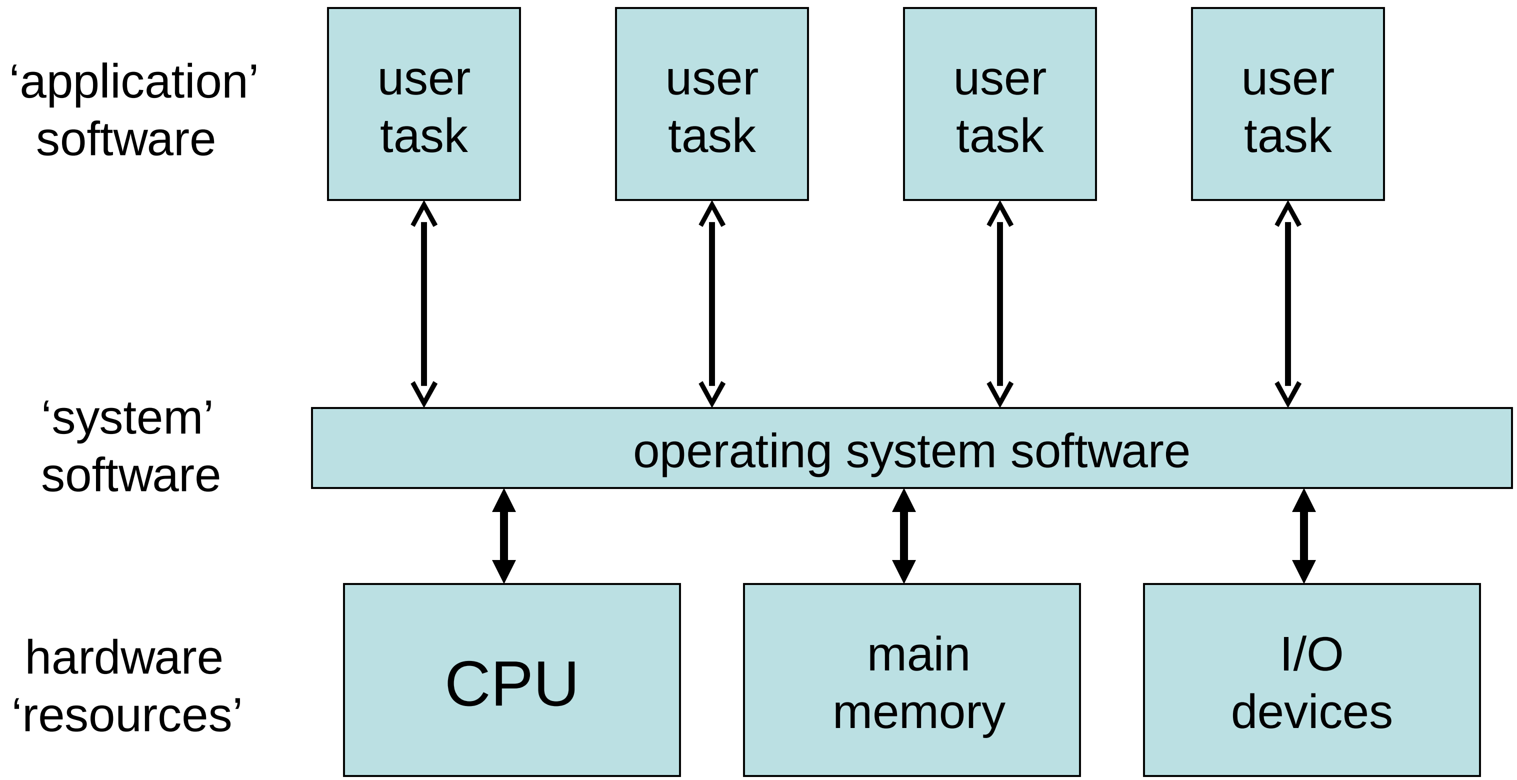
