# An Implementation Of Multiprocessor Linux

*This document describes the implementation of a simple SMP Linux kernel extension and how to use this to develop SMP Linux kernels for architectures other than the Intel MP v1.1 architecture for Pentium and 486 processors.*

Alan Cox, 1995

The author wishes to thank Caldera Inc. ( http://www.caldera.com ) whose donation of an ASUS dual Pentium board made this project possible, and Thomas Radke, whose initial work on multiprocessor Linux formed the backbone of this project.

## 1 Background: The Intel MP specification.

Most IBM PC style multiprocessor motherboards combine Intel 486 or Pentium processors and glue chipsets with a hardware/software specification. The specification places much of the onus for hard work on the chipset and hardware rather than the operating system.

The Intel Pentium processors have a wide variety of inbuilt facilities for supporting multiprocessing, including hardware cache coherency, built in interprocessor interrupt handling and a set of atomic test and set, exchange and similar operations. The cache coherency in particular makes the operating system's job far easier.

The specification defines a detailed configuration structure in ROM that the boot up processor can read to find the full configuration of the processors and buses. It also defines a procedure for starting up the other processors.

## 2 Mutual Exclusion Within A Single Processor Linux Kernel

For any kernel to function in a sane manner it has to provide internal locking and protection of its own tables to prevent two processes updating them at once and for example allocating the same memory block. There are two strategies for this within current Unix and Unixlike kernels. Traditional Unix systems from the earliest of days use a scheme of 'Coarse Grained Locking' where the entire kernel is protected by a small number of locks only. Some modern systems use fine grained locking. Because fine grained locking has more overhead it is normally used only on multiprocessor kernels and real time kernels. In a real time kernel the fine grained locking reduces the amount of time locks are held and reduces the critical (to real time programming at least) latency times.

Within the Linux kernel certain guarantees are made. No process running in kernel mode will be pre-empted by another kernel mode process unless it voluntarily sleeps. This ensures that blocks of kernel code are effectively atomic with respect to other processes and greatly simplifies many operations. Secondly interrupts may pre-empt a kernel running process, but will always return to that process. A process in kernel mode may disable interrupts on the processor and guarantee such an interruption will not occur. The final guarantee is that an interrupt will not be pre-empted by a kernel task. That is interrupts will run to completion or be pre-empted by other interrupts only.

The SMP kernel chooses to continue these basic guarantees in order to make initial implementation

and deployment easier. A single lock is maintained across all processors. This lock is required to access the kernel space. Any processor may hold it and once it is held may also re-enter the kernel for interrupts and other services whenever it likes until the lock is relinquished. This lock ensures that a kernel mode process will not be pre-empted and ensures that blocking interrupts in kernel mode behaves correctly. This is guaranteed because only the processor holding the lock can be in kernel mode, only kernel mode processes can disable interrupts and only the processor holding the lock may handle an interrupt.

Such a choice is however poor for performance. In the longer term it is necessary to move to finer grained parallelism in order to get the best system performance. This can be done hierarchically by gradually refining the locks to cover smaller areas. With the current kernel highly CPU bound process sets perform well but I/O bound task sets can easily degenerate to near single processor performance levels. This refinement will be needed to get the best from Linux/SMP.

## 2.1 Changes To The Portable Kernel Components

The kernel changes are split into generic SMP support changes and architecture specific changes necessary to accommodate each different processor type Linux is ported to.

### 2.1.1 Initialisation

The first problem with a multiprocessor kernel is starting the other processors up. Linux/SMP defines that a single processor enters the normal kernel entry point start_kernel(). Other processors are assumed not to be started or to have been captured elsewhere. The first processor begins the normal Linux initialisation sequences and sets up paging, interrupts and trap handlers. After it has obtained the processor information about the boot CPU, the architecture specific function

**void smp_store_cpu_info(int processor_id)**

is called to store any information about the processor into a per processor array. This includes things like the bogomips speed ratings.

Having completed the kernel initialisation the architecture specific function

**void smp_boot_cpus(void)**

is called and is expected to start up each other processor and cause it to enter start_kernel() with its paging registers and other control information correctly loaded. Each other processor skips the setup except for calling the trap and irq initialisation functions that are needed on some processors to set each CPU up correctly. These functions will probably need to be modified in existing kernels to cope with this.

