SMP Implementation for OpenBSD/sgi

Takuya ASADA
syuu@openbsd.org

Abstract

We started to implement SMP support for the OpenBSD/sgi port and selected the SGI Octane as the first target machine. We are now at the point where we have almost finished, except for the implementation of some non-critical features. SMP support is now operational on SGI Octane systems and the code has been merged into the OpenBSD tree.

1 Introduction

A year ago, I was working to add SMP and 64 bit support to a BSD-based embedded operating system. The target device was based on the MIPS64 architecture. At that time only FreeBSD had started to support SMP on MIPS, however only some of their code had been merged into the public repository. As a result, I tried to implement SMP support in a step-by-step manner, referring to the SMP implementations on other architectures.

That implementation was proprietary, but I wanted to contribute something to the BSD community using the knowledge gained. I decided to implement SMP from scratch again and tried to find a suitable MIPS multiprocessor machine as a target device. I found a cheap SGI Octane, which is capable of two cores and is supported by OpenBSD. I obtained an SGI Octane2 and have been working on it since April 2009. I wrote about my progress and provided patches in my blog. Miod Vallat discovered it and contacted me to suggest that my code be merged into OpenBSD's main repository.

I become an OpenBSD developer in September 2009 and started merging the code. I participated in the Hardware Hackathon (h2k9), held at Coimbra, Portugal in November 2009. I worked with Miod Vallat and Joel Sing and we finally reached the point where a process could be woken up on the secondary processor. Development continued after the hackathon and we are now at the point where we have almost finished the SMP implementation, except for some non-critical features (e.g. switching processors within ddb via "ddb machine ddbcpu<#>").

2 OpenBSD SMP overview

2.1 History

The OpenBSD project started work on SMP support for various architectures in February 2000. As result, we now have SMP support on the i386, amd64, mvme88k, sparc64 and macppc platforms.

Recently, sgi was added to the list of officially supported architectures but without SMP support, despite the SGI hardware range including multiprocessor servers and workstations. We started to implement SMP support for OpenBSD/sgi and selected a SGI Octane as the first target machine.

2.2 Current status

OpenBSD SMP now works correctly, however performance is still limited. The stage of our implementation is similar to the early versions of FreeBSD and NetBSD. A significant amount of work still needs to be done in order to improve performance and scalability.

2.2.1 Giant lock

OpenBSD employs a Big Giant Lock model. In this model, we need to acquire a lock every time we enter the kernel, in order to prevent kernel code from executing on multiple processors at the same time. We can preserve legacy code with this model, however it does not scale well compared to more recent implementations that are based on fine-grained lock models.

2.2.2 Lock primitives

We have rwlock and mutex to implement MP-safe kernel components. An rwlock provides a multi-reader, single-writer locking mechanism to ensure mutual exclusion between different processes. It is a sleep mutex, so the current process will go to sleep if the lock is busy. This is useful in situations such as block I/O, where a process will take time to complete an operation.

A mutex provides a non-recursive, interrupt-aware spinning mechanism to ensure mutual exclusion between different processors. It spinlocks and modifies the interrupt priority level (SPL) at the same time. If old code uses SPL operations for synchronization they should be replaced by a mutex, but it cannot be used recursively. We can also use it to protect shared data within a SMP kernel.

Many subsystems are being rewritten to be giant lock free using these primitives, however this is still a work in progress.

2.2.3 Scheduler

The OpenBSD scheduler is based on the traditional 4.4BSD scheduler, with some enhancements for SMP support. Each processor has its own runqueue. The scheduler balances and dispatches processes to idle processors.

2.2.4 rthread

An rthread is a 1:1 native kernel thread library, implemented using the rfork system call. In the kernel, a thread is realized as a normal process with a special flag, which is set in rfork. rthreads is not the default thread library yet, due to performance and stability issues, and it is still a work in progress. The default thread library is libpthread, which is a userland thread implementation that cannot benefit from SMP.