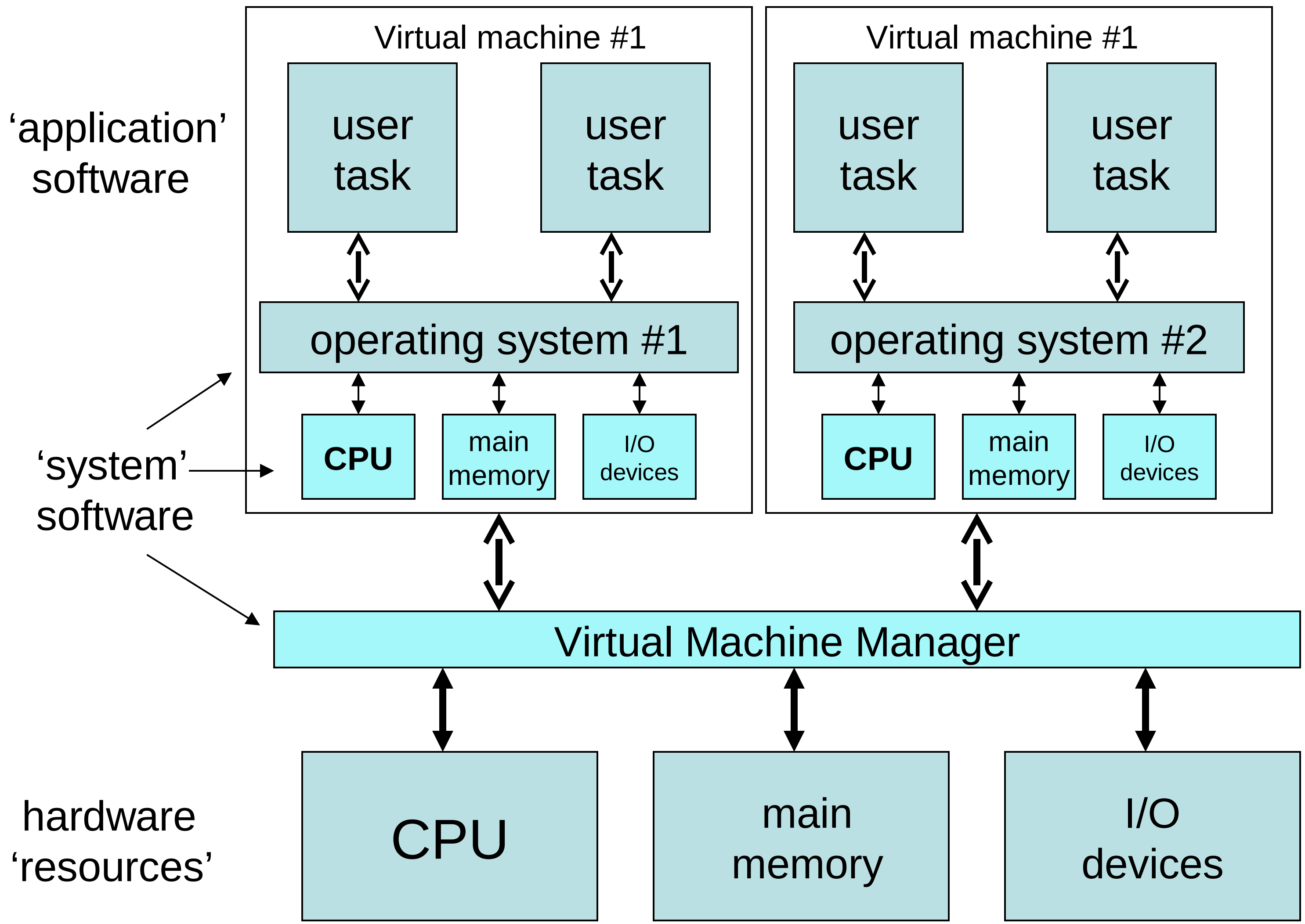
Virtualization Technology

