Implementación Sencilla de un Hipervisor con soporte Intel VT-x

Creación de un entorno invitado

sys_env_mkguest() [kern/syscall.c] comprueba si el procesador tiene soporte de VMX y crea un nuevo entorno invitado.

En **kern/env.c** se implementan las funciones de gestión de los entornos: **envid2env()**, **env_init()**, **env_init_percpu()**, **env_setup_vm()**, **env_guest_alloc()**, **env_guest_free()**, **env_alloc()**, **region_alloc()**, **load_icode()**, **env_create()**, **env_free()**, **env_destroy()**, **env_pop_tf()** y **env_run()**.

Función **vmx_check_support() [vmm/vmx.c]** para comprobar si la cpu tiene soporte VMX. Para ello, hay que leer el bit 5 del registro ECX (si está a 1, hay soporte) a través de la función CPUID.

Función **vmx_check_ept() [vmm/vmx.c]** para comprobar si está soportada la paginación extendida (EPT). Para ello, hay que comprobar en el registro MSR IA32_VMX_PROCBASED_CTLS el bit 63 que indica si los controles VMX secundarios están activados (bit a 1). Además, se comprueba en el registro MSR IA32_VMX_PROCBASED_CTLS2 el bit 33 que indica si el EPT está disponible.

Función **sched_yield() [kern/sched.c]** para que un hilo renuncie a la CPU y de paso al siguiente. Función **vmxon() [kern/sched.c]** para arrancar el modo de ejecución: vmx root.

Doble traducción de dirección:

app -> VMM -> host
va -> guest_pa -> pa

Mapeo del bootloader y kernel del huésped

Función **sys_ept_map() [kern/syscall.c]** para mapear una página del VMM a una MV. Para ello, hay que comprobar que:

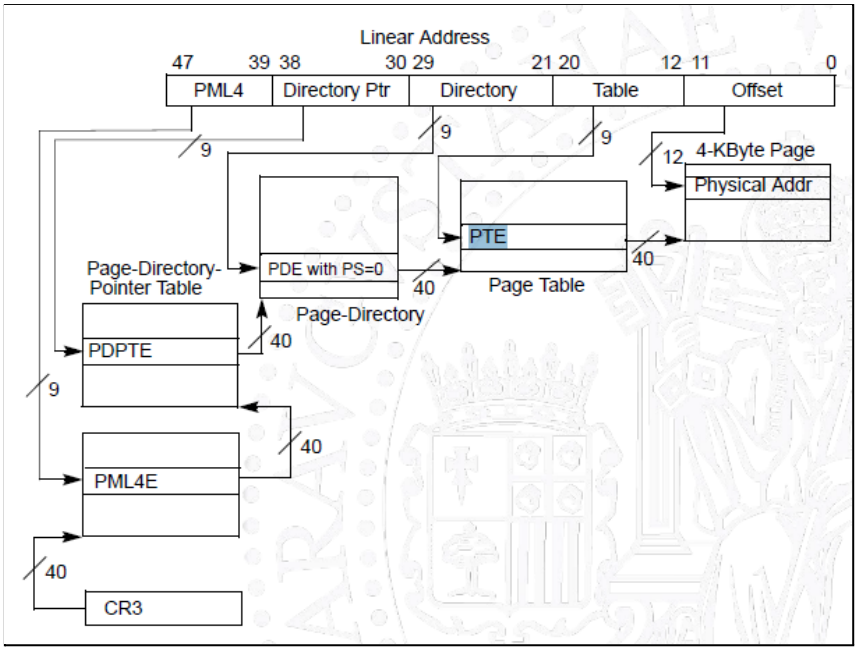
1. Existen los entornos
2. La alineación de las páginas y si superan los límites
3. Existe la página a mapear
4. Los permisos son correctos
5. Se mapea y se comprueba si ha ido bien

Función **ept_page_insert() [vmm/ept.c]** añade la entrada de la página mapeada en el EPT. Para ello, hay que:

1. Buscar el PTE final a partir de la dirección gpa (linear address).
2. Si
    a. está ocupada la entrada PTE por una página:
        i. Si overwrite es 0, no se sobreescribe la página.
        ii. Si overwrite es 1, se sobreescribe la página.
    b. no está ocupada, se escribe la página.

