Bad Concurrency

Misadventures in Concurrent and Parallel programming, plus random comments on software performance and various contributions.

SUNDAY, 8 JANUARY 2012

Building a CPU Topology on MacOS X

Within a project I've been working on I've had the need to simulate the capabilities of Linux's /proc/cpuinfo on Mac OS. Specifically I needed to build a topology of the CPUs on a given system. I.e. I need to map the operating system's processors to hardware threads, then build a picture of which cores and sockets those threads reside. For example my Mac looks something like:

```
CPU0 (Thread 0) ---+
                   |---> Core 0 ---+
CPU1 (Thread 1) ---+               |
                                   | ----> Socket 0
CPU2 (Thread 2) ---+               |
                   |---> Core 1 ---+
CPU3 (Thread 3) ---+
```

While this sounds very simple, it's actually fraught with a number of little niggles. Not only did it require getting down and dirty with a bit of C++ and X86 Assembly, it also required writing a MacOS kernel extension.

The first step was to understand what information was available from the CPU. Intel exposes an instruction called CPUID. The is the primary mechanism for getting information about the CPU. There is a raft of information available from listing of the CPU features available (e.g. hyperthreading) to sizes of the various levels of cache and the associated cache lines. To access the CPUID instruction we need a little bit of inline assembler.

ABOUT ME

Michael Barker

View my complete profile

JAVA CODE GEEKS

TWITTER

BLOG ARCHIVE

```
#include <stdio.h>

static void cpuid(int op1, int op2, int *data) {
    asm("cpuid"
        : "=a" (data[0]), "=b" (data[1]), "=c" (data[3]), "=d" (data[2])
        : "a"(op1), "c"(op2));
}

int main(int argc, char** argv) {
    int values[5];
    cpuid(0, 0, values);
    printf("Vendor String: %s\n", (char*) &values[1]);
}
```

This Gist brought to you by GitHub. cpuid-0.cpp view raw

The code shows how to get the vendor string from the CPU. On my Mac I get the following:

```
// Output:
Vendor String: GenuineIntel
```

For those unfamiliar with Intel inline assembly, the Intel CPU defines a number of registers. The ones used for the CPUID instruction are EAX, EBX, ECX, and EDX (referenced as RAX, RBX, etc if using 64 bit instructions via the REX extension). These used for both input and output. An inline asm segment consists of 3 parts. The first part is the instruction to be executed. In this case the `"cpuid"` instruction. The second line defines the output parameters. The snippet `"=a" (data[0])` means store the result in the EAX register in the variable `data[0]`. The `"=a"` refers to the 2nd letter of the register designation. The 3rd and final section are the input parameters. The CPUID instruction takes 2 parameters, one in EAX and one in ECX.

The particular CPUID reference that provides information needed to building the topology is `0xB (11)` - the extended topology enumeration leaf. The data returned from this instruction is:

```
     0                   1                   2                   3
     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
EAX | Shift |                     Reserved                          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
EBX |   No. Process at this level   |            Reserved           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
ECX |   Level No.   |  Level type   |            Reserved           |
```

▼ 2012 (6)
- ► July (1)
- ► May (1)
- ► April (1)
- ► February (2)
- ▼ January (1)
  - Building a CPU Topo MacOS X

► 2011 (12)
► 2010 (5)
► 2009 (6)
► 2008 (4)
► 2007 (1)

```
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
EDX |                           x2APIC ID                           |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The extended topology enumeration leaf is one of the CPUID indexes that also makes use of ECX as an input parameter. This indicates the level of the CPU that you wish to work at. I started at level 0 and worked my way up. The 'Level type' describes whether the current level is a CPU thread (1) or a physical core (2) or invalid (0). All values greater than 2 are reserved, presumably for later use. The 2 other useful values are the `x2APIC ID` and `Shift`. The `x2APIC ID` is a unique identifier for the smallest logic unit in the CPU. The `Shift` value is used to group `x2APIC ID` values into units at the next logical level. This is done by shifting the `x2APIC ID` right by the value specified by `Shift`. For example on my using the following code on my workstation (2 sockets, 8 cores, 16 threads):