Each additional CPU then calls the architecture specific function

**void smp_callin(void)**

which does any final setup and then spins the processor while the boot up processor forks off enough idle threads for each processor. This is necessary because the scheduler assumes there is always something to run. Having generated these threads and forked init the architecture specific

**void smp_commence(void)**

function is invoked. This does any final setup and indicates to the system that multiprocessor mode is now active. All the processors spinning in the smp_callin() function are now released to

run the idle processes, which they will run when they have no real work to process.

### 2.1.2 Scheduling

The kernel scheduler implements a simple but very effective task scheduler. The basic structure of this scheduler is unchanged in the multiprocessor kernel. A processor field is added to each task, and this maintains the number of the processor executing a given task, or a magic constant (NO_PROC_ID) indicating the job is not allocated to a processor.

Each processor executes the scheduler itself and will select the next task to run from all runnable processes not allocated to a different processor. The algorithm used by the selection is otherwise unchanged. This is actually inadequate for the final system because there are advantages to keeping a process on the same CPU, especially on processor boards with per processor second level caches.

Throughout the kernel the variable 'current' is used as a global for the current process. In Linux/SMP this becomes a macro which expands to current_set[smp_processor_id()]. This enables almost the entire kernel to be unaware of the array of running processors, but still allows the SMP aware kernel modules to see all of the running processes.

The fork system call is modified to generate multiple processes with a process id of zero until the SMP kernel starts up properly. This is necessary because process number 1 must be init, and it is desirable that all the system threads are process 0.

The final area within the scheduling of processes that does cause problems is the fact the uniprocessor kernel hard codes tests for the idle threads as task[0] and the init process as task[1]. Because there are multiple idle threads it is necessary to replace these with tests that the process id is 0 and a search for process ID 1, respectively.

### 2.1.3 Memory Management

The memory management core of the existing Linux system functions adequately within the multiprocessor framework providing the locking is used. Certain processor specific areas do need changing, in particular invalidate() must invalidate the TLBs of all processors before it returns.

### 2.1.4 Miscellaneous Functions

The portable SMP code rests on a small set of functions and variables that are provided by the processor specification functionality. These are

**int smp_processor_id(void)**

which returns the identity of the processor the call is executed upon. This call is assumed to be valid at all times. This may mean additional tests are needed during initialisation.

**int smp_num_cpus;**

This is the number of processors in the system.

**void smp_message_pass(int target, int msg, unsigned long data, int wait)**

This function passes messages between processors. At the moment it is not sufficiently defined to sensibly document and needs cleaning up and further work. Refer to the processor specific code documentation for more details.

## 2.2 Architecture Specific Code For the Intel MP Port

The architecture specific code for the Intel port splits fairly cleanly into four sections. Firstly the initialisation code used to boot the system, secondly the message handling and support code, thirdly the interrupt and kernel syscall entry function handling and finally the extensions to standard kernel facilities to cope with multiple processors.

### 2.2.1 Initialisation

The Intel MP architecture captures all the processors except for a single processor known as the 'boot processor' in the BIOS at boot time. Thus a single processor enters the kernel bootup code. The first processor executes the bootstrap code, loads and uncompresses the kernel. Having unpacked the kernel it sets up the paging and control registers then enters the C kernel startup.

The assembler startup code for the kernel is modified so that it can be used by the other processors to do the processor identification and various other low level configurations but does not execute those parts of the startup code that would damage the running system (such as clearing the BSS segment).

In the initialisation done by the first processor the arch/i386/mm/init code is modified to scan the low page, top page and BIOS for intel MP signature blocks. This is necessary because the MP signature blocks must be read and processed before the kernel is allowed to allocate and destroy the page at the top of low memory. Having established the number of processors it reserves a set of pages to provide a stack come boot up area for each processor in the system. These must be allocated at startup to ensure they fall below the 1Mb boundary.

Further processors are started up in smp_boot_cpus() by programming the APIC controller registers and sending an inter-processor interrupt (IPI) to the processor. This message causes the target processor to begin executing code at the start of any page of memory within the lowest 1Mb, in 16bit real mode. The kernel uses the single page it allocated for each processor to use as stack. Before booting a given CPU the relocatable code from trampoline.S and trampoline32.S is copied to the bottom of its stack page and used as the target for the startup.

The trampoline code calculates the desired stack base from the code segment (since the code segment on startup is the bottom of the stack), enters 32bit mode and jumps to the kernel entry assembler. This as described above is modified to only execute the parts necessary for each processor, and then to enter start_kernel(). On entering the kernel the processor initialises its trap and interrupt handlers before entering smp_callin(), where it reports its status and sets a flag that causes the boot processor to continue and look for further processors. The processor then spins until smp_commence() is invoked.

