

The Implementation of SMP in VSTa 1.6.8

Status Report

Warren Toomey, Bond University
wkt@tuhs.org

21st November 2003

Abstract

This document outlines the changes that were made to VSTa 1.6.8 to support SMP, and why they were made. At present, there are still several bugs to be fixed in the SMP implementation.

1 Introduction

VSTa is an elegant microkernel that supports threads, process address spaces, IPC and kernel pre-emption. VSTa provides both spinlock and semaphore abstractions to synchronise separate kernel entities and to guarantee atomicity on kernel activities. However, VSTa 1.6.8 does not support any SMP platform, in that it cannot start multiple processors and co-ordinate their activities.

Initial SMP support for the Pentium platform has been added to VSTa 1.6.8. We will explain how this was achieved, the architecture of the SMP code, how it was implemented, how VSTa was modified to support SMP, and what SMP issues are still outstanding.

2 Modifications to Existing VSTa Code

In order to support SMP on the Pentium platform, several modifications needed to be made to the original VSTa 1.6.8 kernel source code. The main areas of modification are as follows:

1. VSTa keeps a `percpu` structure that details information about the system's CPU and the thread that is executing on that CPU. In an SMP environment, there needs to be a `percpu` structure for each CPU.
2. The "`nextcpu`" concept in `preempt()` is not required, as every CPU will execute `preempt()`.
3. On the Pentium architecture, each CPU needs its own global descriptor table and its own task state selector.

4. VSTa keeps a single idle stack, used when context switching. Each CPU needs its own idle stack.
5. It is useful to allow a CPU to send an interprocessor interrupt (IPI) to another CPU. VSTa was modified to have one new interrupt type for IPIs.
6. The implementation of spinlocks in VSTa assumes that a lock's value can simply be incremented without an atomic test-and-set operation. This was rewritten for an SMP environment.

Details of these changes are given below, and a file-by-file description of the code changes are given in Section 8.

2.1 The percpu structure

The percpu structure, defined in *include/sys/percpu.h*, holds machine-independent information that is specific to a CPU, such as a pointer to the executing thread, the state of the CPU, if pre-emption is required and other time statistics.

VSTa 1.6.8 has a single percpu structure called *cpu*. For SMP VSTa, there needs to be multiple percpu structures, one per processor. These can be implemented in at least two ways. The first approach is to create an array of *N* structures, and then to have each CPU index into the array according to its identity.

This approach is currently in use in the implementation of SMP VSTa, but it has considerable overhead. Each CPU must continually obtain its own identity to index the percpu array. This identity cannot be kept in a global variable, as these are shared by all CPUs. The CPU identity could be stored in local variables, but this requires that it be passed from function to function, which would require substantial rewriting of the VSTa function interfaces. As the percpu structure is used frequently, efforts must be made to reduce the overhead of this approach.

An alternative approach relies on the fact that each CPU has its own MMU and thus will have separate virtual to physical page mappings. Using the separate MMUs, we can allocate a separate page frame to each CPU but map all the frames to the same virtual address in kernel memory. Then, when a CPU accesses the single *cpu* structure, it is actually accessing its own private instance of the structure. Percpu structure access thus does not require array indexing nor the constant determination of the CPU's identity. Unfortunately, the design of the Intel x86 architecture combined with the use of separate segments for kernel-mode and user-mode have made this alternative next to impossible to implement. Although most of the code to implement "private" pages has been done, I believe it would take significant structural change to VSTa in order to implement it correctly.

2.2 The *nextcpu* Concept

The *preempt()* routine in *kern/sched.c* seems to have a CPU try to find the best CPU amongst the list of CPUs to pre-empt. This seems to be superfluous, as every CPU is going to try to execute this code. I have simply replaced it with a call to *nudge()* on the current processor. The *pc_next* field has been removed from the percpu structure.

