4000];
    void* stack2[4000];

    /* activate clones */
    __clone(loop,stack1+4000,CLONE_VM,1);
    __clone(loop,stack2+4000,CLONE_VM,2);

    /* Closed loop */
    loop(0);
};
```

Symmetrical Multiprocessing or Cluster Computing?

Algorithms to run distributed processing in parallel are only half the story. It matters at least as much whether lot of data is to be exchanged between the processing units or not. The computer system needed for a computer animation, for example, might use what's known as a “render farm”. This consists of individual computers linked by Fast Ethernet (a loose-coupled cluster). A computer used for weather simulation is likely to be the completely different, however. Because of the way finite element analysis works, massive communication needs to takes place between processing nodes so such tasks can only be run on multiprocessor systems. Indeed, since the node to node communications requirements are so high, only multiprocessing supercomputers can handle this kind of job, the Wildfire GS320 supercomputer, for example, offering node to node data transfer rates of a staggering 51.2GB/s.

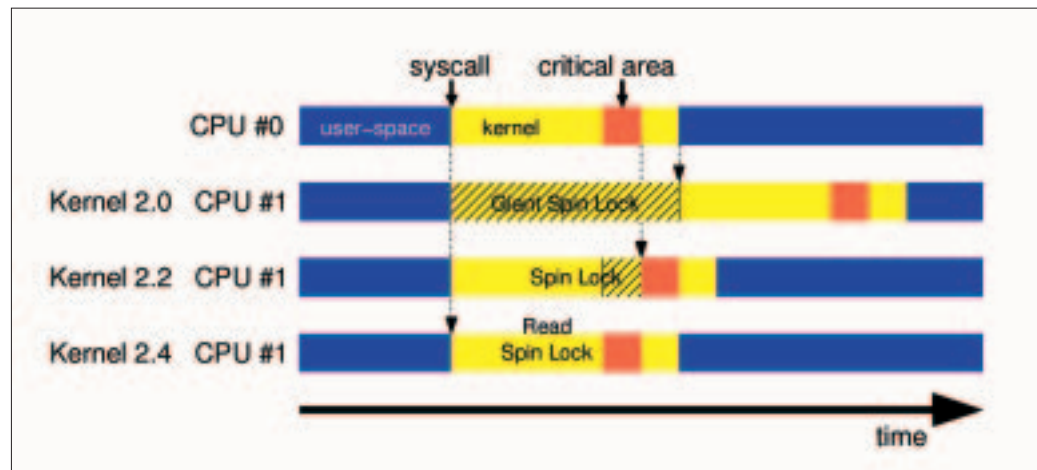
It therefore makes sense to use multiprocessor systems if the applications can actually be run in parallel, but do not permit distribution over several computers. The classic Apache Webserver is a good example of this: the parent process launches several child processes, which wait for incoming HTTP requests and respond to these independently of each other. This means that exactly as many queries can be processed at the same time as there are processors available. This task could of course also be dealt with by a cluster of cheaper single computers, but with an upstream load-balancer and downstream server for the common database, performance can go through the floor: high I/O performance is also a benchmark for a good SMP system.

Classic Unix systems are another good example of the domain of application for multiway computers: many users can log onto a server and start their applications via an X-terminal. The operating system itself then takes care of load distribution.