A first look at some aspects of Intel’s ‘Vanderpool’ initiative

What is a Virtual Machine?



What is a Virtual Machine?



Background

- The ‘Virtual Machine’ concept isn’t new – IBM mainframes implemented it in 1960s
- Features of ‘Classical Virtualization’:
  - FIDELITY: software’s execution on the ‘virtual’ machine is identical -- except for timing -- to its execution on actual hardware
  - PERFORMANCE: the vast majority of a guest’s instructions are executed without any intervention
  - SAFETY: all hardware resources are controlled by the Virtual Machine Manager

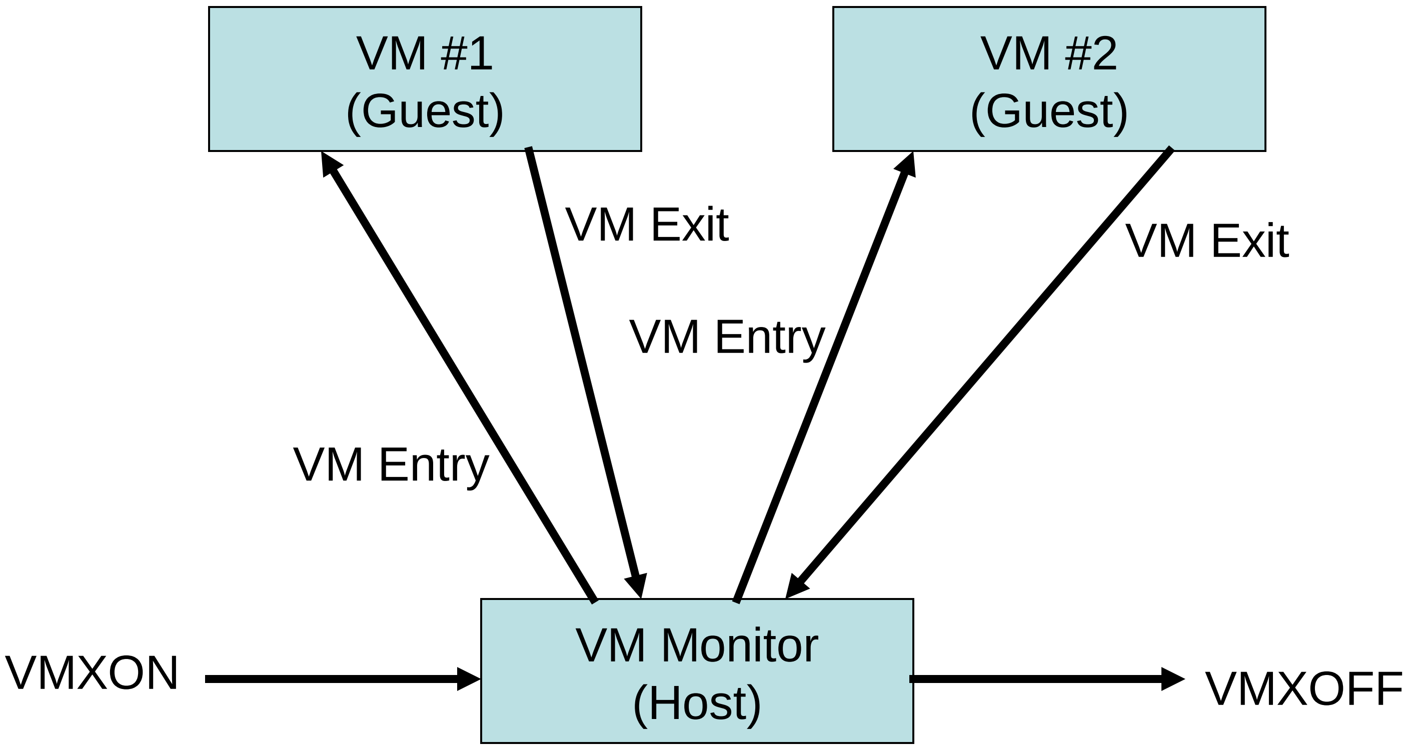
x86 poses some problems

- Certain x86 instructions were impossible to truly ‘virtualize’ in that classical sense
- For example, the ‘smsw’ instruction can be executed at any privilege-level, and in any processor mode, revealing to software the current hardware status (e.g., PE, PG, ET)
- Intel’s Vanderpool Project endeavored to remedy this (using new processor modes)