2.3 GDTs, IDTs and TSSs

Initially, I had hoped that each CPU could share the same Global Descriptor Table, the same Interrupt Descriptor Table and the same Task Segment Selector. This turned out not to be the case for the GDT and TSS. Once a TSS has been used, it is marked as 'Busy', and another CPU cannot pass through the same TSS entry. Therefore, the code in *init_trap()* had to be modified to create a different TSS for each CPU. As the TSS is linked in to the GDT, this also forced each CPU to have its own GDT. Fortunately, the IDT can be shared amongst all the CPUs.

2.4 Kernel Stacks and the Idle Stack

Each CPU needs its own stack to boot, its own kernel stack when in kernel mode but running a user-mode thread, and an idle stack when context switching between threads. At present, the boot processor has its own boot stack, and there is a separate boot stack which each other CPU uses in turn to initialise itself. Synchronisation is used to ensure that this second stack is used by only one CPU at a time.

The idle stack is more difficult. It is used in *resume()* as a part of the context switch. Each CPU needs its own idle stack. The global idle stack has been removed, and an idle stack has been added to the percpu structure. The files that manipulate and use the idle stack (*kern/proc.c*, *kern/sched.c*, *mach/locore.h* and *mach/trap.c*) have been modified accordingly.

2.5 Interprocessor Interrupts

In an ideal SMP environment, all CPUs will receive all external interrupts. This is not the case when running the Bochs simulator with two or more simulated CPUs. Only the first processor will receive external interrupts. This prevents the other processors from completing I/O, which is not too drastic as the first processor can do this. However, it does prevent the other processors from pre-empting a thread which has exceeded its timeslice.

To rectify this, conditional code has been added to the *interrupt()* function for the boot processor to redistribute a clock tick to the other processors via an interprocessor interrupt. The code can be compiled out via a flag in *make/makefile*.

2.6 VSTa Spinlocks

Spinlocks in VSTa are defined in the *mach/mutex.h* file. Although VSTa provides the spinlock abstraction, on a uniprocessor platform there will never be any contention for a spinlock¹. Thus, the implementation of *p_lock()* and *p_lock_void()* assert that the lock is zero (available) and then sets the lock value to 1 (acquired).

SMP VSTa dispenses with the assertion, as one processor may have acquired the lock. The simple lock increment code has now been replaced with a true spinlock using the Pentium *xchg* instruction. However, the caller of *p_lock()* and *p_lock_void()* must disable interrupts beforehand, otherwise an interrupt handler may try to reacquire the same spinlock, and wait indefinitely for the lock to be released. Recursive spinlocks have not been implemented yet; it can be argued

¹This assume that interrupts are disabled before acquiring the spinlock.

that allowing recursive spinlocks also allows the kernel designer to be lax about the use of these abstractions.

No modification was required for the spinlock release code, *v_lock()*, as the lock's value can simply be decremented as is done already.

3 Implementing the Intel SMP Architecture

The new VSTa code that probes for CPUs, sends initialisation messages to each CPU, and the bootstrap code for each CPU, is in four new files in the *os/mach* directory:

mp.c This file holds the SMP probe and CPU initialisation code, along with associated support routines.

mp.h This file holds preprocessor defines and structure definitions used by *mp.c*.

mp_ipi.h This file defines the different interprocessor interrupts (IPIs) that can be delivered.

mps.s This file holds the assembly code used to bootstrap a CPU from 16-bit to 32-bit mode.

The SMP code is heavily based on the SMP code written by Jesús M. Álvarez Llorente for Minix (<http://webepcc.unex.es/~jalvarez/minixsmp/>), and who has given written consent to the use of his code. Although the architecture of Minix is significantly different to VSTa, most of the techniques used to probe for an SMP system and to boot the Application processors are identical. The technical report that Jesús wrote about SMP Minix is excellent and I highly recommend reading it, or the Babelfish translation into English².

In the Intel SMP architecture, only one CPU initially starts, and it is called the Boot System Processor or BSP. The remaining CPUs (known as Application Processors, or APs) remain halted until the BSP sends the command to start them.

Each processor has its own interrupt controller, known as the local APIC. For the purposes of the VSTa SMP implementation, the local APIC allows a processor to determine its identity, and to send IPIs to other processors. The local APIC is mapped into the same set of physical locations for each CPU. The Intel SMP architecture also requires an I/O APIC, which manages the delivery of external interrupts to the CPUs. At present, we do not use the I/O APIC, and it retains its default behaviour.