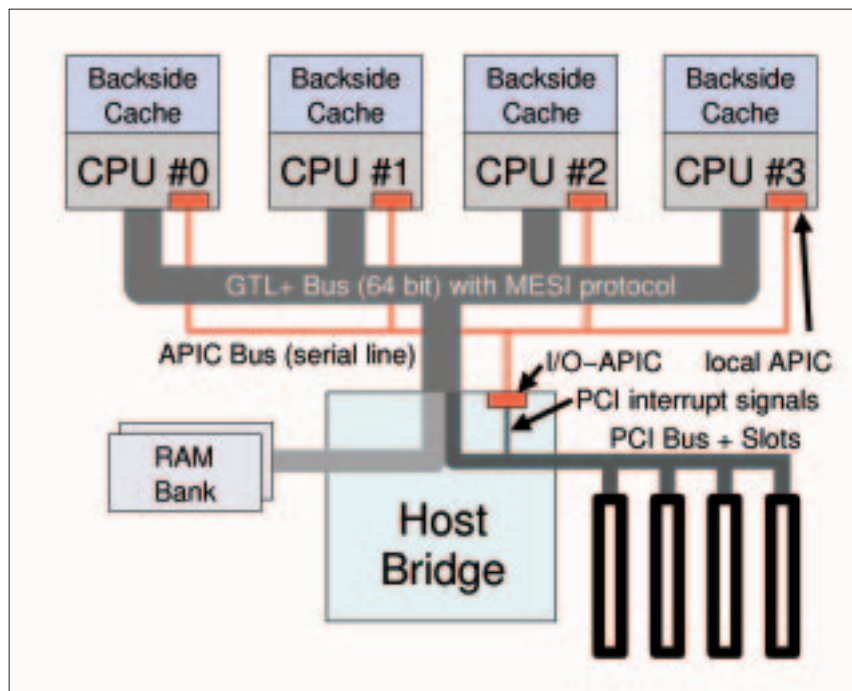
FEATURE

SMP PRINCIPLES

Fig. 1: Comparison between Giant Lock in kernel 2.0, Spin Lock in kernel 2.2 and Read Write Spin Lock in kernel 2.4 (see text).



MESI and APICs make for cache coherence and exchange of messages.



Symmetrical multiprocessing in the Intel IA32

With only a few exceptions, all Linux platforms offer SMP support. The essential basic features of the hardware requirements are covered in the Intel Multiprocessor Specification (MPS) 1.4:

- Cache coherence
- Inter-processor communication

Cache coherence mean that a specific piece of data in the cache of any CPU displays the same value. This means that if an altered value is written back to the main memory, then the other processors, with the aid of the MESI protocol (named after the four statuses Modified, Exclusive, Shared and Invalid), check to see whether the data is in their cache and, if appropriate, take on the new value.

The so-called "local APICs" take care of inter-processor communication: each CPU has an Advance Programmable Interrupt Controller, and with its help can trigger an "Inter Processor Interrupt" in another processor. This is done in order to show that a new message is waiting for it (maybe to initiate a re-scheduling). According to MPS v1.4 this means a maximum of 16 processors can talk to each other. In addition to this, an I/O-APIC integrated in the host bridge announces peripheral interrupts queuing to one or more processors.

scheduling" will get into the next developer kernel (Version 2.5).

With the exception of the scheduler, by the way, the processors share all the data structures in the kernel. Unlike a single processor system, though, the process structure in an SMP kernel contains two additional items, these relating to the processor to which the application was most recently assigned.

In addition to everything else, a protection mechanism has to be in place in multi-CPU systems for what's known as "critical areas". Device drivers in particular have such areas, which are simply code fragments that, during their execution, access system resources – the program sequence for initialising a DMA transfer is a good example of this. If this type of code fragment were to be run on more than one processor at the same time, the most likely outcome would be a system crash.

Giant Lock: the tale of the slow Linux SMP

In kernel version 2.0, the protection of critical areas was achieved by means of a simple trick: At any time, only one processor was able to be active at kernel level – other CPUs were temporarily prevented from entering kernel mode (see Figure 1). This "Giant Lock" mechanism had a particularly bad effect on system performance when running I/O-intensive applications since in such cases the processes might need to be in kernel mode especially frequently. For this reason mechanisms were introduced in kernel 2.2 that only block a processor if it wants to go into a critical area that at that same moment is being processed by another CPU.

Atomic Operations

One critical area, for example, is the manipulation of a value. The C-Compiler will interpret the instruction `cnt=cnt+1` as follow: "Read value from memory, add 1, then write value back". If two processors do this at the same time then there could well be a problem, with the resulting value of `cnt`

Cache in on multiple processors

In practice, one hundred per cent increases in performance in dual processor systems compared to single processors ones are normally only achieved when running server applications. In theory, however, it is even possible to more than double the speed. This is because n processors have n times as much cache memory as a single CPU, and in cyclical processes, this therefore makes displacement of data more unlikely. For example, each byte of a 4 Mbyte field is to be increased cyclically by the value 1. If only one processor with one megabyte of cache handles this task, then the fast buffer memory is practically ineffective. But, if the task is distributed over four CPUs, then each processor keep its "Working Set" completely in the cache and executes the operations at a corresponding faster speed.

