Soporte a los sistemas operativos multitarea y a la virtualización

Área de Arquitectura y Tecnología de Computadores – Versión 3.1.112009, 25/09/2023

Objetivos de la sesión

Esta sesión trata aspectos relacionados con el soporte por parte de la CPU a los sistemas operativos multitarea y a la virtualización.

En una primera parte de la sesión se muestran los principales mecanismos de protección que dan soporte a los sistemas operativos multitarea. Incluyen al menos los siguientes tipos de protección:

- Niveles de privilegio. Evitan que las tareas puedan ejecutar instrucciones privilegiadas. El sistema operativo, en cambio, puede ejecutar cualquier instrucción del juego de instrucciones, incluyendo las privilegiadas.
- Protección de memoria. Evita que una tarea pueda acceder al área de memoria de otra tarea o al área de memoria del sistema operativo.

Para ilustrar los mecanismos de protección anteriores se realizarán varios programas sobre un sistema operativo Linux.

Finalmente, se tratan algunos de los conceptos de virtualización explicados en teoría.

Conocimientos y materiales necesarios

Para poder realizar esta sesión, el alumno debe:

- Revisar los apuntes de teoría correspondientes al apartado dedicado al soporte de la CPU a los sistemas operativos multitarea y a la virtualización.
- Durante la sesión se plantearán una serie de preguntas que deben responderse en el correspondiente cuestionario en el campus virtual. Puedes abrir el cuestionario en otra pestaña del navegador pinchando en el enlace mientras mantienes pulsada la tecla Ctrl .

La sesión práctica se realizará sobre la máquina virtual Linux de la asignatura.

1. La protección en el sistema operativo Linux

Para probar la protección de los sistemas operativos emplearemos la máquina virtual Linux de la asignatura.

Algunas pruebas que realizaremos en Linux pueden exigir ejecutar determinadas instrucciones en ensamblador. En ese caso incluiremos instrucciones ensamblador dentro del programa C. La sintaxis es análoga a la del ejemplo siguiente:

```
#include <stdio.h>
int main()
{
   printf("Hello\n");

   __asm__
   (
     "mov %ax, 0\n\t"
     "add %ax, %cx"
   );
}
```

Observa cómo se pone cada instrucción ensamblador en una línea y todas las líneas salvo la última se terminan con \n\t . Aunque no se aprecia bien en el código de ejemplo, justo antes y después de la palabra asm hay dos guiones bajos consecutivos.

- Arranca la máquina virtual Linux.
- Cambia al directorio de tu repositorio y sincronízalo con el repositorio de Bitbucket para actualizar los cambios que se hubiesen realizado con anterioridad.

```
$> git pull
```

- Descarga los ficheros necesarios para la práctica y descomprímelos con estas órdenes:

```
$> wget http://rigel.atc.uniovi.es/grado/2ac/files/sesion2-4.tar.gz
$> tar xvfz sesion2-4.tar.gz
```

- Añade los ficheros al índice git y confirma los cambios con estas órdenes:

```
$> git add sesion2-4
$> git commit
```

- Cámbiate al directorio sesion2-4 .

1.1. Comprobación de los niveles de privilegio

Los niveles de privilegio impiden que las tareas puedan ejecutar determinadas instrucciones de la arquitectura. Si intentan hacerlo la instrucción correspondiente da lugar a una excepción.

En los sistemas Linux, y Unix en general, cuando un programa termina por una excepción se puede generar un archivo de nombre core dentro del mismo directorio en el que se encuentra el ejecutable. Dicho archivo contiene una imagen de la memoria del programa y de sus registros, lo que permite el análisis *post mortem* de las causas del problema. Sin embargo, en algunas sistemas dicho archivo no se genera por defecto, pues muchos usuarios no lo usan.

- Ejecuta la orden ulimit -c unlimited para forzar la generación del archivo core en caso de excepción.

Pasemos a comprobar la existencia de niveles de privilegio.