3.1 The Files *mach/mp.h* and *mach/mp_ipi.h*

These two files define constants and structures used in other C files. *mach/mp.h* is used only by *mach/mp.c*, and *mach/mp_ipi.h* is only used by *mach/trap.c*.

mach/mp_ipi.h defines the set of IPI messages that can be sent between processors. At present, there is a single IPI message used by the BSP to deliver clock ticks to the APs when we are running on the Bochs simulator.

mach/mp.h defines several structures which are used when probing the system to determine if it is SMP-capable, and to determine how many CPUs and APICs it has. The file also contains the definitions of the local APIC and I/O APIC registers.

²I can provide the Babelfish translation as a text file (no pictures) on request.

3.2 The File mach/mp.c

This file contains most of the new code to VSTa to deal with the Intel SMP architecture, specifically for detecting if a system has SMP and initialising the Application Processors. We will summarise each function, and only give a discussion if a function has an important aspect or if it contains an implication for the rest of the system.

LOCAL_APIC_WRITE() Writes a 32-bit value into one of the local APIC registers.

LOCAL_APIC_READ() Reads a 32-bit value from one of the local APIC registers.

map_apics() Map the local and I/O APICs for all CPUs into kernel virtual addresses. This has to be done as the APICs have physical addresses above the 2G mark, and will be invisible once user threads execute. We duplicate the code to do the mapping from *mach/ominline.h*, as this code will only map pages into the kernel utility area. This function is called at the end of *init_machdep()*.

apic_error_status() Return the error value from the local APIC's status register.

get_cpuid() Obtain the CPU's unique identity from the local APIC as an integer. This function is used heavily in the system by such macros as *curthread*, and to index into the *cpu[]* array. At present it is an ordinary C function, but it should be optimised and converted into an inline function.

dump_local_apic() A debug routine to dump the contents of the local APIC's registers.

cmos_write() Write data into a CMOS register. This is used by *send_init_ipi()* to ensure that each AP does a warm restart and not a cold restart.

cmos_read() Read data from a CMOS register.

enable_apic_ints() Enable the local APIC and allow it to receive interrupts from the I/O APIC.

wait_for_ipi_completion() After sending an IPI via the local APIC to another CPU, wait for that IPI to complete. This requires that the other CPU acknowledge the IPI with *ack_ipi()*.

send_init_ipi() In the Intel SMP architecture, a CPU begins executing instructions after two special IPIs have been sent to it: the Init IPI and the Startup IPI. This function sends the Init IPI to an AP. The AP's reset vector is set to point at the "trampoline" code, which it will execute before entering *ap_main()*.

send_startup_ipi() Send the Startup IPI to an AP.

send_ipi() Send a general IPI to another CPU. The IPI command and any parameter are stored in a global array for the receiving CPU to read.

recv_ipi() Obtain the IPI command and its parameter from the global array. The function does not touch the local APIC, as it is invoked by the CPU having received an interrupt.

ack_ipi() Return an acknowledgment to the received IPI via the local APIC.

find_mpfp() Find the MP Floating Pointer structure in the system's BIOS. This indicates if the system is SMP capable. If so, the MPFP structure either has a "canned" description of the type of SMP system, or it points to another data structure with more detailed information about the SMP system.

process_mpct() Follow the pointer from the MPFP to the MP Configuration Table. Determine the number of CPUs and local APICs that the system has, and return the number of CPUs.

find_trampoline() Find a low-memory zone suitable for allocating the “trampoline” code, which is used to bootstrap each AP. The memory zone must be aligned on a 4K boundary and be all zeroes. Once it is found, the bootstrap code is copied from its current location in the VSTa kernel down to low-memory.

probe_smp() This function is called from *init_machdep()*. It runs *find_mpfp()* and *process_mpct()* to determine if the system is SMP capable, how many CPUs it has, and checks that the VSTa kernel has been compiled to support the number of CPUs found. If it returns, *init_machdep()* then does *map_apics()* if there are two or more CPU. Then, the BSP’s percpu structure is initialised with *init_percpu()*. The BSP then calls *init_trap()* to initialise the GDT structure, the TSS structures for each CPU, and the IDT structure. After this, the BSP can finally call *init_smp()*.