VT-x

- Virtualization Technology for x86 CPUs
- Two new processor execution-modes
  - VMX ‘root’ mode (for VM Managers)
  - VMX ‘non-root’ mode (for VM Guests)
- Ten new hardware instructions
- A six-part VMCS data-structure
- A variety of control-options for VMs

Interaction of VMs and VMM



VMCS

- Virtual Machine Control Structure
  - A six-part data-structure (fits in a page-frame)
  - One VMCS for each VM, one for the Monitor
  - CPU is told physical address of each VMCS
  - Software must first “initialize” each VMCS
  - Then no further direct access to a VMCS
  - Access is indirect (via VMX instructions)
  - One VMCS is “active”, others are “inactive”

Six logical groups

- Organization of contents in the VMCS:
  - The ‘Guest-State’ area
  - The ‘Host-State’ area
  - The VM-execution Control fields
  - The VM-exit Control fields
  - The VM-entry Control fields
  - The VM-exit Information fields

The ten VMX instructions

- VMXON and VMXOFF
- VMPTRLD and VMPTRST
- VMCLEAR
- VMWRITE and VMREAD
- VMLAUNCH and VMRESUME
- VMCALL

Capabilities are model-specific

- Intel’s Virtualization Technology is under continuing development (experimentation)
- Each iteration is identified by a version-ID
  - Example: Pentium-D 900-series (ver 0x3)
  - Example: Core-2 Duo (ver 0x07)
- Software can discover the processor’s VMX capabilities by reading from MSRs
- But the **rdmsr** instruction is ‘privileged’

Types of files

- UNIX systems implement ordinary files for semi-permanent storage of programs/data
- But UNIX systems also implement several kinds of ‘special’ files (such as device-files and symbolic links) which enable users to employ familiar commands and functions (e.g., open(), read(), write(), and close()) when working with other kinds of objects

‘virtual’ files

- Among the various types of ‘special’ files are the so-called ‘pseudo’ files
- Unlike ordinary files which hold information that is ‘static’, the pseudo-files don’t ‘store’ any information at all – but they ‘produce’ information that is created dynamically at the moment when they are being read
- Traditionally they’re known as ‘/proc’ files

Text in ‘/proc’ files

- Usually the data produced by reading from a ‘/proc’ file consists of pure ASCII text (a few exceptions exist, however)
- This means you can view the contents of a ‘/proc’ file without having to write a special application program – just use ‘cat’!
- For example:

```
$ cat /proc/version
```

More ‘/proc’ examples

- $ cat /proc/cpuinfo
- $ cat /proc/modules
- $ cat /proc/meminfo
- $ cat /proc/iomem
- $ cat /proc/devices
- $ cat /proc/self/maps

[Read the ‘man-page’ for details: $ man proc]

Create your own pseudo-files

- You can use our ‘newinfo.cpp’ wizard to create ‘boilerplate’ code for a module that will create a new pseudo-file when you ‘install’ the module into a running kernel
- The module’s ‘payload’ is a function that will get called by the operating system if an application tries to ‘read’ from that file
- The ‘get_info()’ function has full privileges!

The ‘asm’ construct

- When using C/C++ for systems programs, we sometimes need to employ processor-specific instructions (e.g., to access CPU registers or the current stack area)
- Because our high-level languages strive for ‘portability’ across different hardware platforms, these languages don’t provide direct access to CPU registers or stack

gcc/g++ extensions

- The GNU compilers support an extension to the language which allows us to insert assembler code into our instruction-stream
- Operands in registers or global variables can directly appear in assembly language, like this (as can immediate operands):

```
int     count = 4;          // global variable
asm("  movl   count , %eax ");
asm("  imull  $5, %eax, %ecx ");
```

Local variables

- Variables defined as local to a function are more awkward to reference by name with the ‘asm’ construct, because they reside on the stack and require the generation of offsets from the %ebp register-contents
- A special syntax is available for handling such situations in a manner that gcc/g++ can decipher

Template

- The general construct-format is as follows:

```
asm(  instruction-template
    : output-operand
    : input-operand
    : clobber-list );
```

