6

Mecanismo: Ejecución directa limitada

Para virtualizar la CPU, el SO necesita, de alguna manera, compartir la CPU física entre muchas tareas que pareciera que corren al mismo tiempo. La idea básica es simple: correr un proceso por un ratito, luego correr otro y así sucesivamente. La virtualización se alcanzó de esta forma, gracias al tiempo compartido de CPU.

De todas formas, hay algunos desafíos al momento de generar dicha maquinaria de virtualización. El primero es el desempeño: ¿cómo podemos implementar la virtualización sin agregar una sobrecarga excesiva al sistema? El segundo es el control: ¿cómo podemos correr un proceso eficientemente y al mismo tiempo mantener el control de la CPU? El control es particularmente importante para el SO, porque está a cargo de los recursos; sin control, un proceso simplemente podría correr por siempre y apoderarse de la máquina, o acceder a información que no debería estar autorizado a acceder. Por lo tanto obtener un buen desempeño mientras se mantiene el control es uno de los desafíos centrales al construir un sistema operativo.

LA CUESTIÓN: CÓMO VIRTUALIZAR EFICIENTEMENTE LA CPU CONSERVANDO EL CONTROL

El SO debe virtualizar la CPU de una manera eficiente mientras se conserva el control sobre el sistema operativo. Para hacerlo, se requerirá soporte tanto del hardware como del sistema operativo. El sistema operativo a menudo utilizará un poco de hardware prudente para realizar su trabajo de manera efectiva.

6.1 Una técnica básica: Ejecución directa limitada

Para hacer que un programa corra tan rápido como uno esperaría, no es de extrañar que los/as desarrolladores/as de SOs inventaran una técnica, que llamamos **ejecución directa limitada**. La parte de la

idea "ejecución directa" es simple: solo hay que correr el programa directamente en la CPU. Así, cuando el SO desee comenzar a ejecutar un programa, solo crea una entrada para él en la lista de procesos, le asigna un poco de memoria, carga el código del programa en memoria (desde el disco), ubica su punto de entrada (es decir, la rutina `main()` o algo similar), salta a ella, y comienza a correr el código del usuario. La Figura 6.1 muestra el protocolo básico de la ejecución directa (sin ningún límite, todavía), usando una llamada normal y retornando al `main()` del programa y luego de vuelta al kernel.

SO	Programa
Crear una entrada en la lista de procesos	
Asignarle memoria al programa	
Cargar el programa en memoria	
Colocar argc/argv en el stack	
Limpiar los registros	
Ejecutar la **llamada** a main()	
	Correr main()
	Ejecutar el **retorno** de main
Liberar la memoria del proceso	
Borrarlo de la lista de procesos	

Figure 6.1: **Protocolo de la Ejecución Directa (Sin Límites)**

Suena fácil, ¿no? Pero este enfoque genera un par de problemas en nuestro intento de virtualizar la CPU. El primero es simple: si solo corremos un programa, ¿cómo puede el SO asegurar que dicho programa no hace nada que no queramos que haga, y que todavía corra eficientemente? El segundo: cuando estamos corriendo un proceso, ¿cómo lo detiene el SO y cambia a otro proceso, así implementando el **tiempo compartido** que se requiere virtualizar la CPU?

En respuesta a estas preguntas, a continuación, tendremos una mucho mejor idea de lo que se necesita para virtualizar la CPU. Al desarrollar estas técnicas, también veremos de dónde surge la parte "limitada" del nombre; sin límites en la ejecución de programas, el SO no estaría en control de nada y por lo tanto sería "solo una biblioteca" - ¡Una posición muy triste para un aspirante a sistema operativo!

6.2 Problema #1: Operaciones Restringidas

La ejecución directa tiene la obvia ventaja de ser rápida; el programa corre nativamente en el hardware de la CPU y por lo tanto ejecuta tan rápido como uno esperaría. Pero correr en la CPU genera un problema: ¿qué pasa si el proceso desea realizar algún tipo de operación restringida, como hacer un pedido de E/S a un disco, u obtener acceso a más recursos del sistema como CPU o memoria?

LA CUESTIÓN: CÓMO REALIZAR OPERACIONES RESTRINGIDAS
Un proceso debe poder realizar E/S y otras operaciones restringidas, pero sin darle al proceso un completo control sobre el sistema. ¿Cómo pueden el SO y el hardware trabajar juntos para lograrlo?

Un enfoque sería simplemente dejar que cualquier proceso haga lo que quiera en términos de E/S y otras operaciones relacionadas. Sin embargo, hacerlo evitaría la construcción de muchos tipos de sistemas que son deseables. Por ejemplo, si deseamos construir un sistema de archivos que verifique los permisos antes de otorgar acceso a un archivo, no podemos simplemente permitir que cualquier usuario emita E/Ss al disco; si lo hiciéramos, un proceso podría simplemente leer o escribir el disco entero y así todas las protecciones se perderían.

Por lo tanto, el enfoque que adoptamos es introducir un nuevo modo de procesador, conocido como **modo usuario**; el código que corre en modo usuario está restringido en lo que puede hacer. Por ejemplo, cuando se corre en modo usuario, un proceso no puede emitir solicitudes de E/S; hacerlo llevaría a que el procesador genere una excepción; y el SO probablemente mataría el proceso.