init_smp() This function must be called after the GDT, TSS and IDT structures are initialised. Use *find_trampoline()* to copy the trampoline code down to low-memory. Enable the local APIC for the BSP. For each AP, set the global variable *ap_cpuid* to that CPU’s identity, and then send the Init and Startup IPIs to the AP. Loop waiting for the AP to signal that it is up when the value of *ap_cpuid* returns to zero. This ensures that each AP in turn initialises, and synchronises use of the stack reserved for booting of the APs.

ap_main() This is the entrypoint of the AP into the VSTa kernel, and much of the code is similar to that performed by *init_machdep()*. However, the AP does not have to initialise the kernel’s heap, nor most of the kernel’s data structures.

Set up CR0 as per the BSP. Switch to the Boot Processor’s page table and turn on paging. Load the IDT set up by the boot processor. Run *init_percpu()* to initialise this AP’s percpu structure. Set up the TSS that the BSP initialised for us. Flush the segment registers to point at the GDT for this processor; each CPU has its own GDT, because it has to contain a TSS unique to each CPU. Enable delivery of interrupts from the I/O APIC, but keep interrupts disabled for now. At this point we can reset *ap_cpuid* to zero to allow the BSP to start another AP.

The AP is ready to enter the kernel, but it must wait for the BSP to finish the initialisation of the kernel’s data structures. Once the BSP has set the *upyet* flag, the AP can spinlock the run queue and call *swtch()* to select a thread to run.

3.3 The File mach/mps.s

This file contains the code used to bootstrap each AP. The machine code is copied into low memory by *find_trampoline()*. The code clears the general registers, sets up enough segment registers to continue in real-mode, loads the GDT and IDT in use by the BSP, jumps into protected mode (setting a new CS register), loads the segment registers that VSTa uses when it is in kernel mode, points the stack pointer at a stack reserved for AP booting, resets some CP flags and then calls *ap_main()*.

4 Simulation on Bochs

The main development work on the SMP implementation is being done on the Bochs simulator (<http://bochs.sourceforge.net>). Bochs simulates the Pentium/PC architecture, and it can simulate up to 8 CPUs together with the requisite local and I/O APICs.

Bochs has a built-in debugger which allows the developer to do several things such as:

- stop the simulation at any time,
- inspect the contents of the CPUs' registers,
- disassemble instructions in memory,
- setting instruction breakpoints and memory watchpoints,
- tracing the execution of code, and
- allowing the simulated operating system to drop into the debugger via the use of pseudo I/O ports.

Bochs can run using several user interfaces. The X11 interface allows VGA graphics and normal text modes (colours, bold, underline etc.) The text-based interface displays the text display using an 80x25 'curses' display, allowing execution output to be captured using Unix tools such as *script(1)*. Bochs also simulates the parallel printer port, and the VSTa kernel has been modified to output diagnostic information to the printer port, which is then captured to a normal text file.

While Bochs has been extremely useful for debugging the SMP implementation, it has some bugs and flaws. Most notably, Bochs will only send external interrupts to the boot processor. The other simulated processors in Bochs never receive external interrupts, although they do receive traps and exceptions. As well, after several kernel page faults, Bochs ask the user if they wish to invoke the debugger, but after asking for this to occur Bochs goes into an infinite loop.

4.1 Downloading Bochs

Bochs can be downloaded from its CVS area on the sourceforge CVS server as follows:

```
cvscvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/bochs login
cvscvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/bochs co bochs
```

If you are asked for a password, simply hit 'Enter'. A directory called *bochs* will be created and the source code will be placed in this directory.