3 SGI Octane overview

3.1 History

An SGI Octane is a high-end graphics workstation, which is based on the MIPS architecture and is designed to run IRIX. The original version named ”Octane” was manufactured between 1997 and 2000, with an updated version named ”Octane 2” being manufactured between 2000 and 2004. Both models share almost the same architecture which can be configured as a multiprocessor system. There are uniprocessor models and dual processor models of both the Octane and Octane2.

Hardware documentation is limited since SGI never openly disclosed a detailed hardware specification. As a result we can only refer to the Octane-specific header files found on IRIX (/usr/include/sys/RACER/), and Linux kernel patches for Octane (http://www.linux-mips.org/~skylark/).

Figure 1: SGI Octane image

Processors	MIPS R10000/R12000 175-400MHz x 1 or 2
Memory	128MB-4GB SDRAM
Graphics	3D Graphics board
Sound	Digital Audio board
Storage	4/9GB Ultra Fast/Wide SCSI HDD
Communications	100BASE-TX port x 1, RS422/RS433 x 2, Parallel port x 1

Table 1: SGI Octane specification

3.2 MIPS R10000

The SGI Octane has one or two R10000 processors - in this section we describe their features and the challenges involved with implementing SMP on this processor.

3.2.1 Coherent cache and operations for multiprocessor

One of the most important features of this processor is its fully coherent cache. The other important feature is the synchronization system between multiple processors. There are some instructions for synchronization:

SYNC operation The SYNC instruction is used for ordering loads and stores from shared memory. It waits for preceding loads and stores to complete.

LL/SC operations The LL/SC instructions are the implementation of Load-Link/Store-Conditional operations on the MIPS architecture. We need it to implement lock-free atomic read-modify-write operations. The LL operation loads a value from a memory location to a register. The SC operation checks if the value has been changed since LL executed by using the cache line. If it has changed then it stores zero to the register, meaning that SC has failed. Otherwise it continues to store the value to the memory location.

We use these instructions often when implementing atomic functions and lock functions.

3.2.2 TLB Consistency

Software TLB The MIPS TLB is a software TLB, which means that the operating system must manage the TLBs. For example, if a processor causes a TLB miss exception then the hardware does not do anything else. Instead, the operating system must handle the exception, lookup the page table and find the physical address paired with requested virtual address. This means that all software TLB management functions need to be MP-safe, including the data structures they use.

ASID: Address Space Identifier On the MIPS architecture each TLB entry is tagged with an 8 bit Address Space Identifier (ASID). Each process should be assigned an individual identifier and the current process identifier should be set when a context switch occurs. This allows the processor to distinguish between TLB entries that are for the current process and those which are not. This can improve system performance since we do not need to flush TLB entries on every context switch.

However, in order to make the memory management routine MP-safe, the ASID first needs to be managed on a per-processor basis. The next step is to think about what will happen if a TLB entry could remain, even after switching to a different process. If that process moves to another processor and invalidates or updates the same entry, there will be an inconsistency. You can use ASID to solve this problem. We do not need to flush the TLB via a TLB shootdown in this case, instead we can change the ASID assigned to the process and the entry will no longer be used.

3.3 MPCONF

MPCONF is a bank of hardware registers used to control secondary processors and retrieve configuration information from them. Each secondary processor has one entry in MPCONF, with each entry having the following registers:

MAGIC	magic number (0xbaddeed2 if cpu exists)
PRID	processor revision ID
PHYSID	physical CPU ID
VIRTID	virtual CPU ID
MP_SCACHESZ	secondary cache size
FANLOADS	unknown
LAUNCH	launch function address (entry point) / cpu spin up trigger
RNDVZ	rendezvous function address
STACKADDR	stack pointer address
LPARAM	arguments for LAUNCH
RPARAM	arguments for RNDVZ
IDLEFLAG	unknown

Table 2: MPCONF registers

To spinup a secondary processor we need to access the LAUNCH register as a minimum.