En contraste al modo usuario está el **modo kernel**, en el que el SO (o kernel) corre. En este modo, el código que corre puede hacer lo que quiera, incluyendo operaciones privilegiadas tales como emitir solicitudes E/S y ejecutar todo tipo de instrucciones restringidas.

Sin embargo, todavía nos queda un desafío: ¿qué debe hacer un proceso de usuario cuando quiera realizar una operación privilegiada, como leer del disco? Para permitir esto, prácticamente todo el hardware moderno les proporciona a los programas de usuario la capacidad de que realicen una **llamada al sistema**. Usadas por primera vez en máquinas antiguas como Atlas [K+61,L78], las llamadas al sistema permiten que el kernel exponga cuidadosamente ciertas piezas clave de funcionalidad a los programas de usuario, por ejemplo acceder al sistema de archivos, crear y destruir procesos, comunicarse con otros procesos, y asignando más memoria. La mayoría de los sistemas operativos proporcionan unos cientos de llamadas (consultar el estándar POSIX para obtener más detalles [P10]); los primeros sistemas Unix exhibieron un subconjunto más conciso de alrededor de veinte llamadas.

Para ejecutar una llamada al sistema, un programa debe ejecutar una instrucción **trap** especial. Esta instrucción simultáneamente salta al kernel y eleva el nivel de privilegios; una vez en el kernel, el sistema ahora puede realizar cualquier operación privilegiada que sea necesaria (si está permitida), y así hacer el trabajo requerido para el proceso de llamada. Cuando finaliza, el SO llama a una instrucción

APARTE: ¿POR QUÉ LAS LLAMADAS AL SISTEMAS SON COMO LLAMADAS A PROCEDIMIENTOS?

Quizás te estés preguntando por qué una llamada a sistema, tal como `open()` o `read()`, se ve exactamente como una llamada típica a un procedimiento en C; es decir, si solo se ve como una llamada a procedimiento, ¿cómo sabe el sistema que es una llamada al sistema, y hace todo correcto? La razón es simple: *es* una llamada a un procedimiento, pero escondida en esa llamada está la famosa instrucción trap. Más especificamente, cuando llamás a `open()` (por ejemplo), estás ejecutando una llamada a procedimiento en la biblioteca de C. Ahí, tanto para `open()` o como para cualquier otra llamada al sistema provista, la biblioteca usa una convención de llamada acordada con el kernel para poner los argumentos de `open()` en una ubicación conocida (por ejemplo, en el stack o en registros específicos), también coloca el número de llamada al sistema en una ubicación conocida (nuevamente, en el stack o en un registro) y luego ejecuta la intrucción trap mencionada anteriormente. El código de la biblioteca, luego de la trap, desempaca los valores de retorno y devuelve el control al programa que emitió la llamada al sistema. Por lo tanto, las partes de la biblioteca de C que hacen las llamadas al sistema son programadas a mano en assembly, ya que necesitan seguir las convenciones muy cuidadosamente para procesar argumentos y retornar los valores correctamente, así como ejecutar la instrucción trap de hardware específico. Y ahora sabés porqué vos, personalmente, no tenés que escribir código assembly para trapear al SO; alguien más ya escribió ese assembly por vos.

TIP: UTILIZAR LA TRANSFERENCIA DE CONTROL PROTEGIDA

El hardware ayuda al SO proporcionando diferentes modos de ejecución. En el **modo usuario**, las aplicaciones no tienen acceso completo a los recursos de hardware. En **modo kernel**, el SO tiene acceso a todos los recursos de la máquina. También se proporcionan instrucciones especiales para **trapear** al kernel y para **retornar de la trap** a programas en modo usuario, así como instrucciones que permiten al SO indicarle al hardware dónde reside la **tabla de traps** en la memoria.

especial de **retorno de la trap** que, como era de esperar, vuelve al programa de usuario que realiza la llamada y, al mismo tiempo, reduce el nivel de privilegios al de modo usuario.

El hardware debe tener un poco de cuidado al ejecutar una trap, ya que debe asegurarse de guardar suficientes registros del programa que hizo la llamada para poder regresar correctamente cuando el sis-

tema operativo emita la instrucción de retorno de la trap.En x86, por ejemplo, el procesador enviará el contador del programa, las flags y algunos otros registros a un **kernel stack** por proceso; al retornar de la trap se sacará estos valores del stack y reanudará la ejecución del programa de modo usuario (consulte los manuales de sistemas Intel [I11] para más detalles). Otros sistemas de hardware utilizan convenciones diferentes, pero los conceptos básicos son similares en todas las plataformas.

Hay un detalle importante que queda fuera de esta discusión: ¿cómo sabe la trap qué código ejecutar dentro del SO? Claramente, el proceso que realiza la llamada no puede especificar una dirección a la que saltar (como lo haría al realizar una llamada de procedimiento); hacerlo permitiría a los programas saltar a cualquier parte del kernel, lo que claramente es una **Muy Mala Idea**[1]. Por lo tanto, el kernel debe controlar cuidadosamente qué código se ejecuta en una instrucción trap.