4.2 Patching Bochs

For some reason, Bochs will crash when we try to perform some operation on the local APIC. Edit the file *cpu/apic.cc*, find this line around line 145 and comment it out:

```
BX_PANIC(("get_type called on base class"));
```

4.3 Compiling Bochs for Two CPUs

Bochs is configured via the now-standard `./configure` method. This is the command that can be used to configure Bochs to have two CPUs:

```
./configure --with-x11 --with-term --enable-processors=2 \  
--enable-debugger --enable-disasm --enable-iodebug \  
--enable-reset-on-triple-fault=0
```

With this done, you can simply `make` to build the *bochs* executable. The result is an executable called *bochs*. At this point it might be wise to rename this to be *2bochs*.

4.4 Compiling Bochs for One CPU

After building a 2-CPU version of Bochs, it is also a good idea to build a Bochs executable that simulates a single CPU. VSTa can boot on this with no issues, and this is the usual way to modify and rebuild the kernel. After renaming *bochs* to *2bochs*, you should now `make dist-clean`.

The command that can be used to configure Bochs to have one CPU is:

```
./configure --with-x11 --with-term
```

With this done, you can simply `make` to build the *bochs* executable. The result is an executable called *bochs*.

4.5 Executing Bochs

To execute, Bochs requires:

- a disk image and an optional floppy image,
- a configuration file called *.bochsrc*,
- the location of the BIOS image and the VGA ROM image defined in *.bochsrc*.

It is recommended that you keep two configuration files: one for booting *bochs* and the other for booting *2bochs*. An example of a configuration file to boot *2bochs* is given in Section 9. This has had all comments removed. Note that the BIOS image is different for *bochs* vs. *2bochs*, but all other configuration lines can stay the same. Note also that in this configuration, VSTa is booted via a GRUB floppy. The *menu.lst* file on the GRUB floppy has an option for booting normal VSTa, or a VSTa kernel with SMP support.

Once you have built two configuration files, then you can run Bochs as follows:

```
% bochs -q -f .bochsrc          or  
% 2bochs -q -f .2bochsrc
```

As *2bochs* has been compiled with the debugger, you are immediately taken into the debugger. You must type in 'c' and Enter to continue with the execution.

For more details on how to compile, configure and use Bochs, please see the Bochs website at <http://bochs.sourceforge.net>.

5 Compiling VSTa for SMP and non-SMP Platforms

Two new C preprocessor defines have been used to conditionally compile VSTa for SMP and non-SMP platforms. These are:

SMP This define indicates that the platform is an SMP platform. The **SMP** symbol must be given a numeric value of 2 or more, indicating the number of CPUs that the SMP platform has. At boot time, the file `mach/mp.c` will check that the platform has no more than **SMP** CPUs. If the **SMP** symbol is not defined, then VSTa is compiled for a non-SMP platform where there is one CPU.

BOCHS This define indicates that VSTa will be run on the Bochs simulator. This allows the kernel to choose to call out to the Bochs debugger by writing 16-bit words to certain I/O ports as follows:

```
outportw(0x8a00, 0x8a00);
outportw(0x8a00, 0x8ae0);
```

The **BOCHS** symbol will also cause the boot processor on a simulated SMP platform to send IPIs to the other CPUs so as to redistribute clock interrupts to those processors.

Therefore, to compile VSTa on Bochs with two simulated CPUs, you would modify the `makefile` in the `os/make` directory to have:

```
# Build options for kernel, also use .h from this dir
COPTS=-DKERNEL -DSMP=2 -DBOCHS -DKDB -DDEBUG $(INCS)
```

A non-SMP `makefile` would instead look like:

```
# Build options for kernel, also use .h from this dir
COPTS=-DKERNEL -DKDB -DDEBUG $(INCS)
```

6 Current Defects in SMP VSTa

At present, VSTa is able to boot on a Bochs simulator with two CPUs, allowing `root` to login and recompile some of the `*.o` files in `/src/os/make`. However, there are a number of lurking bugs and race conditions which need to be fixed. Here is a list of known symptoms:

1. Many of the VSTa servers are receiving duplicate **M_DISCONNECT** messages. When they receive a duplicate message, the `struct file` pointer that is returned by `hash_lookup()` is **NULL**. The servers then dereference this **NULL** pointer in their `dead_client()` routine and crash. The only workaround at present is to recompile all of the servers to return out of `dead_client()` when they are passed a **NULL** `struct file` pointer.
2. VSTa occasionally crashes just after entering `Tpgflt()` with a stack pointer value within the kernel machine code area. This seems to indicate that something is writing over the TSS area for each CPU.