Loop to read VMX MSRs

```
# This assembly language loop, executing at ring0, reads the eleven
# VMX-Capability MSRs (Model-Specific Registers) and stores their
# values in a memory-array consisting of eleven 64-bit array-entries

        .text
        xor     %rbx, %rbx          # initialize the array-index
        mov     $0x480, %ecx        # initial MSR register-index
nxmsr:
        rdmsr                       # read Model-Specific Register
        mov     %eax, msr0x480+0(, %rbx, 8)     # bits 31..0
        mov     %edx, msr0x480+4(, %rbx, 8)     # bits 63..32
        inc     %ecx                # next MSR register-index
        inc     %rbx                # increment the array-index
        cmp     $11, %rbx           # index exceeds array-size?
        jb      nxmsr               # no, then read another MSR

        .data
msr0x480:  .space   88              # enough for 11 quadwords
```

Using the ‘asm’ construct

```
// Here we use inline assembly language (and the ‘asm’ construct) to
// include a loop to read those MSRs within a “C” language module

#define  MSR_EFER         0x480     // initial MSR register-index

unsigned long     msr0x480[ 11 ];   // declared as a global array

asm(      “  xor      %%rbx, %%rbx                          \n”\
          “  mov      %0, %%ecx                             \n”\
          “nxmsr:     rdmsr                                 \n”\
          “  mov      %%eax, msr0x480+0( , %%rbx, 8)        \n”\
          “  mov      %%edx, msr0x480+4( , %%rbx, 8)        \n”\
          “  inc      %%ecx                                 \n”\
          “  inc      %%rbx                                 \n”\
          “  cmp      $11, %%rbx                            \n”\
          “  jb       nxmsr                                 \n”\
          :: “i” (MSR_EFER) : “ax”, “bx”, “cx”, “dx” );
```

Our ‘vmxmsrs.c’ LKM

- We created a Linux Kernel Module that lets users see the values in the eleven VMX-Capability Model Specific Registers
- Our module implements a ‘pseudo’ file in the ‘/proc’ directory
- You can view that file’s contents by using the ‘cat’ command, like this:

```
$ cat /proc/vmxmsrs
```

Using the LKM

- We use our ‘mmake.cpp’ utility to compile any Linux Kernel Module for kernel 2.6.x:

  $ mmake vmxmsrs

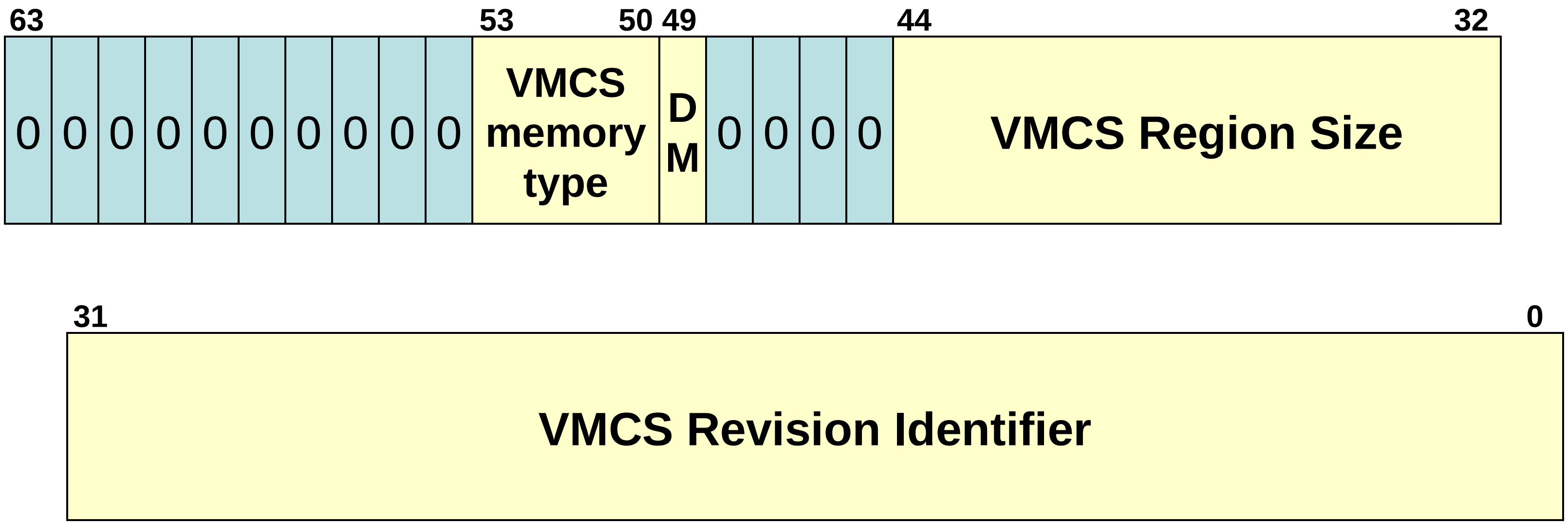
- We use the ‘/sbin/insmod’ command to install the compiled kernel-object:

  $ /sbin/insmod vmxmsrs.ko

- We can view the privileged information from those MSRs:

  $ cat /proc/vmxmsrs

VMX Basic MSR



DM = Dual-Monitor treatment of SMM supported (1=yes, 0=no)

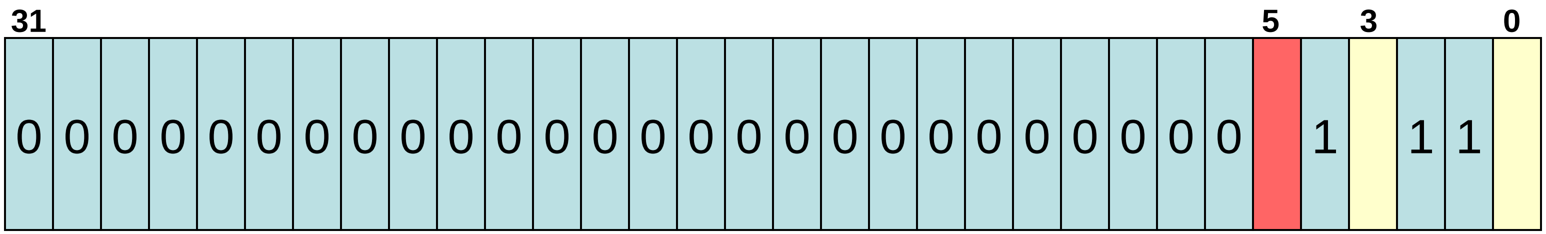
Codes for **memory-type** used for VMCS access:

0000 = Strong UnCacheable (UC)

0110 = Write Back (WB)

(no other values are currently defined for use here)

Pin-based execution controls



Bit 0: External-Interrupt Exiting (1=yes, 0=no)
Bit 3: NMI Exiting (1=yes, 0=no)
Bit 5: Virtual NMIs (1=yes, 0=no)

= this bit has no function (but must have a fixed-value)

= this bit’s function is programmable on our Core-2 Duo cpus

= this bit’s function is unavailable on our Core-2 Duo cpus

CPU-based execution controls

31	30	29	28			25	24	23		21	20	19							12	11	10	9		7				3	2		0
0				0	1				0				0	0	1	1	1	1					1		1	1	1			1	0

Core-2 Duo
(1=yes, 0=no)

Bit 2: Interrupt-window exiting
Bit 3: Use TSC offsetting
Bit 7: HLT exiting
Bit 9: INVLPG exiting
Bit 10: MWAIT exiting
Bit 11: RDPMC exiting
Bit 12: RDTSC exiting

Bit 19: CR8-load exiting
Bit 20: CR8-store exiting
Bit 21: Use TPR shadow
Bit 23: Mov-DR exiting
Bit 24: Unconditional I/O exiting
Bit 25: Activate I/O bitmaps
Bit 28: Use MSR bitmaps
Bit 29: MONITOR exiting
Bit 30: PAUSE exiting

In-class exercise

- Can you write an LKM named ‘sysregs.c’ that will create a pseudo-file which lets a user see the current contents of the five processor Control Registers (CR0, CR2, CR3, CR4, CR8) available on machines that implement EM64T, using the Linux ‘cat’ command:

```
$ cat  /proc/sysregs
```

(Hint: You can try using the ‘asm’ construct)