El kernel lo hace configurando una **tabla de traps** en el momento de booteo. Cuando la máquina arranca, lo hace en modo privilegiado (kernel) y, por lo tanto, es libre de configurar el hardware de la máquina según sea necesario. Una de las primeras cosas que hace el SO es decirle al hardware qué código debe ejecutar cuando ocurren ciertos eventos excepcionales. Por ejemplo, ¿qué código debe ejecutarse cuando se produce una interrupción del disco duro, cuando se produce una interrupción del teclado o cuando un programa realiza una llamada al sistema? El sistema operativo informa al hardware de la ubicación de estos gestores de traps, generalmente con algún tipo de instrucción especial. Una vez que se informa al hardware, recuerda la ubicación de estas rutinas, hasta que se reinicia la máquina y, por lo tanto, el hardware sabe qué hacer (es decir, a qué código saltar) cuando tienen lugar las llamadas del sistema y otros eventos excepcionales.

Para especificar la llamada al sistema exacta, en general se asigna un **número de llamada al sistema** a cada llamada al sistema. Por lo tanto, el código de usuario es responsable de colocar el número de llamada al sistema deseado en un registro o en una ubicación específica en el stack; el SO, al encargarse de la llamada al sistema dentro del controlador de traps, examina este número, asegura que es válido y, si lo es, ejecuta el código correspondiente. Este nivel de indirección sirve como una forma de protección; el código de usuario no puede especificar una dirección exacta a la que saltar, sino que debe solicitar un servicio en particular a través de un número.

[1] Imaginate saltar al código para acceder a un archivo, pero justo después de una verificación de permisos; de hecho, es probable que tal habilidad le permita a un/a programador/a astuto/a hacer que el kernel ejecute secuencias de código arbitrarias [S07]. En general, tratá de evitar Muy Malas Ideas como esta.

TIP: CUIDADO CON LAS ENTRADAS DE USUARIOS EN SISTEMAS SEGUROS

A pesar de que nos hemos esforzado mucho por proteger el SO durante las llamadas al sistema (agregando un mecanismo de trap de hardware y asegurando que todas las llamadas al SO se enruten a través de él), todavía hay muchos otros aspectos para implementar un sistema operativo seguro que debemos considerar. Uno de ellos es el manejo de argumentos en el límite de la llamada al sistema; el SO debe verificar lo que pasa el usuario y asegurarse de que los argumentos estén especificados correctamente, o rechazar la llamada. Por ejemplo, con una llamada al sistema `write()`, el usuario especifica una dirección de un búfer como fuente de la llamada de escritura. Si el usuario (accidental o maliciosamente) pasa una dirección "incorrecta" (por ejemplo, una dentro de la porción del kernel del espacio de direcciones), el SO debe detectar esto y rechazar la llamada. De lo contrario, un usuario podría leer toda la memoria del kernel; dado que la memoria (virtual) del kernel también suele incluir toda la memoria física del sistema, este pequeño desliz permitiría a un programa leer la memoria de cualquier otro proceso del sistema.
En general, un sistema seguro debe tratar los inputs de los usuarios con mucha sospecha. No hacerlo indudablemente conducirá a un software fácil de piratear, una sensación deseperante de que el mundo es un lugar inseguro y espantoso, y la pérdida de seguridad laboral para el/la desarrollador/a de SOs que confió demasiado.

Un último tema aparte: poder ejecutar la instrucción para decirle al hardware dónde están las tablas de traps es una capacidad muy poderosa. Por lo tanto, como habrás adivinado, también es una operación **privilegiada**. Si se intenta ejecutar esta instrucción en modo usuario, el hardware no lo permitirá y probablemente podés adivinar lo que va a suceder (pista: adios, programa transgresor!). Para reflexionar: ¿qué cosas horribles podrías hacerle a un sistema si pudieras instalar su propia tabla de traps? ¿Podrías tomar el control de la máquina?

La línea de tiempo (con el tiempo aumentando hacia abajo, en la Figura 6.2) resume el protocolo. Suponemos que cada proceso tiene un stack de kernel (o kernel stack) donde los registros (incluidos los registros de propósito general y el contador del programa) se guardan y se restauran (por el hardware) cuando se entra o sale del kernel.

Hay dos fases en el protocolo de ejecución directa limitada (**LDE**). En el primero (en el momento de boot), el kernel inicializa la tabla de traps y la CPU recuerda su ubicación para su uso futuro. El kernel lo hace mediante una instrucción privilegiada (todas las instrucciones

SO @ boot (al iniciar) (modo kernel)	Hardware	
inicializar tabla de traps		
	guardar dir. del gestor de llamadas al sistema	
SO @ corriendo modo kernel	**Hardware**	**Programa modo usuario**
Crear entrada en la lista de procesos		
Asignar memoria al programa y cargarlo en memoria		
Poner argc/argv en el stack		
Llenar el kernel stack con reg/PC		
retornar de la trap		
	restaurar registros (del kernel stack)	
	cambiar a modo usuario	
	saltar a main	
		correr main() ...
		realizar la llamada al sistema
		trap al SO
	guardar registros (en el kernel stack)	
	cambiar a modo kernel	
	saltar al trap handler	
Encargarse de la trap		
Hacer el trabajo de la llamada al sistema		
retornar de la trap		
	restaurar registros (del kernel stack)	
	cambiar a modo usuario	
	saltar al PC	
		... retornar de main
		trap (por exit())
Liberar la memoria		
Borrarlo de la lista de procesos		

Figure 6.2: **Protocolo de la Ejecución Directa Limitada**

privilegiadas están resaltadas en negrita). En el segundo (cuando se ejecuta un proceso), antes de usar una instrucción de retorno de la trap para iniciar la ejecución del proceso, el kernel configura algunas cosas (por ejemplo, asigna un nodo en la lista de procesos, asigna memoria); luego cambia la CPU a modo usuario y comienza a ejecutar el proceso. Cuando el proceso desea emitir una llamada al sistema, vuelve a entrar en el SO, que se encarga y una vez más devuelve el control al proceso a través de un retorno de la trap. El proceso luego completa su trabajo y retorna de `main()`; esto generalmente retornará a algún código auxiliar que saldrá correctamente del programa (por ejemplo, con la llamada al sistema `exit()`, que trapea al SO). En este punto, el sistema operativo se limpia y hemos terminado.