3. Just as VSTa begins to execute the boot servers, occasionally we get a crash on the *iret* instruction in *retuser()*, as the CS segment is invalid. However, if we make it into the boot servers, this does not occur.
4. The *cons* server occasionally dies with a page fault at EIP=0x1062af. I have yet to find out which routine & instruction this is.
5. There are other random crashes in the kernel in *free()* and in *free_sched_node()*.

7 Things Left to Do

1. Find and fix the bugs that are causing the kernel to crash.
2. Find and fix the duplicate disconnect problem.
3. There will be race conditions; find and fix them.
4. Improve performance, especially with *get_cpuid()*. Determine if there is significant lock contention anywhere, and try to minimise the contention.
5. Get the system to boot on real SMP hardware, where all CPUs get external interrupts. Perhaps fix Bochs to deliver external interrupts to all CPUs.
6. Boot the system on 4-CPU and 8-CPU platforms, and fix any bugs that may be discovered.
7. Determine the scalability of the VSTa microkernel as the number of CPUs increase.

8 Changes to VSTa at Nov 19 2003

This section summarises the main changes to the VSTa 1.6.8 kernel source code as at November 19, 2003.

8.1 Changes in include/sys/

percpu.h

- Several preprocessor defines have been changed to use the global variable *cpu* as an array of *percpu* structures, not a single *cpu* structure.
- The *percpu* structure has been modified to incorporate a per-CPU idle stack, instead of a globally visible idle stack. The *pc_num* field has been removed, as there is now a *get_cpuid()* function to identify a CPU.

8.2 Changes in include/mach/

vm.h

- The number of task selectors in the global descriptor table now varies with the number of CPUs, as each CPU has its own task selector.

trap.h

- A new interrupt type has been created for inter-processor interrupts (IPIs).

gdt.h

- The number of ISA interrupts has been increased by 1 to account for the new IPI interrupt type.

8.3 Changes in os/kern/

misc.c

- Two new functions *par_puts()* and *parprintf()* allow kernel debugging messages to be sent out to the parallel port. These can be directed to an output file using the Bochs simulator.
- *panic()* has been rewritten to acquire a spinlock before printing the error message to the console. This ensures that the output is not garbled by simultaneous output.
- *assfail()* has been rewritten to include some of the *panic()* code, again to spinlock while writing output to the console.

proc.c

- The concept of *nextcpu* has been removed from *preempt()*. There is now only one single ready queue.
- There is an array of *cpu* structures, and *get_cpuid()* is used to index into the correct one for each CPU.
- The function *free_proc()* has been altered to deal with a possible race condition if two or more threads in a process attempt to *free_proc()* at the same time.

vm_page.c

- An assertion check that walks a list will fail in an SMP environment due to a lack of locking. For now the assertion check has been disabled on SMP systems.

xclock.c

- There is an array of *cpu* structures, and *get_cpuid()* is used to index into the correct one for each CPU.

8.4 Changes in os/mach/

dbg.c

- A new function *par_putc()* outputs characters to the parallel printer port. This allows kernel output to be captured by Bochs. The code does not poll the status register, and so may not work on real hardware.
- *dbg_enter()* has been modified to acquire a spinlock before entering the debugger. This ensures that only one CPU is in the debugger at any time. A normal VSTa spinlock was not used as this affects the interrupt mask.
- A new function *dbg_switchcpu()* allows a CPU which is in the debugger to relinquish the spinlock, thus letting other CPUs in to the kernel debugger.

genassym.c

- The call to *open()* in this program has been modified to truncate the output file. This was done at a time when I was hoping to cross-compile VSTa. However, this never eventuated, and so this change could be undone.