- Edita el archivo 2-4priv.c proporcionado para visualizar su contenido.
- Compila el programa anterior y ejecútalo. Verás que Linux indica que ha habido un error de ejecución ¿Qué ha ocurrido? ¿Cuál es la razón? Responde en el cuestionario pregunta 1.
- Para comprobar la respuesta vamos a ejecutar el depurador gdb que es el habitual de los sistemas Linux. Ejecuta la siguiente orden.

```
$> gdb 2-4priv core
```

  Una vez ejecutada la orden anterior el depurador nos muestra la causa de la excepción y nos proporciona una interfaz de comandos propia que permite introducir *breakpoints*, visualizar el contenido de la memoria y registros, etc. En la última línea observarás que hace referencia a la función main() , lo que no da muchas pistas sobre la instrucción causante de la excepción. La razón es que el archivo ejecutable no contiene información de depuración.

- Sal del depurador con la orden quit .
- Borra el archivo core y compila ahora el programa 2-4priv.c usando la opción -g , la cual añade dentro del archivo ejecutable información para la depuración.

  Debes borrar siempre el archivo core , pues si ya existe NO SE SOBREESCRIBE.

- Ejecuta de nuevo el programa y analiza la causa de la excepción empleando el depurador gdb . Ahora te indicará una línea en el archivo fuente. Observarás que no hace referencia exactamente a la instrucción en ensamblador, sino al bloque ensamblador. La explicación es que el bloque en su conjunto se interpreta como una única instrucción de alto nivel.
- Para ver la instrucción ensamblador que produce la excepción teclea layout asm en gdb. Verás que se abre una ventana con la traducción a ensamblador del código del programa y que el programa está detenido sobre la instrucción concreta que produjo la excepción, en este caso cli .

  Si quieres volver a ver el código del programa en lenguaje C, teclea layout prev en gdb.

- Sal de la interfaz de comandos de gdb y borra el archivo core .

Como habrás podido observar, el sistema operativo Linux impide la ejecución de la instrucción de deshabilitación de interrupciones. Si no fuese así, cualquier tarea que se ejecutase sobre Linux podría provocar un mal funcionamiento del sistema. Para que Linux pueda proporcionar esta protección la CPU debe dar el soporte necesario.

A continuación vamos a comprobar la protección de memoria de Linux.

1.2. Comprobación de la protección de memoria

La memoria está protegida en Linux, de tal forma que una tarea sólo puede acceder a las áreas de memoria que tiene asignadas.

- Edita el archivo 2-4mem1.c proporcionado para visualizar su contenido.
- Compila el programa 2-4mem1.c con la opción de depuración y a continuación ejecútalo. ¿Qué ha ocurrido? ¿Cuál es la razón? Responde en el cuestionario pregunta 2.
- Para comprobar la respuesta ejecuta el depurador con la orden gdb 2-4mem1 core . ¿Qué instrucción da lugar a la excepción? Responde en el cuestionario pregunta 3.
- Recuerda salir del depurador con la orden quit y borrar el archivo core generado.
- ¿Por qué crees que la instrucción que asigna la dirección de memoria al puntero p no genera una excepción? Responde en el cuestionario pregunta 4

A partir de la prueba anterior hemos comprobado que la memoria está protegida en Linux, gracias al soporte proporcionado por el procesador. Cuando una tarea intenta acceder a una dirección que no le pertenece se produce una excepción.

Las excepciones son muy beneficiosas durante el proceso de depuración de programas pues permiten detectar problemas que de otra manera serían muy difíciles de encontrar.

- Edita un programa de nombre 2-4mem2.c con el siguiente código.

```
#include <stdio.h>

int main()
{
   int *p = NULL; /* Pointers should be initialized to NULL */

   printf("%d\n", *p);
}
```

  La constante NULL se emplea para inicializar punteros en C y habitualmente toma el valor cero.

- ¿Qué instrucción crees que dará lugar a una excepción? Responde en el cuestionario pregunta 5.
- Compila y ejecuta el programa anterior. Utiliza el gdb para identificar la instrucción que da lugar a una excepción. ¿Coincide con la que habías predicho? Recuerda salir del depurador con la orden quit y borrar el archivo core generado.