6.3 Problema #2: Cambio de procesos

El siguiente problema con la ejecución directa es lograr un intercambio de procesos. Intercambiar procesos debería ser simple, ¿verdad? El SO debería decidir detener un proceso e iniciar otro. ¿Cuál es el problema? En realidad, es un poco complicado: específicamente, si un proceso se está ejecutando en la CPU, por definición, significa que el SO *no* está corriendo. Si el SO no se está ejecutando, ¿cómo puede hacer algo? (pista: no puede). Si bien esto suena casi filosófico, es un problema real: claramente no hay forma de que el SO tome una acción si no se está ejecutando en la CPU. Llegamos así a nuestro problema.

LA CUESTIÓN: CÓMO RECUPERAR EL CONTROL DE LA CPU
¿Cómo puede el sistema operativo **recuperar el control** de la CPU para que pueda cambiar entre procesos?

Un Enfoque Cooperativo: Esperar las Llamadas al Sistema

Un enfoque que han adoptado algunos sistemas en el pasado (por ejemplo, las primeras versiones del SO de Macintosh [M11] o el antiguo sistema Xerox Alto [A79]) se conoce como el enfoque **cooperativo**. En este estilo, el SO confía en que los procesos del sistema se comporten de manera razonable. Se supone que los procesos que se ejecutan durante demasiado tiempo ceden periódicamente la CPU para que el SO pueda decidir ejecutar alguna otra tarea.

Así, podrías preguntarte, ¿cómo un proceso amigable cede la CPU en este mundo utópico? La mayoría de los procesos, transfieren el control de la CPU al SO con bastante frecuencia haciendo **llamadas al sistema**, por ejemplo, para abrir un archivo y luego leerlo, o para

enviar un mensaje a otra máquina, o para crear un nuevo proceso. Los sistemas como este a menudo incluyen una llamada al sistema explícita **yield**, que no hace nada más que transferir el control al sistema operativo para que pueda ejecutar otros procesos.

Las aplicaciones también transfieren el control al SO cuando hacen algo ilegal. Por ejemplo, si una aplicación se divide por cero o intenta acceder a la memoria a la que no debería poder acceder, generará una **trap** al SO. El cual volverá a tener el control de la CPU (y probablemente terminará el proceso infractor).

Por lo tanto, en un sistema de programación cooperativo, el SO recupera el control de la CPU esperando una llamada del sistema o una operación ilegal de algún tipo. También podrías estar pensando: ¿no es este enfoque pasivo menos que ideal? ¿Qué sucede, por ejemplo, si un proceso (ya sea malicioso o simplemente lleno de errores) termina en un bucle infinito y nunca realiza una llamada al sistema? ¿Qué puede hacer ahí el SO?

Un enfoque no Cooperativo: el SO toma el Control

Sin alguna ayuda adicional del hardware, resulta que el SO no puede hacer mucho cuando un proceso se niega a realizar llamadas al sistema (o errores) y así, devolver el control al SO. De hecho, en el enfoque cooperativo, su único recurso cuando un proceso se atasca en un bucle infinito es recurrir a la antigua solución para todos los problemas en los sistemas informáticos: **reiniciar la máquina**. Por lo tanto, llegamos nuevamente a un subproblema de nuestra búsqueda general para obtener el control de la CPU.

LA CUESTIÓN: CÓMO RECUPERAR EL CONTROL SIN COOPERACIÓN
¿Cómo puede el SO obtener el control de la CPU incluso si los procesos no son cooperativos? ¿Qué puede hacer el SO para garantizar que un proceso fraudulento no se apodere de la máquina?

La respuesta resulta simple y fue descubierta por varias personas que construían sistemas informáticos hace muchos años: una **interrupción por tiempo** [M+63]. Se puede programar un dispositivo temporizador para generar una interrupción cada tantos milisegundos; cuando se genera la interrupción, el proceso que se está ejecutando actualmente se detiene y se ejecuta un **gestor de interrupciones** preconfigurado en el SO. En este punto, el SO ha recuperado el control de la CPU y, por lo tanto, puede hacer lo que le plazca: detener el proceso actual e iniciar uno diferente.

Como comentamos antes con las llamadas al sistema, el SO debe informar al hardware sobre qué código ejecutar cuando se produce la interrupción por tiempo; por lo tanto, en el momento del boot, el

sistema operativo hace exactamente eso. En segundo lugar, también durante la secuencia de arranque, el sistema operativo debe iniciar el temporizador, que por supuesto es una operacion privilegiada. Una vez que el temporizador ha comenzado, el SO puede estar seguro de que el control finalmente será devuelto y, por lo tanto, el SO es libre de ejecutar programas de usuario. El temporizador también se puede apagar (también una operación privilegiada), algo que discutiremos más adelante cuando entendamos la concurrencia con más detalle.