3.4 HEART

HEART consists of a bus and an interrupt controller device. On an Octane most devices are connected to the HEART widget. HEART provides two kinds of registers for interrupt handling - IMR and ISR. IMR stands for Interrupt Mask Register, which allows interrupts to be masked or unmasked on a per-processor basis. Each bit in the mask represents an IRQ. There are four IMR registers, allowing for up to four processors to be supported.

ISR stands for In-Service Register and it is divided into three registers, which are set-only, clear-only and read-only register. If an interrupt handler has completed then it needs to clear the interrupt. The CLR_ISR register is used for this purpose. The SET_ISR is the opposite of the CLR_ISR register; writing to it triggers an interrupt. This will not generally be used to handle external interrupts, however it can be used to trigger Inter-Processor Interrupts (IPIs). IRQs 46-49 are reserved for this purpose.

Memory configuration registers (SDRAM_MODE, MEMCFG0-3 ...)
Interrupt registers (IMR0-3, SET_ISR, CLR_ISR, ISR ...)
Misc registers (COUNT, COMPARE, PRID ...)

Table 3: HEART registers

HEART also contains another useful register named PRID. This register allows the ID of the current processor to be obtained.

4 Tasks for SMP support on Octane

4.1 Support multiple cpu_info and processor related macros

We have a structure named cpu_info for per-processor data storage. There are a number of macros and functions to support using cpu_info.

The curcpu() macro can be used to access a pointer to the current processor's cpu_info structure. In the original code this macro simply returned a reference to a statically allocated cpu_info structure named "cpu_info_primary". We changed this so that the cpu_info structure is allocated dynamically, with a pointer to this structure being stored in an array indexed by processor ID. This structure is allocated per-processor when a processor executes its initialization sequence.

The cpu_number() macro can be used as a processor identifier. Initially this was defined to be "0" since there was only one processor. We changed the marco so that it fetches the ID of the current processor from a hardware register (PRID on HEART, as described above).

```
/* uniprocessor code */
extern struct cpu_info cpu_info_primary;
#define curcpu() (&cpu_info_primary)
#define cpu_number() 0
...

/* multiprocessor code */
extern struct cpu_info *cpu_info[];
#define curcpu() (cpu_info[cpu_number()])
#define cpu_number() \
    (*(uint64_t *)HW_CPU_NUMBER_REG)
...
```

We are planning to use a hardware register (LLAddr in cp0) to store the pointer to the cpu_info structure, in order to improve curcpu() performance. This code has been implemented but not yet merged.

4.2 Move per-processor data into cpu_info

In the original code the following information was stored in global variables even though it is per-processor information: running process information, interrupt priority level, pending interrupt, trapdebug, interrupt depth, etc...

All of these variables were moved into the cpu_info structure and the code which referenced these variables was modified to reflect the changes. Since some assembly code refers to these variables, we implemented an assembly macro version of curcpu(), named GET_CPU_INFO(). One exception was the astpending global variable, which was moved into the per process data structure.

4.3 Lock primitives

4.3.1 rwlock

To use a rwlock in a SMP kernel, you should implement rw_cas(), which is a compare and swap function. A stub for rw_cas() is defined by default, but it is only a uniprocessor implementation.

4.3.2 mutex

Mutexes can be implemented using splraise(), splx() and the rw_cas() function. A pseudo code implementation is as follows:

```
mtx_init(mtx, wantipl) {
    mtx->mtx_lock = 0;
    mtx->mtx_wantipl = wantipl;
    mtx->mtx_oldipl = IPL_NONE;
}

mtx_enter_try(mtx) {
    s = splraise(mtx->mtx_wantipl);
    if (compare_and_swap(&mtx->mtx_lock, 0, 1)
        == success) {
        mtx->mtx_oldipl = s;
        return 1;
    } else {
        splx(s);
        return 0;
    }
}

mtx_enter(mtx) {
    while (mtx_enter_try() == 0)
        /* loop */ ;
}

mtx_leave(mtx) {
    mtx->mtx_lock = 0;
    splx(mtx->mtx_oldipl);
}
```

4.3.3 mp_lock

An mp_lock is a spin lock primitive for giant locks. Unlike a mutex, mp_lock can be used recursively on same processor. This allows interrupts or exceptions to be handled whilst holding a giant lock.