not being as expected. In order to prevent this effect, assembler instructions an enable "atomic execution", which means the manipulation cannot be interrupted. The code for the above example can be found in the kernel file

/usr/src/linux/include/asm-i386/atomic.h:

```
static __inline__ void atomic_inc(atomic_t *v)
{
    __asm__ __volatile__(LOCK "incl %0"
        : "=m" (v->counter) : "m" (v->counter));}
```

Source and target are in the memory and are processed by only one command – so no other processor has a chance to disturb the manipulation.

Spin Locks: in the queue loop

Frequently, critical areas of a more complex nature are executed as atomic operations, our DMA transfer code sequence mentioned previously is a good example of this. Each of these critical areas is assigned what's known as a Spin Lock: If a processor goes into a critical area, the lock is (atomically) activated. If the lock was already occupied by another processor, then the CPU affected checks the lock continually in a program loop, until the critical area has been left by the other processor and the lock has been deactivated. The processor does squander valuable computing time in the queue loop, but normally the block is much too short to make it worthwhile transferring other tasks from the CPU.

Read Write Spin Locks

Very often, processors dealing with critical areas only need to read data – this can obviously be done by several CPUs at the same time: for this purpose a *rwlock* is able to block only processors which want to actively write in the critical area. This fact was all but ignored by the driver developers with kernel 2.2 and Read Write Spin Locks were therefore seldom used. This was remedied in 2.4 and therefore is another reason the later kernel can offer better performance – particularly in computers with eight or more processors.

SMP in practice

The increase in speed brought about by the use of several processors largely depends on the application. In a dual-Celeron system, for example, the Skyvase-Benchmark (pvmpov) runs some 60 per cent faster than with just one CPU. With the kernel compilation option *MAKE="make -j3"* this can be increased to as much as 90 per cent. Under some circumstances two processors can, as the result of caching effects, even end up being be more than twice as fast as a single processor (see box - Cache in on multiple processors).

Linux SMP has a great deal to offer and can increase application execution speeds, particularly server applications, to a greater extent than you might think. Its development is far from over, though, and you can expect to see enhancements in future kernel releases that will make it even more efficient.

Further reading

Linux-Threads FAQ:

<http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html>

Linux-SMP HOWTO:

<http://www.irisa.fr/prive/dmenetre/smp-faq/smp-howtohtml>

Intel MPS v1.4:

<http://developer.intel.com/design/PentiumIII/manuals/>

Mutexes, Semaphores and Status Variables**Mutexes**

*MUTual EXclusions serve to reserve resources exclusively for threads. By doing this a mutex can only ever be acquired by a single thread – if additional threads attempt to claim this mutex for themselves (*pthread_mutex_lock*), they will be blocked until the current claimant releases the mutex again (*pthread_mutex_unlock*). The fact that acquisition occurs atomically (see later) also ensures that only one thread can ever occupy a mutex at any one same time.*

Condition variables

*Condition variables serve to await the onset of specified conditions and/or to display their fulfilment. They are therefore used to synchronise threads. Condition variables are linked to mutexes in order to ensure that he conditions can only ever be altered by one thread at a time. Modification of the condition variables is reported to the waiting threads with *pthread_cond_broadcast()* with the aid of *pthread_cond_wait*.*

Semaphores

Semaphores (or lighthouses) are the more general versions of Mutexes and condition variables. Unlike mutexes, semaphores don't just allow two conditions (acquired/released), but as many as required, where the following counting operations are allowed:

- *atomic incrementing he counter by one followed by waiting until value becomes zero again*
- *atomic decrementing the counter*

Semaphores were originally developed to avoid the problem of "lost signals".