init.c

- The global integer *ncpu* now reflects either the value of the SMP preprocessor macro, or the value 1.
- *cpu* is now an array not a single structure.
- A new function *init_percpu()* now performs the FPU probing and the initialisation of the percpu structure. Each CPU calls this function as it starts up.
- Several *parprintf()* statements have been added to *init_machdep()*; they are purely for debugging and can be removed.
- *init_machdep()* calls *probe_smp()* to determine if the system has SMP. If this is true, a call to *map_apics()* follows to map the local and I/O APICs into virtual memory.
- If an SMP system has been found, a call to *init_smp()* is made at the end of *init_machdep()* to start each of the other CPUs.

locore.h

- A new inline function *sgdt()* saves the value of the global descriptor table register.
- A new inline function *sidt()* saves the value of the interrupt descriptor table register.
- A new inline function *outportw()* outputs a 16-bit value to an I/O port.
- Both *idle_stack()* and *on_idle_stack()* have been modified to use the *pc_id_stk* and *pc_id_top* arrays which are now part of the percpu struct. This allows each CPU to have its own idle stack.

locore.s

- The bootstrap code for VSTa can no longer use the idle stack, as the percpu structures have not yet been initialised. Instead, there is now a boot stack to do this.
- Interprocessor interrupts (IPIs) require another interrupt vector in VSTa. They have been allocated the vector 48.

machproc.c

- *cpu* is now an array not a single structure.
- VSTa used to use a global pointer called *tss*. Each CPU needs its own *tss*, and so this has been changed to an array in *resume()*. A local pointer *mytss* is set to the correct external structure.

makefile.inc

- This has been modified to assemble *mps.s* and *mp.c*.

mutex.c

- Several assertions have been `#ifdef`'d out when SMP is enabled, as they assume that a spinlock must be zero. This is not true in an SMP environment.
- The function *vall_sema()* has been modified to acquire the semaphore's spinlock before waking up all of the sleepers on the semaphore.

mutex.h

- *cpu* is now an array not a single structure. This affects several of the spinlock macros.
- The functions *p_lock()*, *p_lock_void()* and *cp_lock()* have been modified to actually spin whilst acquiring the spinlock. This is performed using the Intel `xchg` instruction.

syscall.c

- *cpu* is now an array not a single structure.

trap.c

- The function *page_fault()* has been modified to drop directly into the kernel debugger when a kernel-mode page fault occurs.
- Several *parprintf()* statements have been added to provide more diagnostic output.
- The function *setup_gdt()* now creates a separate TSS for each CPU, as once a TSS has been entered it cannot be re-entered by another CPU.
- The function *interrupt()* has been modified to receive IPIs from another processor. At present, there is only a single IPI message which redistributes clock ticks from the boot processor to the other CPUs. On the Bochs simulator, only the boot processor receives external interrupts. The function *interrupt()* has also been modified to pass these to the other CPUs using IPIs, but only when the preprocessor **BOCHS** symbol is defined.

9 Example Bochs Configuration File

This is the Bochs configuration file that I use to boot *2bochs* and run the SMP VSTa kernel. To convert this to work with *bochs*, simply change the `romimage:` line to point at a BIOS image which is suitable for only one CPU.

```
floppya: 1_44=vsta.flp, status=inserted
ata0-master: type=disk, path="newvsta_340m.dsk", cylinders=665, heads=16, spt=63
boot: floppy
megs: 32
romimage: file=bios/BIOS-bochs-2-processors, address=0xf0000
vgaromimage: bios/VGABIOS-lgpl-latest
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=0, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=0, ioaddr1=0x1e8, ioaddr2=0x3e8, irq=11
ata3: enabled=0, ioaddr1=0x168, ioaddr2=0x368, irq=9
floppy_bootsig_check: disabled=0
log: bochsout.txt
error: action=report
info: action=report
debug: action=ignore
debugger_log: -
parport1: enabled=1, file="parport.out"
vga_update_interval: 100000
keyboard_serial_delay: 250
keyboard_paste_delay: 100000
floppy_command_delay: 500
ips: 2000000
pit: realtime=1
mouse: enabled=0
private_colormap: enabled=0
fullscreen: enabled=0
screenmode: name="sample"
keyboard_mapping: enabled=0, map=
i440fxsupport: enabled=0
```