Función **ept_lookup_gpa() [vmm/ept.c]** devuelve el PTE final a partir de una gpa (guest physical address). Para ello, hay que:

1. Comprobar que exista eptrt (EPT root)
2. Con la función pml4e_walk() invoca el Page Table Walker desde el nivel más alto (pml4, page map level 4) que se encarga de obtener a partir de la gpa (linear address) el PTE final.
3. Antes de devolver el PTE final en epte_out, hay que establecer los permisos de las entradas EPT intermedias (PML4E, PDPTE y PDE) a __EPTE_FULL (porque lo pone en la cabecera).



Función **map_in_guest() [usr/vmm.c]** para mapear una región de un fichero (ejecutable) en la MV. Para ello, hay que comprobar que:

1. Con PGOFF se comprueba que la dirección gpa está alineada.
    a. Si no está alineada, se alinea con ROUNDDOWN
2. En un bucle se recorre toda la región a mapear (de tamaño filesz) saltando de page en page:
    a. Se reserva espacio en UTEMP (espacio de memoria temporal)
    b. Se busca la dirección base de la región en el file.
    c. Se lee una página de la región del file.
    d. Se escribe la página leída en UTEMP.
    e. Se escribe la página de UTEMP en el guest.
    f. Se elimina la página de UTEMP (haciendo que solo haya una página mapeada en UTEMP en un instante dado).
3. En otro bucle se recorre el resto de memoria (hasta memsz) saltando de page en page:
    a. Se coge una page vacía de UTEMP (todo a 0's).
    b. Se escribe la página vacía de UTEMP en el guest.
    c. Se elimina la página de UTEMP.

Función **copy_guest_kern_gpa() [usr/vmm.c]** para mapear un fichero (ejecutable) en la MV, es decir, que lee la cabecera del ELF y después invoca recursivamente a map_in_guest() para mapear todas las regiones. Para ello, hay que:

1. Abrir el fichero ELF con permisos de lectura.
2. Leer los 512 bytes que ocupa el gestor de arranque (según el enunciado).
3. Se comprueba que el número mágico de la cabecera corresponda con el número mágico de los ficheros ELF (ELF_MAGIC). Este número sirve para asegurar que el fichero es de tipo ELF.
4. *ph* es el iterador que recorre las regiones del fichero a mapear. *eph* es la última región que debe ser mapeada (límite del bucle for).
5. Para cada región se comprueba que sea de tipo ELF_PROG_LOAD, lo que significa que esa región debe ser mapeada en memoria.
    a. Si es así, se mapea la región con la función map_in_guest().

Implementación de vmlaunch y vmresume

Función **asm_vmrun() [vmm/vmx.c]** para arrancar una MV, es decir, copiar los registros del trapframe a los registros físicos y lanzar la MV (con vmlaunch si es la primera vez que se ejecuta la MV, y con vmresume en caso contrario). Para ello, en ensamblador:

1. Cargamos los registros físicos con el contenido del trapframe de la MV.
2. Lanzamos la MV con vmlaunch o vmresume según corresponda.

Gestionando la terminación de una MV

Función **vmexit() [vmm/vmx.c]** para sacar de ejecución a una MV, se comporta de una u otra manera en función de la forma en la que la MV dejó de ejecutar. Para ello:

1. Se comprueba en la estructura VMCS (Virtual-Machine Control Structure) la razón de salida de ejecución.
    a. Con la función vmcs_read32() se accede a un campo de dicha estructura.
    b. El campo que nos interesa es VMCS_32BIT_VMEXIT_REASON.

Mapa multi-arranque (e820)

Función **handle_vmcall() [vmm/vmexits.c]** que es el manejador que captura la excepción producida por la instrucción vmcall (solicita un mapa de memoria “falso” para la MV) desde el guest. Para ello:

1. Si el trap es VMX_VMCALL_MBMAP
    a. Ese mapa de memoria multiboot se divide en tres segmentos: low memory, IO memory, high memory
        i. Características de la low memory
            - Size = 20 porque es el tamaño mínimo
            - Base_addr = 0x0 porque comienza al principio
            - Length = IOPHYSMEM porque termina cuando comienza la IO memory
            - Type = MB_TYPE_USABLE porque puede usarse
        ii. Características de la IO memory
            - Size = 20 porque es el tamaño mínimo
            - Base_addr = IOPHYSMEM porque comienza tras la low memory