Having started each processor up the smp_commence( ) function flips a flag. Each processor spinning in smp_callin() then loads the task register with the task state segment (TSS) of its idle thread as is needed for task switching.

### 2.2.2 Message Handling and Support Code

The architecture specific code implements the smp_processor_id() function by querying the APIC logical identity register. Because the APIC isn't mapped into the kernel address space at boot, the initial value returned is rigged by setting the APIC base pointer to point at a suitable constant. Once the system starts doing the SMP setup (in smp_boot_cpus()), the APIC is mapped with a

vremap() call and the apic pointer is adjusted appropriately. From then on the real APIC logical identity register is read.

Message passing is accomplished using a pair of IPIs on interrupt 13 (unused by the 80486 FPUs in SMP mode) and interrupt 16. Two are used in order to separate messages that cannot be processed until the receiver obtains the kernel spinlock from messages that can be processed immediately. In effect IRQ 13 is a fast IRQ handler that does not obtain the locks, and cannot cause a reschedule, while IRQ 16 is a slow IRQ that must acquire the kernel spinlocks and can cause a reschedule. This interrupt is used for passing on slave timer messages from the processor that receives the timer interrupt to the rest of the processors, so that they can reschedule running tasks.

### 2.2.3 Entry And Exit Code

A single spinlock protects the entire kernel. The interrupt handlers, the syscall entry code and the exception handlers all acquire the lock before entering the kernel proper. When the processor is trying to acquire the spinlock it spins continually on the lock with interrupts disabled. This causes a specific deadlock problem. The lock owner may need to send an invalidate request to the rest of the processors and wait for these to complete before continuing. A processor spinning on the lock would not be able to do this. Thus the loop of the spinlock tests and handles invalidate requests. If the invalidate bit for the spinning CPU is set the processor invalidates its TLB and atomically clears the bit. When the spinlock is obtained that processor will take an IPI and in the IPI test the bit and skip the invalidate as the bit is clear.

One complexity of the spinlock is that a process running in kernel mode can sleep voluntarily and be pre-empted. A switch from such a process to a process executing in user space may reduce the lock count. To track this the kernel uses a syscall_count and a per process lock_depth parameter to track the kernel lock state. The switch_to() function is modified in SMP mode to adjust the lock appropriately.

The final problem is the idle thread. In the single processor kernel the idle thread executes 'hlt' instructions. This saves power and reduces the running temperature of the processors when they are idle. However it means the process spends all its time in kernel mode and would thus hold the kernel spinlock. The SMP idle thread continually reschedules a new task and returns to user mode. This is far from ideal and will be modified to use 'hlt' instructions and release the spinlock soon. Using 'hlt' is even more beneficial on a multiprocessor system as it almost completely takes an idle processor off the bus.

Interrupts are distributed by an i82489 APIC. This chip is set up to work as an emulation of the traditional PC interrupt controllers when the machine boots (so that an Intel MP machine boots one CPU and PC compatible). The kernel has all the relevant locks but does not yet reprogram the 82489 to deliver interrupts to arbitrary processors as it should. This requires further modification of the standard Linux interrupt handling code, and is particularly messy as the interrupt handler behaviour has to change as soon as the 82489 is switched into SMP mode.

### 2.2.4 Extensions To Standard Facilities

The kernel maintains a set of per processor control information such as the speed of the processor for delay loops. These functions on the SMP kernel look the values up in a per processor array that is set up from the data generated at boot up by the smp_store_cpu_info() function. This includes other facts such as whether there is an FPU on the processor. The current kernel does not handle

floating point correctly, this requires some changes to the techniques the single CPU kernel uses to minimise floating point processor reloads.

The highly useful atomic bit operations are prefixed with the 'lock' prefix in the SMP kernel to maintain their atomic properties when used outside of (and by) the spinlock and message code. Amongst other things this is needed for the invalidate handler, as all CPU's will invalidate at the same time without any locks.

Interrupt 13 floating point error reporting is removed. This facility is not usable on a multiprocessor board, nor relevant to the Intel MP architecture which does not cover the 80386/80387 processor pair.

The /proc filesystem support is changed so that the /proc/cpuinfo file contains a column for each processor present. This information is extracted from the data saved by smp_store_cpu_info().