TIP: LIDIAR CON LA MALA CONDUCTA DE APLICACIONES
Los sistemas operativos a menudo tienen que lidiar con procesos que se comportan mal, aquellos que, ya sea por diseño (malicia) o por accidente (errores), intentan hacer algo que no deberían. En los sistemas modernos, la forma en que el SO intenta manejar tal malversación es simplemente terminar al infractor. ¡Un strike y estás fuera! Quizás brutal, pero ¿qué más debería hacer el SO cuando intenta acceder a la memoria ilegalmente o ejecutar una instrucción ilegal?

Notar que el hardware tiene cierta responsabilidad cuando ocurre una interrupción, en particular para guardar suficiente información del estado del programa que se estaba ejecutando cuando ocurrió la interrupción, de modo que una instrucción posterior de retorno de la trap pueda reanudar el programa en ejecución correctamente. Este conjunto de acciones es bastante similar al comportamiento del hardware durante una trap por llamada al sistema explícita en el kernel, con varios registros que se guardan (por ejemplo, en un stack de kernel) y, por lo tanto, se restauran fácilmente mediante la instrucción de retorno de la trap.

Guardar y Restaurar el Contexto

Ahora que el SO ha recuperado el control, ya sea de forma cooperativa a través de una llamada al sistema o de forma más forzada a través de una interrupción por tiempo, se debe tomar una decisión: si continuar ejecutando el proceso que se está ejecutando actualmente o cambiar a uno diferente. Esta decisión la toma una parte del sistema operativo conocida como **planificador**; discutiremos las políticas de planificación con gran detalle en los próximos capítulos.

Si se toma la decisión de cambiar, el SO ejecuta un fragmento de código de bajo nivel al que llamamos **cambio de contexto**. Un cambio de contexto es conceptualmente simple: todo lo que el SO tiene que hacer es guardar algunos valores de registros para el proceso que se está ejecutando actualmente (en su stack de kernel, por ejemplo) y restaurar algunos para el proceso que pronto se ejecutará (desde su stack de kernel). Así, el SO se asegura de que cuando finalmente

se ejecute la instrucción de retorno de la trap, en lugar de regresar al proceso que se estaba ejecutando, el sistema reanude la ejecución de otro proceso.

Para guardar el contexto del proceso que se está ejecutando actualmente, el SO ejecutará un código assembly de bajo nivel para guardar los registros de propósito general, el PC (*program counter*) y el puntero del stack de kernel del proceso que se está ejecutando actualmente; y luego restaurará dichos registros, PC, y cambiará al stack de kernel del proceso que se está por ejecutar. Al cambiar de stack, el kernel ingresa a la llamada al código de cambio en el contexto de un proceso (el que fue interrumpido) y regresa en el contexto de otro (el que pronto se ejecutará). Cuando el SO finalmente ejecuta una instrucción de retorno de la trap, el proceso que antes se estaba por ejecutar se convierte en el proceso que se está ejecutando actualmente. Y así se completa el cambio de contexto.

TIP: USAR INTERRUPCIONES POR TIEMPO PARA RECUPERAR EL CONTROL

Sumar una interrupción por tiempo le da al SO la capacidad de ejecutarse nuevamente en una CPU incluso si los procesos actúan de manera no cooperativa. Por lo tanto, esta característica de hardware es esencial para ayudar al SO a mantener el control de la máquina.

En la Figura 6.3 se muestra una línea de tiempo de todo el proceso. En este ejemplo, el proceso A se está ejecutando y luego es interrumpido por el temporizador. El hardware guarda sus registros (en su stack de kernel) y entra en el kernel (cambiando a modo kernel). En el gestor de interrupciones del temporizador, el SO decide pasar de ejecutar el Proceso A al Proceso B. En ese punto, llama a la rutina `switch()`, que guarda cuidadosamente los valores de registro actuales (en la estructura del proceso de A), restaura los registros de Procese B (desde su entrada de estructura de proceso), y luego cambia de contexto, específicamente cambiando el puntero del stack para usar el stack de kernel de B (y no el de A). Finalmente, el SO retorna de la trap, que restaura los registros de B y comienza a correrlo.

Notar que hay dos tipos de guardado y restauración de registros que ocurren durante este protocolo. La primera es cuando ocurre la interrupción por tiempo; en este caso, los registros de usuario del proceso en ejecución son guardados implícitamente por el hardware, utilizando el stack de kernel de ese proceso. El segundo es cuando el SO decide cambiar de A a B; en este caso, los registros del kernel son guardados explícitamente por el software (es decir, el SO), pero esta vez en la memoria en la estructura del proceso. La última acción

TIP: REINICIAR ES ÚTIL
Anteriormente, notamos que la única solución para los bucles infinitos (y comportamientos similares) bajo la preferencia cooperativa es reiniciar la máquina. Si bien uno se puede burlar de este truco, los/as investigadores/as han demostrado que reiniciar (o en general, comenzar de nuevo con algún software) puede ser una herramienta muy útil para construir sistemas robustos [C+04].
Específicamente, el reinicio es útil porque devuelve el software a un estado conocido y probablemente más testeado. Los reinicios también recuperan recursos viciados o perdidos (por ejemplo, memoria) que de otra manera podrían ser difíciles de manejar. Finalmente, los reinicios son fáciles de automatizar. Por todas estas razones, no es raro en los servicios de Internet de clúster a gran escala que el software de administración del sistema reinicie periódicamente conjuntos de máquinas para reestablecerlas y así obtener las ventajas enumeradas anteriormente.
Por lo tanto, la próxima vez que reinicies, no estás simplemente realizando un truco feo. Más bien, estás utilizando un enfoque testeado para mejorar el comportamiento de un sistema informático. ¡Bien hecho!

hace que el sistema corra como si recién hubiera trapeado al kernel desde B, y no desde A.