        - Length = EXTPHYSMEM - IOPHYSMEM porque termina cuando comienza la high memory
        - Type = MB_TYPE_RESERVED porque no es accesible
    iii. Características de la highmemory
        - Size = 20 porque es el tamaño mínimo
        - Base_addr = EXTPHYSMEM porque comienza al principio
        - Length = glnfo->phys_sz - EXTPHYSMEM porque termina al final de la memoria física asignada a la MV
        - Type = MB_TYPE_USABLE porque puede usarse
  b. Previo al mapa de memoria multiboot está una pequeña cabecera (multiboot_info_t)
    i. Se indica el tamaño del mapa: tamaño total de los tres segmentos
    ii. Se indica la dirección base del mapa: la dirección base que ya nos daban sumado al offset de la cabecera
    iii. Se indican los flags: MB_FLAG_MMAP
  c. Se pide una página de memoria libre con page_alloc()
  d. Se traslada el mapa de memoria multiboot creado anteriormente a la página obtenida del paso anterior. Se emplea la función memmove() primero para la cabecera y segundo para el mapa.
  e. Se deja constancia de la nueva página en el EPT añadiendo una nueva entrada con ept_page_insert().
  f. Tal y como indica el comentario de la función, se devuelve un puntero de la nueva región en el registro %rbx
2. Si el trap es VMX_VMCALL_IPCSEND
  a. Se invoca a la syscall para llamar a SYS_ipc_try_send() [explicado más abajo]
3. Si el trap es VMX_VMCALL_IPCRECV
  a. Se invoca a la syscall para llamar a SYS_ipc_recv() [explicado más abajo]

Función **handle_cpuid() [vmm/vmexits.c]** para cargar en el TrapFrame de la MV las características de la CPU (será empleada para comprobar el soporte a *long mode*). Para ello:

1. Se toma el registro RAX del TrapFrame
2. Se invoca a cpuid(), pasándole como primer parámetro el registro anterior, para conocer las características de la CPU
3. Se actualiza el TrapFrame con la información obtenida de la invocación anterior

Función **bc_pgfault() [fs/bc.c]** que trae un sector de disco a un bloque de memoria:

1. Obtenemos la nueva dirección de memoria donde se va a escribir el sector del disco
2. Se reserva una página de memoria
3. Se obtiene el sector de disco a traer a la memoria
  a. Si es el SO host se utiliza la función ide_read()
  b. Si es el SO guest se utiliza la función host_read()
    i. En ambas funciones, se pasa el sector de disco que queremos leer (sec_no), la dirección de memoria donde queremos escribirlo (new_addr) y el tamaño del sector (nsecs)

Función **flush_block() [fs/bc.c]** que expulsa un bloque de memoria a un sector de disco:

1. Obtenemos la dirección de memoria donde se encuentra el bloque a expulsar
2. Se comprueba
    a. Si el bloque existe y está sucio, hay que escribirlo en disco
        i. Si es el SO host se utiliza la función ide_write()
        ii. Si es el SO guest se utiliza la función host_write()
            1. En ambas funciones, se pasa el sector de disco que queremos escribir (sec_no), la dirección de memoria del bloque (addr) y el tamaño del sector (nsecs)
    b. Si el bloque existe y no está sucio, no hace falta escribirlo en disco
    c. Si el bloque no existe, no tiene sentido hacer un flush

Función **ipc_host_send() [lib/ipc.c]** en vez de realizar la llamada de sys_ipc_try_send() directamente lo hace por medio de la instrucción en ensamblador vmcall, para que se produzca excepción y la capture handle_vmcall() entrando en el case VMX_VMCALL_IPCRECV:

1. Ejecuta vmcall con el parámetro VMX_VMCALL_IPCRECV

Función **ipc_host_recv() [lib/ipc.c]** en vez de realizar la llamada de sys_ipc_recv() directamente lo hace por medio de la instrucción en ensamblador vmcall, para que se produzca excepción y la capture handle_vmcall() entrando en el case VMX_VMCALL_IPCSEND:

1. Ejecuta vmcall con el parámetro VMX_VMCALL_IPCSEND

Función sys_ipc_try_send()

Función sys_ipc_recv()

Errores solucionados

Error del spin_unlock() solucionando invocando lock_kernel() en la función i386_init() de kern/init.c