```c
#include <stdio.h>

static void cpuid(int op1, int op2, int *data) {
    asm("cpuid"
        : "=a" (data[0]), "=b" (data[1]), "=c" (data[2]), "=d" (data[3])
        : "a"(op1), "c"(op2));
}

int main (int argc, const char * argv[]) {
    int values[4];
    int level_type = 0;
    int i = 0;

    while (1) {
      cpuid(0xB, i++, values);

      level_type = values[2] >> 8 & 0xFF;
      if (level_type == 0) break;

      printf("Shift: %d, Count: : %d, Level Id: %d, Level Type: %d, x2APIC: %d, Group: %d\n"
             values[0] & 0xF, values[1] & 0xFFFF,
             values[2] & 0xFF, level_type,
             values[3], values[3] >> (values[0] & 0xF));
    }
}
```

This Gist brought to you by GitHub. cpuid.c view raw

Outputs the following:

```
Shift: 1, Count: 2, Level Id: 0, Level Type: 1, x2APIC: 33, Group: 16
Shift: 5, Count: 8, Level Id: 1, Level Type: 2, x2APIC: 33, Group: 1
```

This indicates that the hardware thread indicated by APIC 33 has the core id of 16 and socket id of 1. The socket and core ids aren't really references, but values that indicate threads have have the same id share that unit. This gets me most of the way there, I can group all of my threads into a hierarchy of cores and sockets. However there is one small niggle remaining. I need this code to run all of the CPUs on my system. On Linux this is reasonably simple, you count the number of CPUs then iterate through all of them and use the `pthread_setaffinity_np(...)`

to specify specify which CPU to run the CPUID instructions on.

However, on Mac OS X actual thread affinity is not supported; just a mechanism logically grouping or separating threads, powerful in its own right, but not what I'm after here. This is where the Kernel module comes in. The XNU Kernel defines a funky little method called `mp_rendevous(...)`. This method takes in a number function pointers. The key one (action_func), is run once on each of the CPUs. So to get the topology information for all of the CPUs we can use it like so:

```
extern "C" {
/* from sys/osfmk/i386/mp.c */

extern void mp_rendezvous(void (*setup_func)(void *),
                          void (*action_func)(void *),
                          void (*teardown_func)(void *),
                          void *arg);

extern int cpu_number(void);
}

typedef struct {
    int os_id;
    int socket;
    int core;
    int apic;
} cpu_info_t;

static void topologyHelperFunction(void* data) {

    int values[4];
    cpu_info_t* cpus = (cpu_info_t*) data;
    int cpu = cpu_number();

    cpus[cpu].os_id = cpu;

    cpuid(0xB, 0, values);
    cpus[cpu].core = values[3] >> values[0] & 0xF

    cpuid(0xB, 1, values);
    cpus[cpu].socket = values[3] >> values[0] & 0xF
    cpus[cpu].apic = values[3];
}

static void getCpuidData() {
    int ncpu;
    size_t len = sizeof(int);
    sysctlbyname("hw.ncpu", &ncpu, &len, NULL, 0);

    cpu_info_t* values = (cpu_info_t*) malloc(sizeof(cpu_info_t) * ncpus);

    mp_rendevous(NULL, (void (*)(void *))topologyHelperFunction,
                 NULL, (void*) values);
}
```

This Gist brought to you by GitHub. all_cpus.c view raw

Because the method `mp_rendevous()` is defined in the kernel the code above needs to be packaged as a kernel extension. Even then, getting access to the method is interesting. It's not defined in a header that can be easily included, however it is available at link time when compiling a kernel module. Therefore in order compile correctly, it's necessary to provide your own forward declaration of the method. The same is true of the `cpu_number()`. Calling into the kernel method from user space requires use of the IOKit framework, but I'll leave the details of that as an exercise for the reader.

Posted by Michael Barker at 19:44

No comments:

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)