Para darte una mejor idea de cómo se realiza dicho cambio, la Figura 6.4 muestra el código de cambio de contexto de xv6. Fijate si podés encontrarle sentido (vas a tener que saber un poco de x86, así como algo de xv6). Las estructuras del contexto “old” y “new” se encuentran en las estructuras del proceso antiguo y nuevo, respectivamente.

6.4 ¿Te preocupa la concurrencia?

Algunos/as de ustedes, como lectores/as atentos/as y reflexivos/as, pueden estar pensando ahora: ”Hmm ... ¿qué pasa cuando, durante una llamada al sistema, ocurre una interrupción del temporizador?” o ”¿Qué pasa cuando manejas una interrupción y ocurre otra? ¿No se vuelve difícil de manejar en el kernel? ” Buenas preguntas, ¡Todavía hay esperanzas en vos!

La respuesta es sí, el SO debe preocuparse por lo que sucede si, durante la interrupción o el manejo de traps, se produce otra interrupción. Este, de hecho, es exactamente el tema de toda la segunda parte de este libro, sobre concurrencia; aplazaremos una discusión detallada hasta entonces.

Para despertar tu apetito, esbozaremos algunos conceptos básicos

SO @ boot (al iniciar) (modo kernel)	Hardware	
inicializar tabla de traps		
	guardar direcciones de gestor de llamadas al sistema y del gestor del temporizador	
iniciar temporizador de interrupciones		
	iniciar temporizador interrumpir la CPU en X ms	
SO @ corriendo modo kernel	**Hardware**	**Programa modo usuario**
		Proceso A
		...
	interrupción por tiempo guardar regs(A) → k-stack(A) cambiar a modo kernel saltar al gestor de traps	
encargarse de la trap Llamar a la rutina switch() guardar regs(A) → proc t(A) restaurar regs(B) ← proc t(B) cambiar al k-stack(B) retornar de la trap (a B)	(del kernel stack)	
	restaurar regs(B) ← k-stack(B) cambiar a modo usuario Saltar al PC de B	
		Proceso B
		...

Figure 6.3: **Protocolo de la Ejecución Directa Limitada (Interrupción por tiempo)**

de cómo el SO maneja estas situaciones complicadas. Una cosa simple que puede hacer un SO es deshabilitar las interrupciones durante el manejo de interrupciones; al hacerlo, se asegura que cuando se procesa una interrupción, no llegará otra a la CPU. Por supuesto, el SO debe tener cuidado al hacerlo; deshabilitar las interrupciones durante demasiado tiempo podría provocar la pérdida de interrupciones, lo cual es (en términos técnicos) malo.

Los sistemas operativos también han desarrollado una serie de esquemas de **bloqueo** sofisticados para proteger el acceso simultáneo a las estructuras de datos internas. Esto permite que se realicen múltiples actividades dentro del kernel al mismo tiempo, lo que es

```
1  # void swtch(struct context **old,
2  #             struct context *new);
3  #
4  # Save current register context in old
5  # and then load register context from new.
6  .globl swtch
7  swtch:
8  # Save old registers
9  movl 4(%esp), %eax      # put old ptr into eax
10 popl 0(%eax)            # save the old IP
11 movl %esp, 4(%eax)      # and stack
12 movl %ebx, 8(%eax)      # and other registers
13 movl %ecx, 12(%eax)
14 movl %edx, 16(%eax)
15 movl %esi, 20(%eax)
16 movl %edi, 24(%eax)
17 movl %ebp, 28(%eax)
18 # Load new registers
19 movl 4(%esp), %eax      # put new ptr into eax
20 movl 28(%eax), %ebp     # restore other registers
21 movl 24(%eax), %edi
22 movl 20(%eax), %esi
23 movl 16(%eax), %edx
24 movl 12(%eax), %ecx
25 movl 8(%eax), %ebx
26 movl 4(%eax), %esp      # stack is switched here
27 pushl 0(%eax)           # return addr put in place
28 ret \                   # finally return into new ctxt
```

Figure 6.4: **El Código de Cambio de Contexto de xv6**

particularmente útil en multiprocesadores. Sin embargo, como veremos en la próxima parte de este libro sobre concurrencia, dicho bloqueo puede ser complicado y dar lugar a una variedad de errores interesantes y difíciles de encontrar.