The same processor can acquire a mp_lock more than twice, simply by incrementing the lock counter. Other processors will not be able to acquire the mp_lock until all locks have been released at which point the counter will return to zero. Note that when the lock first succeeds the counter value will be two - this provides protection for the mp_lock structure when unlocking.

A pseudo code implementation is as follows:

```
__mp_lock_init(mpl) {
    mpl->mpl_cpu = NULL;
    mpl->mpl_count = 0;
}

__mp_lock(mpl) {
    while (1) {
        disable_interrupt();
        if (compare_and_swap(&mpl->mpl_count, 0, 1)
         == success)
            mpl->mpl_cpu = curcpu();
        if (mpl->mpl_cpu == curcpu()) {
            mpl->mpl_count++;
            enable_interrupt();
            break;
        }
        enable_interrupt();
    }
}

__mp_unlock(mpl) {
    disable_interrupt();
    if (--mpl->mpl_count == 1) {
        mpl->mpl_cpu = NULL;
        mpl->mpl_count = 0;
    }
    enable_interrupt();
}
```

4.4 Acquiring giant lock

We need to acquire giant lock at trap(), software interrupts and hardware interrupts, prior to entering the kernel context.

4.5 Atomic operations

The original code already had atomic operations, however some extra operations were added to facilitate the SMP implementation.

4.6 Spin up secondary processors

Spin up code is needed in order to boot secondary processors. You need to determine which processors are to be booted after cpuattach() and set the appropriate flags within their cpu_info structure.

Late in the boot process the kernel calls a function named cpu_boot_secondary_processors(). This iterates over the cpu_info array to find and spin up target processors. In order to spin up secondary processors we need to set various parameters (entry point address, stack address, etc...) and turn on the processor by writing to the multiprocessor controller hardware registers.

The SGI Octane has control registers called MPCONF for this purpose. The spin up function writes values to these registers. The actual steps taken by our implementation are as follows:

1. calculate base address of MPCONF entry using cpuid
2. allocate bootstrap stack
3. set allocated stack address to STACKADDR
4. set cpu_info pointer to LPARAM
5. set secondary processor entry point address to LAUNCH

4.7 Secondary processor entry point

When a secondary processor spins up it needs to run some initialization code. This is similar to the primary processor boot sequence, however we do not need to perform kernel initialization since this has already been done by the primary processor at boot time.

Our implementation executes the following in order to enable process scheduling on the secondary processor:

1. disable interrupts
2. load global register
3. cache initialization
4. TLB initialization
5. clock initialization
6. ipi initialization
7. interrupt controller initialization
8. enable interrupt
9. start scheduling

4.8 IPI: Inter-Processor Interrupt

IPI is a type of interrupt used by one processor to send an interrupt to another processor. On Octane, we use HEART to implement IPIs, which are then used for TLB shootdowns and cpu_unidle().

4.9 Per-processor ASID management

As described previously, the MIPS TLB has a tag on each entry to identify which process it belongs to. Since it is only 8 bits in size (much smaller than the process ID) the kernel has to maintain a PID to ASID map. Although the original code used global variables for this purpose, we changed it to use per-processor variables. The pmap structure also has an ASID entry, which we also changed to be per-processor since it is not necessary to share ASIDs between processors.

4.10 TLB shootdown

When multiple processors share the same page table entries we will hit TLB consistency problem. While each processor runs different processes they will not share page table entries because each process has individual page tables. However, in the following cases it will share the entries:

- kernel pages
- page shared by multiple processes, each process may run on a different processor at the same time
- a process that has multiple kernel threads, each process may run on different processor at the same time

Since typical hardware does not have a mechanism to keep the TLB consistent between multiple processors automatically, we have to keep it consistent within software. There is an algorithm called 'TLB shootdown' which can achieve this using IPIs. The basic concept is as follows:

1. block all other active processors (these can be identified by checking the active flag) via IPI
2. change the TLB entry on the local processor
3. invalidate the old entry on all other active processors

Whilst this process works, it is slow and does not scale well because it often blocks multiple processors. Platform dependent techniques can be used in order to reduce blocking time or frequency. We referred to FreeBSD/mips, which has implemented a simple TLB shootdown.