La inicialización de punteros a NULL permite detectar cuando se escribe de forma incorrecta en un puntero no inicializado, lo cual es un error bastante común cuando se programa en C.

2. Virtualización en x86-64

Los ordenadores de la sala de prácticas utilizan la arquitectura x86-64 (a veces llamada x64), que es una evolución de 64 bits sobre la arquitectura de 32 bits desarrollada por Intel y denominada habitualmente x86. La arquitectura x86 no estaba pensada para ser virtualizada y no cumplía los requisitos de Popek y Goldberg para una virtualización eficiente. En concreto, había instrucciones sensibles que no eran privilegiadas.

El funcionamiento habitual de la arquitectura x86 define distintos niveles de privilegio a los que denomina anillos. Habitualmente, el núcleo del sistema operativo funciona en el anillo 0, que es el de más privilegio, y las aplicaciones, en el anillo 3, con menor privilegio. Los primeros programas de virtualización movieron el sistema operativo al anillo 1 y ejecutaban el hipervisor o Virtual Machine Monitor (VMM) en el nivel 0. Sin embargo, esto no bastaba para implementar la técnica de *trap-and-emulate* debido a las instrucciones sensibles que no provocaban una excepción cuando se ejecutaban en el anillo 1, con lo que había que utilizar también traducción binaria para lidiar con estas instrucciones, lo que reducía el rendimiento.

Para evitar este problema, tanto AMD como Intel introdujeron unas extensiones de soporte a la virtualización, denominadas respectivamente AMD-V e Intel VT-x, que añadían un nuevo modo de operación de la CPU, llamado *Guest Mode* en AMD y *VMX Root Mode* en Intel. Este modo tiene más privilegio que el anillo 0 y permite que el hipervisor o Virtual Machine Monitor (VMM) se ejecute en este nivel (a veces llamado anillo -1) contrastando los sistemas operativos que se ejecutan en el anillo 0.

En las prácticas se utiliza VirtualBox como VMM. ¿Qué tipo de VMM (1 o 2) es VirtualBox y por qué? Responde en el cuestionario pregunta 6.

Vamos a analizar algunas opciones de VirtualBox.

- Abre VirtualBox y para la máquina virtual Linux vete a la pestaña Aceleración dentro de **Configuración > Sistema** . ¿Se está utilizando VT-x/AMD-V? Responde en el cuestionario pregunta 7.

Como verás, en esa ventana hay una opción para seleccionar el tipo de interfaz de paravirtualización. Como has estudiado en clase, la paravirtualización consiste en modificar el sistema operativo para que incluya llamadas a funciones del hipervisor. Por lo tanto, estas funciones son distintas dependiendo del tipo de hipervisor que se esté utilizando. Los sistemas operativos modernos suelen ofrecer estas funciones de paravirtualización, pero cada sistema operativo ofrece llamadas a sólo algunos tipos de hipervisores.

VirtualBox puede aprovechar de algunas funciones de paravirtualización para mejorar aún más el rendimiento. Entre las opciones que ofrece VirtualBox están:

- Ninguna: no utiliza estas extensiones.
- Predeterminada: selecciona la opción en función del sistema operativo invitado seleccionado durante la creación de la máquina virtual.
- Heredada: es para máquinas virtuales creadas con versiones antiguas de VirtualBox.
- Mínima: simplemente le indica al sistema operativo invitado que se está ejecutando dentro de un hipervisor y le da cierta información sobre la frecuencia de algunos elementos del sistema. Es necesaria cuando el sistema operativo invitado es Mac OS X.
- Hyper-V: utiliza las llamadas al hipervisor Hyper-V creado por Microsoft.
- KVM: utiliza las llamadas al hipervisor KVM (Kernel-based Virtual Machine) desarrollado para virtualizar sistemas Linux.