6.5 Resumen

Hemos descripto algunos mecanismos clave de bajo nivel para implementar la virtualización de CPU, un conjunto de técnicas a las que nos referimos colectivamente como **ejecución directa limitada**. La idea básica es sencilla: simplemente ejecutar el programa que se desea correr en la CPU, pero primero hay que asegurarse de configurar el hardware para limitar lo que el proceso puede hacer sin la

APARTE: CUÁNTO TIEMPO TOMAN LOS CAMBIOS DE CONTEXTO
Una pregunta que naturalmente podrías hacerte es: ¿cuánto tiempo toma algo como un cambio de contexto? ¿O incluso una llamada al sistema? Para aquellos/as de ustedes que tengan curiosidad, existe una herramienta llamada **lmbench** [MS96] que mide exactamente esas cosas, así como algunas otras medidas de desempeño que podrían ser relevantes.
Los resultados han mejorado bastante con el tiempo, siguiendo aproximadamente el rendimiento del procesador. Por ejemplo, en 1996 ejecutando Linux 1.3.37 en una CPU P6 de 200 MHz, las llamadas al sistema tomaban aproximadamente 4 microsegundos y un cambio de contexto aproximadamente 6 microsegundos [MS96]. Los sistemas modernos funcionan casi un orden de magnitud mejor, con resultados de menos de microsegundos en sistemas con procesadores de 2 o 3 GHz.
Cabe señalar que no todas las acciones del sistema operativo rastrean el rendimiento de la CPU. Como observó Ousterhout, muchas operaciones del SO consumen mucha memoria y el ancho de banda de la memoria no ha mejorado tan drásticamente como la velocidad del procesador a lo largo del tiempo [O90]. Por lo tanto, dependiendo de su carga de trabajo, es posible que comprar el último y mejor procesador no acelere tu SO tanto como esperarías.

ayuda del SO.

Este enfoque general también se aplica en la vida real. Por ejemplo, aquellos de ustedes que tienen hijos/as, o, al menos, han oído hablar de niños/as, pueden estar familiarizados con el concepto de proteger una habitación para bebés: cerrar con llave los gabinetes que contienen cosas peligrosas y cubrir los enchufes. Cuando la habitación esté así preparada, se puede dejar que deambulen libremente, con la certeza de que los aspectos más peligrosos han sido restringidos.

De manera análoga, el SO prepara la CPU para que sea "a prueba de bebés", primero (durante el tiempo de boot) configurando los controladores de traps e iniciando un temporizador de interrupciones, y luego ejecutando solo procesos en un modo restringido. Al hacerlo, el SO puede sentirse bastante seguro de que los procesos pueden ejecutarse de manera eficiente, solo requiriendo la intervención del SO para realizar operaciones privilegiadas o cuando han monopolizado la CPU durante demasiado tiempo y, por lo tanto, deben cambiarse.

Así, tenemos los mecanismos básicos para virtualizar la CPU. Sin embargo, una pregunta importante que queda sin responder: ¿qué proceso deberíamos ejecutar en un momento dado? Es esta pregunta la que el planificador debe responder y, por lo tanto, el siguiente tema

de nuestro estudio.

APARTE: TÉRMINOS CLAVE DE VIRTUALIZACIÓN DE LA CPU (MECANISMOS)

- La CPU debe admitir al menos dos modos de ejecución: un **modo usuario** restringido y un **modo kernel** privilegiado (no restringido).
- Las aplicaciones de usuario típicas se ejecutan en modo de usuario y utilizan una **llamada al sistema** para **trapear** al kernel y solicitar servicios del sistema operativo.
- La instrucción trap guarda cuidadosamente el estado del registro, cambia el estado del hardware al modo kernel y salta al sistema operativo a un destino preestablecido: la **tabla de traps**.
- Cuando el SO termina de dar servicio a una llamada al sistema, regresa al programa de usuario a través de otra instrucción especial de **retorno de la trap**, que reduce el privilegio y devuelve el control a la instrucción después de la trampa que saltó al sistema operativo.
- El SO debe configurar las tablas de traps en el momento del booteo y asegurarse de que los programas de usuario no puedan modificarlas fácilmente. Todo esto es parte del protocolo de **ejecución directa limitada** que ejecuta programas de manera eficiente pero sin perder el control del SO.
- Una vez que un programa se está ejecutando, el SO debe utilizar mecanismos de hardware para garantizar que el programa de usuario no se ejecute para siempre, es decir, la **interrupción por temporizador**. Este es un enfoque **no-cooperativo** para la planificación de CPU.
- A veces, el SO, durante una interrupción del temporizador o una llamada al sistema, puede desear cambiar de ejecutar el proceso actual a uno diferente, una técnica de bajo nivel conocida como **cambio de contexto**.

6.6 Referencias

[A79] "Alto User's Handbook" de Xerox. Centro de Investigación de Xerox Palo Alto, septiembre de 1979. Disponible: `http://history-computer.com/Library/AltoUsersHandbook.pdf`. *Un sistema increíble, muy adelantado a su época. Se hizo famoso porque Steve Jobs visitó, tomó notas y construyó Lisa y eventualmente Mac.*

[C+04] "Microreboot — A Technique for Cheap Recovery" por G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. OSDI '04, San Francisco, CA, diciembre de 2004. *Un artículo excelente que señala hasta dónde se puede llegar con el reinicio en la construcción de sistemas más robustos.*

[I11] "Intel 64 and IA-32 Architectures Software Developer's Manual" por Volumen 3A y 3B:System Programming Guide. Intel Corporation, enero de 2011. *Este es solo un manual aburrido, pero a veces son útiles.*

[K+61] "One-Level Storage System" de T. Kilburn, DBG Edwards, MJ Lanigan, FH Sumner. IRE Transactions on Electronic Computers, abril de 1962. *El Atlas fue pionero en gran parte de lo que se ve en los sistemas modernos. Pero este documento no es el mejor para leer. Si solo leés uno, podrías probar la perspectiva histórica a continuación [L78].*