4.11 Lazy FPU handling

In the original code FPU registers were not saved during a context switch, rather they are only saved when another process starts using FP. This is called lazy FPU handling and tries to reduce the frequency of FP context switches in order to save CPU time.

However, it does not work well in an SMP environment. On an SMP system a process can be migrated to another processor if the current processor is busy. If a process is switched without saving its FPU context, we will lose the data stored within these registers. We can delay register saving until the process is migrated to another processor, however we decided not to implement lazy FP handling on SMP for now.

4.12 Per-processor clock

We need a per-processor clock to make process scheduling work correctly. On Octane this is easily achieved since each processor has its own internal clock. All we needed to do was change the clock driver to maintain clock information per-processor rather than globally.

5 Ideas implementing SMP

We have faced a number of issues while implementing SMP, which we managed to solve. Some of the issues and solutions are presented here.

5.1 Writing assembly code using only 2 registers

There was a small challenge in implementing GET_CPU_INFO(): the TLB miss handler skips a context switch, thus we could only use two registers named "k0" and "k1", which are reserved for kernel operations [1]. To fulfill this requirement this macro takes two arguments which specifies the registers that are to be used:

```
/* uniprocessor code */
tlb_miss:
    PTR_L    k1, curprocpaddr
    dmfc0    k0, COP_0_BAD_VADDR
...

/* multiprocessor code */
tlb_miss:
    GET_CPU_INFO(k1, k0)
    PTR_L    k1, CI_CURPROCPADDR(k1)
    dmfc0    k0, COP_0_BAD_VADDR
...
```

5.2 Dynamic memory allocation without using a virtual address

To support an arbitrary number of processors, the cpu_info structure and the bootstrap kernel stack for secondary processors should be allocated dynamically. However, using malloc() to allocate them causes a problem since malloc() returns a virtual address when a physical address is required. If we try to use a virtual address for the bootstrap stack, it may be used before the TLB is initialized, thus causing the processor fault.

If we use a virtual address for cpu_info it causes an infinite fault loop, due to the fact that the TLB miss handler refers to the cpu_info structure - the TLB miss handler causes a TLB miss exception, which re-runs the TLB miss handler, looping forever.

To avoid these problems we implemented a wrapper function which allocates memory dynamically using malloc() or uvm_pglistalloc() (depending on the allocation size), before obtaining and returning the physical address for the memory allocated.

5.3 Reduce frequency of TLB shootdown

In the FreeBSD/mips pmap implementation, a TLB shootdown is performed for every TLB invalidate and update, however the shot processor decides if the TLB invalidate/update is needed or not. Our implementation was very inefficient, so we modified the code so that a TLB shootdown is only performed if it really is necessary. Otherwise we just increment the ASID generation, etc.

5.4 Calculate size of struct pmap on boot time

As described earlier, the pmap structure should have ASID entries for all processors. However, we do not know how many processors will exist in the system at the time of compilation. The kernel detects the number of processors during the boot process, hence we can calculate the size of the pmap structure at boot time.

We defined the pmap structure with pm_asid[1] as follows:

```
typedef struct pmap {
    int pm_count;
    simple_lock_data_t pm_lock;
    struct pmap_statistics pm_stats;
    struct segtab *pm_segtab;
    struct pmap_asid_info pm_asid[1];
} *pmap_t;
```

Then we use PMAP_SIZEOF(n) instead of sizeof(struct pmap) when allocating its size.

```
#define PMAP_SIZEOF(x) \
    (ALIGN(sizeof(struct pmap) + \
    (sizeof(struct pmap_asid_info) * ((x) - 1))))

pool_init(&pmap_pmap_pool,
    PMAP_SIZEOF(ncpusfound),
    0, 0, 0,"pmappl", NULL);
```

This idea is borrowed from NetBSD/alpha, which also has an ASID (although they call it 'ASN' instead).