Los sistemas Linux tienen, lógicamente, extensiones para hacer llamadas al hipervisor KVM pero también para Hyper-V, ya que Microsoft aportó al núcleo de Linux el código necesario para que funcionasen mejor los invitados Linux en los anfitriones Windows, algo que utiliza, por ejemplo, en Azure, su nube de máquinas virtuales, que permite ejecutar máquinas virtuales bajo demanda en servidores de Microsoft.

La opción por defecto en la sección de paravirtualización es Predeterminada . Vamos a comprobar que, como estamos ejecutando un sistema invitado Linux, eso se traduce en que se utiliza KVM.

- La orden dmesg permite obtener los mensajes que genera el núcleo. Durante el inicio del sistema, el núcleo indica si se está utilizando paravirtualización, así que vamos a buscar algún mensaje relativo a la paravirtualización con esta orden:

```
$> dmesg | grep paravirt
```

  Como verás, se generan dos líneas que te indican que se está utilizando KVM. ¿Qué dice la primera línea? Responde en el cuestionario pregunta 8.

2.1. Número de procesadores

Una de las ventajas de la virtualización es que se pueden repartir los recursos de una máquina física entre varias máquinas virtuales de manera controlada. Uno de los recursos más fáciles de repartir es la CPU en sistemas multinúcleos, ya que se puede limitar el número de núcleos que utiliza la máquina virtual. Vamos a comprobar cómo afecta esto al rendimiento.

- En primer lugar, vamos a comprobar cuántos núcleos tiene el procesador físico. Abre el administrador de tareas de Windows pulsando Ctrl + Shift + Esc . Si no te aparecen pestañas, pulsa en el botón **Más detalles** . Escoge la pestaña **Rendimiento** y en la sección de CPU, mira cuántos núcleos te indica. Responde en el cuestionario pregunta 9.
- En VirtualBox, vete a la sección **Procesador** dentro del menú **Configuración > Sistema** de la máquina virtual. Si no están asignados dos procesadores, debes cambiar el valor, pero para ello tendrás que detener antes la máquina virtual y arrancarla después.
- Vamos a comprobar que ese número de núcleos es el que se ve en la máquina virtual. Ejecuta la orden nproc --all . Comprueba que te sale lo mismo que en la respuesta anterior.
- Abre Visual Studio y crea un proyecto de consola de Visual C++ de tipo **Aplicación de consola Win32** llamado **Cpuburn** . En la función main del programa copia este código:

```
for (;;)
{
   // Infinite loop to burn CPU
}
```

  La idea es que tenemos un bucle infinito que simplemente para ocupar una CPU.

- Ejecuta el programa **CpuBurn** y asegúrate, mirando el administrador de tareas, de que uno de los núcleos está ocupado mientras lo estás ejecutando.
- Mientras se está ejecutando CpuBurn , ejecuta en la máquina virtual el benchmark bench_fp_mt , creado en la sesión 1.2, mientras observas qué ocurre con el reparto de la CPU en la pestaña Procesos del administrador de tareas. ¿Qué porcentaje de CPU ocupa VirtualBox y qué porcentaje el proceso CpuBurn ? Responde en el cuestionario pregunta 10
- Cierra la máquina virtual, cambia el número de núcleos de la máquina virtual a 1 y vuelve a arrancarla.
- Mientras sigue ejecutándose CpuBurn , vuelve a ejecutar el benchmark bench_fp_mt mientras observas qué ocurre con el reparto de la CPU en la pestaña Procesos del administrador de tareas. ¿Qué porcentaje de CPU ocupa ahora VirtualBox y qué porcentaje el proceso CpuBurn ? Responde en el cuestionario pregunta 11
- Cierra el programa CpuBurn . Limpia el proyecto, cierra el Visual Studio, comprime el directorio del proyecto con el nombre 2-4CpuBurn.zip y añade el archivo resultante al repositorio.

Archivos de la práctica

En tu repositorio deben tener los archivos 2-4priv.c , 2-4mem1.c , 2-4mem2.c y 2-4CpuBurn.zip . Recuerda sincronizar tu repositorio local con el repositorio en Bitbucket.