[L78] "The Manchester Mark I and Atlas: A Historical Perspective" por S. H. Lavington. Communications of the ACM, 21:1, enero de 1978. *Una historia del desarrollo temprano de las computadoras y los esfuerzos pioneros de Atlas.*

[M+63] "A Time-Sharing Debugging System for a Small Computer" por J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider. AFIPS '63 (Primavera), mayo de 1963, Nueva York, EE.UU. *"La tarea básica de la rutina de reloj del canal 17 es decidir si remover el usuario actual del núcleo, y si es así decidir a qué programa de usuario cambiar cuando se vaya."*

[MS96] "lmbench: Portable tools for performance analysis" por Larry McVoy y Carl Staelin. Conferencia técnica anual de USENIX, enero de 1996. *Un artículo divertido sobre cómo medir una serie de cosas diferentes sobre el SO y su rendimiento. Descargá lmbench y probalo.*

[M11] "Mac OS 9" de Apple Computer, Inc. Enero de 2011. http://en.wikipedia.org/wiki/ Mac OS 9 . *Probablemente incluso se pueda encontrar un emulador de OS 9 si querés; ¡probalo, es una pequeña y divertida Mac!*

[O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" por J. Ousterhout. Conferencia de verano de USENIX, junio de 1990. *Un artículo clásico sobre la naturaleza del rendimiento del sistema operativo.*

[P10] "The Single UNIX Specification, Version 3" de The Open Group, mayo de 2010. Disponible: http://www.unix.org/version3/. *Este es difícil y doloroso de leer, así que evitalo si podés. A menos que al-*

guien te pague por leerlo. ¡O simplemente tenés tanta curiosidad que no puedés evitarlo!

[S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" por Hovav Shacham. CCS '07, octubre de 2007. *Una de esas ideas asombrosas y alucinantes que ves en la investigación de vez en cuando. El autor muestra que si podés saltar al código arbitrariamente, básicamente podés unir cualquier secuencia de código que desees (dada una base de código grande); leé el documento para tener los detalles. La técnica hace que sea aún más difícil defenderse contra ataques maliciosos, por desgracia.*

6.7 Tarea (Medición)

APARTE: TAREAS DE MEDICIONES
Las tareas de medición son pequeños ejercicios en los que se escribe código para ejecutar en una máquina real, con el fin de medir algún aspecto del rendimiento del sistema operativo o del hardware. La idea detrás de estas tareas es brindarte un poco de experiencia práctica con un sistema operativo real.

En esta tarea, vas a medir los costos de una llamada al sistema y un cambio de contexto. Medir el costo de una llamada al sistema es relativamente fácil. Por ejemplo, podés llamar repetidamente a una simple llamada al sistema (por ejemplo, realizar una lectura de 0 bytes) y medir el tiempo que tarda; dividir el tiempo por el número de iteraciones le da una estimación del costo de una llamada al sistema.

Una cosa que deberías tener en cuenta es la precisión y exactitud de tu temporizador. Un temporizador típico que se puede usar es `gettimeofday()`; leé la man page para más detalles. Lo que vas a ver ahí es que `gettimeofday()` devuelve el tiempo en microsegundos desde 1970; sin embargo, esto no significa que el temporizador sea preciso al microsegundo. Medí las llamadas consecutivas a `gettimeofday()` para saber qué tan preciso es realmente el temporizador; esto te va a decir cuántas iteraciones de tu prueba de llamada al sistema nula tendrá que ejecutar para obtener un buen resultado de medición. Si `gettimeofday()` no es lo suficientemente preciso, podés considerar usar la instrucción `rdtsc`, disponible en máquinas x86.

Medir el costo de un cambio de contexto es un poco más complicado. El banco de pruebas de lmbench lo hace ejecutando dos procesos en una sola CPU y configurando dos UNIX pipes entre ellos; una pipe es solo una de las muchas formas en que los procesos de un sistema UNIX pueden comunicarse entre sí. Luego, el primer proceso emite una escritura en la primera pipe y espera una lectura en la segunda; al ver el primer proceso esperando que se lea algo de la segunda pipe, el SO pone el primer proceso en el estado bloqueado y cambia al otro proceso, que lee desde la primera pipe y luego escribe en la segunda. Cuando el segundo proceso intenta leer desde la primera pipe nuevamente, se bloquea y, por lo tanto, continúa el ciclo de comunicación de ida y vuelta. Midiendo el costo de comunicarse de esta manera repetidamente, lmbench puede hacer una buena estimación del costo de un cambio de contexto. Podés intentar recrear algo similar aquí, utilizando pipes, o quizás algún otro mecanismo de comunicación como sockets UNIX.

Una dificultad para medir el costo del cambio de contexto surge en sistemas con más de una CPU; lo que debe hacer en un sistema de este tipo es asegurarse de que sus procesos de cambio de contexto estén ubicados en el mismo procesador. Afortunadamente, la mayoría de los sistemas operativos tienen llamadas para vincular un proceso a un procesador en particular; en Linux, por ejemplo, la llamada `sched_setaffinity()` es lo que estás buscando. Al garantizar que ambos procesos estén en el mismo procesador, te estás asegurando de medir el costo de detener un proceso y restaurar otro en la misma CPU.