5.5 Fighting against deadlock

It was hard to find the cause of deadlocks that occurred when both processors were running concurrently. Because they are a result of timing bugs caused by two conflicting processors, we need to be able to determine what happened on both processors at that time. It was likely that there is no way to use JTAG ICE on Octane, hence we only used printf() for debugging.

In order to figure out which lock the processors were stacking on, we placed counters and debug prints in every type of spinlock. If the lock continues to spin too much it will print a message.

It might seem like a good idea to execute the kernel debugger at that moment, in order to reveal its stacktrace.

Unfortunately, this is not overly useful since we have not yet implemented ddb machine ddbcpu<#> - this means that we cannot switch processor from within ddb. As a result, we can only see the status of the blocked processor and cannot determine the status of the locked processor.

We temporarily implemented code which executes ddb on the locked processor using an IPI, but this was useless because we could only get the stacktrace from ddb after the IPI interrupt had occurred.

We then decided to record the return address when we acquire a lock. By printing the return addresses out as debug messages we can determine who acquired the lock when the lock blocks. Most deadlocks were revealed to occur in the rendezvous point of the TLB shootdown function, so we placed counters and debug printing code there.

It was hard to read the debug messages output from two processors on one console, hence we hacked up a serial driver for the second serial port and separated the output per-processor - the first processor (cpu0) outputs to the first serial port and the second processor (cpu1) outputs to the second serial port.

If a TLB shootdown causes a deadlock, it is important to check whether the blocked processor has accepted an IPI or not. We dumped the cp0 status register to check the interrupt enable bit and the interrupt mask status. We also checked that the current IPL is equal to the interrupt mask on the processor.

We iterated over these approaches again and again, until we finally found and fixed the following bugs:

5.5.1 Interrupt and exception blocks IPI

If CPU A triggers a fault and causes a TLB shootdown, and CPU B is interrupted after CPU A got the fault but before CPU A sends the IPI, a deadlock would occur. This occurs since CPU A has acquired the kernel lock before the TLB shootdown and is waiting for CPU B to execute its IPI handler. CPU B blocks the IPI interrupt because interrupts have been disabled and it then waits for CPU A to release the kernel lock, which will never happen. This could occur during software interrupts, hardware interrupts and traps. To avoid this problem IPI interrupts were enabled in all handlers.

5.5.2 splhigh() blocks IPI

If CPU A triggers a fault and causes a TLB shootdown, CPU B has masked interrupts via splhigh() and has tried to lock the kernel after CPU A has triggered the fault but before sending the IPI, a deadlock would occur. The reason is almost the same as the interrupt/exception - CPU B blocks IPI interrupts because it has been masked via splhigh, so it waits forever. We redefined IPL_IPI to be higher than IPL_HIGH in order to prevent masking IPIs at splhigh. After this change an IPI can interrupt the operation, even when at IPL_HIGH.

6 Development status and future works

We are now at the point where we have almost finished, except for the implementation of some non-critical features (e.g. machine ddbcpu<#>). SMP is now operational on SGI Octane systems and the code has been merged into the OpenBSD tree. You can download it from an OpenBSD CVS repository and try it now.

I am also interested in the machine independent part of SMP implementations and I wish to work on these areas too.

7 Acknowledgments

Thanks to Miod Vallat for inviting me to the OpenBSD project. He helped every time I faced a difficulty. I really appreciate his help.

Thanks to Naoki Hamada for providing me with an Octane. He brought me lots of knowledge about kernel hacking and also supported writing this paper.

Thanks to Joel Sing for working together at the OpenBSD hardware hackathon. He took me to the hackathon room. I even could not reach there without his help.

Also thanks to all OpenBSD developers who helped with this work, and Livedoor coworkers for supporting me to work on OpenBSD project.