Per-CPU variables

Consult the submit server for deadline date and time

1 Overview

Data that is local to a processor can be useful. In GeekOS, there are key variables that are frequently accessed on a per-processor basis:

1. The current thread, via the `CURRENT_THREAD` macro or `get_current_thread()` function
2. The index of the current cpu using `Get_CPU_ID()`, although this is most often used in the process of indexing into the current thread.

One could imagine more per-processor data, for example, the head of a per-processor run queue. A scheduler that has run queues distributed across processors would avoid grabbing a global run queue lock on every interrupt. This would avoid most of the lock contention currently in geekos, where the kthreadlock or globallock covers the thread queues and thread objects.

Currently, the way to get per-processor information is to:

1. Disable preemption: important because we don't want to be preempted in this process. Currently, this is accomplished by disabling interrupts.
2. Figure out which processor we're on: this asks the APIC for its id, which varies by processor.
3. Use the processor id as an index into a larger table, such as `g_currentThreads`.
4. Re-enable preemption / interrupts.

A common approach to storing per-processor information is instead to:

1. Execute an instruction that (in a single assembly instruction) operates over data known to be "owned" by the processor.

If the entirety of the operation can be completed in one instruction, preemption need not be disabled: preemption can only occur between instructions. Concurrent access by other processors isn't possible because the per-cpu area is "owned". By "owned" I mean that no other processor is allowed to access the data, or equivalently, the processor has this region locked, permanently.

The trick to permit a single instruction to access per-cpu memory on x86 is to use either fs or gs segments. While "cs," "ds," and "ss," have a well-defined purpose used transparently by the processor, "fs" and "gs" do not. The only way to access variables relative to the base of the fs or gs is to use an assembly instruction that mentions the segment explicitly.

These segments are loaded into the processor as indexes into the Global Descriptor Table (GDT). The GDT is a per-processor data structure that in unmodified GeekOS is shared across all processors. To make a per-cpu segment work, we will need to switch to having per-cpu GDTs that are identical except for the segment descriptor at a particular index.

This becomes a bit complicated because each user process has an entry in the GDT that refers to its LDT (in userContext) and defines the cs and ds,ss,es,fs,gs for that user process. Before the VM project, it is critically important that this reference to the LDT be present in the GDT on each processor, which is simple when there is only one GDT for the whole machine, but more complicated now that there will be a GDT per processor. After the VM project, it is possible to have just one global LDT for all user processes, since they all have the same base and limit. Each of the GDT's can reference this single LDT.

Your task is to make the following changes.

1.1 percpu.c

Implement:

```
void Init_PerCPU(int cpu);
int PerCPU_Get_CPU(void);
struct Kernel_Thread *PerCPU_Get_Current(void);
```

Define a struct to hold per-cpu data and initialize it in `Init_PerCPU()`. Access it in the `PerCPU_Get` functions. Setters are not required in percpu.c, since the CPU can't be changed and the current thread is set in assembly.

1.2 main.c

Call `Init_PerCPU()`.

1.3 smp.c

Call `Init_PerCPU()`. Modify `get_current_thread()`.

1.4 percpu.asm

In percpu.asm, we will redefine the following macros to use the per-cpu area you'll define:

- `Get_Current_Thread_To_EAX`
- `Set_Current_Thread_From_EBX`
- `Push_Current_Thread_PTR`

These will be single instructions.

The nasm compiler will warn about redefining these macros, but will accept the redefinition. For this reason, percpu.asm is included just after these macros are given their original definition in lowlevel.asm. You may delete or comment the original versions if the warning message offends you.

1.5 defs.asm

This file notes which GDT entries are associated with which segments. GDT entry 1 is the kernel code segment. GDT entry 2 is the kernel data segment. You'll need to define a GDT entry for the per-cpu data segment.

1.6 lowlevel.asm

In Handle_Interrupt, load gs appropriately. Base your changes on the code that loads ds and es.

1.7 gdt.c

Allocate a segment descriptor for the per-cpu region. (If implementing with paging, this is where you would allocate a segment descriptor for a global user ldt.) Initialize the user ldt.

1.8 kthread.c

In Setup_Kernel_Thread, push an initial gs appropriately.

1.9 segment.c

Add a function comparable to Init_LDT_Descriptor that will initialize an LDT in each processor's GDT, useful for wherever Init_LDT_Descriptor is called.

2 Hints

You may assume that the maximum number of processors shall be 8. This is in practice the size of of the array of GDTs in `s_GDT`. Elsewhere, `MAX_CPUS` is defined a bit higher.

3 Further Reading

This design is heavily influenced by the Linux implementation of percpu variables on x86.

Per-CPU

Much of the complexity of implementing per-cpu variables is in allowing code to allocate on each of the segments, despite architecture-specific implementations. As such, there's not a lot of description of how the trick works.

- `http://ftp.dei.uc.pt/pub/linux/kernel/people/christoph/sf2011/percpu-collab2011.pdf`
- `http://www.makelinux.net/ldd3/chp-8-sect-5`
- `http://lwn.net/Articles/198184/` - concise description of linux design.
- `https://0xax.gitbooks.io/linux-insides/content/Concepts/per-cpu.html`
- `http://lxr.free-electrons.com/source/Documentation/this_cpu_ops.txt` - "Inner working of this_cpu operations"
- `http://lxr.free-electrons.com/source/arch/x86/include/asm/percpu.h` http://lxr.free-electrons.com/source/ar
- `http://lxr.free-electrons.com/source/arch/x86/include/asm/segment.h`
- `http://www.ccs.neu.edu/course/cs5600f15/parent/unix-xv6/proc.h` - a different OS project's implementation of similar features.
- `http://www2.mta.ac.il/~carmi/Teaching/2014-5A-OS/Slides/Lec%2013%20Per%20CPU%20variables.pdf` - detailed on the xv6 percpu variable implementation.
- `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/osfmk/i386/cpu_data.h` - search for "per-cpu" and note how a per-cpu variable is used to disable preemption.

The GDT

- `http://wiki.osdev.org/Segmentation`
- `http://wiki.osdev.org/Global_Descriptor_Table`
- `http://wiki.osdev.org/GDT_Tutorial` - ignore the detailed code.

Inline assembly

Note that NASM orders the operands of an instruction using "operation destination, source", while gcc's inline assembler uses "operation source, destination". Do not be confused.

- `http://wiki.osdev.org/Inline_Assembly` - Probably the most appropriate overview.
- `http://wiki.osdev.org/Inline_Assembly/Examples`
- `http://ericw.ca/notes/a-tiny-guide-to-gcc-inline-assembly.html` - Likely clearest.
- `http://www.osdever.net/tutorials/view/a-brief-tutorial-on-gcc-inline-asm` - Probably too detailed.

- `http://asm.sourceforge.net/articles/rmiyagi-inline-asm.txt` - Describes all constraints; too detailed.